Oracle9*i*

Database Utilities

Release 2 (9.2)

March 2002

Part No. A96652-01

ORACLE®

Oracle9*i* Database Utilities, Release 2 (9.2)

Part No. A96652-01

Primary Author: Kathy Rich

Contributors: Lee Barton, Ellen Batbouta, Janet Blowney, George Claborn, Jay Davison, William Fisher, Dean Gagne, John Galanes, John Kalogeropoulos, Jonathan Klein, Cindy Lim, Eric Magrath, Brian McCarthy, Ray Pfau, Rich Phillips, Paul Reilly, Mike Sakayeda, Francisco Sanchez, Jim Stenoish

Contents

2 Import

Part II SQL*Loader

3 SQL*Loader Concepts

4 SQL*Loader Command-Line Reference

6 Field List Reference

7 Loading Objects, LOBs, and Collections

9 Conventional and Direct Path Loads

10 SQL*Loader Case Studies

Part III External Tables

11 External Tables Concepts

12 External Tables Access Parameters

Part IV Other Utilities

13 DBVERIFY: Offline Database Verification Utility

14 DBNEWID Utility

15 Using the Metadata API

Part V Appendixes

A SQL*Loader Syntax Diagrams

B DB2/DXT User Notes

C Backus-Naur Form Syntax

Index

List of Examples

List of Figures

List of Tables

Send Us Your Comments

Oracle9*i* Database Utilities, Release 2 (9.2)

Part No. A96652-01

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this document. 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?

If you find any errors or have any other suggestions for improvement, please indicate the document title and part number, and the chapter, section, and page number (if available). You can send comments to us in the following ways:

- Electronic mail: nedc-doc_us@oracle.com
- FAX: 603-897-3825 Attn: Oracle9*i* Database Utilities Documentation
- Postal service:

  Oracle Corporation
  Oracle9*i* Database Utilities Documentation
  One Oracle Drive
  Nashua, NH 03062-2804
  USA

If you would like a reply, please give your name, address, telephone number, and (optionally) electronic mail address.

If you have problems with the software, please contact your local Oracle Support Services.

Preface

This document describes how to use the Oracle9*i* database utilities for data transfer, data maintenance, and database administration.

This preface contains these topics:

- Audience
- Documentation Accessibility
- Organization
- Related Documentation
- Conventions

Audience

This document 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 and Import utilities
- Load data into Oracle tables from operating system files using SQL*Loader or from external sources using the external tables feature
- Extract and manipulate complete representations of the metadata for database objects, using the Metadata API
- Maintain the internal database identifier (DBID) and the database name (DBNAME) for an operational database, using the DBNEWID utility.

To use this manual, you need a working knowledge of SQL and Oracle fundamentals, information that is contained in *Oracle9i Database Concepts*. In addition, SQL*Loader requires that you know how to use the file management facilities of your operating system.

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible, with good usability, to the disabled community. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Standards will continue to evolve over time, and Oracle Corporation is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For additional information, visit the Oracle Accessibility Program Web site at

`http://www.oracle.com/accessibility/`

Accessibility of Code Examples in Documentation JAWS, a Windows screen reader, may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, JAWS may not always read a line of text that consists solely of a bracket or brace.

Organization

This document contains:

Part I, "Export and 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 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 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 Command-Line Reference"

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

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 datafiles, tables and columns, the location of data, the type and format of data to be loaded, and more.

Chapter 6, "Field List Reference"

This chapter describes the field list section of a SQL*Loader control file. The field list provides information about fields being loaded, such as position, datatype, conditions, and delimiters.

Chapter 7, "Loading Objects, LOBs, and Collections"

This chapter describes how to load column objects in various formats. It also discusses how to load object tables, REF columns, LOBs, and collections.

Chapter 8, "SQL*Loader Log File Reference"

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

Chapter 9, "Conventional and Direct Path Loads"

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

Chapter 10, "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.

Part III, "External Tables"

Chapter 11, "External Tables Concepts"

This chapter describes basic concepts about external tables.

Chapter 12, "External Tables Access Parameters"

This chapter describes the access parameters used to interface with the external tables API.

Part IV, "Other Utilities"

Chapter 13, "DBVERIFY: Offline Database Verification Utility"

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

Chapter 14, "DBNEWID Utility"

This chapter describes how to use the DBNEWID utility to change the name or ID, or both, for a database.

Chapter 15, "Using the Metadata API"

This chapter describes the Metadata API, which you can use to extract and manipulate complete representations of the metadata for database objects.

Part V, "Appendixes"

Appendix A, "SQL*Loader Syntax Diagrams"

This appendix provides diagrams of the SQL*Loader syntax.

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.

Appendix C, "Backus-Naur Form Syntax"

This appendix explains the symbols and conventions of the variant of Backus-Naur Form (BNF) used in text descriptions of syntax diagrams.

Related Documentation

For more information, see the following Oracle resources.

The Oracle9*i* documentation set, especially:

- *Oracle9i Database Concepts*
- *Oracle9i SQL Reference*
- *Oracle9i Database Administrator's Guide*

Many books in the documentation set use the sample schemas of the seed database, which is installed by default when you install the Oracle database server. Refer to *Oracle9i Sample Schemas* for information on how these schemas were created and how you can use them.

In North America, printed documentation is available for sale in the Oracle Store at

```
http://oraclestore.oracle.com/
```

Customers in Europe, the Middle East, and Africa (EMEA) can purchase documentation from

```
http://www.oraclebookshop.com/
```

Other customers can contact their Oracle representative to purchase printed documentation.

To download free release notes, installation documentation, white papers, or other collateral, please visit the Oracle Technology Network (OTN). You must register online before using OTN; registration is free and can be done at

```
http://otn.oracle.com/admin/account/membership.html
```

If you already have a username and password for OTN, then you can go directly to the documentation section of the OTN Web site at

```
http://otn.oracle.com/docs/index.htm
```

To access the database documentation search engine directly, please visit

`http://tahiti.oracle.com`

Conventions

This section describes the conventions used in the text and code examples of this documentation set. It describes:

- Conventions in Text
- Conventions in Code Examples

Conventions in Text

We use various conventions in text to help you more quickly identify special terms. The following table describes those conventions and provides examples of their use.

Convention	Meaning	Example
Bold	Bold typeface indicates terms that are defined in the text or terms that appear in a glossary, or both.	When you specify this clause, you create an **index-organized table.**
Italics	Italic typeface indicates book titles or emphasis.	*Oracle9i Database Concepts* Ensure that the recovery catalog and target database do *not* reside on the same disk.
`UPPERCASE monospace (fixed-width) font`	Uppercase monospace typeface indicates elements supplied by the system. Such elements include parameters, privileges, datatypes, RMAN keywords, SQL keywords, SQL*Plus or utility commands, packages and methods, as well as system-supplied column names, database objects and structures, usernames, and roles.	You can specify this clause only for a `NUMBER` column. You can back up the database by using the `BACKUP` command. Query the `TABLE_NAME` column in the `USER_TABLES` data dictionary view. Use the `DBMS_STATS.GENERATE_STATS` procedure.

Convention	Meaning	Example
`lowercase monospace (fixed-width) font`	Lowercase monospace typeface indicates executables, filenames, directory names, and sample user-supplied elements. Such elements include computer and database names, net service names, and connect identifiers, as well as user-supplied database objects and structures, column names, packages and classes, usernames and roles, program units, and parameter values. **Note:** Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.	Enter `sqlplus` to open SQL*Plus. The password is specified in the `orapwd` file. Back up the datafiles and control files in the `/disk1/oracle/dbs` directory. The `department_id`, `department_name`, and `location_id` columns are in the `hr.departments` table. Set the `QUERY_REWRITE_ENABLED` initialization parameter to `true.` Connect as `oe` user. The `JRepUtil` class implements these methods.
`lowercase italic monospace (fixed-width) font`	Lowercase monospace italic font represents placeholders or variables.	You can specify the *`parallel_clause`*. Run `U`*`old_release`*`.SQL` where *`old_release`* refers to the release you installed prior to upgrading.

Conventions in Code Examples

Code examples illustrate SQL, PL/SQL, SQL*Plus, or other command-line statements. They are displayed in a monospace (fixed-width) font and separated from normal text as shown in this example:

```
SELECT username FROM dba_users WHERE username = 'MIGRATE';
```

The following table describes typographic conventions used in code examples and provides examples of their use.

Convention	Meaning	Example
[ ]	Brackets enclose one or more optional items. Do not enter the brackets.	`DECIMAL (`*`digits`*` [ , `*`precision`*` ])`
{ }	Braces enclose two or more items, one of which is required. Do not enter the braces.	`{ENABLE | DISABLE}`

|	A vertical bar represents a choice of two or more options within brackets or braces. Enter one of the options. Do not enter the vertical bar.	`{ENABLE | DISABLE}`
		`[COMPRESS | NOCOMPRESS]`

Convention	Meaning	Example
`. . .`	Horizontal ellipsis points indicate either:	
	- That we have omitted parts of the code that are not directly related to the example	`CREATE TABLE ... AS` *`subquery`*`;`
	- That you can repeat a portion of the code	`SELECT` *`col1`*`,` *`col2`*`, ... ,` *`coln`* `FROM employees;`

.
.
.
	Vertical ellipsis points indicate that we have omitted several lines of code not directly related to the example.	`SQL> SELECT NAME FROM V$DATAFILE;`
		`NAME`
		`------------------------------------`
		`/fsl/dbs/tbs_01.dbf`
		`/fs1/dbs/tbs_02.dbf`
		`.`
		`.`
		`.`
		`/fsl/dbs/tbs_09.dbf`
		`9 rows selected.`

Other notation	You must enter symbols other than brackets, braces, vertical bars, and ellipsis points as shown.	`acctbal NUMBER(11,2);`
		`acct    CONSTANT NUMBER(4) := 3;`

`Italics`	Italicized text indicates placeholders or variables for which you must supply particular values.	`CONNECT SYSTEM/`*`system_password`*
		`DB_NAME =` *`database_name`*

`UPPERCASE`	Uppercase typeface indicates elements supplied by the system. We show these terms in uppercase in order to distinguish them from terms you define. Unless terms appear in brackets, enter them in the order and with the spelling shown. However, because these terms are not case sensitive, you can enter them in lowercase.	`SELECT last_name, employee_id FROM employees;`
		`SELECT * FROM USER_TABLES;`
		`DROP TABLE hr.employees;`

`lowercase`	Lowercase typeface indicates programmatic elements that you supply. For example, lowercase indicates names of tables, columns, or files.
	Note: Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.
		`SELECT last_name, employee_id FROM employees;`
		`sqlplus hr/hr`
		`CREATE USER mjones IDENTIFIED BY ty3MU9;`

What's New in Database Utilities?

This section describes new features of the Oracle9*i* database utilities and provides pointers to additional information. To help those who are upgrading to the current release, this section also describes features that were introduced in Oracle8*i*.

The information is divided into the following sections:

- Oracle9i Utilities New Features for Release 9.2
- Oracle9i Utilities New Features for Release 9.0.1
- Oracle8i Utilities New Features

Oracle9*i* Utilities New Features for Release 9.2

The following sections describe new and enhanced features that were introduced in Oracle9*i*, release 2.

Export and Import Utilities

The following is a list of new and enhanced features for the Export and Import utilities:

- New `OBJECT_CONSISTENT` parameter for Export that lets you export each object in its own read-only transaction, even if it is partitioned. See OBJECT_CONSISTENT on page 1-24.
- New `STREAMS_CONFIGURATION` parameter for Import that allows you to import any general streams metadata that may be present in the export dump file. See STREAMS_CONFIGURATION on page 2-28.

- New STREAMS_INSTANTIATION parameter for Import that allows you to import streams instantiation metadata that may be present in the export dump file. See STREAMS_INSTANTIATION on page 2-28.

SQL*Loader Utility

The following is a list of new and enhanced features for SQL*Loader:

- A new date cache feature reduces the actual number of date conversions done when many duplicate date values are present in the input data. This can increase performance during direct path loads. See Specifying a Value for the Date Cache on page 9-22.
- Overriding the default attribute-value constructor by creating one or more user-defined constructors. See Loading Column Objects with User-Defined Constructors on page 7-8.

External Tables

The following is a list of new and enhanced features for external tables:

- A new date cache feature reduces the actual number of date conversions done when many duplicate date values are present in the input data. This can increase performance during direct path loads. See Performance Hints When Using External Tables on page 11-6.

DBNEWID Utility

DBNEWID is a new database utility that can change the internal database identifier (DBID) and the database name (DBNAME) for an operational database. See Chapter 14, "DBNEWID Utility" for more information.

Metadata API

The Metadata API provides a centralized, simple, and flexible means for performing the following tasks:

- Extracting complete definitions of database objects (metadata) as either XML or creation DDL
- Transforming the metadata through industry-standard XSLT (Extensible Stylesheet Language Transformation).
- Generating SQL DDL to re-create the database objects

The Metadata API was available as of Oracle9*i*, release 9.0.1; however, it was documented in a different book. As of release 9.2, it is documented in this manual. See Chapter 15, "Using the Metadata API" for more information.

Oracle9*i* Utilities New Features for Release 9.0.1

The following sections describe new and enhanced features that were introduced for Oracle9*i* utilities.

Export and Import Utilities

The following is a list of new and enhanced features for the Export and Import utilities:

- Enhanced export/import functions for precalculated optimizer statistics. For more information, see:
  - STATISTICS on page 1-27 for information about use of this parameter with the Export utility
  - STATISTICS on page 2-27 for information about use of this parameter with the Import utility
  - Importing Statistics on page 2-68
- Addition of new parameters, `RESUMABLE`, `RESUMABLE_NAME`, `RESUMABLE_TIMEOUT`, `FLASHBACK_SCN`, and `FLASHBACK_TIME`. For more information, see the descriptions of these parameters beginning in Export Parameters on page 1-13 and in Import Parameters on page 2-14.
- Export mode can be used to dump out all tables in a tablespace. See TABLESPACES on page 1-30.
- Pattern matching of table names during export. See TABLES on page 1-28.
- Pattern matching of table names during import. See TABLES on page 2-29.
- Reduced character set translations on Import. See Character Set Conversion on page 2-55.

SQL*Loader Utility

The following is a list of new and enhanced features for SQL*Loader:

- SQL*Loader enhancements that allow for correctly loading integer and zoned / packed decimal datatypes across platforms. SQL*Loader can now do the following:
  - Load binary integer data created on a platform whose byte ordering is different than that of the target platform
  - Load binary floating-point data created on a platform whose byte ordering is different than that of the target platform (if the floating-point format used by source and target systems is the same)
  - Specify the size, in bytes, of a binary integer and load it regardless of the target platform's native integer size
  - Specify that integer values are to be treated as signed or unsigned quantities
  - Accept EBCDIC-based zoned or packed decimal data encoded in IBM format

  For more information on these enhancements, see the following:

  - INTEGER(n) on page 6-8
  - DECIMAL on page 6-11
  - ZONED on page 6-10
  - Loading Data Across Different Platforms on page 6-36
- Support for loading `XML` columns. See Loading LOBs on page 7-18.
- Support for loading object tables with a subtype. See Loading Object Tables with a Subtype on page 7-13.
- Support for loading column objects with a derived subtype. See Loading Column Objects with a Derived Subtype on page 7-4.
- SQL*Loader support for Unicode. This support includes the following:
  - Use of the UTF16 character set in the SQL*Loader datafile
  - Support of character-length semantics in the SQL*Loader control file
  - Use of SQL*Loader to load data into columns of datatype `NCHAR`, `NVARCHAR2`, and `NCLOB` when the national character set is AL16UTF16
  - Specifying byte order (big endian or little endian) for SQL*Loader datafile

See Also:

- Handling Different Character Encoding Schemes on page 5-16
- SQL*Loader Datatypes on page 6-7
- Byte Ordering on page 6-37

- SQL*Loader extensions for support of datetime and interval datatypes as specified in the ANSI SQL 92 standards document. This support includes the ability to:

  - Load datetime and interval datatypes for both conventional and direct path modes of SQL*Loader
  - Perform datetime and interval datatype conversions between SQL*Loader client and database server
  - Load datetime and interval datatypes using the direct path API

  For more information, see Datetime and Interval Datatypes on page 6-16.

- New functionality that allows users to specify the `UNSIGNED` parameter for the binary integers, `SMALLINT` and `INTEGER(`*n*`)`. For more information, see SMALLINT on page 6-9 and INTEGER(n) on page 6-8.
- New functionality that allows a length specification to be applied to the `INTEGER` parameter; for example, `INTEGER(`*n*`)`. See INTEGER(n) on page 6-8.
- New multithreaded loading functionality for direct path loads that, when possible, converts column arrays to stream buffers and performs stream buffer loading in parallel. For more information, see Optimizing Direct Path Loads on Multiple-CPU Systems on page 9-23.
- New `COLUMNARRAYROWS` parameter that lets you specify a value for the number of column array rows in direct path loads. And a new `STREAMSIZE` parameter that lets you specify the size of direct path stream buffers. For more

information, see Specifying the Number of Column Array Rows and Size of Stream Buffers on page 9-21.

- Addition of RESUMABLE, RESUMABLE_NAME, and RESUMABLE_TIMEOUT parameters to enable and disable resumable space allocation. See Command-Line Parameters on page 4-3.

External Tables

The Oracle9*i* external tables feature is a complement to existing SQL*Loader functionality. It allows you to access data in external sources as if it were in a table in the database.

See Also:

- Chapter 11, "External Tables Concepts"
- Chapter 12, "External Tables Access Parameters"

DBVERIFY Utility

The DBVERIFY utility now has an additional command-line interface that allows you to specify a table segment or index segment for verification. It checks to make sure that a row chain pointer is within the segment being verified. See Using DBVERIFY to Validate a Segment on page 13-4.

Oracle8*i* Utilities New Features

The Oracle8*i* new features described in this section comprise the overall effort to optimize data transfer, maintenance, and administration. The features described in this section were added for releases 8.1.5, 8.1.6, and 8.1.7.

Export Utility

The following are new or enhanced Export features:

- Export of subpartitions. See Table-Level and Partition-Level Export on page 1-12.
- The ability to specify multiple dump files for an export command. See the parameters FILE on page 1-20 and FILESIZE on page 1-21.
- The ability to specify a query for the SELECT statements that Export uses to unload tables. See QUERY on page 1-24.
- The maximum number of bytes in an export file on each volume of tape has been increased. See VOLSIZE on page 1-32.
- The ability to export tables containing LOBs and objects, even if direct path is specified on the command line. See Invoking a Direct Path Export on page 1-52.
- The ability to export and import precalculated optimizer statistics instead of recalculating the statistics at import time. (This feature is only applicable to certain exports and tables.) See STATISTICS on page 1-27.
- Developers of domain indexes can export application-specific metadata associated with an index using the new ODCIIndexGetMetadata method on the ODCIIndex interface. See the *Oracle9i Data Cartridge Developer's Guide* for more information.
- Export of transportable tablespace metadata. See TRANSPORT_TABLESPACE on page 1-31.

Import Utility

The following are new or enhanced Import features:

- Import of subpartitions. See Table-Level and Partition-Level Import on page 2-49.
- The ability to specify multiple dump files for an Import command. See the parameters FILE on page 2-21 and FILESIZE on page 2-21.
- 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 2-31.
- The maximum number of bytes in an export file on each volume of tape has been increased. See VOLSIZE on page 2-34.
- Support for fine-grained access control. See Considerations When Importing Database Objects on page 2-56.
- 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 2-27.
- Import of transportable tablespace metadata. See TRANSPORT_TABLESPACE on page 2-33.

SQL*Loader Utility

The following are new or enhanced SQL*Loader features:

- There is now a PRESERVE parameter for use with CONTINUEIF THIS and CONTINUEIF NEXT.

  If the PRESERVE parameter is not used, the continuation field is removed from all physical records when the logical record is assembled. That is, data values are allowed to span the records with no extra characters (continuation characters) in the middle.

  If the PRESERVE parameter is used, the continuation field is kept in all physical records when the logical record is assembled.

  See Using CONTINUEIF to Assemble Logical Records on page 5-27.
- DATE fields that contain only whitespace are loaded as NULL fields and, therefore, no longer cause an error. See Datetime and Interval Datatypes on page 6-16.
- As of release 8.1.5, the behavior of certain DDL clauses and restrictions has been changed to provide object support. Be sure to read Chapter 7, "Loading Objects, LOBs, and Collections" for a complete description of how this now works.

Additionally, you should be sure to read the information in the following sections:

– Specifying Filler Fields on page 6-6

– Using the WHEN, NULLIF, and DEFAULTIF Clauses on page 6-32

– Applying SQL Operators to Fields on page 6-50

Part I

Export and Import

The chapters in this section describe the Oracle Export and Import utilities:

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 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 Import object support. It also provides several examples of Import sessions.

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).

This chapter discusses the following topics:

- What Is the Export Utility?
- Before Using Export
- Invoking Export
- Export Modes
- Getting Online Help
- Export Parameters
- Example Export Sessions
- Using the Interactive Method
- Warning, Error, and Completion Messages
- Exit Codes for Inspection and Display
- Conventional Path Export Versus Direct Path Export
- Invoking a Direct Path Export
- Network Considerations
- Character Set and Globalization Support Considerations
- Instance Affinity and Export
- Considerations When Exporting Database Objects

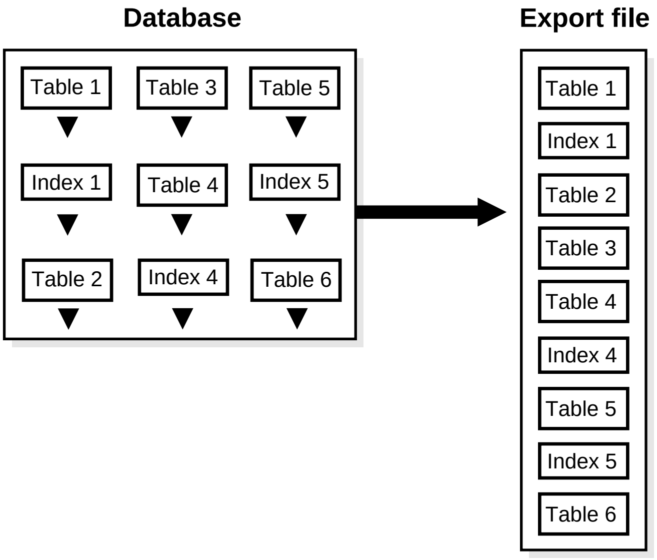
- Transportable Tablespaces
- Exporting from a Read-Only Database
- Using Export and Import to Partition a Database Migration
- Using Different Releases and Versions of Export

What Is the Export Utility?

The Export utility 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.

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. The extracted data is written to an Export file, as illustrated in Figure 1–1.

Figure 1–1 Exporting a Database



An Export file is an Oracle binary-format dump file that is typically located on disk or tape. The dump files can be transferred using FTP or physically transported (in the case of tape) to a different site. The files can then be used with the Import utility to transfer data between databases that are on systems not connected through a network. The files can also be used as backups in addition to normal backup procedures.

Export dump files can only be read by the Oracle Import utility. The version of the Import utility cannot be earlier than the version of the Export utility used to create the dump file.

You can also display the contents of an export file without actually performing an import. To do this, use the Import `SHOW` parameter. See SHOW on page 2-27 for more information.

To load data from ASCII fixed-format or delimited files, use the SQL*Loader utility.

See Also:

- Using Different Releases and Versions of Export on page 1-61
- Chapter 2 for information about the Import utility
- Part II of this manual for information about the SQL*Loader utility
- *Oracle9i Replication* for information on how to use the Export and Import utilities to facilitate certain aspects of Oracle Advanced Replication, such as offline instantiation

Before Using Export

Before you begin using Export, be sure you take care of the following items (described in detail in the following sections):

- Run the `catexp.sql` or `catalog.sql` script
- Ensure there is sufficient disk or tape storage to write the export file
- Verify that you have the required access privileges

Running catexp.sql or catalog.sql

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 filenames 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 in the data dictionary
- Creates the EXP_FULL_DATABASE role
- Assigns all necessary privileges to the EXP_FULL_DATABASE role
- Assigns EXP_FULL_DATABASE to the DBA role
- Records the version of catexp.sql that has been installed

Ensuring Sufficient Disk Space

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. You can find table sizes 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 in partitioned tables.

See Also: *Oracle9i Database Reference* for more information about dictionary views

Verifying 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.

The following schema names are reserved and will not be processed by Export:

- ORDSYS
- MDSYS
- CTXSYS
- ORDPLUGINS
- LBACSYS

Invoking Export

You can invoke Export and specify parameters by using any of the following methods:

- Command-line entries
- Interactive Export prompts
- Parameter files

Before you use one of these methods to invoke Export, be sure to read the descriptions of the available parameters. See Export Parameters on page 1-13.

Command-Line Entries

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

```
exp username/password PARAMETER=value
```

or

```
exp username/password PARAMETER=(value1,value2,...,valuen)
```

The number of parameters cannot exceed the maximum length of a command line on the system.

Interactive Export Prompts

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

```
exp username/password
```

Export will display commonly used parameters with a request for you to enter a value. This method exists for backward compatibility and is not recommended because it provides less functionality than the other methods. See Using the Interactive Method on page 1-43.

Parameter Files

You can specify all valid parameters and their values in a parameter file. Storing the parameters in a file allows them to be easily modified or reused, and is the recommended method for invoking Export. If you use different parameters for different databases, you can have multiple parameter files.

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 first example does not specify the *`username/password`* on the command line to illustrate that you can specify them in the parameter file, although, for security reasons, this is not recommended.

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

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

The following example shows a partial parameter file listing:

```
FULL=y
FILE=dba.imp
GRANTS=y
INDEXES=y
CONSISTENT=y
```

Note: 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.

You can specify a parameter file at the same time that you are entering parameters 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 which parameters take precedence. For example, assume the parameter file `params.dat` contains the parameter `INDEXES=y` and Export is invoked with the following line:

```
exp username/password 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 parameter file.

See Also:

- Export Parameters on page 1-13 for descriptions of the Export parameters
- Exporting and Importing with Oracle Net on page 1-53 for information on how to specify an export from a remote database

Invoking Export As SYSDBA

`SYSDBA` is used internally and has specialized functions; its behavior is not the same as for generalized users. Therefore, you should not typically need to invoke Export as `SYSDBA,` except in the following situations:

- At the request of Oracle technical support
- When using transportable tablespaces (see Transportable Tablespaces on page 1-59)

To invoke Export as `SYSDBA,` use the following syntax, adding any desired parameters or parameter filenames:

```
exp \'username/password AS SYSDBA\'
```

Optionally, you could also specify an instance name:

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

If either the username or password is omitted, Export will prompt you for it.

This example shows the entire connect string enclosed in single quotation marks and backslashes. This is because the string, `AS SYSDBA,` contains a blank, a situation for which most operating systems require that the entire connect string be placed in quotation marks or marked as a literal by some method. Some operating systems also require that quotation marks on the command line be preceded by an escape character. In this example, backslashes are used as the escape character. If the backslashes were not present, the command-line parser that Export uses would not understand the quotation marks and would remove them before calling Export.

See your Oracle operating system-specific documentation for more information about special and reserved characters on your system.

If you prefer to use the Export interactive mode, see Using the Interactive Method on page 1-43 for more information.

Export Modes

The Export utility provides four modes of export:

- Full
- User (Owner)
- Table
- Tablespace

All users can export in table mode and user mode. Users with the `EXP_FULL_DATABASE` role (privileged users) can export in all modes. Table 1–1 shows the objects that are exported and imported in each mode. Also see Processing Restrictions on page 1-13.

To specify one of these modes, use the appropriate parameter (`FULL, OWNER, TABLES,` or `TABLESPACES`) when you invoke Export. See Export Parameters on page 1-13 for information on the syntax for each of these parameters.

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 Conventional Path Export Versus Direct Path Export on page 1-50.

See Also:

- *Oracle9i Database Administrator's Guide*
- *Oracle9i Database Concepts* for an introduction to the transportable tablespaces feature

Table 1–1 Objects Exported and Imported in Each Mode

Object	Table Mode	User Mode	Full Database Mode	Tablespace Mode
Analyze cluster	No	Yes	Yes	No
Analyze tables / statistics	Yes	Yes	Yes	Yes
Application contexts	No	No	Yes	No
Auditing information	Yes	Yes	Yes	No
B-tree, bitmap, domain functional indexes	Yes[1]	Yes[1]	Yes	Yes
Cluster definitions	No	Yes	Yes	Yes
Column and table comments	Yes	Yes	Yes	Yes
Database links	No	Yes	Yes	No
Default roles	No	No	Yes	No
Dimensions	No	Yes	Yes	No
Directory aliases	No	No	Yes	No
External tables (without data)	Yes	Yes	Yes	No
Foreign function libraries	No	Yes	Yes	No
Indexes owned by users other than table owner	Yes (Privileged users only)	Yes	Yes	Yes
Index types	No	Yes	Yes	No

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

Object	Table Mode	User Mode	Full Database Mode	Tablespace Mode
Java resources and classes	No	Yes	Yes	No
Job queues	No	Yes	Yes	No
Nested table data	Yes	Yes	Yes	Yes
Object grants	Yes (Only for tables and indexes)	Yes	Yes	Yes
Object type definitions used by table	Yes	Yes	Yes	Yes
Object types	No	Yes	Yes	No
Operators	No	Yes	Yes	No
Password history	No	No	Yes	No
Postinstance actions and objects	No	No	Yes	No
Postschema procedural actions and objects	No	Yes	Yes	No
Posttable actions	Yes	Yes	Yes	Yes
Posttable procedural actions and objects	Yes	Yes	Yes	Yes
Preschema procedural objects and actions	No	Yes	Yes	No
Pretable actions	Yes	Yes	Yes	Yes
Pretable procedural actions	Yes	Yes	Yes	Yes
Private synonyms	No	Yes	Yes	No
Procedural objects	No	Yes	Yes	No
Profiles	No	No	Yes	No
Public synonyms	No	No	Yes	No
Referential integrity constraints	Yes	Yes	Yes	No

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

Object	Table Mode	User Mode	Full Database Mode	Tablespace Mode
Refresh groups	No	Yes	Yes	No
Resource costs	No	No	Yes	No
Role grants	No	No	Yes	No
Roles	No	No	Yes	No
Rollback segment definitions	No	No	Yes	No
Security policies for table	Yes	Yes	Yes	Yes
Sequence numbers	No	Yes	Yes	No
Snapshot logs	No	Yes	Yes	No
Snapshots and materialized views	No	Yes	Yes	No
System privilege grants	No	No	Yes	No
Table constraints (primary, unique, check)	Yes	Yes	Yes	Yes
Table data	Yes	Yes	Yes	No
Table definitions	Yes	Yes	Yes	Yes
Tablespace definitions	No	No	Yes	No
Tablespace quotas	No	No	Yes	No
Triggers	Yes	Yes[2]	Yes[3]	Yes
Triggers owned by other users	Yes (Privileged users only)	No	No	No
User definitions	No	No	Yes	No
User proxies	No	No	Yes	No
User views	No	Yes	Yes	No

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

Object	Table Mode	User Mode	Full Database Mode	Tablespace Mode
User-stored procedures, packages, and functions	No	Yes	Yes	No

[1] Nonprivileged 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] Nonprivileged and privileged users can export and import all triggers owned by the user, even if they are on tables owned by other users.

[3] A full export does not export triggers owned by schema SYS. You must manually re-create SYS triggers either before or after the full import. Oracle Corporation recommends that you re-create them after the import in case they define actions that would impede progress of the import.

Table-Level and Partition-Level Export

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

- **Table-level Export:** exports all data from the specified tables
- **Partition-level Export:** exports only data from the specified source partitions or subpartitions

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

Table-Level Export

In table-level Export, you can export an entire table (partitioned or nonpartitioned) along with its indexes and other table-dependent objects. If the table is partitioned, all of its partitions and subpartitions are also exported. This applies to both direct path Export and conventional path Export. You can perform a table-level export in any Export mode.

Partition-Level Export

In partition-level Export, you can export one or more specified partitions or subpartitions of a table. You can only perform a partition-level export in Table mode.

For information on how to specify table-level and partition-level Exports, see TABLES on page 1-28.

Processing Restrictions

The following restrictions apply when you process data with the Export and Import utilities:

- Java classes, resources, and procedures that are created using Enterprise Java Beans (EJBs) are not placed in the export file.
- Constraints that have been altered using the RELY keyword lose the RELY attribute when they are exported.
- When a type definition has evolved and then data referencing that evolved type is exported, the type definition on the import system must have evolved in the same manner.

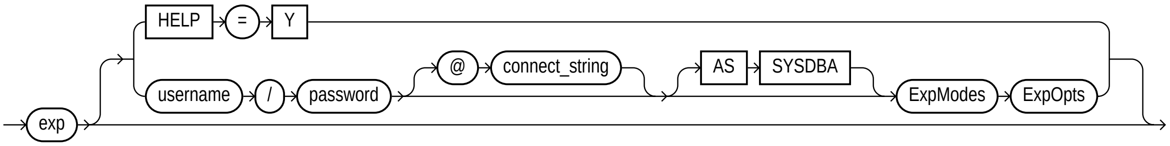
Getting Online Help

Export provides online help. Enter `exp help=y` on the command line to invoke it.

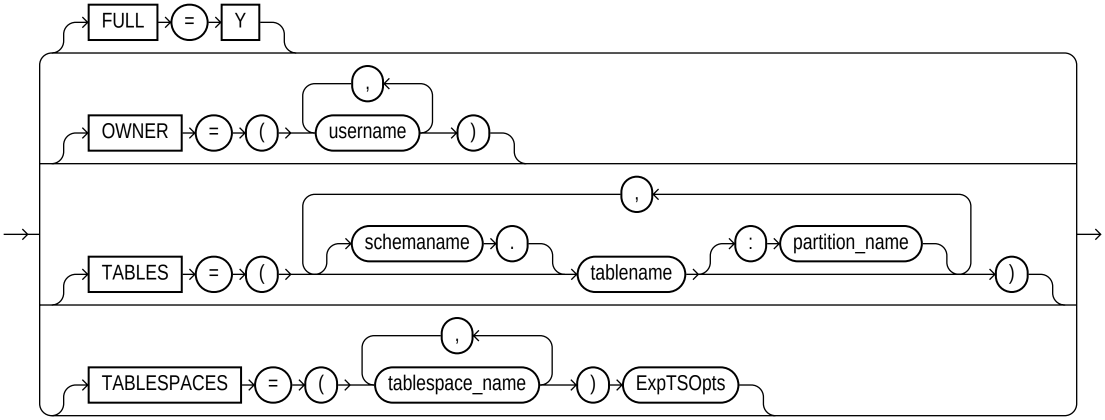
Export Parameters

The following diagrams show the syntax for the parameters that you can specify in the parameter file or on the command line. Following the diagrams are descriptions of each parameter.

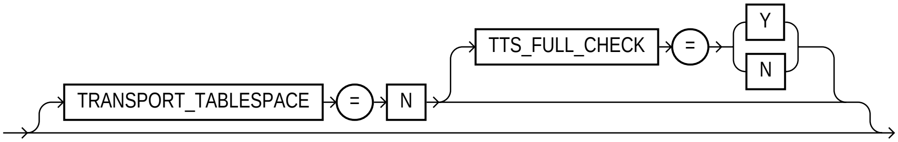
Export_start



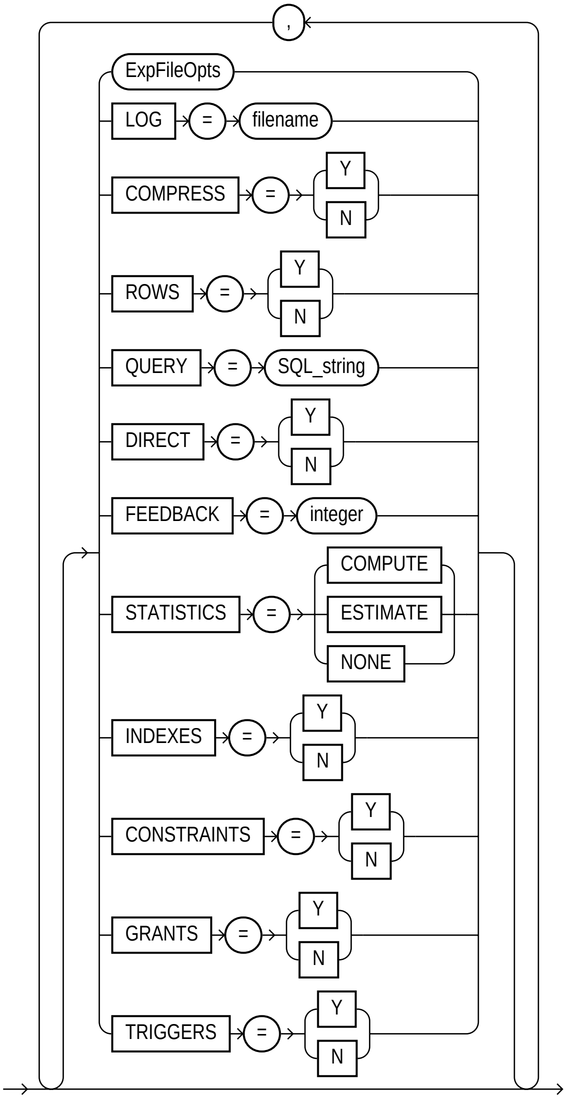
ExpModes



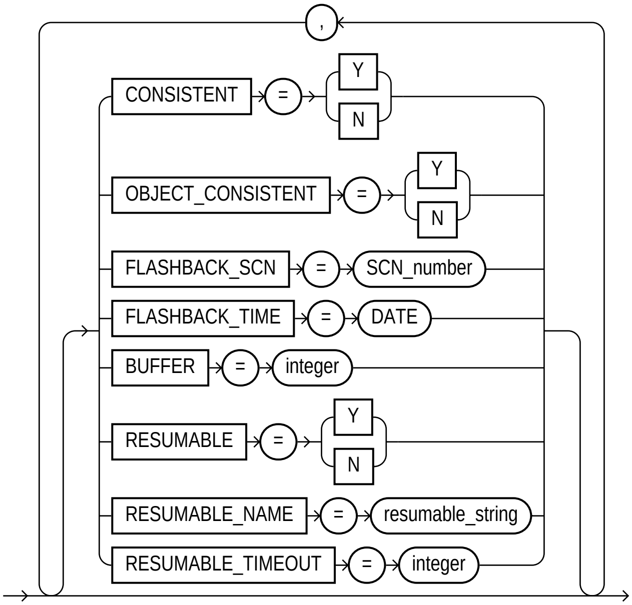
ExpTSOpts (tablespaces_spec)



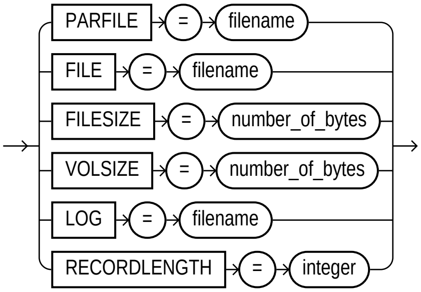
ExpOpts


,
ExpFileOpts
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

ExpOpts_continued



ExpFileOpts



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 columns of type LONG, LOB, BFILE, REF, ROWID, LOGICAL ROWID, or DATE 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.

Example: Calculating Buffer Size

This section shows an example of how to calculate buffer size.

The following table is created:

```
CREATE TABLE sample (name varchar(30), weight number);
```

The maximum size of the name column is 30, plus 2 bytes for the indicator. The maximum size of the weight column is 22 (the size of the internal representation for Oracle numbers), plus 2 bytes for the indicator.

Therefore, the maximum row size is 56 (30+2+22+2).

To perform array operations for 100 rows, a buffer size of 5600 should be specified.

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. Therefore, 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, values of initial extent size and next extent size at the time of export 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 exp 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 use CONSISTENT=n, 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.

Table 1–2 shows a sequence of events by two users: user1 exports partitions in a table and user2 updates data in that table.

Table 1–2 Sequence of Events During Updates by Two Users

TIme Sequence	User1	User2
1	Begins export of TAB:P1	No activity

Table 1–2 (Cont.) Sequence of Events During Updates by Two Users

TIme Sequence	User1	User2
2	No activity	Updates TAB:P2 Updates TAB:P1 Commits transaction
3	Ends export of TAB:P1	No activity
4	Exports TAB:P2	No activity

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, the user2 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 usage 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:

- CONSISTENT=y is unsupported for exports that are performed when you are connected as user SYS or you are using AS SYSDBA, or both.
- Export of certain metadata may require the use of the SYS schema within recursive SQL. In such situations, the use of CONSISTENT=y will be ignored. Oracle Corporation recommends that you avoid making metadata changes during an export process in which CONSISTENT=y is selected.
- 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.
- A "snapshot too old" error occurs when rollback space is 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.

See Also: OBJECT_CONSISTENT on page 1-24

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 information about direct path Exports, including security and performance considerations, see Invoking a Direct Path Export on page 1-52.

FEEDBACK

Default: `0` (zero)

Specifies that Export should display a progress meter in the form of a period for *`n`* number of rows exported. For example, if you specify `FEEDBACK=10,` Export displays a period 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. Because Export supports multiple export files (see the parameter FILESIZE on page 1-21), you can specify multiple filenames to be used. For example:

```
exp scott/tiger FILE = dat1.dmp, dat2.dmp, dat3.dmp FILESIZE=2048
```

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 FILE parameter, 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

Default: Data is written to one file until the maximum size, as specified in Table 1–3, is reached.

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 1-20 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 the space requirements of your export file exceed the available disk space, Export will abort, and you will have to repeat the Export after making sufficient disk space available.

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

Table 1–3 shows that the maximum size for dump files depends on the operating system you are using and on the release of the Oracle database server that you are using.

Table 1–3 Maximum Size for Dump Files

Operating System	Release of Oracle Server	Maximum Size
Any	Prior to 8.1.5	2 gigabytes
32-bit	8.1.5	2 gigabytes
64-bit	8.1.5 and later	Unlimited
32-bit with 32-bit files	Any	2 gigabytes
32-bit with 64-bit files	8.1.6 and later	Unlimited

Note: The maximum value that can be stored in a file is dependent on your operating system. You should verify this maximum value in your Oracle 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 KB (number of kilobytes). For example, FILESIZE=2KB is the same as FILESIZE=2048. Similarly, MB specifies megabytes (1024 * 1024) and GB specifies gigabytes (1024**3). B remains the shorthand for bytes; the number is not multiplied to obtain the final file size (FILESIZE=2048B is the same as FILESIZE=2048).

FLASHBACK_SCN

Default: none

Specifies the system change number (SCN) that Export will use to enable flashback. The export operation is performed with data consistent as of this specified SCN.

See Also: *Oracle9i Application Developer's Guide - Fundamentals* for more information about using flashback

FLASHBACK_TIME

Default: none

Specifies a time. Export finds the SCN that most closely matches the specified time. This SCN is used to enable flashback. The export operation is performed with data consistent as of this SCN.

See Also: *Oracle9i Application Developer's Guide - Fundamentals* for more information about using flashback

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 to have 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 mode or user mode. In full database mode, all grants on a table are exported. In user mode, only those granted by the owner of the table are exported. System privilege grants are always exported.

HELP

Default: none

Displays a description of the Export parameters. Enter exp help=y on the command line to invoke it.

INDEXES

Default: y

Specifies whether or not the Export utility exports indexes.

LOG

Default: none

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

```
exp SYSTEM/password LOG=export.log
```

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

OBJECT_CONSISTENT

Default: n

Specifies whether or not the Export utility uses the SET TRANSACTION READ ONLY statement to ensure that the data exported is consistent to a single point in time and does not change during the export. If OBJECT_CONSISTENT is set to y, each object is exported in its own read-only transaction, even if it is partitioned. In contrast, if you use the CONSISTENT parameter, then there is only one read-only transaction.

See Also: CONSISTENT on page 1-18

OWNER

Default: none

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: none

Specifies a filename for a file that contains a list of Export parameters. For more information on using a parameter file, see Invoking Export on page 1-5.

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 that 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 less than 1600, he could do the following (this example is UNIX-based):

```
exp scott/tiger TABLES=emp QUERY=\"WHERE job=\'SALESMAN\' and sal \<1600\"
```

Note: Because 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 quotation marks or marked as a literal by some method. Operating system reserved characters also need to be preceded by an escape character. See your Oracle operating system-specific documentation for information about special and reserved characters on your system.

When executing this query, Export builds a SQL SELECT statement similar to the following:

```
SELECT * FROM emp WHERE job='SALESMAN' and sal <1600;
```

The values specified for the QUERY parameter are applied to all tables (or table partitions) listed in the TABLE parameter. For example, the following statement will unload rows in both emp and bonus that match the query:

```
exp scott/tiger TABLES=emp,bonus QUERY=\"WHERE job=\'SALESMAN\' and sal\<1600\"
```

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

```
SELECT * FROM emp WHERE job='SALESMAN' and sal <1600;

SELECT * FROM bonus WHERE job='SALESMAN' and sal <1600;
```

If a table is missing the columns specified in the QUERY clause, an error message will be produced, and no rows will be exported for the offending table.

Restrictions

- The parameter QUERY cannot be specified for full, user, or 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 cannot determine from the contents of the export file whether the data is the result of a QUERY export.

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 buffer size. For more information about the buffer size default value, see your Oracle operating system-specific documentation.

You can set `RECORDLENGTH` to any value equal to or greater than your system's buffer size. (The highest value is 64 KB.) 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.

See your Oracle operating system-specific documentation to determine the proper value or to create a file with a different record size.

RESUMABLE

Default: `n`

The `RESUMABLE` parameter is used to enable and disable resumable space allocation. Because this parameter is disabled by default, you must set `RESUMABLE=y` in order to use its associated parameters, `RESUMABLE_NAME` and `RESUMABLE_TIMEOUT`.

See Also:

- *Oracle9i Database Concepts*
- *Oracle9i Database Administrator's Guide* for more information about resumable space allocation

RESUMABLE_NAME

Default: `'User USERNAME (USERID), Session SESSIONID, Instance INSTANCEID'`

The value for this parameter identifies the statement that is resumable. This value is a user-defined text string that is inserted in either the USER_RESUMABLE or DBA_RESUMABLE view to help you identify a specific resumable statement that has been suspended.

This parameter is ignored unless the RESUMABLE parameter is set to y to enable resumable space allocation.

RESUMABLE_TIMEOUT

Default: 7200 seconds (2 hours)

The value of the parameter specifies the time period during which an error must be fixed. If the error is not fixed within the timeout period, execution of the statement is aborted.

This parameter is ignored unless the RESUMABLE parameter is set to y to enable resumable space allocation.

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 Import parameter STATISTICS on page 2-27 and Importing Statistics on page 2-68.

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

However, the precalculated optimizer statistics will not be used at export time if a table has columns with system-generated names.

The precalculated optimizer statistics are flagged as questionable at export time if:

- There are row errors while exporting
- The client character set or NCHAR character set does not match the server character set or NCHAR character set
- A QUERY clause is specified

- Only certain partitions or subpartitions are exported

> **Note:** Specifying `ROWS=n` does not preclude saving the precalculated statistics in the Export file. This allows you to tune plan generation for queries in a nonproduction database using statistics from a production database.

See Also: *Oracle9i Database Concepts*

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:

- *`schemaname`* specifies the name of the user's schema from which to export the table or partition. The schema names `ORDSYS`, `MDSYS`, `CTXSYS`, and `ORDPLUGINS` are reserved by Export.
- *`tablename`* specifies the name of the table or tables to be exported. Table-level export lets you export entire partitioned or nonpartitioned tables. If a table in the list is partitioned and you do not specify a partition name, all its partitions and subpartitions are exported.

  The table name can contain any number of '%' pattern matching characters, which can each match zero or more characters in the table name against the table objects in the database. All the tables in the relevant schema that match the specified pattern are selected for export, as if the respective table names were explicitly specified in the parameter.
- *`partition_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 preceding is in the form:

```
schemaname.tablename:partition_name
schemaname.tablename:subpartition_name
```

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. If the specified table is not partitioned, the *partition_name* is ignored and the entire table is exported.

See Example Export Session Using Partition-Level Export on page 1-41 for several examples of partition-level exports.

Note: 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

The following restrictions apply to table names:

- By default, table names in a database are stored as uppercase. If you have a table name in mixed-case or lowercase, and you want to preserve case-sensitivity for the table name, you must enclose the name in quotation marks. The name must exactly match the table name stored in the database.

  Some operating systems require that quotation marks on the command line be preceded by an escape character. The following are examples of how case-sensitivity can be preserved in the different Export modes.

  – In command-line mode:

    ```
    TABLES='\"Emp\"'
    ```

  – In interactive mode:

    ```
    Table(T) to be exported: "Emp"
    ```

  – In parameter file mode:

    ```
    TABLES='"Emp"'
    ```

- 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 does not export the tables dept and mydata:

```
TABLES=(emp#, dept, mydata)
```

However, given the following line, the Export utility exports all three tables because emp# is enclosed in quotation marks:

```
TABLES=("emp#", dept, mydata)
```

> **Note:** Some operating systems require single quotation marks rather than double quotation marks, or the reverse; 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.

For a TABLES parameter that specifies multiple schema.tablename:(sub)partition_name arguments, Export attempts to purge duplicates before processing the list of objects.

TABLESPACES

Default: none

The TABLESPACES parameter specifies that all tables in the tablespace be exported to the Export dump file. This includes all tables contained in the list of tablespaces and all tables that have a partition located in the list of tablespaces. Indexes are exported with their tables, regardless of where the index is stored.

You must have the EXP_FULL_DATABASE role to use TABLESPACES to export all tables in the tablespace.

When TABLESPACES is used in conjunction with TRANSPORT_TABLESPACE=y, you can specify a limited list of tablespaces to be exported from the database to the export file.

TRANSPORT_TABLESPACE

Default: n

When specified as y, this parameter enables the export of transportable tablespace metadata.

See Also:

- Transportable Tablespaces on page 1-59
- *Oracle9i Database Administrator's Guide*
- *Oracle9i Database Concepts*

TRIGGERS

Default: y

Specifies whether or not the Export utility exports triggers.

TTS_FULL_CHECK

Default: FALSE

When TTS_FULL_CHECK is set to TRUE, Export verifies that a recovery set (set of tablespaces to be recovered) has no dependencies (specifically, IN pointers) on objects outside the recovery set, and vice versa.

USERID (username/password)

Default: none

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

If you connect as user SYS, you must also specify AS SYSDBA in the connect string. Your operating system may require you to treat AS SYSDBA as a special string, in which case the entire string would be enclosed in quotation marks. See Invoking Export As SYSDBA on page 1-7 for more information.

See Also:

- *Oracle9i Heterogeneous Connectivity Administrator's Guide*
- The user's guide for your Oracle Net protocol for information about specifying the `@connect_string` for Oracle Net

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 Oracle operating system-specific documentation for more information.

The `VOLSIZE` value can be specified as a number followed by KB (number of kilobytes). For example, `VOLSIZE=2KB` is the same as `VOLSIZE=2048.` Similarly, MB specifies megabytes (1024 * 1024) and GB 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/password OWNER=jones TABLES=scott.emp
```

Similarly, `OWNER` and `TABLES` conflict with `FULL=y.`

Example Export Sessions

This section provides examples of the following types of Export sessions:

- Example Export Session in Full Database Mode
- Example Export Session in User Mode
- Example Export Sessions in Table Mode
- Example Export Session Using Partition-Level Export

In each example, you are shown how to use both the command-line method and the parameter file method.

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

Export Messages

```
Export: Release 9.2.0.1.0 - Production on Wed Feb 27 16:52:15 2002

(c) Copyright 2002 Oracle Corporation.  All rights reserved.


Connected to: Oracle9i Enterprise Edition Release 9.2.0.1.0 - Production
With the Partitioning and Oracle Data Mining options
JServer Release 9.2.0.1.0 - Production
Export done in WE8DEC character set and AL16UTF16 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 PUBLIC type synonyms
. exporting private type synonyms
. 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            AQ$_INTERNET_AGENTS          0 rows exported
. . exporting table       AQ$_INTERNET_AGENT_PRIVS          0 rows exported
. . 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       LOGSTDBY$APPLY_MILESTONE          0 rows exported
. . exporting table        LOGSTDBY$APPLY_PROGRESS
. . exporting partition                             P0          0 rows exported
. . exporting table                LOGSTDBY$EVENTS          0 rows exported
. . exporting table            LOGSTDBY$PARAMETERS          0 rows exported
. . exporting table                 LOGSTDBY$PLSQL          0 rows exported
. . exporting table                   LOGSTDBY$SCN          0 rows exported
. . exporting table                  LOGSTDBY$SKIP          0 rows exported
. . exporting table      LOGSTDBY$SKIP_TRANSACTION          0 rows exported
. . exporting table        REPCAT$_AUDIT_ATTRIBUTE          2 rows exported
. . exporting table           REPCAT$_AUDIT_COLUMN          0 rows exported
. . exporting table           REPCAT$_COLUMN_GROUP          0 rows exported
. . exporting table               REPCAT$_CONFLICT          0 rows exported
. . exporting table                    REPCAT$_DDL          0 rows exported
. . exporting table             REPCAT$_EXCEPTIONS          0 rows exported
. . exporting table              REPCAT$_EXTENSION          0 rows exported
. . exporting table                REPCAT$_FLAVORS          0 rows exported
. . exporting table         REPCAT$_FLAVOR_OBJECTS          0 rows exported
. . exporting table              REPCAT$_GENERATED          0 rows exported
. . exporting table         REPCAT$_GROUPED_COLUMN          0 rows exported
. . exporting table      REPCAT$_INSTANTIATION_DDL          0 rows exported
. . exporting table            REPCAT$_KEY_COLUMNS          0 rows exported
. . exporting table           REPCAT$_OBJECT_PARMS          0 rows exported
. . exporting table           REPCAT$_OBJECT_TYPES         28 rows exported
. . exporting table       REPCAT$_PARAMETER_COLUMN          0 rows exported
. . exporting table               REPCAT$_PRIORITY          0 rows exported
. . exporting table         REPCAT$_PRIORITY_GROUP          0 rows exported
. . exporting table      REPCAT$_REFRESH_TEMPLATES          0 rows exported
. . exporting table                 REPCAT$_REPCAT          0 rows exported
. . exporting table              REPCAT$_REPCATLOG          0 rows exported
. . exporting table              REPCAT$_REPCOLUMN          0 rows exported
. . exporting table         REPCAT$_REPGROUP_PRIVS          0 rows exported
. . exporting table              REPCAT$_REPOBJECT          0 rows exported
. . exporting table                REPCAT$_REPPROP          0 rows exported
. . exporting table              REPCAT$_REPSCHEMA          0 rows exported
. . exporting table             REPCAT$_RESOLUTION          0 rows exported
. . exporting table      REPCAT$_RESOLUTION_METHOD         19 rows exported
. . exporting table  REPCAT$_RESOLUTION_STATISTICS          0 rows exported
. . exporting table    REPCAT$_RESOL_STATS_CONTROL          0 rows exported
. . exporting table          REPCAT$_RUNTIME_PARMS          0 rows exported
. . exporting table              REPCAT$_SITES_NEW          0 rows exported
. . exporting table           REPCAT$_SITE_OBJECTS          0 rows exported
. . exporting table              REPCAT$_SNAPGROUP          0 rows exported
. . exporting table       REPCAT$_TEMPLATE_OBJECTS          0 rows exported
. . exporting table         REPCAT$_TEMPLATE_PARMS          0 rows exported
. . exporting table     REPCAT$_TEMPLATE_REFGROUPS          0 rows exported
. . exporting table         REPCAT$_TEMPLATE_SITES          0 rows exported
. . exporting table        REPCAT$_TEMPLATE_STATUS          3 rows exported
. . exporting table       REPCAT$_TEMPLATE_TARGETS          0 rows exported
. . exporting table         REPCAT$_TEMPLATE_TYPES          2 rows exported
. . exporting table    REPCAT$_USER_AUTHORIZATIONS          0 rows exported
. . exporting table       REPCAT$_USER_PARM_VALUES          0 rows exported
. . exporting table        SQLPLUS_PRODUCT_PROFILE          0 rows exported
. about to export OUTLN’s tables via Conventional Path ...
. . exporting table                            OL$          0 rows exported
. . exporting table                       OL$HINTS          0 rows exported
. . exporting table                       OL$NODES          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 synonyms
. exporting views
. exporting referential integrity constraints
. exporting stored procedures
. exporting operators
. exporting indextypes
. exporting bitmap, functional and extensible indexes
. exporting posttables actions
. exporting triggers
. exporting materialized views
. 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 statistics
Export terminated successfully without warnings.
```

Example Export Session in User Mode

User mode exports can be used to 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 9.2.0.1.0 - Production on Wed Feb 27 17:01:06 2002

(c) Copyright 2002 Oracle Corporation.  All rights reserved.


Connected to: Oracle9i Enterprise Edition Release 9.2.0.1.0 - Production
With the Partitioning and Oracle Data Mining options
JServer Release 9.2.0.1.0 - Production
Export done in WE8DEC character set and AL16UTF16 NCHAR character set
. exporting pre-schema procedural objects and actions
. exporting foreign function library names for user SCOTT
. exporting PUBLIC type synonyms
. exporting private type synonyms
. 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 materialized views
. exporting snapshot logs
. exporting job queues
. exporting refresh groups and children
. exporting dimensions
. exporting post-schema procedural objects and actions
. exporting statistics
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=schemaname.tablename.

If schemaname is not specified, Export defaults to the previous schema name 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/password TABLES=(a, scott.b, c, mary.d)
```

A user with the EXP_FULL_DATABASE role can also export dependent objects that are owned by other users. A nonprivileged 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: DBA Exporting Tables for Two Users

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

Parameter File Method

```
> exp SYSTEM/password 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/password FILE=expdat.dmp TABLES=(scott.emp,blake.dept) GRANTS=y-
INDEXES=y
```

Export Messages

```
Export: Release 9.2.0.1.0 - Production on Wed Feb 27 17:01:35 2002

(c) Copyright 2002 Oracle Corporation.  All rights reserved.
```

```
Connected to: Oracle9i Enterprise Edition Release 9.2.0.1.0 - Production
With the Partitioning and Oracle Data Mining options
JServer Release 9.2.0.1.0 - Production
Export done in WE8DEC character set and AL16UTF16 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: User Exports Tables That He Owns

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 9.2.0.1.0 - Production on Wed Feb 27 17:01:38 2002

(c) Copyright 2002 Oracle Corporation.  All rights reserved.


Connected to: Oracle9i Enterprise Edition Release 9.2.0.1.0 - Production
With the Partitioning and Oracle Data Mining options
JServer Release 9.2.0.1.0 - Production
Export done in WE8DEC character set and AL16UTF16 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 3: Using Pattern Matching to Export Various Tables

In this example, pattern matching is used to export various tables for users `scott` and `blake`.

Parameter File Method

```
> exp SYSTEM/password PARFILE=params.dat
```

The `params.dat` file contains the following information:

```
FILE=misc.dmp
TABLES=(scott.%P%,blake.%,scott.%S%)
```

Command-Line Method

```
> exp SYSTEM/password FILE=misc.dmp TABLES=(scott.%P%,blake.%,scott.%S%)
```

Export Messages

```
Export: Release 9.2.0.1.0 - Production on Wed Feb 27 17:01:40 2002

(c) Copyright 2002 Oracle Corporation.  All rights reserved.


Connected to: Oracle9i Enterprise Edition Release 9.2.0.1.0 - Production
With the Partitioning and Oracle Data Mining options
JServer Release 9.2.0.1.0 - Production
Export done in WE8DEC character set and AL16UTF16 NCHAR character set

About to export specified tables via Conventional Path ...
Current user changed to SCOTT
. . exporting table                           DEPT          4 rows exported
. . exporting table                            EMP         14 rows exported
Current user changed to BLAKE
. . exporting table                           DEPT          8 rows exported
. . exporting table                        MANAGER          4 rows exported
Current user changed to SCOTT
. . exporting table                          BONUS          0 rows exported
. . exporting table                       SALGRADE          5 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: Exporting a Table Without Specifying a Partition

Assume emp is a table that is partitioned on employee name. There are two partitions, m and z. 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 9.2.0.1.0 - Production on Wed Feb 27 17:01:53 2002

(c) Copyright 2002 Oracle Corporation.  All rights reserved.


Connected to: Oracle9i Enterprise Edition Release 9.2.0.1.0 - Production
With the Partitioning and Oracle Data Mining options
JServer Release 9.2.0.1.0 - Production
Export done in WE8DEC character set and AL16UTF16 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: Exporting a Table with a Specified Partition

Assume emp is a table that is partitioned on employee name. There are two partitions, m and z. 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 9.2.0.1.0 - Production on Wed Feb 27 17:01:55 2002

(c) Copyright 2002 Oracle Corporation.  All rights reserved.


Connected to: Oracle9i Enterprise Edition Release 9.2.0.1.0 - Production
With the Partitioning and Oracle Data Mining options
JServer Release 9.2.0.1.0 - Production
Export done in WE8DEC character set and AL16UTF16 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: Exporting a Composite Partition

Assume `emp` is a partitioned table with two partitions, `m` and `z.` Table `emp` is partitioned using the composite method. Partition `m` has subpartitions `sp1` and `sp2,` and partition `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 PARFILE=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 9.2.0.1.0 - Production on Wed Feb 27 17:22:47 2002

(c) Copyright 2002 Oracle Corporation.  All rights reserved.


Connected to: Oracle9i Enterprise Edition Release 9.2.0.1.0 - Production
With the Partitioning and Oracle Data Mining options
JServer Release 9.2.0.1.0 - Production
Export done in WE8DEC character set and AL16UTF16 NCHAR character set

About to export specified tables via Conventional Path ...
. . exporting table                            EMP
. . exporting composite partition                    M
. . exporting subpartition                         SP1          1 rows exported
. . exporting subpartition                         SP2          3 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 command-line interactive method does not provide prompts for all Export functionality and is provided only for backward compatibility. If you want to use an interactive interface to the Export utility, it is recommended that you use the Oracle Enterprise Manager (OEM) Export Wizard.

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

When you invoke Export interactively, the response given by Export depends on what you enter at the command line. Table 1–4 shows the possibilities.

Table 1–4 Invoking Export Using the Interactive Method

You enter...	Export's Response
exp *username*/*password@instance* as sysdba	Starts an Export session

Table 1–4 (Cont.) Invoking Export Using the Interactive Method

You enter...	Export's Response
`exp` *`username`*`/`*`password@instance`*	Starts an Export session
`exp` *`username`*`/`*`password`* `as sysdba`	Starts an Export session
`exp` *`username`*`/`*`password`*	Starts an Export session
`exp` *`username@instance`* `as sysdba`	Prompts for password
`exp` *`username@instance`*	Prompts for password
`exp` *`username`*	Prompts for password
`exp` *`username`* `as sysdba`	Prompts for password
`exp / as sysdba`	No prompt for password, operating system authentication is used
`exp /`	No prompt for password, operating system authentication is used
`exp` *`/@instance`* `as sysdba`	No prompt for password, operating system authentication is used
`exp` *`/@instance`*	No prompt for password, operating system authentication is used

In Export interactive mode, you are not prompted to specify whether you want to connect as SYSDBA or *`@instance`*. You must specify AS SYSDBA and/or *`@instance`* with the username.

Additionally, 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.

Before you invoke Export using AS SYSDBA, be sure to read Invoking Export As SYSDBA on page 1-7 for information about correct command-line syntax.

After Export is invoked, it displays the following prompts. 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 Enter.

```
Export: Release 9.2.0.1.0 - Production on Wed Feb 27 17:02:03 2002
```

```
(c) Copyright 2002 Oracle Corporation.  All rights reserved.


Connected to: Oracle9i Enterprise Edition Release 9.2.0.1.0 - Production
With the Partitioning and Oracle Data Mining options
JServer Release 9.2.0.1.0 - Production
Enter array fetch buffer size: 4096 >
Export file: expdat.dmp >
(1)E(ntire database), (2)U(sers), or (3)T(ables): (2)U >
Export grants (yes/no): yes >
Export table data (yes/no): yes >
Compress extents (yes/no): yes >
Export done in WE8DEC character set and AL16UTF16 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 PUBLIC type synonyms
. exporting private type synonyms
. 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            AQ$_INTERNET_AGENTS          0 rows exported
. . exporting table       AQ$_INTERNET_AGENT_PRIVS          0 rows exported
. . 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       LOGSTDBY$APPLY_MILESTONE          0 rows exported
. . exporting table        LOGSTDBY$APPLY_PROGRESS
. . exporting partition                             P0          0 rows exported
. . exporting table                LOGSTDBY$EVENTS          0 rows exported
. . exporting table            LOGSTDBY$PARAMETERS          0 rows exported
. . exporting table                 LOGSTDBY$PLSQL          0 rows exported
. . exporting table                   LOGSTDBY$SCN          0 rows exported
. . exporting table                  LOGSTDBY$SKIP          0 rows exported
. . exporting table      LOGSTDBY$SKIP_TRANSACTION          0 rows exported
. . exporting table        REPCAT$_AUDIT_ATTRIBUTE          2 rows exported
. . exporting table           REPCAT$_AUDIT_COLUMN          0 rows exported
. . exporting table           REPCAT$_COLUMN_GROUP          0 rows exported
. . exporting table               REPCAT$_CONFLICT          0 rows exported
. . exporting table                    REPCAT$_DDL          0 rows exported
. . exporting table             REPCAT$_EXCEPTIONS          0 rows exported
. . exporting table              REPCAT$_EXTENSION          0 rows exported
. . exporting table                REPCAT$_FLAVORS          0 rows exported
. . exporting table         REPCAT$_FLAVOR_OBJECTS          0 rows exported
. . exporting table              REPCAT$_GENERATED          0 rows exported
. . exporting table         REPCAT$_GROUPED_COLUMN          0 rows exported
. . exporting table      REPCAT$_INSTANTIATION_DDL          0 rows exported
. . exporting table            REPCAT$_KEY_COLUMNS          0 rows exported
. . exporting table           REPCAT$_OBJECT_PARMS          0 rows exported
. . exporting table           REPCAT$_OBJECT_TYPES         28 rows exported
. . exporting table       REPCAT$_PARAMETER_COLUMN          0 rows exported
. . exporting table               REPCAT$_PRIORITY          0 rows exported
. . exporting table         REPCAT$_PRIORITY_GROUP          0 rows exported
. . exporting table      REPCAT$_REFRESH_TEMPLATES          0 rows exported
. . exporting table                 REPCAT$_REPCAT          0 rows exported
. . exporting table              REPCAT$_REPCATLOG          0 rows exported
. . exporting table              REPCAT$_REPCOLUMN          0 rows exported
. . exporting table         REPCAT$_REPGROUP_PRIVS          0 rows exported
. . exporting table              REPCAT$_REPOBJECT          0 rows exported
. . exporting table                REPCAT$_REPPROP          0 rows exported
. . exporting table              REPCAT$_REPSCHEMA          0 rows exported
. . exporting table             REPCAT$_RESOLUTION          0 rows exported
. . exporting table      REPCAT$_RESOLUTION_METHOD         19 rows exported
. . exporting table  REPCAT$_RESOLUTION_STATISTICS          0 rows exported
. . exporting table    REPCAT$_RESOL_STATS_CONTROL          0 rows exported
. . exporting table          REPCAT$_RUNTIME_PARMS          0 rows exported
. . exporting table              REPCAT$_SITES_NEW          0 rows exported
. . exporting table           REPCAT$_SITE_OBJECTS          0 rows exported
. . exporting table              REPCAT$_SNAPGROUP          0 rows exported
. . exporting table       REPCAT$_TEMPLATE_OBJECTS          0 rows exported
. . exporting table          REPCAT$_TEMPLATE_PARMS         0 rows exported
. . exporting table      REPCAT$_TEMPLATE_REFGROUPS         0 rows exported
. . exporting table          REPCAT$_TEMPLATE_SITES         0 rows exported
. . exporting table         REPCAT$_TEMPLATE_STATUS         3 rows exported
. . exporting table        REPCAT$_TEMPLATE_TARGETS         0 rows exported
. . exporting table          REPCAT$_TEMPLATE_TYPES         2 rows exported
. . exporting table     REPCAT$_USER_AUTHORIZATIONS         0 rows exported
. . exporting table        REPCAT$_USER_PARM_VALUES         0 rows exported
. . exporting table         SQLPLUS_PRODUCT_PROFILE         0 rows exported
. about to export OUTLN’s tables via Conventional Path ...
. . exporting table                             OL$         0 rows exported
. . exporting table                        OL$HINTS         0 rows exported
. . exporting table                        OL$NODES         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 synonyms
. exporting views
. exporting referential integrity constraints
. exporting stored procedures
. exporting operators
. exporting indextypes
. exporting bitmap, functional and extensible indexes
. exporting posttables actions
. exporting triggers
. exporting materialized views
. 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 statistics
Export terminated successfully without warnings.
```

Restrictions

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

- In user mode, Export prompts for all usernames 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 Enter.
- 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 the `beth` 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 describes the different types of messages issued by Export and how to save them in a log file.

Log File

You can capture all Export messages in a log file, either by using the `LOG` parameter (see LOG on page 1-23) or, for those systems that permit it, by redirecting the Export output to a file. The Export utility writes a log of detailed information about successful unloads and any errors that may occur. Refer to your Oracle operating system-specific documentation for information on redirecting output.

Warning Messages

Export does not terminate after recoverable 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 recoverable errors are known as warnings.

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

For example, if a nonexistent 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.

Nonrecoverable Error Messages

Some errors are nonrecoverable 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 nonrecoverable error message:

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

Completion Messages

When an export completes without errors, Export displays the following message:

```
Export terminated successfully without warnings
```

If one or more recoverable errors occurs but Export is able to continue to completion, Export displays the following message:

```
Export terminated successfully with warnings
```

If a nonrecoverable error occurs, Export terminates immediately and displays the following message:

```
Export terminated unsuccessfully
```

> **See Also:** *Oracle9i Database Error Messages* and your Oracle operating system-specific documentation

Exit Codes for Inspection and Display

Export provides the results of an export operation immediately upon completion. Depending on the platform, Export may report the outcome in a process exit code as well as recording the results in the log file. This enables you to check the outcome from the command line or script. Table 1–5 shows the exit codes that get returned for various results.

Table 1–5 Exit Codes for Export

Result	Exit Code
Export terminated successfully without warnings	`EX_SUCC`
Export terminated successfully with warnings	`EX_OKWARN`
Export terminated unsuccessfully	`EX_FAIL`

For UNIX, the exit codes are as follows:

```
EX_SUCC   0
EX_OKWARN 0
EX_FAIL   1
```

Conventional Path Export Versus Direct Path Export

Export provides two methods for exporting table data:

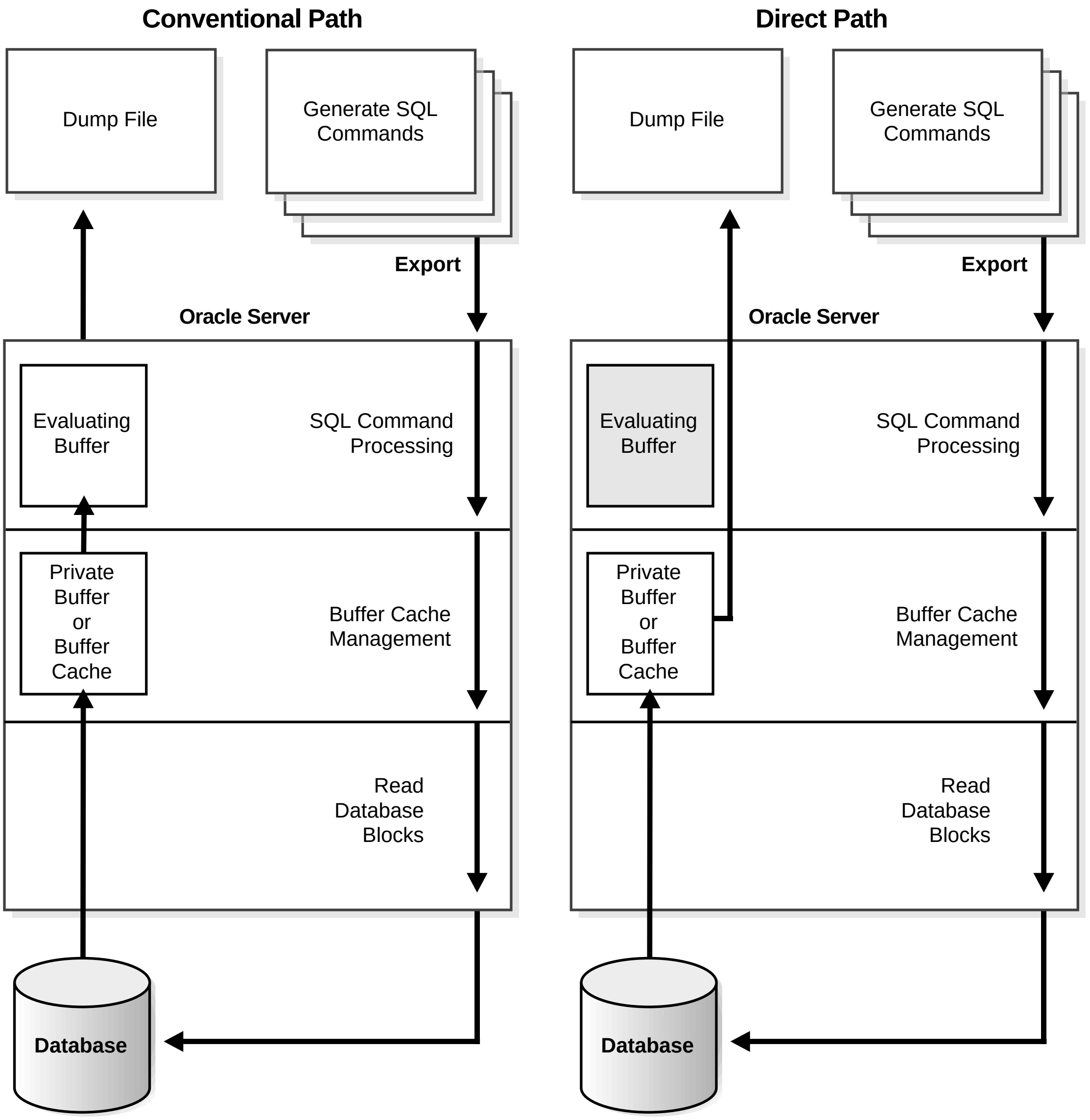
- Conventional path Export
- Direct path Export

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 evaluating 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 is much faster than conventional path Export because 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 on page 1-51 illustrates how data extraction differs between conventional path Export and direct path Export.

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



Invoking a Direct Path Export

To invoke a direct path Export, you must use either the command-line method or a parameter file. You cannot invoke a direct path Export using the interactive method.

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.

Additionally, be aware that the Export parameter `BUFFER` applies only to conventional path Exports. For direct path Export, use the `RECORDLENGTH` parameter to specify the size of the buffer that Export uses for writing to the export file.

In versions of Export prior to 8.1.5, you could not use direct path Export for tables containing objects and LOBs. If you tried to, their rows were not exported. This behavior has changed. Rows in tables that contain objects and LOBs will now be exported using conventional path, even if direct path was specified. Import will correctly handle these conventional path tables within direct path dump files.

Security Considerations for Direct Path Exports

Virtual Private Database (VPD) and Oracle Label Security are not enforced during direct path Exports.

The following users are exempt from Virtual Private Database and Oracle Label Security enforcement regardless of the export mode, application, or utility used to extract data from the database:

- The database user `SYS`
- Database users granted the Oracle9*i* `EXEMPT ACCESS POLICY` privilege, either directly or through a database role

This means that *any* user who is granted the `EXEMPT ACCESS POLICY` privilege is completely exempt from enforcement of VPD and Oracle Label Security. This is a powerful privilege and should be carefully managed. This privilege does not affect the enforcement of traditional object privileges such as `SELECT`, `INSERT`, `UPDATE`, and `DELETE`. These privileges are enforced even if a user has been granted the `EXEMPT ACCESS POLICY` privilege.

See Also:

- Support for Fine-Grained Access Control on page 1-59
- *Oracle9i Application Developer's Guide - Fundamentals*

Performance Issues for Direct Path Exports

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 depends 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.)

The following values are generally recommended for `RECORDLENGTH:`

- Multiples of the file system I/O block size
- Multiples of `DB_BLOCK_SIZE`

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 Oracle Net

With Oracle Net, 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 Import with Oracle Net, 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 Oracle Net protocol.

See Also:

- *Oracle9i Net Services Administrator's Guide*
- *Oracle9i Heterogeneous Connectivity Administrator's Guide*

Character Set and Globalization Support Considerations

This section describes the behavior of Export and Import with respect to globalization support.

Character Set Conversion

The Export utility always exports user data, including Unicode data, in the character sets of the Export server. The character sets are specified at database creation.

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

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 system 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.

Note: Data definition language (DDL), such as a CREATE TABLE command, is exported in the client character set.

Effect of Character Set Sorting Order on Conversions

If the export character set has a different sorting order than the import character set, then tables that are partitioned on character columns may yield unpredictable results. For example, consider the following table definition, which is produced on a database having an ASCII character set:

```
CREATE TABLE partlist
   (
   part     VARCHAR2(10),
   partno   NUMBER(2)
   )
```

```
PARTITION BY RANGE (part)
  (
  PARTITION part_low VALUES LESS THAN ('Z')
    TABLESPACE tbs_1,
  PARTITION part_mid VALUES LESS THAN ('z')
    TABLESPACE tbs_2,
  PARTITION part_high VALUES LESS THAN (MAXVALUE)
    TABLESPACE tbs_3
  );
```

This partitioning scheme makes sense because `z` comes after `Z` in ASCII character sets.

When this table is imported into a database based upon an EBCDIC character set, all of the rows in the `part_mid` partition will migrate to the `part_low` partition because `z` comes before `Z` in EBCDIC character sets. To obtain the desired results, the owner of `partlist` must repartition the table following the import.

See Also: *Oracle9i Database Globalization Support Guide*

Multibyte Character Sets and Export and Import

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

You can use instance affinity to associate jobs with instances in databases you plan to export and import. Be aware that there may be some compatibility issues if you are using a combination of releases 8.0, 8.1, and 9*i*.

See Also:

- *Oracle9i Database Administrator's Guide*
- *Oracle9i Database Reference*
- *Oracle9i Database Migration*

Considerations When Exporting Database Objects

The following sections describe points you should consider 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.

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. You must move the library's executable files 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 tablespaces, 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 first create the offline bitmapped tablespaces 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 (DBA) must move the directories containing the specified files to a new location where they can be accessed.

External Tables

The contents of external tables are not included in the export file. Instead, only the table specification (name, location) is included in full database and user mode export. You must manually move the external data and update the table specification if the database is moved to a new location.

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 `table1` belonging to user `scott` 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 are 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 Also:** *Oracle9i Application Developer's Guide - Advanced Queuing*

Exporting Synonyms

You should be cautious when exporting compiled objects that reference a name used as a synonym and as another object. Exporting and importing these objects will force a recompilation that could result in changes to the object definitions.

The following example helps to illustrate this problem:

```
CREATE PUBLIC SYNONYM emp FOR scott.emp;

CONNECT blake/paper;
CREATE TRIGGER t_emp BEFORE INSERT ON emp BEGIN NULL; END;
CREATE VIEW emp AS SELECT * FROM dual;
```

If the database in the preceding example were exported, the reference to `emp` in the trigger would refer to `blake`'s view rather than to `scott`'s table. This would cause an error when Import tried to reestablish the `t_emp` trigger.

Possible Export Errors Related to Java Synonyms

If an export operation attempts to export a synonym named `DBMS_JAVA` when there is no corresponding `DBMS_JAVA` package or when Java is either not loaded or loaded incorrectly, the export will terminate unsuccessfully. The error messages that are generated include, but are not limited to, the following: EXP-00008, ORA-00904, and ORA-29516.

If Java is enabled, make sure that both the `DBMS_JAVA` synonym and `DBMS_JAVA` package are created and valid before rerunning the export.

If Java is not enabled, remove Java-related objects before rerunning the export.

Support for Fine-Grained Access Control

You can export tables with fine-grained access control policies enabled. When doing so, consider the following:

- The user who imports from an export file containing such tables must have the appropriate privileges (specifically, the `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.
- If fine-grained access control is enabled on a `SELECT` statement, then conventional path Export may not export the entire table because fine-grained access may rewrite the query.
- Only user `SYS,` or a user with the `EXPORT_FULL_DATABASE` role enabled or who has been granted `EXEMPT ACCESS POLICY`, can perform direct path Exports on tables having fine-grained access control.

> **See Also:** *Oracle9i Application Developer's Guide - Fundamentals* for more information about fine-grained access control

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 and 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 CD-ROMs.

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

Export provides the following parameters to enable export of transportable tablespace metadata.

- `TABLESPACES`
- `TRANSPORT_TABLESPACE`

See TABLESPACES on page 1-30 and TRANSPORT_TABLESPACE on page 1-31 for more information.

> **See Also:**
>
> - *Oracle9i Database Administrator's Guide* for details about managing transportable tablespaces
> - *Oracle9i Database Concepts* for an introduction to transportable tablespaces

Exporting from a Read-Only Database

To extract metadata from a source database, Export uses queries that contain ordering clauses (sort operations). For these queries to succeed, the user performing the export must be able to allocate on-disk sort segments. For these sort segments to be allocated in a read-only database, the user's temporary tablespace should be set to point at a temporary, locally managed tablespace.

> **See Also:** *Oracle9i Data Guard Concepts and Administration* for more information on setting up this environment

Using Export and Import to Partition a Database Migration

When you use the Export and Import utilities to migrate a large database, it may be more efficient to partition the migration into multiple export and import jobs. If you decide to partition the migration, be aware of the following advantages and disadvantages.

Advantages of Partitioning a Migration

Partitioning a migration has the following advantages:

- Time required for the migration may be reduced because many of the subjobs can be run in parallel.
- The import can start as soon as the first export subjob completes, rather than waiting for the entire export to complete.

Disadvantages of Partitioning a Migration

Partitioning a migration has the following disadvantages:

- The export and import processes become more complex.
- Support of cross-schema references for certain types of objects may be compromised. For example, if a schema contains a table with a foreign key constraint against a table in a different schema, you may not have the required parent records when you import the table into the dependent schema.

How to Use Export and Import to Partition a Database Migration

To perform a database migration in a partitioned manner, take the following steps:

1. For all top-level metadata in the database, issue the following commands:
   a. `exp dba/password FILE=full FULL=y CONSTRAINTS=n TRIGGERS=n ROWS=n INDEXES=n`
   b. `imp dba/password FILE=full FULL=y`
2. For each schema*n* in the database, issue the following commands:
   a. `exp dba/password OWNER=scheman FILE=scheman`
   b. `imp dba/password FILE=scheman FROMUSER=scheman TOUSER=scheman IGNORE=y`

All exports can be done in parallel. When the import of `full.dmp` completes, all remaining imports can also be done in parallel.

Using Different Releases and Versions of Export

This section describes compatibility issues that relate to using different releases of Export and the Oracle database server.

Whenever you are moving data between different releases of the Oracle database server, the following basic rules apply:

- The Import utility and the database to which data is being imported (the target database) must be the same version.
- The version of the Export utility must be equal to the lowest version of the source or target database.

  For example, to create an export file for an import into a higher release database, use a version of the Export utility that is equal to the source database. Conversely, to create an export file for an import into a lower release database, use a version of the Export utility that is equal to the version of the target database. The following information is for specific versions:

  – When you create an Oracle version 6 export file from an Oracle7 database by running the Oracle version 6 Export utility against the Oracle7 database server, you must first run the `catexp6.sql` script on the Oracle7 database. This script creates the export views that make the database look, to Export, like an Oracle version 6 database.

  – When you create an Oracle7 export file from an Oracle8*i* database by running the Oracle7 Export utility against the Oracle8*i* database, you must first run the `catexp7.sql` script on the Oracle8*i* database. This script creates the export views that make the database look, to Export, like an Oracle8*i* database.

  Note: The `catexp6.sql` and `catexp7.sql` scripts must be run by user `SYS.` These scripts only need to be run once.

  – In general, you can use the Export utility from any Oracle8 release to export from an Oracle9*i* server and create an Oracle8 export file. See Creating Oracle Release 8.0 Export Files from an Oracle9i Database on page 1-64.

Restrictions When Using Different Releases and Versions of Export and Import

The following restrictions apply when you are using different releases of Export and Import:

- Export dump files can be read only by the Import utility because they are stored in a special binary format.
- Any export dump file can be imported into a higher release of the Oracle database server.

- The Import utility can read export dump files created by Export release 5.1.22 and higher.
- The Import utility cannot read export dump files created by the Export utility of a higher maintenance release or version. For example, a release 8.1 export dump file cannot be imported by a release 8.0 Import utility, and a version 8 export dump file cannot be imported by a version 7 Import utility.
- The Oracle version 6 (or earlier) Export utility cannot be used against an Oracle8 or higher database.
- Whenever a lower version of the Export utility runs with a higher version of the Oracle database server, categories of database objects that did not exist in the lower version are excluded from the export. For example, partitioned tables did not exist in the Oracle database server version 7. So, if you need to move a version 8 partitioned table to a version 7 database, you must first reorganize the table into a nonpartitioned table.
- Export files generated by Oracle9*i* Export, either direct path or conventional path, are incompatible with earlier releases of Import and can be imported only with Oracle9*i* Import. When backward compatibility is an issue, use the earlier release or version of the Export utility against the Oracle9*i* database.

Examples of Using Different Releases of Export and Import

Table 1–6 shows some examples of which Export and Import releases to use when moving data between different releases of the Oracle database server.

Table 1–6 Using Different Releases of Export and Import

Export from->Import to	Use Export Release	Use Import Release
7.3.3 -> 8.1.6	7.3.3	8.1.6
8.1.6 -> 8.1.6	8.1.6	8.1.6
8.1.5 -> 8.0.6	8.0.6	8.0.6
8.1.7 -> 8.1.6	8.1.6	8.1.6
8.1.7 -> 7.3.4	7.3.4[1]	7.3.4
9.0.1 -> 8.1.6	8.1.6	8.1.6
9.0.1 -> 9.0.2	9.0.1	9.0.2

[1] If catexp7.sql has never been run on the 8.1.7 database, then you must do that first to create the Oracle7 data dictionary views. Only then can you successfully use Export release 7.3.4 on the release 8.1.7 database.

Creating Oracle Release 8.0 Export Files from an Oracle9*i* Database

You do not need to take any special steps to create an Oracle release 8.0 export file from an Oracle9*i* database. However, the following features are not supported when you use Export release 8.0 on an Oracle9*i* database:

- Export does not export rows from tables containing objects and LOBs when you have specified a direct path load (`DIRECT=y`).
- Export does not export dimensions.
- Functional and domain indexes are not exported.
- Secondary objects (tables, indexes, sequences, and so on, created in support of a domain index) are not exported.
- Views, procedures, functions, packages, type bodies, and types containing references to new Oracle9*i* features may not compile.
- Objects whose DDL is implemented as a stored procedure rather than SQL are not exported.
- Triggers whose action is a `CALL` statement are not exported.
- Tables containing logical `ROWID` columns, primary key refs, or user-defined `OID` columns will not be exported.
- Temporary tables are not exported.
- Index-organized tables (IOTs) revert to an uncompressed state.
- Partitioned IOTs lose their partitioning information.
- Index types and operators are not exported.
- Bitmapped, temporary, and UNDO tablespaces are not exported.
- Java sources, classes, and resources are not exported.
- Varying-width `CLOB`s, collection enhancements, and LOB-storage clauses for `VARRAY` columns or nested table enhancements are not exported.
- Fine-grained access control policies are not preserved.
- External tables are not exported.

Possible Errors When Using Different Releases and Versions

This section briefly discusses some of the error messages you might receive if incompatible releases or versions of the Export utility and the Oracle database server are used.

EXP-24

```
EXP-24: Export views not installed, please notify your DBA
Cause: The necessary export views were not installed.
Action: Ask the DBA to install the required views.
```

EXP-23

```
EXP-23: Import views not installed, please notify your DBA
Cause: The necessary import views were not installed.
Action: Ask the DBA to install the required views.
```

EXP-37

```
EXP-37: Export views not compatible with database version
Cause: The Export utility is at a higher version than the database version.
Action: Use the same version of the Export utility as the database.
```

See Also:

- *Oracle9i Database Error Messages*
- Restrictions When Using Different Releases and Versions of Export and Import on page 1-62
- Using Export Files from a Previous Oracle Release on page 2-70

2

Import

This chapter describes how to use the Import utility to read an export file into an Oracle database. Import only reads files created by the Export utility. For information on how to export a database, see Chapter 1. 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:

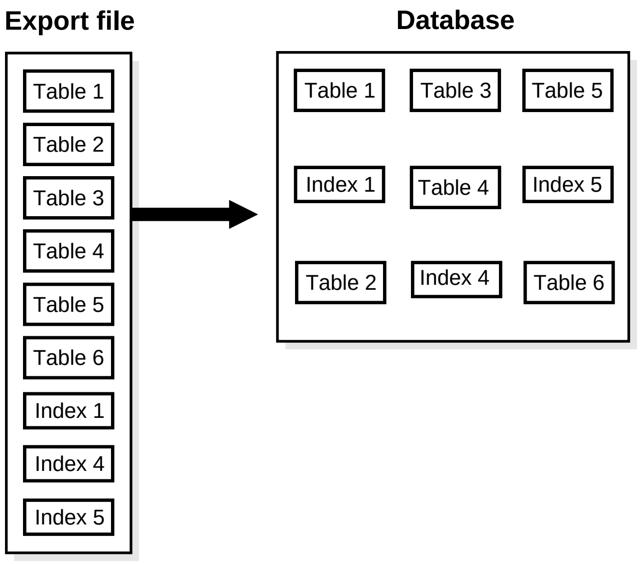
- Network Considerations
- Character Set and Globalization Support Considerations
- Considerations When Importing Database Objects
- Materialized Views and Snapshots
- Transportable Tablespaces
- Storage Parameters
- Dropping a Tablespace
- Reorganizing Tablespaces
- Importing Statistics
- Using Export and Import to Partition a Database Migration
- Using Export Files from a Previous Oracle Release

What Is the Import Utility?

The Import utility reads the object definitions and table data from an Export dump file. It inserts the data objects into an Oracle database.

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

Figure 2–1 Importing an Export File



Export dump files can only be read by the Oracle Import utility. The version of the Import utility cannot be earlier than the version of the Export utility used to create the dump file.

Import can read export files created by Export release 5.1.22 and higher.

To read load data from ASCII fixed-format or delimited files, use the SQL*Loader utility.

See Also:

- Chapter 1 for information about the Export utility
- Part II of this manual for information about the SQL*Loader utility

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 is 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.

> **Note:** For the reasons mentioned previously, it is a good idea to disable referential constraints when importing into an existing table. You can then reenable the constraints after the import is completed.

Before Using Import

Before you begin using Import, be sure you take care of the following items:

- Run the `catexp.sql` or `catalog.sql` script
- Verify that you have the required access privileges

Additionally, before you begin using Import, you should read the following sections:

- Importing into Existing Tables on page 2-8
- Effect of Schema and Database Triggers on Import Operations on page 2-9

Running catexp.sql or catalog.sql

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 Oracle9*i*.

> **Note:** The actual names of the script files depend on your operating system. The script filenames and the method for running them are described in your Oracle operating system-specific documentation.

`The catexp.sql` or `catalog.sql` script needs 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.

Verifying Access Privileges

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

To use Import, you need the privilege `CREATE SESSION` to log on to the Oracle database 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, keep in mind that if the export file was created by a user with `EXP_FULL_DATABASE` privilege, then you must have `IMP_FULL_DATABASE` privilege to import it. Both of these privileges are typically assigned to DBAs.

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	Required Privilege (Privilege Type, If Applicable)
Clusters	CREATE CLUSTER (System) and Tablespace Quota, or UNLIMITED TABLESPACE (System)
Database links	CREATE DATABASE LINK (System) and CREATE SESSION (System) on remote database
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)
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 (including comments and audit options)	CREATE TABLE (System) and Tablespace Quota, or UNLIMITED TABLESPACE (System)
Views	CREATE VIEW (System) and SELECT (Object) on the base table, or 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	The object must exist in the user's schema, *or* the user must have the object privileges with the `WITH GRANT OPTION` *or*, the user must have the `IMP_FULL_DATABASE` role enabled.
System privileges	User must have the `SYSTEM` privilege 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

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, although 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
- Change `DEFAULT` column values

> **Note:** When tables are manually created before data is imported, the `CREATE TABLE` statement in the export dump file will fail because the table already exists. To avoid this failure and continue loading data into the table, set the import parameter `IGNORE=y`. Otherwise, no data will be loaded into the table because of the table creation error.

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. For example, if table emp has a referential integrity constraint on the mgr column that verifies that 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 reenabled 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, perform several imports from an export file instead of one. First, import tables that are the targets of referential checks. Then, import the tables that reference them. This option works if tables do not reference each other in a circular fashion, and if a table does not reference itself.

Effect of Schema and Database Triggers on Import Operations

Triggers that are defined to trigger on DDL events for a specific schema or on DDL-related events for the database are system triggers. These triggers can have detrimental effects on certain Import operations. For example, they can prevent successful re-creation of database objects, such as tables. This causes errors to be returned that give no indication that a trigger caused the problem.

Database administrators and anyone creating system triggers should verify that such triggers do not prevent users from performing database operations for which they are authorized. To test a system trigger, take the following steps:

1. Define the trigger.
2. Create some database objects.
3. Export the objects in table or user mode.

4. Delete the objects.
5. Import the objects.
6. Verify that the objects have been successfully re-created.

> **Note:** A full export does not export triggers owned by schema SYS. You must manually re-create SYS triggers either before or after the full import. Oracle Corporation recommends that you re-create them after the import in case they define actions that would impede progress of the import.

Invoking Import

You can invoke Import and specify parameters by using any of the following methods:

- Command-line entries
- Interactive Import prompts
- Parameter files

Before you use one of these methods to invoke Import, be sure to read the descriptions of the available parameters. See Import Parameters on page 2-14.

Command-Line Entries

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

```
imp username/password PARAMETER=value
```

or

```
imp username/password PARAMETER=(value1,value2,...,valuen)
```

The number of parameters cannot exceed the maximum length of a command line on the system.

Interactive Import Prompts

If you prefer to let Import prompt you for the value of each parameter, you can use the following syntax to start Import in interactive mode:

```
imp username/password
```

Import will display each parameter with a request for you to enter a value. This method exists for backward compatibility and is not recommended because it provides less functionality than the other methods. See Using the Interactive Method on page 2-44 for more information.

Parameter Files

You can specify all valid parameters and their values in a parameter file. Storing the parameters in a file allows them to be easily modified or reused, and is the recommended method for invoking Import. If you use different parameters for different databases, you can have multiple parameter files.

Create the parameter file using any flat file text editor. The command-line option `PARFILE=filename` tells Import to read the parameters from the specified file rather than from the command line. For example:

```
imp PARFILE=filename
imp username/password PARFILE=filename
```

The first example does not specify the *username/password* on the command line to illustrate that you can specify them in the parameter file, although, for security reasons, this is not recommended.

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

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

The following example shows a partial parameter file listing:

```
FULL=y
FILE=dbay
INDEXES=y
CONSISTENT=y
```

Note: 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. Import ignores all characters to the right of the pound (#) sign.

You can specify a parameter file at the same time that you are entering parameters 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 which parameters take precedence. For example, assume the parameter file params.dat contains the parameter INDEXES=y and Import is invoked with the following line:

```
imp username/password 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 parameter file.

See Also:

- Import Parameters on page 2-14 for descriptions of the Import parameters
- Exporting and Importing with Oracle Net on page 2-54 for information on how to specify an import from a remote database

Invoking Import As SYSDBA

SYSDBA is used internally and has specialized functions; its behavior is not the same as for generalized users. Therefore, you should not typically need to invoke Import as SYSDBA, except in the following situations:

- At the request of Oracle technical support
- When importing a transportable tablespace set

To invoke Import as SYSDBA, use the following syntax, adding any desired parameters or parameter filenames:

```
imp \'username/password AS SYSDBA\'
```

Optionally, you could also specify an instance name:

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

If either the username or password is omitted, Import will prompt you for it.

This example shows the entire connect string enclosed in quotation marks and backslashes. This is because the string, AS SYSDBA, contains a blank, a situation for which most operating systems require that the entire connect string be placed in quotation marks or marked as a literal by some method. Some operating systems also require that quotation marks on the command line be preceded by an escape character. In this example, backslashes are used as the escape character. If the backslashes were not present, the command-line parser that Export uses would not understand the quotation marks and would remove them before calling Export.

See your Oracle operating system-specific documentation for more information about special and reserved characters on your system.

If you prefer to use the Import interactive mode, see Using the Interactive Method on page 2-44.

Import Modes

The Import utility provides four modes of import.

- Full—Only users with the IMP_FULL_DATABASE role can import in this mode, which imports a full database export dump file. Use the FULL parameter to specify this mode.
- Tablespace—allows a privileged user to move a set of tablespaces from one Oracle database to another. Use the TRANSPORT_TABLESPACE parameter to specify this mode.
- User (Owner)—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. Use the FROMUSER parameter to specify this mode.
- Table—allows you to import specific tables and partitions. A privileged user can qualify the tables by specifying the schema that contains them. Use the TABLES parameter to specify this mode.

> **Caution:** When you use table mode to import tables that have columns of type ANYDATA, you may receive the following error:
>
> ORA-22370: Incorrect usage of method. Nonexistent type.
>
> This indicates that the ANYDATA column depends on other types that are not present in the database. You must manually create dependent types in the target database before you use table mode to import tables that use the ANYDATA type.

All users can import in table mode and user mode. Users with the `IMP_FULL_DATABASE` role (privileged users) can import in all modes.

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

The objects that are imported depend on the Import mode you choose and the mode that was used during the export.

See Also:

- Import Parameters on page 2-14 for information on the syntax for each of these parameters
- Table 1–1 on page 1-9 for a list of the objects that are exported in the various Export modes

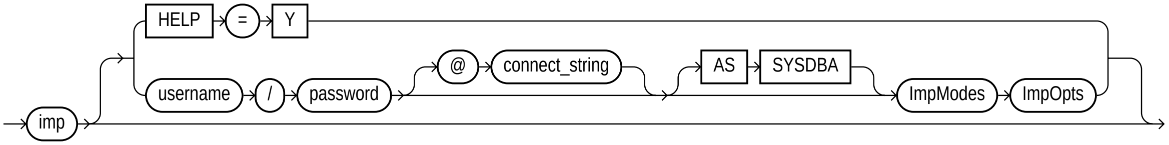
Getting Online Help

Import provides online help. Enter `imp HELP=y` on the command line to invoke it.

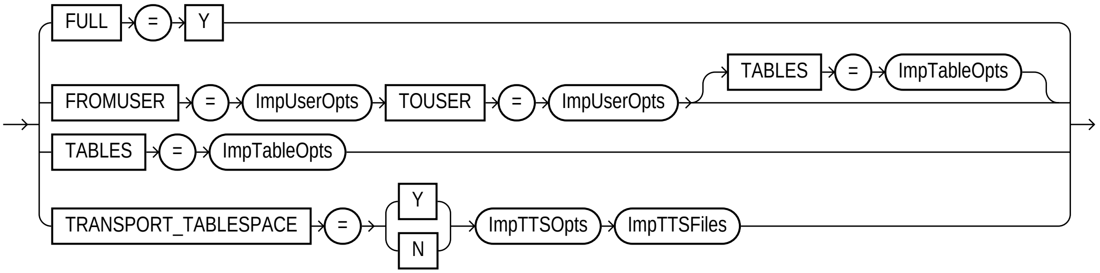
Import Parameters

The following diagrams show the syntax for the parameters that you can specify in the parameter file or on the command line. Following the diagrams are descriptions of each parameter.

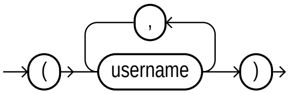
Import_start



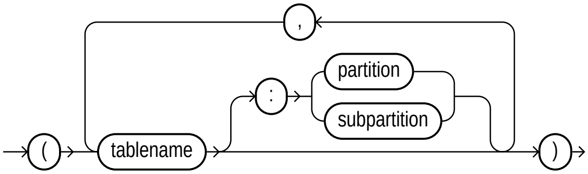
ImpModes



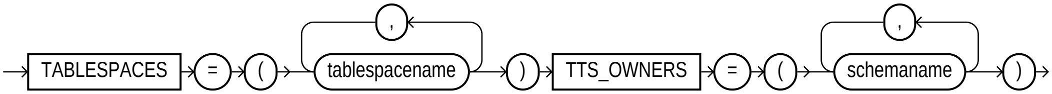
ImpUserOpts



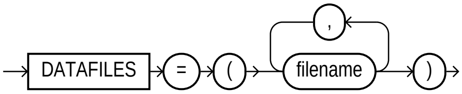
ImpTableOpts



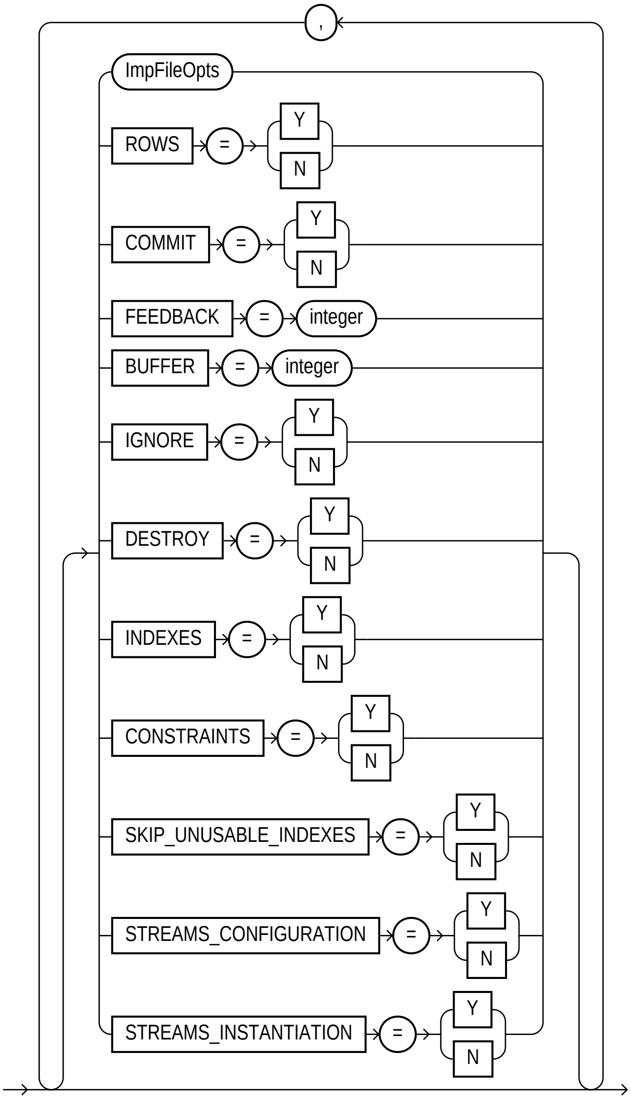
ImpTTSOpts



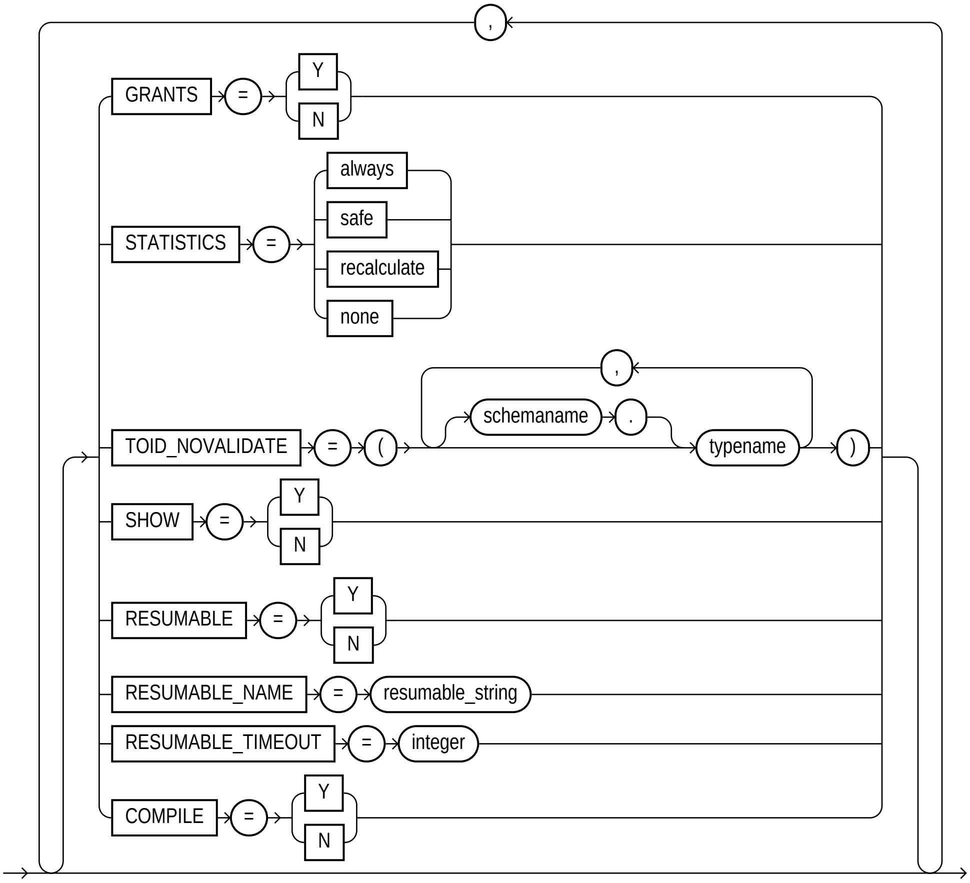
ImpTTSFiles



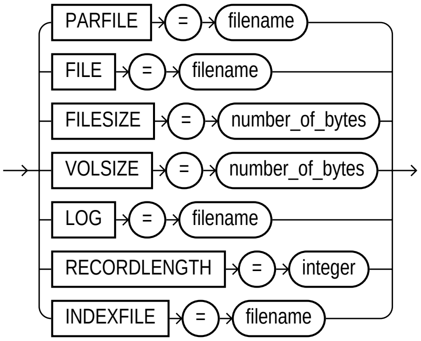
ImpOpts


,
ImpFileOpts
ROWS
=
Y
N
COMMIT
=
Y
N
FEEDBACK
=
integer
BUFFER
=
integer
IGNORE
=
Y
N
DESTROY
=
Y
N
INDEXES
=
Y
N
CONSTRAINTS
=
Y
N
SKIP_UNUSABLE_INDEXES
=
Y
N
STREAMS_CONFIGURATION
=
Y
N
STREAMS_INSTANTIATION
=
Y
N

ImpOpts_continued



ImpFileOpts



The following sections describe parameter functionality and default values.

BUFFER

Default: operating system-dependent

The integer specified for BUFFER is the size, in bytes, of the buffer through which data rows are transferred.

BUFFER 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, UROWID, or DATE 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.

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

CHARSET

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 2-73 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 recoverable error.

If a table does not have a uniqueness constraint, Import could produce duplicate rows when you reimport the data.

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

COMPILE

Default: `y`

Specifies whether or not Import should compile packages, procedures, and functions as they are created.

If `COMPILE=n`, these units are compiled on their first use. For example, packages that are used to build domain indexes are compiled when the domain indexes are created.

> **See Also:** Importing Stored Procedures, Functions, and Packages on page 2-60

CONSTRAINTS

Default: y

Specifies whether or not table constraints are to be imported. The default is to import constraints. If you do not want constraints to be imported, you must set the parameter value to n.

Note that primary key constraints for index-organized tables (IOTs) and object tables are always imported.

DATAFILES

Default: none

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

See Also: TRANSPORT_TABLESPACE on page 2-33

DESTROY

Default: n

Specifies whether or not the existing datafiles 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 statement, which causes Import to reuse the original database's datafiles 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 system (for testing or other purposes), the Import utility will overwrite the first database's datafiles when it creates the tablespace. In this situation you should use the default, DESTROY=n, so that an error occurs if the datafiles 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 datafiles without replacing them.

Caution: 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 period for *n* number of rows imported. For example, if you specify FEEDBACK=10, Import displays a period 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. Because Export supports multiple export files (see the following description of the FILESIZE parameter), you may need to specify multiple filenames to be imported. For example:

```
imp scott/tiger IGNORE=y FILE = dat1.dmp, dat2.dmp, dat3.dmp FILESIZE=2048
```

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

Default: operating-system dependent

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 Oracle operating system-specific documentation before specifying FILESIZE.

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

For information on the maximum size of dump files, see FILESIZE on page 1-21.

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 multischema, user-mode export dump file).

Schema names that appear inside functional indexes, functions, procedures, triggers, type bodies, views, and so on, are *not* affected by FROMUSER or TOUSER processing. Only the *name* of the object is affected. After the import has completed, items in any TOUSER schema should be manually checked for references to old (FROMUSER) schemas, and corrected if necessary.

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 2-32). However, if you omit specifying TOUSER, Import will:

- Import objects into the FROMUSER schema if the export file is a full dump or a multischema, 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: none

Displays a description of the Import parameters. Enter imp HELP=y on the command line to invoke it.

IGNORE

Default: n

Specifies how object creation errors should be handled. If you accept the default, IGNORE=n, Import logs or displays object creation errors before continuing.

If you specify IGNORE=y, Import overlooks object creation errors when it attempts to create database objects, and continues without reporting the errors.

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 CONSTRAINTS=n. If you do a full import with the CONSTRAINTS=n, no constraints for any tables are imported.

If a table already exists and IGNORE=y, then rows are imported into existing tables without any errors or messages being given. You might want to import data into tables that already exist in order to use new storage parameters or because you have already created the table in a cluster.

If a table already exists and IGNORE=n, then errors are reported and the table is skipped with no rows inserted. Also, objects dependent on tables, such as indexes, grants, and constraints, will not be created.

> **Caution:** When you import into existing tables, if no column in the table is uniquely indexed, rows could be duplicated.

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 statements.

When this parameter is specified, index-creation statements 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 statements.
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 statements 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: none

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

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 Oracle operating system-specific documentation.

You can set RECORDLENGTH to any value equal to or greater than your system's BUFSIZ. (The highest value is 64 KB.) 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.

You can also use this parameter to specify the size of the Import I/O buffer.

Note: See your Oracle operating system-specific documentation to determine the proper value or to create a file with a different record size.

RESUMABLE

Default: n

The RESUMABLE parameter is used to enable and disable resumable space allocation. Because this parameter is disabled by default, you must set RESUMABLE=y in order to use its associated parameters, RESUMABLE_NAME and RESUMABLE_TIMEOUT.

See Also:

- *Oracle9i Database Concepts*
- *Oracle9i Database Administrator's Guide* for more information about resumable space allocation

RESUMABLE_NAME

Default: 'User USERNAME (USERID), Session SESSIONID, Instance INSTANCEID'

The value for this parameter identifies the statement that is resumable. This value is a user-defined text string that is inserted in either the USER_RESUMABLE or DBA_RESUMABLE view to help you identify a specific resumable statement that has been suspended.

This parameter is ignored unless the RESUMABLE parameter is set to y to enable resumable space allocation.

RESUMABLE_TIMEOUT

Default: 7200 seconds (2 hours)

The value of the parameter specifies the time period during which an error must be fixed. If the error is not fixed within the timeout period, execution of the statement is aborted.

This parameter is ignored unless the RESUMABLE parameter is set to y to enable resumable space allocation.

ROWS

Default: y

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

SHOW

Default: n

When SHOW=y, 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 parameter.

SKIP_UNUSABLE_INDEXES

Default: n

Specifies whether or not Import skips building indexes that were set to the Index Unusable state (by either system or user). 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.

Note: Indexes that are unique and marked Unusable are not allowed to skip index maintenance. Therefore, the SKIP_UNUSABLE_INDEXES parameter has no effect on unique indexes.

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 will fail.

See Also: The ALTER SESSION statement in the *Oracle9i SQL Reference*

STATISTICS

Default: ALWAYS

Specifies what is done with the database optimizer statistics at import time.

The options are:

- `ALWAYS`

  Always import database optimizer statistics regardless of whether or not they are questionable.

- `NONE`

  Do not import or recalculate the database optimizer statistics.

- `SAFE`

  Import database optimizer statistics only if they are not questionable. If they are questionable, recalculate the optimizer statistics.

- `RECALCULATE`

  Do not import the database optimizer statistics. Instead, recalculate them on import.

See Also:

- *Oracle9i Database Concepts* for more information about the optimizer and the statistics it uses
- STATISTICS on page 1-27
- Importing Statistics on page 2-68

STREAMS_CONFIGURATION

Default: `y`

Specifies whether or not to import any general streams metadata that may be present in the export dump file.

See Also: *Oracle9i Streams*

STREAMS_INSTANTIATION

Default: `n`

Specifies whether or not to import streams instantiation metadata that may be present in the export dump file. Specify `y` if the import is part of an instantiation in a Streams environment.

See Also: *Oracle9i Streams*

TABLES

Default: none

Specifies that the Import is a table-mode import and lists the table names and partition and subpartition names to import. Table-mode import lets you import entire partitioned or nonpartitioned tables. The TABLES parameter restricts the import to the specified tables and their associated objects, as listed in Table 1–1 on page 1-9. You can specify the following values for the TABLES parameter:

- *tablename* specifies the name of the table or tables to be imported. If a table in the list is partitioned and you do not specify a partition name, all its partitions and subpartitions are imported. To import all the exported tables, specify an asterisk (*) as the only table name parameter.

  tablename can contain any number of '%' pattern matching characters, which can each match zero or more characters in the table names in the export file. All the tables whose names match all the specified patterns of a specific table name in the list are selected for import. A table name in the list that consists of all pattern matching characters and no partition name results in all exported tables being imported.

- *partition_name* and *subpartition_name* let you restrict the import to one or more specified partitions or subpartitions within a partitioned table.

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

tablename:partition_name

tablename:subpartition_name

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. If the specified table is not partitioned, the *partition_name* is ignored and the entire table is imported.

The number of tables that can be specified at the same time is dependent on command-line limits.

As the export file is processed, each table name in the export file is compared against each table name in the list, in the order in which the table names were specified in the parameter. To avoid ambiguity and excessive processing time, specific table names should appear at the beginning of the list, and more general table names (those with patterns) should appear at the end of the list.

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/password TABLES=(jones.accts, scott.emp, scott.dept)
```

The valid specification to import these tables is as follows:

```
imp SYSTEM/password FROMUSER=jones TABLES=(accts)
imp SYSTEM/password FROMUSER=scott TABLES=(emp,dept)
```

For a more detailed example, see Example Import of Using Pattern Matching to Import Various Tables on page 2-43.

Note: 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

The following restrictions apply to table names:

- By default, table names in a database are stored as uppercase. If you have a table name in mixed-case or lowercase, and you want to preserve case-sensitivity for the table name, you must enclose the name in quotation marks. The name must exactly match the table name stored in the database.

  Some operating systems require that quotation marks on the command line be preceded by an escape character. The following are examples of how case-sensitivity can be preserved in the different Import modes.

  - In command-line mode:

    ```
    tables='\"Emp\"'
    ```

  - In interactive mode:

    ```
    Table(T) to be exported: "Exp"
    ```

  - In parameter file mode:

    ```
    tables='"Emp"'
    ```

- 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 Import 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, Import interprets everything on the line after `emp#` as a comment and does not import the tables `dept` and `mydata`:

  ```
  TABLES=(emp#, dept, mydata)
  ```

  However, given the following line, the Import utility exports all three tables because `emp#` is enclosed in quotation marks:

  ```
  TABLES=("emp#", dept, mydata)
  ```

> **Note:** Some operating systems require single quotation marks rather than double quotation marks, or the reverse; 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 Import.

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 TRANSPORT_TABLESPACE on page 2-33 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 preexisting 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. Import 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 as follows:

```
TOID_NOVALIDATE=([schemaname.]typename [, ...])
```

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 preceding example, 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"
[...]
```

Caution: 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/password 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/password FROMUSER=scott,fred TOUSER=joe,ted
```

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/password FROMUSER=scott,adams TOUSER=ted,ted
```

Note that user ted is listed twice.

> **See Also:** FROMUSER on page 2-22 for information about restrictions when using FROMUSER and TOUSER

TRANSPORT_TABLESPACE

Default: n

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

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 TRANSPORT_TABLESPACE on page 2-33.

USERID (username/password)

Default: none

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

or

username/password@instance AS SYSDBA

If you connect as user SYS, you must also specify AS SYSDBA in the connect string. Your operating system may require you to treat AS SYSDBA as a special string, in which case the entire string would be enclosed in quotation marks. See Invoking Import As SYSDBA on page 2-12 for more information.

See Also:

- *Oracle9i Heterogeneous Connectivity Administrator's Guide*
- The user's guide for your Oracle Net protocol for information about specifying the *@connect_string* for Oracle Net

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 Oracle operating system-specific documentation for more information.

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

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 the following scenarios:

- Example Import of Selected Tables for a Specific User
- Example Import of Tables Exported by Another User
- Example Import of Tables from One User to Another
- Example Import Session Using Partition-Level Import
- Example Import of Using Pattern Matching to Import Various Tables

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/password 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/password FILE=dba.dmp FROMUSER=scott TABLES=(dept,emp)
```

Import Messages

```
Import: Release 9.2.0.1.0 - Production on Wed Feb 27 17:20:51 2002

(c) Copyright 2002 Oracle Corporation.  All rights reserved.


Connected to: Oracle9i Enterprise Edition Release 9.2.0.1.0 - Production
With the Partitioning and Oracle Data Mining options
JServer Release 9.2.0.1.0 - Production

Export file created by EXPORT:V09.02.00 via conventional path
import done in WE8DEC character set and AL16UTF16 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/password 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/password FROMUSER=blake TOUSER=scott FILE=blake.dmp -
TABLES=(unit,manager)
```

Import Messages

```
Import: Release 9.2.0.1.0 - Production on Wed Feb 27 17:21:40 2002

(c) Copyright 2002 Oracle Corporation.  All rights reserved.


Connected to: Oracle9i Enterprise Edition Release 9.2.0.1.0 - Production
With the Partitioning and Oracle Data Mining options
JServer Release 9.2.0.1.0 - Production

Export file created by EXPORT:V09.02.00 via conventional path

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

import done in WE8DEC character set and AL16UTF16 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/password PARFILE=params.dat
```

The `params.dat` file contains the following information:

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

Command-Line Method

```
> imp SYSTEM/password FILE=scott.dmp FROMUSER=scott TOUSER=blake TABLES=(*)
```

Import Messages

```
Import: Release 9.2.0.1.0 - Production on Wed Feb 27 17:21:44 2002

(c) Copyright 2002 Oracle Corporation.  All rights reserved.


Connected to: Oracle9i Enterprise Edition Release 9.2.0.1.0 - Production
With the Partitioning and Oracle Data Mining options
JServer Release 9.2.0.1.0 - Production

Export file created by EXPORT:V09.02.00 via conventional path

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

import done in WE8DEC character set and AL16UTF16 NCHAR character set
. importing SCOTT's objects into BLAKE
. . 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
```

Import Messages

```
Export: Release 9.2.0.1.0 - Production on Wed Feb 27 17:22:55 2002

(c) Copyright 2002 Oracle Corporation.  All rights reserved.


Connected to: Oracle9i Enterprise Edition Release 9.2.0.1.0 - Production
With the Partitioning and Oracle Data Mining options
JServer Release 9.2.0.1.0 - Production
Export done in WE8DEC character set and AL16UTF16 NCHAR character set

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, these are `p1` and `p3` of table `emp`:

```
> imp scott/tiger TABLES=(emp:p1,emp:p3) FILE=exmpexp.dat ROWS=y
```

Import Messages

```
Import: Release 9.2.0.1.0 - Production on Wed Feb 27 17:22:57 2002

(c) Copyright 2002 Oracle Corporation.  All rights reserved.


Connected to: Oracle9i Enterprise Edition Release 9.2.0.1.0 - Production
With the Partitioning and Oracle Data Mining options
JServer Release 9.2.0.1.0 - Production
```

```
Export file created by EXPORT:V09.02.00 via conventional path
import done in WE8DEC character set and AL16UTF16 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. 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
```

Import Messages

```
Export: Release 9.2.0.1.0 - Production on Wed Feb 27 17:23:06 2002

(c) Copyright 2002 Oracle Corporation.  All rights reserved.


Connected to: Oracle9i Enterprise Edition Release 9.2.0.1.0 - Production
With the Partitioning and Oracle Data Mining options
JServer Release 9.2.0.1.0 - Production
Export done in WE8DEC character set and AL16UTF16 NCHAR character set

About to export specified tables via Conventional Path ...
. . exporting table                            EMP
. . exporting composite partition                  P1
. . exporting subpartition                      P1_SP1          2 rows exported
. . exporting subpartition                      P1_SP2         10 rows exported
. . exporting subpartition                      P1_SP3          7 rows exported
. . exporting composite partition                  P2
. . exporting subpartition                      P2_SP1          4 rows exported
. . exporting subpartition                      P2_SP2          2 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 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
```

Import Messages

```
Import: Release 9.2.0.1.0 - Production on Wed Feb 27 17:23:07 2002

(c) Copyright 2002 Oracle Corporation.  All rights reserved.


Connected to: Oracle9i Enterprise Edition Release 9.2.0.1.0 - Production
With the Partitioning and Oracle Data Mining options
JServer Release 9.2.0.1.0 - Production

Export file created by EXPORT:V09.02.00 via conventional path
import done in WE8DEC character set and AL16UTF16 NCHAR character set
. importing SCOTT's objects into SCOTT
. . importing subpartition             "EMP":"P1_SP2"         10 rows imported
. . importing subpartition             "EMP":"P1_SP3"          7 rows imported
. . importing subpartition             "EMP":"P2_SP1"          4 rows imported
. . importing subpartition             "EMP":"P2_SP2"          2 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 column.

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

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

The following example illustrates these steps.

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

Export: Release 9.2.0.1.0 - Production on Wed Feb 27 17:22:19 2002

(c) Copyright 2002 Oracle Corporation.  All rights reserved.


Connected to: Oracle9i Enterprise Edition Release 9.2.0.1.0 - Production
```

```
With the Partitioning and Oracle Data Mining options
JServer Release 9.2.0.1.0 - Production
Export done in WE8DEC character set and AL16UTF16 NCHAR character set

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> connect scott/tiger
Connected.
SQL> drop table emp cascade constraints;
Statement processed.
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_1,
    16>   partition dept_mid values less than (25)
    17>     tablespace tbs_2,
    18>   partition dept_high values less than (35)
    19>     tablespace tbs_3
    20>   );
Statement processed.
SQL> exit

> imp scott/tiger tables=emp file=empexp.dat ignore=y

Import: Release 9.2.0.1.0 - Production on Wed Feb 27 17:22:25 2002

(c) Copyright 2002 Oracle Corporation.  All rights reserved.


Connected to: Oracle9i Enterprise Edition Release 9.2.0.1.0 - Production
```

```
With the Partitioning and Oracle Data Mining options
JServer Release 9.2.0.1.0 - Production

Export file created by EXPORT:V09.02.00 via conventional path
import done in WE8DEC character set and AL16UTF16 NCHAR character set
. importing SCOTT's objects into SCOTT
. . 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> connect scott/tiger
Connected.
SQL> select empno, deptno from emp partition (dept_low);
EMPNO      DEPTNO
---------- ----------
      7782         10
      7839         10
      7934         10
3 rows selected.
SQL> select empno, deptno from emp partition (dept_mid);
EMPNO      DEPTNO
---------- ----------
      7369         20
      7566         20
      7788         20
      7876         20
      7902         20
5 rows selected.
SQL> select empno, deptno from emp partition (dept_high);
EMPNO      DEPTNO
---------- ----------
      7499         30
      7521         30
      7654         30
      7698         30
      7844         30
      7900         30
6 rows selected.
SQL> exit;
```

Example Import of Using Pattern Matching to Import Various Tables

In this example, pattern matching is used to import various tables for user `scott`.

Parameter File Method

```
imp SYSTEM/password PARFILE=params.dat
```

The `params.dat` file contains the following information:

```
FILE=scott.dmp
IGNORE=n
GRANTS=y
ROWS=y
FROMUSER=scott
TABLES=(%d%,b%s)
```

Command-Line Method

```
imp SYSTEM/password FROMUSER=scott FILE=scott.dmp TABLES=(%d%,b%s)
```

Import Messages

```
Import: Release 9.2.0.1.0 - Production on Wed Feb 27 17:22:25 2002

(c) Copyright 2002 Oracle Corporation.  All rights reserved.


Connected to: Oracle9i Enterprise Edition Release 9.2.0.1.0 - Production
With the Partitioning and Oracle Data Mining options
JServer Release 9.2.0.1.0 - Production

Export file created by EXPORT:V09.02.00 via conventional path

import done in US7ASCII character set and AL16UTF16 NCHAR character set
import server uses JA16SJIS character set (possible charset conversion)
. importing SCOTT's objects into SCOTT
. . importing table                  "BONUS"          0 rows imported
. . importing table                   "DEPT"          4 rows imported
. . importing table               "SALGRADE"          5 rows imported
Import terminated successfully without warnings.
```

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`* combination on the command line, the Import utility prompts you for this information.

When you invoke Import interactively, the response given by Import depends on what you enter at the command line. Table 2–3 shows the possibilities.

Table 2–3 Invoking Import Using the Interactive Method

You enter...	Import's Response
`imp` *`username/password@instance`* `as sysdba`	Starts an Import session
`imp` *`username/password@instance`*	Starts an Import session
`imp` *`username/password`* `as sysdba`	Starts an Import session
`imp` *`username/password`*	Starts an Import session
`imp` *`username@instance`* `as sysdba`	Prompts for password
`imp` *`username@instance`*	Prompts for password
`imp` *`username`*	Prompts for password
`imp` *`username`* `as sysdba`	Prompts for password

In Import interactive mode, you are not prompted to specify whether you want to connect as SYSDBA or *`@instance`*. You must specify AS SYSDBA and/or *`@instance`* with the username.

Additionally, if you omit the password and allow Import to prompt you for it, you cannot specify the *`@instance`* string as well. You can specify *`@instance`* only with *`username`*.

Before you invoke Import using AS SYSDBA, be sure to read Invoking Import As SYSDBA on page 2-12 for information about correct command-line syntax.

After Import is invoked, it displays the following prompts. You may not see all 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 Enter.

```
Import: Release 9.2.0.1.0 - Production on Wed Feb 27 17:22:37 2002

(c) Copyright 2002 Oracle Corporation.  All rights reserved.


Connected to: Oracle9i Enterprise Edition Release 9.2.0.1.0 - Production
With the Partitioning and Oracle Data Mining options
JServer Release 9.2.0.1.0 - Production

Import file: expdat.dmp >
Enter insert buffer size (minimum is 8192) 30720>
Export file created by EXPORT:V09.02.00 via conventional path

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

import done in WE8DEC character set and AL16UTF16 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): no > y
. importing BLAKE's objects into SYSTEM
. . importing table                          "DEPT"          4 rows imported
. . importing table                       "MANAGER"          3 rows imported
Import terminated successfully without warnings.
```

If you specify `No` at the `Import entire export file(yes/no):` prompt, Import prompts you for a schema name and the table names you want to import for that schema, as follows:

```
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 specify only one schema at a time when you use the interactive method.

Warning, Error, and Completion Messages

This section describes the different types of messages issued by Import and how to save them in a log file.

Log File

You can capture all Import messages in a log file, either by using the LOG parameter or, for those systems that permit it, by redirecting Import's output to a file. The Import utility writes a log of detailed information about successful loads and any errors that may occur.

See Also:

- LOG on page 2-25
- Your Oracle operating system-specific documentation for information on redirecting output

Warning Messages

Import does not terminate after recoverable errors. For example, if an error occurs while importing a table, Import displays (or logs) an error message, skips to the next table, and continues processing. These recoverable errors are known as warnings.

Import also issues a warning whenever it encounters an invalid object.

For example, if a nonexistent table is specified as part of a table-mode import, the Import utility imports all other tables. Then it issues a warning and terminates successfully.

Nonrecoverable Error Messages

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

Completion Messages

When an import completes without errors, Import displays the following message:

```
Import terminated successfully without warnings
```

If one or more recoverable errors occurs but Import is able to continue to completion, Import displays the following message:

```
Import terminated successfully with warnings
```

If a nonrecoverable error occurs, Import terminates immediately and displays the following message:

```
Import terminated unsuccessfully
```

> **See Also:** *Oracle9i Database Error Messages* and your Oracle operating system-specific documentation

Exit Codes for Inspection and Display

Import provides the results of an import operation immediately upon completion. Depending on the platform, Import may report the outcome in a process exit code as well as recording the results in the log file. This enables you to check the outcome from the command line or script. Table 2–4 shows the exit codes that are returned for various results.

Table 2–4 Exit Codes for Import

Result	Exit Code
Import terminated successfully without warnings	EX_SUCC
Import terminated successfully with warnings	EX_OKWARN
Import terminated unsuccessfully	EX_FAIL

For UNIX, the exit codes are as follows:

```
EX_SUCC   0
EX_OKWARN 0
EX_FAIL   1
```

Error Handling During an Import

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.

A "tablespace full" error can suspend the import if the `RESUMABLE=y` parameter is specified.

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 Also:**
>
> - *Oracle9i Application Developer's Guide - Fundamentals*
> - *Oracle9i Database Concepts*

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 datatypes, or 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 these errors occur, 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 errors) *are* reported and processing may stop.

Caution: 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. If a sequence is not dropped before the import, it 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 2-19.

Domain Index Metadata

Domain indexes can have associated application-specific metadata that is imported using 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.

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 2-23 for more information.

Guidelines for Using Table-Level Import

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

- All tables exported using any Export mode (except `TRANSPORT_TABLESPACES`) can be imported.
- Users can import the entire (partitioned or nonpartitioned) table, partitions, or subpartitions from a table-level export file into a (partitioned or nonpartitioned) 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 source data from the export file into the target table. After Import, the target table contains the partition definitions of *all* 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.

Guidelines for Using Partition-Level Import

Partition-level Import can only be specified in table mode. It lets you selectively load data from specified partitions or subpartitions in an export file. Keep the following guidelines in mind when using partition-level import.

- Import always stores the rows according to the partitioning scheme of the target table.
- 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 cannot import a nonpartitioned exported table. However, a partitioned table can be imported from a nonpartitioned 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 partition or subpartition of the target table.
- If ROWS=y (default) and IGNORE=y, but the table already existed before Import, all 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 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 nonpartitioned, 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 nonpartitioned table on the import target system.

Migrating Data Across Partitions and Tables

If you specify a partition name for a composite partition, all subpartitions within the composite partition are used as the source.

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

```
imp SYSTEM/password FILE=expdat.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=expdat.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.

Controlling Index Creation and Maintenance

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

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 creation, 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 statements 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.

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 an UNUSABLE state until the duplicates are removed and the index is rebuilt.

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=expdat.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 noncontiguous, 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 the Oracle database server 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` statement.
5. Do a full database import (`FULL=y`) to restore the entire database.

> **See Also:** *Oracle9i Database Administrator's Guide* for more information about creating databases

Network Considerations

This section describes factors to take into account when using 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 Oracle Net

With Oracle Net, 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 Import with Oracle Net, 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 Oracle Net protocol.

> **See Also:**
>
> - *Oracle9i Net Services Administrator's Guide*
> - *Oracle9i Heterogeneous Connectivity Administrator's Guide*

Character Set and Globalization Support Considerations

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

Character Set Conversion

The following sections describe character conversion as it applies to user data and DDL.

User Data

Data of datatypes `CHAR`, `VARCHAR2`, `NCHAR`, `NVARCHAR2`, `CLOB`, and `NCLOB` are written to the export file directly in the character sets of the source database. If the character sets of the source database are different than the character sets of the import database, a single conversion is performed.

Data Definition Language (DDL)

Up to three character set conversions may be required for DDL 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 export file's character set is different than the Import user session character set, then Import converts the character set to its user session character set. Import can only perform this conversion for single-byte character sets. This means that for multibyte character sets, the import file's character set must be identical to the export file's character set.
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, ensure that the export database, the export user session, the import user session, and the import database all use the same character set.

Import and Single-Byte Character Sets

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 system 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 apparent 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.

Import and Multibyte Character Sets

During character set 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.

See Also: *Oracle9i Database Globalization Support Guide*

Considerations When Importing Database Objects

The following sections describe points you should consider when you import particular database objects.

Importing Object Identifiers

The Oracle database 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 preexisting 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 types's unique identifier (TOID) with the identifier stored in the export file. If those match, Import then compares the type's unique hashcode with that stored in the export file. Import will not import table rows if the TOIDs or hashcodes 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 the TOID and hashcode comparison. See TOID_NOVALIDATE on page 2-31 for more information.

Caution: Be very careful about using TOID_NOVALIDATE, because type validation provides an important capability that helps avoid data corruption. Be sure you are confident of your knowledge of type validation and how it works before attempting to perform an import operation 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, the object type already exists, and the object identifiers, hashcodes, and type descriptors match, no error is reported. If the object identifiers or hashcodes 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, hashcodes, or type descriptors 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, the table already exists, and the object identifiers, hashcodes, and type descriptors 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, hashcodes, or type descriptors 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, consider the following when you import objects from one schema into another schema using the FROMUSER and TOUSER parameters:

- If the FROMUSER object types and object tables already exist on the target system, errors occur because the object identifiers of the TOUSER object types and object tables are already in use. The FROMUSER 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. You can

import one of the tables, but the second table receives an error because the object identifier is already in use.

Importing Existing Object Tables and Tables That Contain Object Types

Users frequently create tables before importing data to reorganize tablespace usage or to 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 object tables and for tables containing columns of objects, each object the table references has its name, structure, and version information written out to the Export file. Export also includes object type information from different schemas, as needed.

Import verifies the existence of each object type required by a table prior to importing the table data. This verification consists of a check of the object type's name followed by a comparison of the object type's structure and version from the import system with that found in the Export file.

If an object type name is found on the import system, but the structure or version do not match that from the Export file, an error message is generated and the table data is not imported.

The Import parameter `TOID_NOVALIDATE` can be used to disable the verification of the object type's structure and version for specific objects.

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 nonrecoverable 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 recoverable 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 nonrecoverable errors occur inserting data in an inner table, 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 log file 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 `ROWID`s for the target database. You should execute the following statement to reset the `ROWID`s to their proper values:

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

See Also: *Oracle9i SQL Reference* for more information about the `ANALYZE TABLE` statement

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 filename 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 directory aliases, if the operating system 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 filenames 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

The behavior of Import when a local stored procedure, function, or package is imported depends upon whether the `COMPILE` parameter is set to `y` or to `n`.

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

If `COMPILE=n`, the procedure, function, or package is still imported, but the original timestamp is lost. The compilation takes place the next time the procedure, function, or package is used.

See Also: COMPILE on page 2-19

Importing Java Objects

When you import Java objects into any schema, the Import utility leaves the resolver unchanged. (The resolver is the list of schemas used to resolve Java full names.) This means that after an import, all user classes are left in an invalid state until they are either implicitly or explicitly revalidated. An implicit revalidation occurs the first time the classes are referenced. An explicit revalidation occurs when the SQL statement `ALTER JAVA CLASS...RESOLVE` is used. Both methods result in the user classes being resolved successfully and becoming valid.

Importing External Tables

Import does not verify that the location referenced by the external table is correct. If the formats for directory and filenames used in the table'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 table and ensure the table's specification is valid on the import system.

Importing Advanced Queue (AQ) Tables

Importing a queue table also imports any underlying queues and the related dictionary information. A queue can be imported only at the granularity level of the queue table. When a queue table is imported, export pretable and posttable action procedures maintain the queue dictionary.

> **See Also:** *Oracle9i Application Developer's Guide - Advanced Queuing*

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.

Import can be used to convert `LONG` columns to `CLOB` columns. To do this, first create a table specifying the new `CLOB` column. When Import is run, the `LONG` data is converted to `CLOB` format. The same technique can be used to convert `LONG RAW` columns to `BLOB` columns.

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. 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 requires 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 Partitioned 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 nonpartitioned index or index partition that is marked Indexes Unusable or is otherwise not suitable.

Support for Fine-Grained Access Control

You can export tables with fine-grained access control policies enabled. When doing so, keep the following considerations in mind:

To restore the fine-grained access control policies, the user who imports from an export file containing such tables must have the following privileges:

- EXECUTE privilege on the DBMS_RLS package so that the tables' security policies can be reinstated.
- EXPORT_FULL_DATABASE role enabled or EXEMPT ACCESS POLICY granted

If a user without the correct privileges attempts to import from an export file that contains tables with fine-grained access control policies, a warning message will be issued. Therefore, it is advisable for security reasons that the exporter and importer of such tables be the DBA.

> **See Also:** *Oracle9i Application Developer's Guide - Fundamentals* for more information about fine-grained access control

Materialized Views 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.

See Also: *Oracle9i Replication* for Import-specific information about migration and compatibility and for more information about snapshots and snapshot logs

Snapshot Log

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

When a `ROWID` snapshot log is exported, `ROWID`s 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 values of the primary keys do retain their meaning upon Import. Therefore, primary key snapshots can do a fast refresh after the import.

See Also: *Oracle9i Replication* for information about primary key snapshots

Snapshots

A snapshot that has been restored from an export file has reverted 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 and TOUSER to import snapshot data, an error will be written to the Import log file and the items will not be imported.

Transportable Tablespaces

Transportable tablespaces let you 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 and 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 CD-ROMs.

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

See Transportable Tablespaces on page 1-59 for information on creating an Export file containing transportable tablespace metadata.

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

- `TRANSPORT_TABLESPACE`
- `TABLESPACES`
- `DATAFILES`
- `TTS_OWNERS`

See TRANSPORT_TABLESPACE on page 2-33, TABLESPACES on page 2-31, DATAFILES on page 2-20, and TTS_OWNERS on page 2-33 for more information.

See Also:

- *Oracle9i Database Administrator's Guide* for details about how to move or copy tablespaces to another database
- *Oracle9i Database Concepts* for an introduction to the transportable tablespaces feature

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`, `VARRAY`, or `OPAQUE` type columns
- Has an index-organized 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 2-67 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, VARRAY, or OPAQUE type columns, LOB, VARRAY, or OPAQUE type 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, and so forth).

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 resides 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 using the Import utility to import data, you may want to create large tables with different storage parameters. 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 1-17.

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 `imp` 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` statement 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, you must first create a table and then import the table again, specifying `IGNORE=y.`

Objects are not imported into the default tablespace if the tablespace does not exist or you do not have the necessary quotas for your default tablespace.

Reorganizing Tablespaces

If a user's quota allows 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, 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.

   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 for joe.
5. Import joe's tables. (By default, Import puts joe's tables into tablespace B.)

Importing Statistics

If statistics are requested at export time and analyzer statistics are available for a table, Export will place the ANALYZE statement to recalculate the statistics for the table into the dump file. In most circumstances, Export will also write the precalculated optimizer statistics for tables, indexes, and columns to the dump file. See the description of the Export parameter STATISTICS on page 1-27 and the Import parameter 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. By default, Import will always use the precalculated statistics that are found in the export dump file.

The Export utility flags certain precalculated statistics as questionable. See the Export parameter, STATISTICS on page 1-27 for more information. In certain situations, the importer might want to import only unquestionable statistics, and may not want to import precalculated statistics in the following situations:

- 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 precalculated statistics. This feature allows plan generation for queries to be tuned in a nonproduction database using statistics from a production database. In these cases, the importer should specify STATISTICS=SAFE.

In certain situations, the importer might want to always use ANALYZE statements rather than precalculated 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 should specify STATISTICS=RECALCULATE to force the recalculation of statistics.

If you do not want any statistics to be established by Import, you should specify STATISTICS=NONE.

Using Export and Import to Partition a Database Migration

When you use the Export and Import utilities to migrate a large database, it may be more efficient to partition the migration into multiple export and import jobs. If you decide to partition the migration, be aware of the following advantages and disadvantages.

Advantages of Partitioning a Migration

Partitioning a migration has the following advantages:

- Time required for the migration may be reduced because many of the subjobs can be run in parallel.
- The import can start as soon as the first export subjob completes, rather than waiting for the entire export to complete.

Disadvantages of Partitioning a Migration

Partitioning a migration has the following disadvantages:

- The export and import processes become more complex.
- Support of cross-schema references for certain types of objects may be compromised. For example, if a schema contains a table with a foreign key constraint against a table in a different schema, you may not have all required parent records when you import the table into the dependent schema.

How to Use Export and Import to Partition a Database Migration

To perform a database migration in a partitioned manner, take the following steps:

1. For all top-level metadata in the database, issue the following commands:

   a. `exp dba/password FILE=full FULL=y CONSTRAINTS=n TRIGGERS=n ROWS=n INDEXES=n`

   b. `imp dba/password FILE=full FULL=y`

2. For each schema*n* in the database, issue the following commands:

   a. `exp dba/password OWNER=scheman FILE=scheman`

   b. `imp dba/password FILE=scheman FROMUSER=scheman TOUSER=scheman IGNORE=y`

All exports can be done in parallel. When the import of `full.dmp` completes, all remaining imports can also be done in parallel.

Using Export Files from a Previous Oracle Release

The following sections describe considerations when you import data from earlier versions of the Oracle database server into an Oracle9*i* server.

> **See Also:** Using Different Releases and Versions of Export on page 1-61

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 Oracle9*i* server.

> **See Also:** *Oracle9i Database Migration*

Check Constraints on DATE Columns

In Oracle9*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 versions prior to Oracle8*i*, data imported from an earlier Oracle database might not have used the `TO_DATE` function. In such cases, the constraints are imported into the Oracle9*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 Oracle9*i* server.

User Privileges

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

CHAR Columns

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

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 Oracle9*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 Oracle9*i* server does make this check, however. As a result, column defaults that could be imported into a version 6 database may not import into Oracle9*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 statement:

```
DESCRIBE user_sys_privs
```

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

See Also: *Oracle9i Database Migration*

Using Oracle Version 5 Export Files

Oracle9*i* Import reads Export dump files created by Oracle release 5.1.22 and higher. 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.

Restrictions When Using Different Releases and Versions of Export and Import

The following restrictions apply when you are using different releases of Export and Import:

- Export dump files can be read only by the Import utility because they are stored in a special binary format.
- Any export dump file can be imported into a higher release of the Oracle database server.
- Export dump files cannot be read by previous versions and releases of the Import utility. Therefore, a release 8.1 export file cannot be imported by a release 8.0 Import utility and a version 8 export dump file cannot be imported by a version 7 Import utility.
- The Import utility can read export dump files created by Export release 5.1.22 and higher.
- The Import utility cannot read export dump files created by the Export utility of a higher maintenance release or version. For example, a release 8.1 export dump

file cannot be imported by a release 8.0 Import utility, and a version 8 export dump file cannot be imported by a version 7 Import utility.

- The Oracle version 6 (or earlier) Export utility cannot be used against an Oracle8 or higher database.
- Whenever a lower version of the Export utility runs with a higher version of the Oracle database server, categories of database objects that did not exist in the lower version are excluded from the export. For example, partitioned tables did not exist in the Oracle database server version 7. So, if you need to move a version 8 partitioned table to a version 7 database, you must first reorganize the table into a nonpartitioned table.
- Export files generated by Oracle9*i* Export, either direct path or conventional path, are incompatible with earlier releases of Import and can be imported only with Oracle9*i* Import. When backward compatibility is an issue, use the earlier release or version of the Export utility against the Oracle9*i* database.
- You cannot import job queues from a release 8.1.7 database into earlier releases of the database. Therefore, you must manually restart your jobs after the import is finished.

The CHARSET Parameter

Default: none

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 database 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 and the export file is ASCII, Import will verify that the user session character set is ASCII. Or, if the export file is EBCDIC, Import will verify that the user session character set is EBCDIC.

If you are using a version of Oracle greater than version 5 or 6, 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.

PartII

SQL*Loader

The chapters in this section describe the SQL*Loader utility:

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 Command-Line Reference"

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

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 datafiles, tables and columns, the location of data, the type and format of data to be loaded, and more.

Chapter 6, "Field List Reference"

This chapter describes the field list section of a SQL*Loader control file. The field list provides information about fields being loaded, such as position, datatype, conditions, and delimiters.

Chapter 7, "Loading Objects, LOBs, and Collections"

This chapter describes how to load objects in various formats, as well as loading object tables and REF columns. This chapter also discusses loading LOBs and columns.

Chapter 8, "SQL*Loader Log File Reference"

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

Chapter 9, "Conventional and Direct Path Loads"

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

Chapter 10, "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.

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 Features

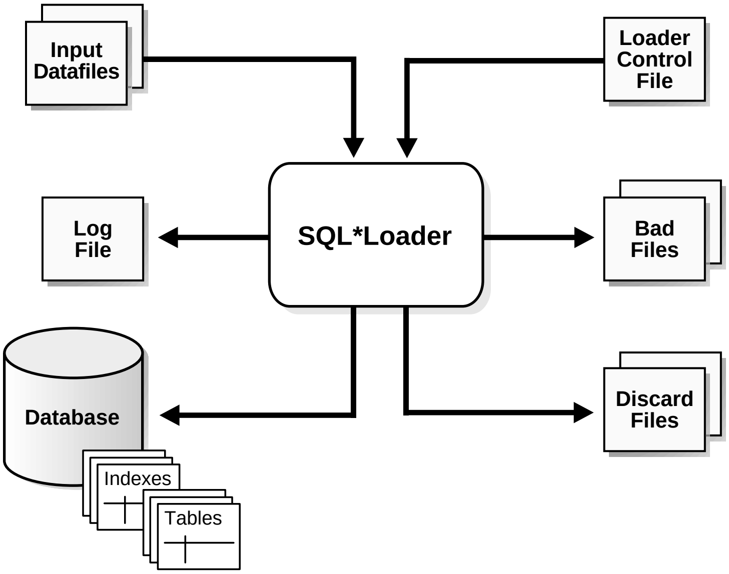
SQL*Loader loads data from external files into tables of an Oracle database. It has a powerful data parsing engine that puts little limitation on the format of the data in the datafile. You can use SQL*Loader to do the following:

- Load data from multiple datafiles during the same load session.
- Load data into multiple tables during the same load session.
- Specify the character set of the data.

- Selectively load data (you can load records based on the records' values).
- Manipulate the data before loading it, using SQL functions.
- Generate unique sequential key values in specified columns.
- Use the operating system's file system to access the datafiles.
- Load data from disk, tape, or named pipe.
- Generate sophisticated error reports, which greatly aids troubleshooting.
- Load arbitrarily complex object-relational data.
- Use secondary datafiles for loading LOBs and collections.
- Use either conventional or direct path loading. While conventional path loading is very flexible, direct path loading provides superior loading performance. See Chapter 9.
- Use a DB2 Load Utility control file as a SQL*Loader control file with few or no changes involved. See Appendix B.

A typical SQL*Loader session takes as input a control file, which controls the behavior of SQL*Loader, and one or more datafiles. The output of SQL*Loader is an Oracle database (where the data is loaded), a log file, a bad file, and potentially, a discard file. An example of the flow of a SQL*Loader session is shown in Figure 3–1.

*Figure 3–1 SQL*Loader Overview*



SQL*Loader Control File

The control file is a text file written in a language that SQL*Loader understands. 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.

Although not precisely defined, a control file can be said to have three sections.

The first section contains session-wide information, for example:

- Global options such as bindsize, rows, records to skip, and so on
- `INFILE` clauses to specify where the input data is located
- Data to be loaded

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 loaded, such as the table name and the columns of the table.

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 (--) that mark the beginning of the comment to the end of the line. 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.
- The `CONSTANT` keyword has special meaning to SQL*Loader and is therefore reserved. To avoid potential conflicts, Oracle Corporation recommends that you do not use the word `CONSTANT` as a name for any tables or columns.

See Also: Chapter 5 for details about control file syntax and semantics

Input Data and Datafiles

SQL*Loader reads data from one or more files (or operating system equivalents of files) specified in the control file. 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. The record format can be specified in the control file with the `INFILE` parameter. If no record format is specified, the default is stream record format.

Note: If data is specified inside the control file (that is, `INFILE *` was specified in the control file), then the data is interpreted in the stream record format with the default record terminator.

Fixed Record Format

A file is in fixed record format when all records in a datafile are the same byte length. Although this format is the least flexible, it results in better performance than variable or stream format. Fixed format is also simple to specify. For example:

```
INFILE datafile_name "fix n"
```

This example 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. Assuming that a period (.) indicates a space, the first physical record is [001,...cd,.] which is exactly eleven bytes (assuming a single-byte character set). The second record is [0002,fghi,\n] followed by the newline character (which is the eleventh byte), and so on. Note that newline characters are not required with the fixed record format.

Note that the length is always interpreted in bytes, even if character-length semantics are in effect for the file. This is necessary because the file could contain a mix of fields, some of which are processed with character-length semantics and others which are processed with byte-length semantics. See Character-Length Semantics on page 5-22.

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, col2)

example.dat:
001,   cd, 0002,fghi,
00003,lmn,
1, "pqrs",
0005,uvwx,
```

Variable Record Format

A file is in variable record format when the length of each record in a character field is included 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 that is to be interpreted as being in variable record format as follows:

```
INFILE "datafile_name" "var n"
```

In this example, *n* specifies the number of bytes in the record length field. If *n* is not specified, SQL*Loader assumes a length of 5 bytes. Specifying *n* larger than 40 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. The first is specified to be 009 (that is, 9) bytes long, the second is 010 bytes long (that is, 10, including a 1-byte newline), and the third is 012 bytes long (also including a 1-byte newline). Note that newline characters are not required with the variable record format. This example also assumes a single-byte character set for the datafile.

The lengths are always interpreted in bytes, even if character-length semantics are in effect for the file. This is necessary because the file could contain a mix of fields, some processed with character-length semantics and others processed with byte-length semantics. See Character-Length Semantics on page 5-22.

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

A file is in stream record format when the records are not specified by size; instead SQL*Loader forms records by scanning for the *record terminator*. Stream record format is the most flexible format, but there can be a negative effect on performance. The specification of a datafile to be interpreted as being in stream record format looks similar to the following:

```
INFILE datafile_name ["str terminator_string"]
```

The *`terminator_string`* is specified as either *`'char_string'`* or *`X'hex_string'`* where:

- *`'char_string'`* is a string of characters enclosed in single or double quotation marks
- *`X'hex_string'`* is a byte string in hexadecimal format

When the *`terminator_string`* contains special (nonprintable) characters, it should be specified as a *`X'hex_string'`*. However, some nonprintable characters can be specified as (*`'char_string'`*) by using a backslash. For example:

- `\n` indicates a line feed

- \t indicates a horizontal tab
- \f indicates a form feed
- \v indicates a vertical tab
- \r indicates a carriage return

If the character set specified with the NLS_LANG parameter for your session is different from the character set of the datafile, character strings are converted to the character set of the datafile. This is done before SQL*Loader checks for the default record terminator.

Hexadecimal strings are assumed to be in the character set of the datafile, so no conversion is performed.

On UNIX-based platforms, if no *terminator_string* is specified, SQL*Loader defaults to the line feed character, \n.

On Windows NT, if no *terminator_string* is specified, then SQL*Loader uses either \n or \r\n as the record terminator, depending on which one it finds first in the datafile. This means that if you know that one or more records in your datafile has \n embedded in a field, but you want \r\n to be used as the record terminator, you must specify it.

Example 3–3 illustrates loading data in stream record format where the terminator string is specified using a character string, '|\n'. The use of the backslash character allows the character string to specify the nonprintable line feed character.

Example 3–3 Loading Data in Stream Record Format

```
load data
infile 'example.dat'  "str '|\n'"
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 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

See Also:

- Assembling Logical Records from Physical Records on page 5-27
- Case Study 4: Loading Combined Physical Records on page 10-14 for an example of how to use continuation fields to form one logical record from multiple physical records

Data Fields

Once a logical record is formed, field setting on the logical record is done. Field setting is a process in which SQL*Loader uses control-file field specifications to determine which parts of logical record data correspond to which control-file fields. It is possible for two or more field specifications to claim the same data. Also, it is possible for a logical record to contain data that is not claimed by any 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 data field's beginning, end, or both, can be specified. This specification form is not the most flexible, but it provides high field-setting performance.
- The strings delimiting (enclosing and/or terminating) a particular data field can be specified. A delimited data field is assumed to start where the last data field ended, unless the byte position of the start of the data field is specified.
- The byte offset and/or the length of the data field 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.
- Length-value datatypes can be used. In this case, the first *n* number of bytes of the data field contain information about how long the rest of the data field is.

See Also:

- SQL*Loader Datatypes on page 6-7
- Specifying Delimiters on page 6-24

LOBFILEs and Secondary Datafiles (SDFs)

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.

For example, you might use LOBFILEs to load employee names, employee IDs, and employee resumes. You could read the employee names and IDs from the main datafiles and you could read the resumes, which can be quite lengthy, from LOBFILEs.

You might also use LOBFILEs to facilitate the loading of XML data. You can use XML columns to hold data that models structured and semistructured data. Such data can be quite lengthy.

Secondary datafiles (SDFs) are similar in concept to 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. Only a collection_fld_spec can name an SDF as its data source.

SDFs are specified using the SDF parameter. The SDF parameter can be followed by either the file specification string, or a FILLER field that is mapped to a data field containing one or more file specification strings.

See Also:

- Loading LOB Data from LOBFILEs on page 7-23
- Secondary Datafiles (SDFs) on page 7-31

Data Conversion and Datatype Specification

During a conventional path load, *data fields* in the datafile are converted into *columns* in the database (direct path loads are conceptually similar, but the implementation is different). There are two conversion steps:

1. SQL*Loader uses the field specifications in the control file to interpret the format of the datafile, parse the input data, and populate the bind arrays that correspond to a SQL `INSERT` statement using that data.
2. The Oracle database server accepts the data and executes the `INSERT` statement to store the data in the database.

The Oracle database server uses the datatype of the column to convert the data into its final, stored form. 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*.

Discarded and Rejected Records

Records read from the input file might not be inserted into the database. Such records are placed in either a bad file or a discard file.

The Bad File

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

SQL*Loader Rejects

Datafile 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.

Oracle Rejects

After a datafile record is accepted for processing by SQL*Loader, it is sent to the Oracle database server for insertion into a table as a row. If the Oracle database server determines that the row is valid, then the row is inserted into the table. If the row is determined to be invalid, then the record is rejected and SQL*Loader puts it in the bad file. The row may be invalid, 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.

See Also:

- Specifying the Bad File on page 5-11
- Case Study 4: Loading Combined Physical Records on page 10-14 for an example use of a bad file

The Discard File

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

See Also:

- Case Study 4: Loading Combined Physical Records on page 10-14
- Specifying the Discard File on page 5-14

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.

See Also:

- Chapter 8, "SQL*Loader Log File Reference"
- Chapter 10, "SQL*Loader Case Studies" for sample log files

Conventional Path Loads, Direct Path Loads, and External Table Loads

SQL*Loader provides the following methods to load data:

- Conventional Path Loads
- Direct Path Loads

- External Table Loads

Conventional Path Loads

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 no more data is left to read), an array insert is executed.

See Also:

- Data Loading Methods on page 9-1
- Bind Arrays and Conventional Path Loads on page 5-44

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. Note also that because LOB data is loaded after the array insert has been performed, `BEFORE` and `AFTER` row triggers may not work as expected for `LOB` columns. This is because the triggers fire before SQL*Loader has a chance to load the LOB contents into the column. For instance, suppose you are loading a `LOB` column, `C1,` with data and that you want a `BEFORE` row trigger to examine the contents of this `LOB` column and derive a value to be loaded for some other column, `C2,` based on its examination. This is not possible because the LOB contents will not have been loaded at the time the trigger fires.

Direct Path Loads

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.

See Also: Direct Path Load on page 9-5

Parallel Direct Path

A parallel direct path load allows multiple direct path load sessions to concurrently load the same data segments (allows intrasegment parallelism). Parallel direct path is more restrictive than direct path.

See Also: Parallel Data Loading Models on page 9-30

External Table Loads

An external table load creates an external table for data in a datafile and executes INSERT statements to insert the data from the datafile into the target table.

The advantages of using external table loads over conventional path and direct path loads are as follows:

- An external table load attempts to load datafiles in parallel. If a datafile is big enough, it will attempt to load that file in parallel.
- An external table load allows modification of the data being loaded by using SQL functions and PL/SQL functions as part of the INSERT statement that is used to create the external table.

See Also:

- Chapter 11, "External Tables Concepts"
- Chapter 12, "External Tables Access Parameters"

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 *Oracle9i Database Concepts* and in the *Oracle9i Database 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 their entirety in a single column position in a row. These objects do not have object identifiers and cannot be referenced.

If the object type of the column object is declared to be nonfinal, then SQL*Loader allows a derived type (or subtype) to be loaded into the column object.

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 (OIDs) for each of the objects in the table. Columns in other tables can refer to these objects by using the OIDs.

If the object type of the object table is declared to be nonfinal, then SQL*Loader allows a derived type (or subtype) to be loaded into the row object.

> **See Also:**
>
> - Loading Column Objects on page 7-1
> - Loading Object Tables on page 7-12

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

VARRAYs 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.

> **See Also:** Loading Collections (Nested Tables and VARRAYs) on page 7-29 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 LOB types:

- `BLOB:` a LOB containing unstructured binary data

- CLOB: a LOB containing character data
- NCLOB: a LOB containing characters in a database national character set
- BFILE: a BLOB stored outside of the database tablespaces in a server-side operating system 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."

> **See Also:** Loading LOBs on page 7-18 for details on using SQL*Loader control file data definition language to load these LOB types

Partitioned Object Support

SQL*Loader supports loading partitioned objects in the database. A partitioned object in an Oracle database 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 2000 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.

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
- A nonpartitioned table

Application Development: Direct Path Load API

Oracle Corporation provides a direct path load API for application developers. See the *Oracle Call Interface Programmer's Guide* for more information.

4

SQL*Loader Command-Line Reference

This chapter describes the command-line parameters used to invoke SQL*Loader. The following topics are discussed:

- Invoking SQL*Loader
- Command-Line Parameters
- Exit Codes for Inspection and Display

Invoking SQL*Loader

When you invoke SQL*Loader, you can specify certain parameters to establish session characteristics. Parameters can be entered in any order, optionally separated by commas. You specify values for parameters, or in some cases, you can accept the default without entering a value.

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
```

If you invoke SQL*Loader without specifying any parameters, SQL*Loader displays a help screen similar to the following. It lists the available parameters and their default values.

```
sqlldr
...
SQL*Loader: Release 9.2.0.1.0 - Production on Wed Feb 27 12:06:17 2002

(c) Copyright 2002 Oracle Corporation.  All rights reserved.
```

```
Usage: SQLLDR keyword=value [,keyword=value,...]

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  (Default 256000)
    silent -- Suppress messages during run (header,feedback,errors,discards,partitions)
    direct -- use direct path                      (Default FALSE)
   parfile -- parameter file: name of file that contains parameter specifications
  parallel -- do parallel load                     (Default FALSE)
      file -- File to allocate extents from
skip_unusable_indexes -- disallow/allow unusable indexes or index partitions  (Default FALSE)
skip_index_maintenance -- do not maintain indexes, mark affected indexes as unusable  (Default
FALSE)
  readsize -- Size of Read buffer                  (Default 1048576)
external_table -- use external table for load; NOT_USED, GENERATE_ONLY, EXECUTE  (Default NOT_
USED)
columnarrayrows -- Number of rows for direct path column array  (Default 5000)
streamsize -- Size of direct path stream buffer in bytes  (Default 256000)
multithreading -- use multithreading in direct path
 resumable -- enable or disable resumable for current session  (Default FALSE)
resumable_name -- text string to help identify resumable statement
resumable_timeout -- wait time (in seconds) for RESUMABLE  (Default 7200)
date_cache -- size (in entries) of date conversion cache  (Default 1000)

PLEASE NOTE: Command-line parameters may be specified either by position or by keywords.
An example of the former case is 'sqlldr scott/tiger foo'; an example of the latter is 'sqlldr
control=foo userid=scott/tiger'.  One may specify parameters by position before but not after
parameters specified by keywords.  For example,'sqlldr scott/tiger control=foo logfile=log' is
allowed, but 'sqlldr scott/tiger control=foo log' is not, even though the position of the
parameter 'log' is correct.
```

Note: The command to invoke SQL*Loader is operating system-dependent. The examples in this chapter use the UNIX-based name, `sqlldr`. See your Oracle operating system-specific documentation for the correct command for your system.

See Also: Command-Line Parameters on page 4-3 for descriptions of all the command-line parameters

Specifying Parameters in the Control File

If the length of the command line exceeds the size of the maximum command line on your system, you can put some command-line parameters in the control file. See OPTIONS Clause on page 5-4 for information on how to do this.

They can also be specified in a separate file specified by the `PARFILE` parameter. These alternative methods are useful for specifying parameters whose values seldom change. Parameters specified in this manner can be overridden from the command line.

See Also:

- Chapter 5 for a detailed description of the SQL*Loader control file
- PARFILE (parameter file) on page 4-10

Command-Line Parameters

This section describes each SQL*Loader command-line parameter. The defaults and maximum values listed for these parameters are for UNIX-based systems. They may be different on your operating system. Refer to your Oracle operating system-specific documentation for more information.

BAD (bad file)

Default: The name of the datafile, with an extension of `.bad`.

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 default is used.

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.

> **See Also:** Specifying the Bad File on page 5-11 for information about the format of bad files

BINDSIZE (maximum size)

Default: To see the default value for this parameter, invoke SQL*Loader without any parameters, as described in Invoking SQL*Loader on page 4-1.

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.

> **See Also:**
>
> - Bind Arrays and Conventional Path Loads on page 5-44
> - READSIZE (read buffer size) on page 4-10

COLUMNARRAYROWS

Default: To see the default value for this parameter, invoke SQL*Loader without any parameters, as described in Invoking SQL*Loader on page 4-1.

Specifies the number of rows to allocate for direct path column arrays. The value for this parameter is not calculated by SQL*Loader. You must either specify it or accept the default.

> **See Also:**
>
> - Using CONCATENATE to Assemble Logical Records on page 5-27
> - Specifying the Number of Column Array Rows and Size of Stream Buffers on page 9-21

CONTROL (control file)

Default: none

CONTROL specifies the name of the SQL*Loader control file that describes how to load data. If a file extension or file type is not specified, it defaults to .ctl. If the filename is omitted, SQL*Loader prompts you for it.

If the name of your SQL*Loader control file contains special characters, your operating system may require that they be preceded by an escape character. Also, if your operating system uses backslashes in its file system paths, you may need to use multiple escape characters or to enclose the path in quotation marks. See your Oracle operating system-specific documentation for more information.

> **See Also:** Chapter 5 for a detailed description of the SQL*Loader control file

DATA (datafile)

Default: The name of the control file, with an extension of `.dat`.

`DATA` specifies the name of the datafile containing the data to be loaded. If you do not specify a file extension or file type, the default is `.dat.`

If you specify a datafile on the command line and also specify datafiles in the control file with `INFILE,` the data specified on the command line is processed first. The first datafile specified in the control file is ignored. All other datafiles specified in the control file are processed.

If you specify a file processing option when loading data from the control file, a warning message will be issued.

DATE_CACHE

Default: Enabled (for `1000` elements). To completely disable the date cache feature, set it to `0`.

`DATE_CACHE` specifies the date cache size (in entries). For example, `DATE_CACHE=5000` specifies that each date cache created can contain a maximum of 5000 unique date entries. Every table has its own date cache, if one is needed. A date cache is created only if at least one date or timestamp value is loaded that requires datatype conversion in order to be stored in the table.

The date cache feature is only available for direct path loads. It is enabled by default. The default date cache size is 1000 elements. If the default size is used and the number of unique input values loaded exceeds 1000, then the date cache feature is automatically disabled for that table. However, if you override the default and specify a nonzero date cache size and that size is exceeded, then the cache is not disabled.

You can use the date cache statistics (entries, hits, and misses) contained in the log file to tune the size of the cache for future similar loads.

See Also: Specifying a Value for the Date Cache on page 9-22

DIRECT (data path)

Default: `false`

`DIRECT` specifies the data path, that is, the load method to use, either conventional path or direct path. A value of `true` specifies a direct path load. A value of `false` specifies a conventional path load.

See Also: Chapter 9, "Conventional and Direct Path Loads"

DISCARD (filename)

Default: The name of the datafile, with an extension of `.dsc`.

`DISCARD` specifies a discard file (optional) to be created by SQL*Loader to store records that are neither inserted into a table nor rejected.

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.

See Also: Discarded and Rejected Records on page 3-10 for information about the format of discard files

DISCARDMAX (integer)

Default: `ALL`

`DISCARDMAX` specifies the number of discard records to allow before data loading is terminated. To stop on the first discarded record, specify one (1).

ERRORS (errors to allow)

Default: To see the default value for this parameter, invoke SQL*Loader without any parameters, as described in Invoking SQL*Loader on page 4-1.

`ERRORS` specifies the maximum number of insert errors to allow. If the number of errors exceeds the value specified for `ERRORS`, then SQL*Loader terminates the load. 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, multitable loads do not terminate immediately if errors exceed the error limit. When SQL*Loader encounters the maximum number of errors for a multitable 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.

EXTERNAL_TABLE

Default: NOT_USED

EXTERNAL_TABLE instructs SQL*Loader whether or not to load data using the external tables option. There are three possible values:

- NOT_USED—the default value. It means the load is performed using either conventional or direct path mode.
- GENERATE_ONLY—places all the SQL statements needed to do the load using external tables, as described in the control file, in the SQL*Loader log file. These SQL statements can be edited and customized. The actual load can be done later without the use of SQL*Loader by executing these statements in SQL*Plus. See Log File Created When EXTERNAL_TABLE=GENERATE_ONLY on page 8-8 for an example of what this log file would look like.
- EXECUTE—attempts to execute the SQL statements that are needed to do the load using external tables. However, if any of the SQL statements returns an error, then the attempt to load stops. Statements are placed in the log file as they are executed. This means that if a SQL statement returns an error, then the remaining SQL statements required for the load will not be placed in the control file.

Note that the external tables option uses directory objects in the database to indicate where all datafiles are stored and to indicate where output files, such as bad files and discard files, are created. You must have READ access to the directory objects containing the datafiles, and you must have WRITE access to the directory objects where the output files are created. If there are no existing directory objects for the location of a datafile or output file, SQL*Loader will generate the SQL statement to create one. Note that if the EXECUTE option is specified, then you must have the CREATE ANY DIRECTORY privilege.

Note: The EXTERNAL_TABLE=EXECUTE qualifier tells SQL*Loader to create an external table that can be used to load data and then execute the INSERT statement to load the data. All files in the external table must be identified as being in a directory object. SQL*Loader is supposed to use directory objects that already exist and that you have privileges to access. However, SQL*Loader does not find the matching directory object. Because no match is found, SQL*Loader attempts to create a temporary directory object. If you do not have privileges to create new directory objects, then the operation fails.

To work around this, use EXTERNAL_TABLE=GENERATE_ONLY to create the SQL statements that SQL*Loader would try to execute. Extract those SQL statements and change references to directory objects to be the directory object that you have privileges to access. Then, execute those SQL statements

When using a multitable load, SQL*Loader does the following:

1. Creates a table in the database that describes all fields in the datafile that will be loaded into any table.
2. Creates an INSERT statement to load this table from an external table description of the data.
3. Executes one INSERT statement for every table in the control file.

To see an example of this, run case study 5 (Case Study 5: Loading Data into Multiple Tables on page 10-18), but add the EXTERNAL_TABLE=GENERATE_ONLY parameter. To guarantee unique names in the external table, SQL*Loader uses generated names for all fields. This is because the field names may not be unique across the different tables in the control file.

See Also:

- Chapter 11, "External Tables Concepts"
- Chapter 12, "External Tables Access Parameters"

Restrictions When Using EXTERNAL_TABLE

The following restrictions apply when you use the EXTERNAL_TABLE qualifier:

- Julian dates cannot be used when you insert data into a database table from an external table through SQL*Loader. To work around this, use TO_DATE and TO_CHAR to convert the Julian date format, as shown in the following example:

  ```
  TO_CHAR(TO_DATE(:COL1, 'MM-DD-YYYY'), 'J')
  ```

- Built-in functions and SQL strings cannot be used for object elements when you insert data into a database table from an external table.

FILE (file to load into)

Default: none

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.

See Also: Parallel Data Loading Models on page 9-30

LOAD (records to load)

Default: All records are loaded.

LOAD specifies the maximum number of logical records to load (after skipping the specified number of records). No error occurs if fewer than the maximum number of records are found.

LOG (log file)

Default: The name of the control file, with an extension of .log.

LOG specifies the log file that SQL*Loader will create to store logging information about the loading process.

MULTITHREADING

Default: true on multiple-CPU systems, false on single-CPU systems

This parameter is available only for direct path loads.

By default, the multithreading option is always enabled (set to true) on multiple-CPU systems. In this case, the definition of a multiple-CPU system is a single system that has more than one CPU.

On single-CPU systems, multithreading is set to `false` by default. To use multithreading between two single-CPU systems, you must enable multithreading; it will not be on by default. This will allow stream building on the client system to be done in parallel with stream loading on the server system.

Multithreading functionality is operating system-dependent. Not all operating systems support multithreading.

See Also: Optimizing Direct Path Loads on Multiple-CPU Systems on page 9-23

PARALLEL (parallel load)

Default: `false`

`PARALLEL` specifies whether direct loads can operate in multiple concurrent sessions to load data into the same table.

See Also: Parallel Data Loading Models on page 9-30

PARFILE (parameter file)

Default: none

`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
```

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.

READSIZE (read buffer size)

Default: To see the default value for this parameter, invoke SQL*Loader without any parameters, as described in Invoking SQL*Loader on page 4-1.

The READSIZE parameter is used *only* when reading data from datafiles. When reading records from a control file, a value of 64K is *always* used as the READSIZE.

The READSIZE parameter lets you specify (in bytes) the size of the read buffer, if you choose not to use the default. The maximum size allowed is 20MB for both direct path loads and conventional path loads.

In the conventional path method, the bind array is limited by the size of the read buffer. Therefore, 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
```

This example enables SQL*Loader to perform reads from the external datafile in chunks of 1,000,000 bytes before a commit is required.

Note: If the READSIZE value specified is smaller than the BINDSIZE value, the READSIZE value will be increased.

The READSIZE parameter has no effect on LOBs. The size of the LOB read buffer is fixed at 64 KB.

See BINDSIZE (maximum size) on page 4-4.

RESUMABLE

Default: false

The RESUMABLE parameter is used to enable and disable resumable space allocation. Because this parameter is disabled by default, you must set RESUMABLE=true in order to use its associated parameters, RESUMABLE_NAME and RESUMABLE_TIMEOUT.

See Also:

- *Oracle9i Database Concepts*
- *Oracle9i Database Administrator's Guide*

RESUMABLE_NAME

Default: 'User USERNAME (USERID), Session SESSIONID, Instance INSTANCEID'

The value for this parameter identifies the statement that is resumable. This value is a user-defined text string that is inserted in either the USER_RESUMABLE or DBA_RESUMABLE view to help you identify a specific resumable statement that has been suspended.

This parameter is ignored unless the RESUMABLE parameter is set to true to enable resumable space allocation.

RESUMABLE_TIMEOUT

Default: 7200 seconds (2 hours)

The value of the parameter specifies the time period during which an error must be fixed. If the error is not fixed within the timeout period, execution of the statement is aborted.

This parameter is ignored unless the RESUMABLE parameter is set to true to enable resumable space allocation.

ROWS (rows per commit)

Default: To see the default value for this parameter, invoke SQL*Loader without any parameters, as described in Invoking SQL*Loader on page 4-1.

Conventional path loads only: ROWS specifies the number of rows in the bind array. See Bind Arrays and Conventional Path Loads on page 5-44.

Direct path loads only: ROWS identifies the number of rows you want to read from the datafile before a data save. The default is to read all rows and save data once at the end of the load. See Using Data Saves to Protect Against Data Loss on page 9-13.

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 similar to the following appears on the screen and is placed in the log file:

```
SQL*Loader: Release 9.2.0.1.0 - Production on Wed Feb 27 14:33:54 2002
```

```
(c) Copyright 2002 Oracle Corporation.  All rights reserved.
```

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 one or more values.

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 values 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` - Disables writing the per-partition statistics to the log file during a direct load of a partitioned table.
- `ALL` - Implements all of the suppression values: `HEADER`, `FEEDBACK`, `ERRORS`, `DISCARDS`, and `PARTITIONS`.

SKIP (records to skip)

Default: No records are skipped.

`SKIP` specifies the number of logical records from the beginning of the file that should not be loaded.

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 Also: Interrupted Loads on page 5-24

SKIP_INDEX_MAINTENANCE

Default: `false`

The `SKIP_INDEX_MAINTENANCE` parameter 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` parameter:

- Applies to both local and global indexes
- Can be used (with the `PARALLEL` parameter) to do parallel loads on an object that has indexes
- Can be used (with the `PARTITION` parameter 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

SKIP_UNUSABLE_INDEXES

Default: `false`

The `SKIP_UNUSABLE_INDEXES` parameter applies to both conventional and direct path loads.

`SKIP_UNUSABLE_INDEXES=true` 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.

STREAMSIZE

Default: To see the default value for this parameter, invoke SQL*Loader without any parameters, as described in Invoking SQL*Loader on page 4-1.

Specifies the size, in bytes, for direct path streams.

> **See Also:** Specifying the Number of Column Array Rows and Size of Stream Buffers on page 9-21

USERID (username/password)

Default: none

`USERID` is used to provide your Oracle *`username/password.`* If it is omitted, you are prompted for it. If only a slash is used, `USERID` defaults to your operating system login.

If you connect as user SYS, you must also specify AS SYSDBA in the connect string. For example:

```
sqlldr \'SYS/password AS SYSDBA\' foo.ctl
```

Note: This example shows the entire connect string enclosed in quotation marks and backslashes. This is because the string, `AS SYSDBA`, contains a blank, a situation for which most operating systems require that the entire connect string be placed in quotation marks or marked as a literal by some method. Some operating systems also require that quotation marks on the command line be preceded by an escape character. In this example, backslashes are used as the escape character. If the backslashes were not present, the command line parser that SQL*Loader uses would not understand the quotation marks and would remove them.

See your Oracle operating system-specific documentation for information about special and reserved characters on your system.

Exit Codes for Inspection and Display

Oracle SQL*Loader provides the results of a SQL*Loader run immediately upon completion. Depending on the platform, SQL*Loader may report the outcome in a process exit code as well as recording the results in the log file. This Oracle SQL*Loader functionality allows for checking the outcome of a SQL*Loader invocation from the command line or script. Table 4–1 shows the exit codes for various results.

Table 4–1 Exit Codes for SQL*Loader

Result	Exit Code
All rows loaded successfully	EX_SUCC
All or some rows rejected	EX_WARN
All or some rows discarded	EX_WARN
Discontinued load	EX_WARN
Command-line or syntax errors	EX_FAIL
Oracle errors nonrecoverable for SQL*Loader	EX_FAIL
Operating system errors (such as file open/close and malloc)	EX_FAIL

For UNIX, the exit codes are as follows:

```
EX_SUCC 0
EX_FAIL 1
```

```
EX_WARN 2
EX_FTL  3
```

For Windows NT, the exit codes are as follows:

```
EX_SUCC 0
EX_WARN 2
EX_FAIL 3
EX_FTL  4
```

If SQL*Loader returns any exit code other than zero, you should consult your system log files and SQL*Loader log files for more detailed diagnostic information.

In UNIX, 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 execution exited with EX_WARN, see logfile" ;;
3) echo "SQL*Loader execution encountered a fatal error" ;;
*) echo "unknown return code";;
esac
```

5

SQL*Loader Control File Reference

This chapter describes the SQL*Loader control file. The following topics are included:

Control File Contents

The SQL*Loader control file is a text file that contains data definition language (DDL) instructions. DDL is used to control the following aspects of a SQL*Loader session:

- 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, and so on) as it loads the data
- How SQL*Loader will manipulate the data being loaded

See Appendix A for syntax diagrams of the SQL*Loader DDL.

To create the SQL*Loader control file, use a text editor such as vi or xemacs.create.

In general, the control file has three main sections, in the following order:

- Session-wide information
- Table and field-list information
- Input data (optional section)

Example 5–1 shows a sample control file.

Example 5–1 Sample Control File

```
1    -- This is a sample control file
2    LOAD DATA
3    INFILE 'sample.dat'
4    BADFILE 'sample.bad'
5    DISCARDFILE 'sample.dsc'
6    APPEND
7    INTO TABLE emp
8    WHEN (57) = '.'
9    TRAILING NULLCOLS
10  (hiredate SYSDATE,
      deptno POSITION(1:2)  INTEGER EXTERNAL(2)
              NULLIF deptno=BLANKS,
       job    POSITION(7:14)  CHAR  TERMINATED BY WHITESPACE
              NULLIF job=BLANKS  "UPPER(:job)",
       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
              "TO_NUMBER(:sal,'$99,999.99')",
       comm   INTEGER EXTERNAL  ENCLOSED BY '(' AND '%'
              ":comm * 100"
    )
```

In this sample control file, the numbers that appear to the left would not appear in a real control file. They are keyed in this sample to the explanatory notes in the following list:

1. This is how comments are entered in a control file. See Comments in the Control File on page 5-4.
2. The LOAD DATA statement tells SQL*Loader that this is the beginning of a new data load. See Appendix A for syntax information.
3. The INFILE clause specifies the name of a datafile containing data that you want to load. See Specifying Datafiles on page 5-7.
4. The BADFILE parameter specifies the name of a file into which rejected records are placed. See Specifying the Bad File on page 5-11.
5. The DISCARDFILE parameter specifies the name of a file into which discarded records are placed. See Specifying the Discard File on page 5-14.
6. The APPEND parameter is one of the options you can use when loading data into a table that is not empty. See Loading Data into Nonempty Tables on page 5-33.

   To load data into a table that is empty, you would use the INSERT parameter. See Loading Data into Empty Tables on page 5-33.
7. The INTO TABLE clause allows you to identify tables, fields, and datatypes. It defines the relationship between records in the datafile and tables in the database. See Specifying Table Names on page 5-32.
8. The WHEN clause specifies one or more field conditions. SQL*Loader decides whether or not to load the data based on these field conditions. See Loading Records Based on a Condition on page 5-35.
9. The TRAILING NULLCOLS clause tells SQL*Loader to treat any relatively positioned columns that are not present in the record as null columns. See Handling Short Records with Missing Data on page 5-37.

10. The remainder of the control file contains the field list, which provides information about column formats in the table being loaded. See Chapter 6 for information about that section of 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. An example of comments in a control file is shown in Case Study 3: Loading a Delimited, Free-Format File on page 10-11.

Specifying Command-Line Parameters in the Control File

The OPTIONS clause is useful when you typically invoke a control file with the same set of options. The OPTIONS clause precedes the LOAD DATA statement.

OPTIONS Clause

The OPTIONS clause allows you to specify runtime parameters in the control file, rather than on the command line. The following parameters can be specified using the OPTIONS clause. These parameters are described in greater detail in Chapter 4.

```
BINDSIZE = n
COLUMNARRAYROWS = n
DIRECT = {TRUE | FALSE}
ERRORS = n
LOAD = n
MULTITHREADING = {TRUE | FALSE}
PARALLEL = {TRUE | FALSE}
READSIZE = n
RESUMABLE = {TRUE | FALSE}
RESUMABLE_NAME = 'text string'
RESUMABLE_TIMEOUT = n
ROWS = n
SILENT = {HEADERS | FEEDBACK | ERRORS | DISCARDS | PARTITIONS | ALL}
SKIP = n
SKIP_INDEX_MAINTENANCE = {TRUE | FALSE}
SKIP_UNUSABLE_INDEXES = {TRUE | FALSE}
STREAMSIZE = n
```

For example:

```
OPTIONS (BINDSIZE=100000, SILENT=(ERRORS, FEEDBACK) )
```

> **Note:** Values specified on the command line override values specified in the `OPTIONS` clause in the control file.

Specifying Filenames and Object Names

In general, SQL*Loader follows the SQL standard for specifying object names (for example, table and column names). The information in this section discusses the following topics:

- Filenames That Conflict with SQL and SQL*Loader Reserved Words
- Specifying SQL Strings
- Operating System Considerations

Filenames That Conflict with SQL and SQL*Loader Reserved Words

SQL and SQL*Loader reserved words must be specified within double quotation marks. The only SQL*Loader reserved word is `CONSTANT.`

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.

> **See Also:** *Oracle9i SQL Reference*

Specifying SQL Strings

You must specify SQL strings within double quotation marks. The SQL string applies SQL operators to data fields.

> **See Also:** Applying SQL Operators to Fields on page 6-50

Operating System Considerations

The following sections discuss situations in which your course of action may depend on the operating system you are using.

Specifying a Complete Path

If you encounter problems when trying to specify a complete path name, it may be due to an operating system-specific incompatibility caused by special characters in the specification. In many cases, specifying the path name within single quotation marks prevents errors.

If not, please see your Oracle operating system-specific documentation for possible solutions.

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 character 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\mydata` contains a double quotation mark. Preceding the double quotation mark with a backslash indicates that the double quotation mark 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 quotation mark in the initial position cannot be preceded by an escape character. Therefore, you should avoid creating strings with an initial quotation mark.

Nonportable 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 you convert to a different operating system, you will probably need to modify these strings. 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 path name, *and* if the version of the Oracle database server running on your operating system implements the backslash escape character for filenames and other nonportable strings, then you must specify double backslashes in your path names and use single quotation marks.

See your Oracle operating system-specific documentation for information about which escape characters are required or allowed.

Escape Character Is Sometimes Disallowed

The version of the Oracle database server running on your operating system may not implement the escape character for nonportable 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 path names such as the following can be specified normally:

```
INFILE 'topdir\mydir\myfile'
```

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.

Specifying Datafiles

To specify a datafile that contains the data to be loaded, use the `INFILE` clause, followed by the filename and optional file processing options string. You can specify multiple files by using multiple `INFILE` clauses.

Note: You can also specify the datafile from the command line, using the `DATA` parameter described in Command-Line Parameters on page 4-3. A filename specified on the command line overrides the first `INFILE` clause 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 5-10.

> **Note:** The information in this section applies only to primary datafiles. It does not apply to LOBFILEs or SDFs.
>
> For information about LOBFILES, see Loading LOB Data from LOBFILEs on page 7-23.
>
> For information about SDFs, see Secondary Datafiles (SDFs) on page 7-31.

The syntax for the `INFILE` clause is as follows:

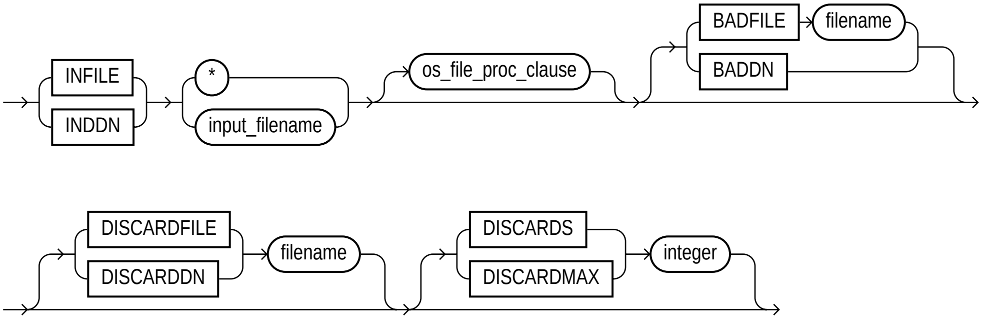


Table 5–1 describes the parameters for the `INFILE` clause.

Table 5–1 Parameters for the INFILE Clause

Parameter	Description
`INFILE` or `INDDN`	Specifies that a datafile specification follows. Note that `INDDN` has been retained for situations in which compatibility with DB2 is required.
`input_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 Object Names on page 5-5.

Table 5–1 (Cont.) Parameters for the INFILE Clause

Parameter	Description
*	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.
os_file_proc_clause	This is the file-processing options string. It specifies the datafile format. It also optimizes datafile reads. The syntax used for this string is specific to your operating system. See Specifying Datafile Format and Buffering on page 5-11.

Examples of INFILE Syntax

The following list shows different ways you can specify `INFILE` syntax:

- Data contained in the control file itself:

```
INFILE  *
```

- Data contained in a file named `foo` with a default extension of `.dat`:

```
INFILE  foo
```

- Data contained in a file named `datafile.dat` with a 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 Object Names on page 5-5.

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. In such a case, the separate bad files and discard files must be declared immediately 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 the default filename and extension mydat2.bad. 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.

Identifying Data in the Control File with BEGINDATA

If the data is included in the control file itself, then the INFILE clause is followed by an asterisk rather than a filename. The actual data is placed in the control file after the load configuration specifications.

Specify the BEGINDATA parameter before the first data record. The syntax is:

```
BEGINDATA
data
```

Keep the following points in mind when using the BEGINDATA parameter:

- If you omit the BEGINDATA parameter 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 parameter.
- Do not use spaces or other characters on the same line as the BEGINDATA parameter, or the line containing BEGINDATA will be interpreted as the first line of data.

- Do not put comments after BEGINDATA, or they will also be interpreted as data.

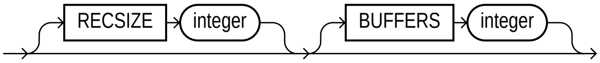
See Also:

- Specifying Datafiles on page 5-7 for an explanation of using INFILE
- Case Study 1: Loading Variable-Length Data on page 10-5

Specifying Datafile Format and Buffering

When configuring SQL*Loader, you can specify an operating system-dependent file processing options string (*os_file_proc_clause*) in the control file to specify file format and buffering.

For example, suppose that your operating system has the following option-string syntax:



In this syntax, 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 8 I/O buffers, you would use the following control file entry:

```
INFILE 'mydata.dat' "RECSIZE 80 BUFFERS 8"
```

For details on the syntax of the file processing options string, see your Oracle operating system-specific documentation.

Note: This example uses the recommended convention of single quotation marks for filenames and double quotation marks for everything else.

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 created and the rejected records are logged.
- If no records are rejected, then the bad file is not created. When this occurs, you must reinitialize the bad file for the next run.
- If the bad file is created, it overwrites any existing file with the same name; ensure that you do not overwrite a file you wish to retain.

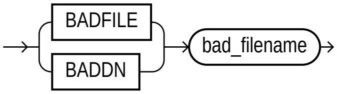
Note: On some systems, a new version of the file is created if a file with the same name already exists. See your Oracle operating system-specific documentation to find out if this is the case on your system.

To specify the name of the bad file, use the BADFILE parameter (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 Parameters on page 4-3.

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 making corrections. For datafiles in stream record format, the record terminator that is found in the datafile is also used in the bad file.

The syntax for the bad file is as follows:



The BADFILE or BADDN parameter specifies that a filename for the bad file follows. (Use BADDN when DB2 compatibility is required.)

The *bad_filename* parameter specifies a valid filename specification for your platform. Any spaces or punctuation marks in the filename must be enclosed in single quotation marks.

Examples of Specifying a Bad File Name

To specify a bad file with filename `foo` and default file extension or file type of `.bad,` enter:

```
BADFILE foo
```

To specify a bad file with filename `bad0001` and file extension or file type of `.rej,` enter either of the following lines:

```
BADFILE bad0001.rej
BADFILE '/REJECT_DIR/bad0001.rej'
```

How Bad Files Are Handled with LOBFILEs and SDFs

Data from LOBFILEs and SDFs is 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` column is left empty (not null with a length of zero (0) bytes). However, when the LOBFILE is being used to load an `XML` column and there is an error loading this LOB data, then the `XML` column is left as null.

Criteria for Rejected Records

A record can be rejected for the following reasons:

1. Upon insertion, the record causes an Oracle error (such as invalid data for a given datatype).
2. The record is formatted incorrectly so that SQL*Loader cannot find field boundaries.
3. The record violates a constraint or tries to make a unique index non-unique.

If the data can be evaluated according to the `WHEN` clause criteria (even with unbalanced delimiters), then it is either inserted or rejected.

Neither a conventional path nor a direct path load will write a row to any table if it is rejected because of reason number 2 in the previous list.

Additionally, a conventional path load will not write a row to any tables if reason number 1 or 3 in the previous list is violated for any one table. The row is rejected for that table and written to the reject file.

The log file indicates the Oracle error for each rejected record. Case Study 4: Loading Combined Physical Records on page 10-14 demonstrates rejected records.

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. (If the discard file is created, it overwrites any existing file with the same name, so be sure 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 from within a control file, specify any of the following: DISCARDFILE *filename*, DISCARDDN *filename* (DB2), DISCARDS, or DISCARDMAX.

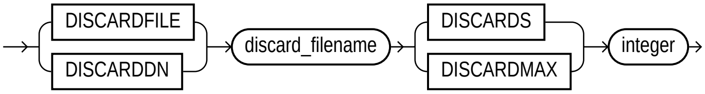
To create a discard file from the command line, specify either DISCARD or DISCARDMAX.

You can specify the discard file directly by specifying its name, or indirectly by specifying the maximum number of discards.

The discard file is created in the same record and file format as the datafile. For datafiles in stream record format, the same record terminator that is found in the datafile is also used in the discard file.

Specifying the Discard File in the Control File

To specify the name of the file, use the DISCARDFILE or DISCARDDN (for DB2-compatibility) parameter, followed by the filename.



The DISCARDFILE or DISCARDDN parameter specifies that a discard filename follows. (Use DISCARDDN when DB2 compatibility is required.)

The *discard_filename* parameter specifies a valid filename specification for your platform. Any spaces or punctuation marks in the filename must be enclosed in single quotation marks.

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.

Specifying the Discard File from the Command Line

See DISCARD (filename) on page 4-6 for information on how to specify a discard file from the command line.

A filename specified on the command line overrides any discard file that you may have specified in the control file.

Examples of Specifying a Discard File Name

The following list shows different ways you can specify a name for the discard file from within the control file:

- To specify a discard file with filename circular and default file extension or file type of .dsc:

  ```
  DISCARDFILE  circular
  ```

- To specify a discard file named notappl with the file extension or file type of .may:

  ```
  DISCARDFILE notappl.may
  ```

- To specify a full path to the discard file forget.me:

  ```
  DISCARDFILE  '/discard_dir/forget.me'
  ```

Criteria for Discarded Records

If there is no INTO TABLE clause specified for a record, the record is discarded. This situation occurs when every INTO TABLE clause 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` clause is specified without a `WHEN` clause. An attempt is made to insert every record into such a table. Therefore, records may be rejected, but none are discarded.

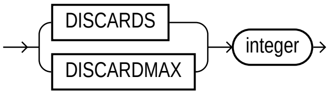
Case Study 7: Extracting Data from a Formatted Report on page 10-28 provides an example of using a discard file.

How Discard Files Are Handled with LOBFILEs and SDFs

Data from LOBFILEs and SDFs is not written to a discard file when there are discarded rows.

Limiting the Number of Discarded Records

You can limit the number of records to be discarded for each datafile by specifying an integer:



When the discard limit (specified with *`integer`*) 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. Or, if you specify the number of discards only 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.

Handling Different Character Encoding Schemes

SQL*Loader supports different character encoding schemes (called character sets, or code pages). SQL*Loader uses features of Oracle's globalization support technology to handle the various single-byte and multibyte character encoding schemes available today.

See Also: *Oracle9i Database Globalization Support Guide*

In general, loading shift-sensitive character data can be much slower than loading simple ASCII or EBCDIC data. The fastest way to load shift-sensitive character data is to use fixed-position fields without delimiters. To improve performance, remember the following points:

- The field data must have an equal number of shift-out/shift-in bytes.
- The field must start and end in single-byte mode.
- It is acceptable for the first byte to be shift-out and the last byte to be shift-in.
- The first and last characters cannot be multibyte.
- If blanks are not preserved and multibyte-blank-checking is required, a slower path is used. This can happen when the shift-in byte is the last byte of a field after single-byte blank stripping is performed.

The following sections provide a brief introduction to some of the supported character encoding schemes.

Multibyte (Asian) Character Sets

Multibyte character sets support Asian languages. Data can be loaded in multibyte format, and database object names (fields, tables, and so on) can be specified with multibyte characters. In the control file, comments and object names can also use multibyte characters.

Unicode Character Sets

SQL*Loader supports loading data that is in a Unicode character set.

Unicode is a universal encoded character set that supports storage of information from most languages in a single character set. Unicode provides a unique code value for every character, regardless of the platform, program, or language. There are two different encodings for Unicode, UTF-16 and UTF-8.

> **Note:** In this manual, you will see the terms UTF-16 and UTF16 both used. The term UTF-16 is a general reference to UTF-16 encoding for Unicode. The term UTF16 (no hyphen) is the specific name of the character set and is what you should specify for the CHARACTERSET parameter when you want to use UTF-16 encoding. This also applies to UTF-8 and UTF8.

The UTF-16 Unicode encoding is a fixed-width multibyte encoding in which the character codes 0x0000 through 0x007F have the same meaning as the single-byte ASCII codes 0x00 through 0x7F.

The UTF-8 Unicode encoding is a variable-width multibyte encoding in which the character codes 0x00 through 0x7F have the same meaning as ASCII. A character in UTF-8 can be 1 byte, 2 bytes, or 3 bytes long.

See Also:

- Case Study 11: Loading Data in the Unicode Character Set on page 10-47
- *Oracle9i Database Globalization Support Guide* for more information on Unicode encoding.

Database Character Sets

The Oracle database server uses the database character set for data stored in SQL CHAR datatypes (CHAR, VARCHAR2, CLOB, and LONG), for identifiers such as table names, and for SQL statements and PL/SQL source code. Only single-byte character sets and varying-width character sets that include either ASCII or EBCDIC characters are supported as database character sets. Multibyte fixed-width character sets (for example, AL16UTF16) are not supported as the database character set.

An alternative character set can be used in the database for data stored in SQL NCHAR datatypes (NCHAR, NVARCHAR, and NCLOB). This alternative character set is called the database national character set. Only Unicode character sets are supported as the database national character set.

Datafile Character Sets

By default, the datafile is in the character set as defined by the NLS_LANG parameter. The datafile character sets supported with NLS_LANG are the same as those supported as database character sets. SQL*Loader supports all Oracle-supported character sets in the datafile (even those not supported as database character sets).

For example, SQL*Loader supports multibyte fixed-width character sets (such as AL16UTF16 and JA16EUCFIXED) in the datafile. SQL*Loader also supports UTF-16 encoding with little endian byte ordering. However, the Oracle database server supports only UTF-16 encoding with big endian byte ordering (AL16UTF16) and only as a database national character set, not as a database character set.

The character set of the datafile can be set up by using the NLS_LANG parameter or by specifying a SQL*Loader CHARACTERSET parameter.

Input Character Conversion

The default character set for all datafiles, if the CHARACTERSET parameter is not specified, is the session character set defined by the NLS_LANG parameter. The character set used in input datafiles can be specified with the CHARACTERSET parameter.

SQL*Loader has the capacity to automatically convert data from the datafile character set to the database character set or the database national character set, when they differ.

When data character set conversion is required, the target character set should be a superset of the source datafile character set. Otherwise, characters that have no equivalent in the target character set are converted to replacement characters, often a default character such as a question mark (?). This causes loss of data.

The sizes of the database character types CHAR and VARCHAR2 can be specified in bytes (byte-length semantics) or in characters (character-length semantics). If they are specified in bytes, and data character set conversion is required, the converted values may take more bytes than the source values if the target character set uses more bytes than the source character set for any character that is converted. This will result in the following error message being reported if the larger target value exceeds the size of the database column:

```
ORA-01401: inserted value too large for column
```

You can avoid this problem by specifying the database column size in characters and by also using character sizes in the control file to describe the data. Another way to avoid this problem is to ensure that the maximum column size is large enough, in bytes, to hold the converted value.

See Also:

- *Oracle9i Database Concepts* for more information about character-length semantics in the database.
- Character-Length Semantics on page 5-22

CHARACTERSET Parameter

Specifying the CHARACTERSET parameter tells SQL*Loader the character set of the input datafile. The default character set for all datafiles, if the CHARACTERSET parameter is not specified, is the session character set defined by the NLS_LANG parameter. Only character data (fields in the SQL*Loader datatypes CHAR, VARCHAR, VARCHARC, numeric EXTERNAL, and the datetime and interval datatypes) is affected by the character set of the datafile.

The CHARACTERSET syntax is as follows:

```
CHARACTERSET char_set_name
```

The *char_set_name* variable specifies the character set name. Normally, the specified name must be the name of an Oracle-supported character set.

For UTF-16 Unicode encoding, use the name UTF16 rather than AL16UTF16. AL16UTF16, which is the supported Oracle character set name for UTF-16 encoded data, is only for UTF-16 data that is in big endian byte order. However, because you are allowed to set up data using the byte order of the system where you create the datafile, the data in the datafile can be either big endian or little endian. Therefore, a different character set name (UTF16) is used. The character set name AL16UTF16 is also supported. But if you specify AL16UTF16 for a datafile that has little endian byte order, SQL*Loader issues a warning message and processes the datafile as big endian.

The CHARACTERSET parameter can be specified for primary datafiles as well as LOBFILEs and SDFs. It is possible to specify different character sets for different input datafiles. A CHARACTERSET parameter specified before the INFILE parameter applies to the entire list of primary datafiles. If the CHARACTERSET parameter is specified for primary datafiles, the specified value will also be used as the default for LOBFILEs and SDFs. This default setting can be overridden by specifying the CHARACTERSET parameter with the LOBFILE or SDF specification.

The character set specified with the CHARACTERSET parameter does not apply to data in the control file (specified with INFILE). To load data in a character set other than the one specified for your session by the NLS_LANG parameter, you must place the data in a separate datafile.

See Also:

- Byte Ordering on page 6-37
- *Oracle9i Database Globalization Support Guide* for more information on the names of the supported character sets
- Control File Character Set on page 5-21
- Case Study 11: Loading Data in the Unicode Character Set on page 10-47 for an example of loading a datafile that contains little endian UTF-16 encoded data

Control File Character Set

The SQL*Loader control file itself is assumed to be in the character set specified for your session by the `NLS_LANG` parameter. If the control file character set is different from the datafile character set, keep the following issue in mind. Delimiters and comparison clause values specified in the SQL*Loader control file as character strings are converted from the control file character set to the datafile character set before any comparisons are made. To ensure that the specifications are correct, you may prefer to specify hexadecimal strings, rather than character string values.

If hexadecimal strings are used with a datafile in the UTF-16 Unicode encoding, the byte order is different on a big endian versus a little endian system. For example, "," (comma) in UTF-16 on a big endian system is X’002c’. On a little endian system it is X’2c00’. SQL*Loader requires that you always specify hexadecimal strings in big endian format. If necessary, SQL*Loader swaps the bytes before making comparisons. This allows the same syntax to be used in the control file on both a big endian and a little endian system.

Record terminators for datafiles that are in stream format in the UTF-16 Unicode encoding default to "\n" in UTF-16 (that is, 0x000A on a big endian system and 0x0A00 on a little endian system). You can override these default settings by using the `"STR '`*`char_str`*`'"` or the `"STR x'`*`hex_str`*`'"` specification on the `INFILE` line. For example, you could use either of the following to specify that `'ab'` is to be used as the record terminator, instead of `'\`*`n`*`'`.

```
INFILE myfile.dat "STR 'ab'"

INFILE myfile.dat "STR x'00410042'"
```

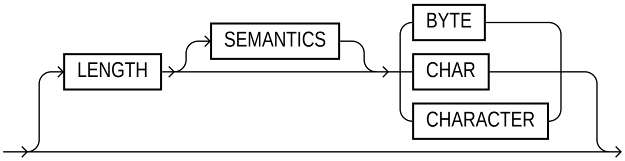
Any data included after the `BEGINDATA` statement is also assumed to be in the character set specified for your session by the `NLS_LANG` parameter.

For the SQL*Loader datatypes (CHAR, VARCHAR, VARCHARC, DATE, and EXTERNAL numerics), SQL*Loader supports lengths of character fields that are specified in either bytes (byte-length semantics) or characters (character-length semantics). For example, the specification CHAR(10) in the control file can mean 10 bytes or 10 characters. These are equivalent if the datafile uses a single-byte character set. However, they are often different if the datafile uses a multibyte character set.

To avoid insertion errors caused by expansion of character strings during character set conversion, use character-length semantics in both the datafile and the target database columns.

Character-Length Semantics

Byte-length semantics are the default for all datafiles except those that use the UTF16 character set (which uses character-length semantics by default). To override the default you can specify CHAR or CHARACTER, as shown in the following syntax:



The LENGTH parameter is placed after the CHARACTERSET parameter in the SQL*Loader control file. The LENGTH parameter applies to the syntax specification for primary datafiles as well as to LOBFILEs and secondary datafiles (SDFs). It is possible to specify different length semantics for different input datafiles. However, a LENGTH specification before the INFILE parameters applies to the entire list of primary datafiles. The LENGTH specification specified for the primary datafile is used as the default for LOBFILEs and SDFs. You can override that default by specifying LENGTH with the LOBFILE or SDF specification. Unlike the CHARACTERSET parameter, the LENGTH parameter can also apply to data contained within the control file itself (that is, INFILE * syntax).

You can specify CHARACTER instead of CHAR for the LENGTH parameter.

If character-length semantics are being used for a SQL*Loader datafile, then the following SQL*Loader datatypes will use character-length semantics:

- CHAR
- VARCHAR

- VARCHARC
- DATE
- EXTERNAL numerics (INTEGER, FLOAT, DECIMAL, and ZONED)

For the VARCHAR datatype, the length subfield is still a binary SMALLINT length subfield, but its value indicates the length of the character string in characters.

The following datatypes use byte-length semantics even if character-length semantics are being used for the datafile, because the data is binary, or is in a special binary-encoded form in the case of ZONED and DECIMAL:

- INTEGER
- SMALLINT
- FLOAT
- DOUBLE
- BYTEINT
- ZONED
- DECIMAL
- RAW
- VARRAW
- VARRAWC
- GRAPHIC
- GRAPHIC EXTERNAL
- VARGRAPHIC

The start and end arguments to the POSITION parameter are interpreted in bytes, even if character-length semantics are in use in a datafile. This is necessary to handle datafiles that have a mix of data of different datatypes, some of which use character-length semantics, and some of which use byte-length semantics. It is also needed to handle position with the VARCHAR datatype, which has a SMALLINT length field and then the character data. The SMALLINT length field takes up a certain number of bytes depending on the system (usually 2 bytes), but its value indicates the length of the character string in characters.

Character-length semantics in the datafile can be used independent of whether or not character-length semantics are used for the database columns. Therefore, the datafile and the database columns can use either the same or different length semantics.

Interrupted Loads

Loads are interrupted and discontinued for a number of reasons. A primary reason is space errors, in which SQL*Loader runs out of space for data rows or index entries. A load might also be discontinued because the maximum number of errors was exceeded, an unexpected error was returned to SQL*Loader from the server, a record was too long in the datafile, or a Control+C was executed.

The behavior of SQL*Loader when a load is discontinued varies depending on whether it is a conventional path load or a direct path load, and on the reason the load was interrupted. Additionally, when an interrupted load is continued, the use and value of the `SKIP` parameter can vary depending on the particular case. The following sections explain the possible scenarios.

> **See Also:** SKIP (records to skip) on page 4-13

Discontinued Conventional Path Loads

In a conventional path load, data is committed after all data in the bind array is loaded into all tables. If the load is discontinued, only the rows that were processed up to the time of the last commit operation are loaded. There is no partial commit of data.

Discontinued Direct Path Loads

In a direct path load, the behavior of a discontinued load varies depending on the reason the load was discontinued.

Load Discontinued Because of Space Errors

If there is one `INTO TABLE` statement in the control file and a space error occurs, the following scenarios can take place:

- If you are loading data into an unpartitioned table, one partition of a partitioned table, or one subpartition of a composite partitioned table, then SQL*Loader commits as many rows as were loaded before the error occurred. This is independent of whether the `ROWS` parameter was specified.
- If you are loading data into multiple subpartitions (that is, loading into a partitioned table, a composite partitioned table, or one partition of a composite

partitioned table), the load is discontinued and no data is saved unless ROWS has been specified. In that case, all data that was previously committed will be saved.

If there are multiple INTO TABLE statements in the control file and a space error occurs on one of those tables, the following scenarios can take place:

- If the space error occurs when you are loading data into an unpartitioned table, one partition of a partitioned table, or one subpartition of a composite partitioned table, SQL*Loader attempts to load data already read from the datafile into other tables. SQL*Loader then commits as many rows as were loaded before the error occurred. This is independent of whether the ROWS parameter was specified. In this scenario, a different number of rows could be loaded into each table; to continue the load you would need to specify a different value for the SKIP parameter for every table. SQL*Loader only reports the value for the SKIP parameter if it is the same for all tables.
- If the space error occurs when you are loading data into multiple subpartitions (that is, loading into a partitioned table, a composite partitioned table, or one partition of a composite partitioned table), the load is discontinued for all tables and no data is saved unless ROWS has been specified. In that case, all data that was previously committed is saved, and when you continue the load the value you supply for the SKIP parameter will be the same for all tables.

Load Discontinued Because Maximum Number of Errors Exceeded

If the maximum number of errors is exceeded, SQL*Loader stops loading records into any table and the work done to that point is committed. This means that when you continue the load, the value you specify for the SKIP parameter may be different for different tables. SQL*Loader only reports the value for the SKIP parameter if it is the same for all tables.

Load Discontinued Because of Fatal Errors

If a fatal error is encountered, the load is stopped and no data is saved unless ROWS was specified at the beginning of the load. In that case, all data that was previously committed is saved. SQL*Loader only reports the value for the SKIP parameter if it is the same for all tables.

Load Discontinued Because a Control+C Was Issued

If SQL*Loader is in the middle of saving data when a Control+C is issued, it continues to do the save and then stops the load after the save completes. Otherwise, SQL*Loader stops the load without committing any work that was not committed already. This means that the value of the SKIP parameter will be the same for all tables.

Status of Tables and Indexes After an Interrupted Load

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 an unusable state. You must drop these indexes before the load can continue. You can re-create the indexes either before continuing or after the load completes.

Other indexes are valid if no other errors occurred. See Indexes Left in an Unusable State on page 9-12 for other reasons why an index might be left in an unusable state.

Using the Log File to Determine Load Status

The SQL*Loader 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.

Continuing Single-Table Loads

When SQL*Loader must discontinue a direct path or conventional path load before it is finished, some rows have probably already been committed or marked with savepoints. To continue the discontinued load, use the SKIP parameter to specify the number of logical records that have already been processed by the previous load. At the time the load is discontinued, the value for SKIP is written to the log file in a message similar to the following:

```
Specify SKIP=1001 when continuing the load.
```

This message specifying the value of the SKIP parameter is preceded by a message indicating why the load was discontinued.

Note that for multiple-table loads, the value of the SKIP parameter is displayed only if it is the same for all tables.

See Also: SKIP (records to skip) on page 4-13

Assembling Logical Records from Physical Records

Because Oracle9*i* supports user-defined record sizes larger than 64 KB (see READSIZE (read buffer size) on page 4-10), the need to break up logical records into multiple physical records is reduced. However, there may still be situations in which you may want to do so. At some point, when you want to combine those multiple physical records back into one logical record, you can use one of the following clauses, depending on your data:

- CONCATENATE
- CONTINUEIF

Using CONCATENATE to Assemble Logical Records

Use CONCATENATE when you want SQL*Loader to always combine the same number of physical records to form one logical record. In the following example, *integer* specifies the number of physical records to combine.

```
CONCATENATE  integer
```

The integer value specified for CONCATENATE determines the number of physical record structures that SQL*Loader allocates for each row in the column array. Because the default value for COLUMNARRAYROWS is large, if you also specify a large value for CONCATENATE, then excessive memory allocation can occur. If this happens, you can improve performance by reducing the value of the COLUMNARRAYROWS parameter to lower the number of rows in a column array.

See Also:

- COLUMNARRAYROWS on page 4-4
- Specifying the Number of Column Array Rows and Size of Stream Buffers on page 9-21

Using CONTINUEIF to Assemble Logical Records

Use CONTINUEIF if the number of physical records to be continued varies. The parameter 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 a pound sign (#) were in byte 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:

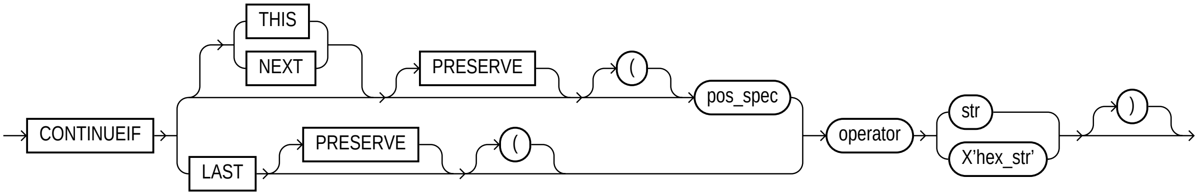


Table 5–2 describes the parameters for CONTINUEIF.

Table 5–2 Parameters for CONTINUEIF

Parameter	Description
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 logical record, continuing until the condition is false.
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.
LAST	This test is similar to THIS, but the test is always against the last nonblank character. If the last nonblank 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.

Table 5–2 (Cont.) Parameters for CONTINUEIF

Parameter	Description
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 zeros.
str	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-str'	A string of bytes in hexadecimal format used in the same way as str. X'1FB033 would represent the three bytes with values 1F, B0, and 33 (hexadecimal).
PRESERVE	Includes 'char_string' or X'hex_string' in the logical record. The default is to exclude them.

Note: The positions in the CONTINUEIF clause refer to positions in each physical record. This is the only time you refer to positions in physical records. All other references are to logical records.

If the PRESERVE parameter is not used, the continuation field is removed from all physical records when the logical record is assembled. That is, data values are allowed to span the records with no extra characters (continuation characters) in the middle.

If the PRESERVE parameter is used, the continuation field is kept in all physical records when the logical record is assembled.

Example 5–2 through Example 5–5 show the use of CONTINUEIF THIS and CONTINUEIF NEXT, with and without the PRESERVE parameter.

Example 5–2 CONTINUEIF THIS Without the PRESERVE Parameter

Assume that you have physical records 14 bytes long and that a period represents a space:

```
%%aaaaaaaa....
%%bbbbbbbb....
..cccccccc....
%%dddddddddd..
%%eeeeeeeeee..
..ffffffffff..
```

In this example, the `CONTINUEIF THIS` clause does not use the `PRESERVE` parameter:

```
CONTINUEIF THIS (1:2) = '%%'
```

Therefore, the logical records are assembled as follows:

```
aaaaaaaa....bbbbbbbb....cccccccc....
dddddddddd..eeeeeeeeee..ffffffffff..
```

Note that columns 1 and 2 (for example, %% in physical record 1) are removed from the physical records when the logical records are assembled.

Example 5–3 CONTINUEIF THIS with the PRESERVE Parameter

Assume that you have the same physical records as in Example 5–2.

In this example, the `CONTINUEIF THIS` clause uses the `PRESERVE` parameter:

```
CONTINUEIF THIS PRESERVE (1:2) = '%%'
```

Therefore, the logical records are assembled as follows:

```
%%aaaaaaaa....%%bbbbbbbb......cccccccc....
%%dddddddddd..%%eeeeeeeeee....ffffffffff..
```

Note that columns 1 and 2 are not removed from the physical records when the logical records are assembled.

Example 5–4 CONTINUEIF NEXT Without the PRESERVE Parameter

Assume that you have physical records 14 bytes long and that a period represents a space:

```
..aaaaaaaa....
%%bbbbbbbb....
```

```
%%cccccccc....
..dddddddddd..
%%eeeeeeeeee..
%%ffffffffff..
```

In this example, the CONTINUEIF NEXT clause does not use the PRESERVE parameter:

```
CONTINUEIF NEXT (1:2) = '%%'
```

Therefore, the logical records are assembled as follows (the same results as for Example 5–2).

```
aaaaaaaa....bbbbbbbb....cccccccc....
dddddddddd..eeeeeeeeee..ffffffffff..
```

Example 5–5 CONTINUEIF NEXT with the PRESERVE Parameter

Assume that you have the same physical records as in Example 5–4.

In this example, the CONTINUEIF NEXT clause uses the PRESERVE parameter:

```
CONTINUEIF NEXT PRESERVE (1:2) = '%%'
```

Therefore, the logical records are assembled as follows:

```
..aaaaaaaa....%%bbbbbbbb....%%cccccccc....
..dddddddddd..%%eeeeeeeeee..%%ffffffffff..
```

See Also: Case Study 4: Loading Combined Physical Records on page 10-14 for 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 data delimiters for those records
- How to handle short records with missing data

Specifying Table Names

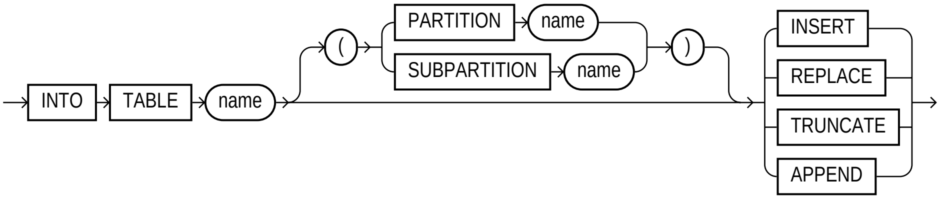
The INTO TABLE clause of the LOAD DATA statement allows you to identify tables, fields, and datatypes. It defines the relationship between records in the datafile and tables in the database. The specification of fields and datatypes is described in later sections.

INTO TABLE Clause

Among its many functions, the INTO TABLE clause 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 syntax is as follows:



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 reserved keyword, if it contains any special characters, or if it is case sensitive.

```
INTO TABLE scott."CONSTANT"
INTO TABLE scott."Constant"
INTO TABLE scott."-CONSTANT"
```

The user must have INSERT privileges for the table being loaded. If the table is not in the user's schema, then the user must either use a synonym to reference the table or include the schema name as part of the table name (for example, scott.emp).

Table-Specific Loading Method

When you are loading a table, you can use the INTO TABLE clause to specify a table-specific loading method (INSERT, APPEND, REPLACE, or TRUNCATE) that applies only to that table. That method 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. The following sections discuss using these options to load data into empty and nonempty tables.

Loading Data into Empty Tables

If the tables you are loading into are empty, use the INSERT option.

INSERT This 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 Study 1: Loading Variable-Length Data on page 10-5 provides an example.

Loading Data into Nonempty Tables

If the tables you are loading into already contain data, you have three options:

- APPEND
- REPLACE
- TRUNCATE

> **Caution:** When REPLACE or TRUNCATE 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.

APPEND If data already exists in the table, SQL*Loader appends the new rows to it. If data does not already exist, the new rows are simply loaded. You must have SELECT privilege to use the APPEND option. Case Study 3: Loading a Delimited, Free-Format File on page 10-11 provides an example.

REPLACE With 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 Study 4: Loading Combined Physical Records on page 10-14 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. For more information on cascaded deletes, see the information about data integrity in *Oracle9i Database 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 *Oracle9i SQL Reference*.

TRUNCATE The SQL TRUNCATE statement quickly and efficiently deletes all rows from a table or cluster, to achieve the best possible performance. For the TRUNCATE statement 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.

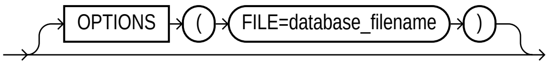
The table must be in your schema, or you must have the DROP ANY TABLE privilege.

> **See Also:** *Oracle9i Database Administrator's Guide* for more information about the TRUNCATE statement

Table-Specific OPTIONS Parameter

The OPTIONS parameter can be specified for individual tables in a parallel load. (It is only valid for a parallel load.)

The syntax for the OPTIONS parameter is as follows:

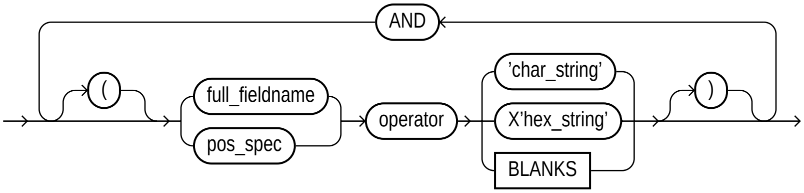


> **See Also:** Parameters for Parallel Direct Path Loads on page 9-33

Loading Records Based on a Condition

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. The syntax for `field_condition` is as follows:



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')
```

See Also:

- Using the WHEN, NULLIF, and DEFAULTIF Clauses on page 6-32 for information about how SQL*Loader evaluates `WHEN` clauses, as opposed to `NULLIF` and `DEFAULTIF` clauses
- Case Study 5: Loading Data into Multiple Tables on page 10-18 provides an example of the `WHEN` clause

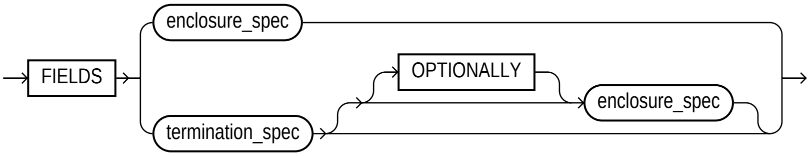
Using the WHEN Clause with LOBFILEs and SDFs

If a record with a LOBFILE or SDF is discarded, SQL*Loader skips the corresponding data in that LOBFILE or SDF.

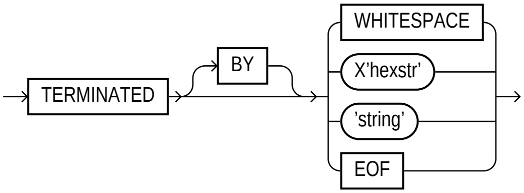
Specifying Default Data Delimiters

If all data fields are terminated similarly in the datafile, you can use the `FIELDS` clause to indicate the default delimiters. The syntax for the `fields_spec`, `termination_spec`, and `enclosure_spec` clauses is as follows:

fields_spec



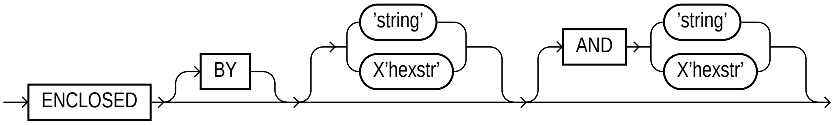
termination_spec



Note: Terminator strings can contain one or more characters. Also, `TERMINATED BY EOF` applies only to loading LOBs from LOBFILE.

---

enclosure_spec



Note: Enclosure strings can contain one or more characters.

You can override the delimiter for any given column by specifying it after the column name. Case Study 3: Loading a Delimited, Free-Format File on page 10-11 provides an example.

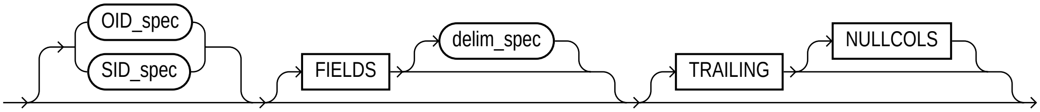
See Also:

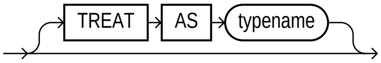
- Specifying Delimiters on page 6-24 for a complete description of the syntax
- Loading LOB Data from LOBFILEs on page 7-23

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 following example), 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 (shown in the following syntax diagram) to determine the course of action.





TRAILING NULLCOLS Clause

The `TRAILING NULLCOLS` clause tells SQL*Loader to treat any relatively positioned columns that are not present in the record as null columns.

For example, consider the following data:

```
10 Accounting
```

Assume that the preceding data is read with the following control file and the record ends after `dname`:

```
INTO TABLE dept
    TRAILING NULLCOLS
( deptno CHAR TERMINATED BY " ",
  dname  CHAR TERMINATED BY WHITESPACE,
  loc    CHAR TERMINATED BY WHITESPACE
)
```

In this case, the remaining `loc` field is set to null. Without the `TRAILING NULLCOLS` clause, an error would be generated due to missing data.

> **See Also:** Case Study 7: Extracting Data from a Formatted Report on page 10-28 for an example of `TRAILING NULLCOLS`

Index Options

This section describes the following SQL*Loader options that control how index entries are created:

- `SORTED INDEXES`
- `SINGLEROW`

SORTED INDEXES Clause

The `SORTED INDEXES` clause 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.

> **See Also:** SORTED INDEXES Clause on page 9-18

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 records into a large table. This option inserts each index entry directly into the index, one record at a time.

By default, SQL*Loader does not use `SINGLEROW` to append records 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 because additional UNDO information is generated for each index insert. This option is suggested for use when either of the following situations exists:

- Available storage is limited.
- The number of records to be loaded is small compared to the size of the table (a ratio of 1:20 or less is recommended).

Benefits of Using Multiple INTO TABLE Clauses

Multiple INTO TABLE clauses allow you to:

- Load data into different tables
- Extract multiple logical records from a single input record
- Distinguish different input record formats
- Distinguish different input row object subtypes

In the first case, it is common for the INTO TABLE clauses to refer to the same table. This section illustrates the different ways to use multiple INTO TABLE clauses and shows you how to use the POSITION parameter.

> **Note:** A key point when using multiple INTO TABLE clauses is that *field scanning continues from where it left off* when a new INTO TABLE clause 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 parameter.

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, assume the data is as follows:

```
1119 Smith      1120 Yvonne
```

```
1121 Albert     1130 Thomas
```

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 Based on Delimiters

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 parameter. That mechanism is described in Distinguishing Different Input Record Formats on page 5-40 and in Loading Data into Multiple Tables on page 5-43.

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  FILLER POSITION(1:1)  INTEGER EXTERNAL,
    deptno POSITION(3:4)  INTEGER EXTERNAL,
    dname  POSITION(8:21) CHAR)
INTO TABLE emp
   WHEN recid <> 1
   (recid  FILLER POSITION(1:1)   INTEGER EXTERNAL,
    empno  POSITION(3:6)   INTEGER EXTERNAL,
    ename  POSITION(8:17)  CHAR,
    deptno POSITION(19:20) INTEGER EXTERNAL)
```

Relative Positioning Based on the POSITION Parameter

The records in the previous example could also be loaded as delimited data. In this case, however, it is necessary to use the `POSITION` parameter. The following control file could be used:

```
INTO TABLE dept
   WHEN recid = 1
   (recid  FILLER INTEGER EXTERNAL TERMINATED BY WHITESPACE,
    deptno INTEGER EXTERNAL TERMINATED BY WHITESPACE,
    dname  CHAR TERMINATED BY WHITESPACE)
INTO TABLE emp
   WHEN recid <> 1
   (recid  FILLER POSITION(1) INTEGER EXTERNAL TERMINATED BY ' ',
    empno  INTEGER EXTERNAL TERMINATED BY ' '
    ename  CHAR TERMINATED BY WHITESPACE,
    deptno INTEGER EXTERNAL TERMINATED BY ' ')
```

The `POSITION` parameter in the second `INTO TABLE` clause is necessary to load this data correctly. It 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`.

Distinguishing Different Input Row Object Subtypes

A single datafile may contain records made up of row objects inherited from the same base row object type. For example, consider the following simple object type and object table definitions in which a nonfinal base object type is defined along with two object subtypes that inherit from the base type:

```
CREATE TYPE person_t AS OBJECT
 (name    VARCHAR2(30),
  age     NUMBER(3)) not final;

CREATE TYPE employee_t UNDER person_t
 (empid   NUMBER(5),
  deptno  NUMBER(4),
  dept    VARCHAR2(30)) not final;

CREATE TYPE student_t UNDER person_t
 (stdid   NUMBER(5),
  major   VARCHAR2(20)) not final;

CREATE TABLE persons OF person_t;
```

The following input datafile contains a mixture of these row objects subtypes. A type ID field distinguishes between the three subtypes. `person_t` objects have a `P` in the first column, `employee_t` objects have an `E`, and `student_t` objects have an `S`.

```
P,James,31,
P,Thomas,22,
E,Pat,38,93645,1122,Engineering,
P,Bill,19,
P,Scott,55,
S,Judy,45,27316,English,
S,Karen,34,80356,History,
E,Karen,61,90056,1323,Manufacturing,
S,Pat,29,98625,Spanish,
S,Cody,22,99743,Math,
P,Ted,43,
E,Judy,44,87616,1544,Accounting,
E,Bob,50,63421,1314,Shipping,
S,Bob,32,67420,Psychology,
E,Cody,33,25143,1002,Human Resources,
```

The following control file uses relative positioning based on the `POSITION` parameter to load this data. Note the use of the TREAT AS clause with a specific object type name. This informs SQL*Loader that all input row objects for the object table will conform to the definition of the named object type.

```
INTO TABLE persons
```

```
REPLACE
WHEN typid = 'P' TREAT AS person_t
FIELDS TERMINATED BY ","
 (typid   FILLER  POSITION(1) CHAR,
  name            CHAR,
  age             CHAR)

INTO TABLE persons
REPLACE
WHEN typid = 'E' TREAT AS employee_t
FIELDS TERMINATED BY ","
 (typid   FILLER  POSITION(1) CHAR,
  name            CHAR,
  age             CHAR,
  empid           CHAR,
  deptno          CHAR,
  dept            CHAR)

INTO TABLE persons
REPLACE
WHEN typid = 'S' TREAT AS student_t
FIELDS TERMINATED BY ","
 (typid   FILLER  POSITION(1) CHAR,
  name            CHAR,
  age             CHAR,
  stdid           CHAR,
  major           CHAR)
```

See Also: Loading Column Objects on page 7-1 for more information on loading object types

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 Study 5: Loading Data into Multiple Tables on page 10-18.

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 parameter is essential for achieving the expected results.

When the POSITION parameter 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 parameter *is* used, multiple INTO TABLE clauses can process the same record in different ways, allowing multiple formats to be recognized in one input file.

Bind Arrays and Conventional Path Loads

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 the Oracle database an INSERT command, the entire array is inserted at one time. After the rows in the bind array are inserted, a COMMIT is issued.

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 uses the direct path API, rather than Oracle's SQL interface.

> **See Also:** *Oracle Call Interface Programmer's Guide* for more information about the concepts of direct path loading

Size Requirements for Bind Arrays

The bind array must 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 Parameters on page 4-3.

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 of Bind Arrays

Large bind arrays minimize the number of calls to the Oracle database server and maximize performance. 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 to be greater than 100 rows generally delivers more modest improvements in performance. The size (in bytes) of 100 rows is typically a good value to use.

In general, any reasonably large size permits SQL*Loader to operate effectively. It is not usually necessary to perform the detailed calculations described in this section. Read this section when you need maximum performance or an explanation of memory usage.

Specifying Number of Rows Versus Size of Bind Array

When you specify a bind array size using the command-line parameter `BINDSIZE` (see BINDSIZE (maximum size) on page 4-4) or the `OPTIONS` clause in the control file (see OPTIONS Clause on page 5-4), 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 size in bytes 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 4-12) or the `OPTIONS` clause in the control file (see OPTIONS Clause on page 5-4).

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 to Determine Bind Array Size

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, as follows:

```
bind array size =
    (number of rows) * (  SUM(fixed field lengths)
                        + SUM(maximum varying field lengths)
                        + ( (number of varying length fields)
                             * (size of length indicator) )
                       )
```

Many fields do not vary in size. These fixed-length fields are the same for each loaded row. For these fields, the maximum length of the field is the field size, in bytes, as described in SQL*Loader Datatypes on page 6-7. There is no overhead for these fields.

The fields that *can* vary in size from row to row are:

- CHAR
- DATE
- INTERVAL DAY TO SECOND
- INTERVAL DAY TO YEAR
- LONG VARRAW
- numeric EXTERNAL
- TIME
- TIMESTAMP
- TIME WITH TIME ZONE
- TIMESTAMP WITH TIME ZONE
- VARCHAR
- VARCHARC
- VARGRAPHIC
- VARRAW
- VARRAWC

The maximum length of these datatypes is described in SQL*Loader Datatypes on page 6-7. The maximum lengths describe the number of bytes 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.

Note: In conventional path loads, LOBFILEs are not included when allocating the size of a bind array.

Determining the Size of the Length Indicator

On most systems, the size of the length indicator is 2 bytes. On a few systems, it is 3 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(1))
BEGINDATA
a
```

This control file loads a 1-byte CHAR using a 1-row bind array. In this example, no data is actually loaded because a conversion error occurs when the character a is loaded into a numeric column (deptno). 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 determine the bind array size.

Calculating the Size of Field Buffers

Table 5–3 through Table 5–6 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 6-7.

Table 5–3 Fixed-Length Fields

Datatype	Size in Bytes (Operating System-Dependent)
INTEGER	The size of the INT datatype, in C
INTEGER(*N*)	*N* bytes
SMALLINT	The size of SHORT INT datatype, in C
FLOAT	The size of the FLOAT datatype, in C
DOUBLE	The size of the DOUBLE datatype, in C
BYTEINT	The size of UNSIGNED CHAR, in C
VARRAW	The size of UNSIGNED SHORT, plus 4096 bytes or whatever is specified as *max_length*
LONG VARRAW	The size of UNSIGNED INT, plus 4096 bytes or whatever is specified as *max_length*
VARCHARC	Composed of 2 numbers. The first specifies length, and the second (which is optional) specifies *max_length* (default is 4096 bytes).
VARRAWC	This datatype is for RAW data. It is composed of 2 numbers. The first specifies length, and the second (which is optional) specifies *max_length* (default is 4096 bytes).

Table 5–4 Nongraphic Fields

Datatype	Default Size	Specified Size
(packed) DECIMAL	None	(N+1)/2, rounded up
ZONED	None	P
RAW	None	L
CHAR (no delimiters)	1	L+S
datetime and interval (no delimiters)	None	L+S
numeric EXTERNAL (no delimiters)	None	L+S

Table 5–5 Graphic Fields

Datatype	Default Size	Length Specified with POSITION	Length Specified with DATATYPE
GRAPHIC	None	L	2*L

Table 5–5 (Cont.) Graphic Fields

Datatype	Default Size	Length Specified with POSITION	Length Specified with DATATYPE
GRAPHIC EXTERNAL	None	L - 2	2*(L-2)
VARGRAPHIC	4Kb*2	L+S	(2*L)+S

Table 5–6 Variable-Length Fields

Datatype	Default Size	Maximum Length Specified (L)
VARCHAR	4Kb	L+S
CHAR (delimited)	255	L+S
datetime and interval (delimited)	255	L+S
numeric EXTERNAL (delimited)	255	L+S

Minimizing Memory Requirements for Bind Arrays

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. Consider the following example:

```
CHAR(10) TERMINATED BY ","
```

With byte-length semantics, this example uses (10 + 2) * 64 = 768 bytes in the bind array, assuming that the length indicator is 2 bytes long and that 64 rows are loaded at a time.

With character-length semantics, the same example uses ((10 * s) + 2) * 64 bytes in the bind array, where "s" is the maximum size in bytes of a character in the datafile character set.

Now consider the following example:

```
CHAR TERMINATED BY ","
```

Regardless of whether byte-length semantics or character-length semantics are used, this example uses (255 + 2) * 64 = 16,448 bytes, because the default maximum size for a delimited field is 255 bytes. This can make a considerable difference in the number of rows that fit into the bind array.

Calculating Bind Array Size for Multiple INTO TABLE Clauses

When calculating a bind array size for a control file that has multiple INTO TABLE clauses, calculate as if the INTO TABLE clauses 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. It is especially important to minimize the buffer allocations for such fields.

Note: Generated data is produced by the SQL*Loader functions CONSTANT, EXPRESSION, RECNUM, SYSDATE, and SEQUENCE. Such generated data does not require any space in the bind array.

6

Field List Reference

This chapter describes the field-list portion of the SQL*Loader control file. The following topics are included:

- Field List Contents
- Specifying the Position of a Data Field
- Specifying Columns and Fields
- SQL*Loader Datatypes
- Specifying Field Conditions
- Using the WHEN, NULLIF, and DEFAULTIF Clauses
- Loading Data Across Different Platforms
- Byte Ordering
- Loading All-Blank Fields
- Trimming Whitespace
- Preserving Whitespace
- Applying SQL Operators to Fields
- Using SQL*Loader to Generate Data for Input

Field List Contents

The field-list portion of a SQL*Loader control file provides information about fields being loaded, such as position, datatype, conditions, and delimiters.

Example 6–1 shows the field list section of the sample control file that was introduced in Chapter 5.

Example 6–1 Field List Section of Sample Control File

```
.
.
.
1  (hiredate  SYSDATE,
2     deptno  POSITION(1:2)  INTEGER EXTERNAL(2)
              NULLIF deptno=BLANKS,
3       job   POSITION(7:14)  CHAR  TERMINATED BY WHITESPACE
              NULLIF job=BLANKS  "UPPER(:job)",
       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
              "TO_NUMBER(:sal,'$99,999.99')",
4      comm   INTEGER EXTERNAL  ENCLOSED BY '(' AND '%'
              ":comm * 100"
    )
```

In this sample control file, the numbers that appear to the left would not appear in a real control file. They are keyed in this sample to the explanatory notes in the following list:

1. SYSDATE sets the column to the current system date. See Setting a Column to the Current Date on page 6-56.
2. POSITION specifies the position of a data field. See Specifying the Position of a Data Field on page 6-3.

   INTEGER EXTERNAL is the datatype for the field. See Specifying the Datatype of a Data Field on page 6-7 and Numeric EXTERNAL on page 6-19.

   The NULLIF clause is one of the clauses that can be used to specify field conditions. See Using the WHEN, NULLIF, and DEFAULTIF Clauses on page 6-32.

   In this sample, the field is being compared to blanks, using the BLANKS parameter. See Comparing Fields to BLANKS on page 6-31.
3. The TERMINATED BY WHITESPACE clause is one of the delimiters it is possible to specify for a field. See TERMINATED Fields on page 6-25.
4. The ENCLOSED BY clause is another possible field delimiter. See Enclosed Fields on page 6-49.

Specifying the Position of a Data Field

To load data from the datafile, SQL*Loader must know the length and location of the field. To specify the position of a field in the logical record, use the `POSITION` clause 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. The start, end, and integer values are always in bytes, even if character-length semantics are used for a datafile.

The syntax for the position specification (pos_spec) clause is as follows:

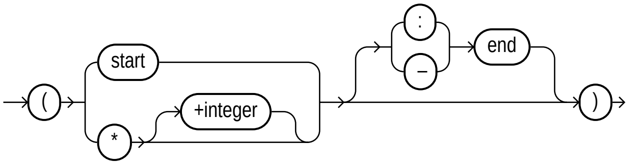


Table 6–1 describes the parameters for the position specification clause.

Table 6–1 Parameters for the Position Specification Clause

Parameter	Description
start	The starting column of the data field in the logical record. The first byte 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, and without a length specification (CHAR(*n*)), is assumed to have a length of 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.
+*integer*	You can use an offset, specified as +*integer*, to offset the current field from the next position after the end of the previous field. A number of bytes, as specified by *+integer*, 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.

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 you use the SQL*Loader advanced SQL string capabilities to load data from a formatted report:

- You look at a printed copy of the report, carefully measuring all character positions, and create your control file.
- The load 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.

See Also: Specifying Delimiters on page 6-24

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 Extracting Multiple Logical Records on page 5-39.

A logical record might 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 of Using POSITION

```
siteid  POSITION (*) SMALLINT
```

```
siteloc POSITION (*) INTEGER
```

If these were the first two column specifications, `siteid` would begin in column 1, 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 next position (27) after the end of `empno` by +2, so it starts in column 29 and continues until a slash is encountered.

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.

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 `RECNUM`, `SEQUENCE`, or `CONSTANT` parameter. See Using SQL*Loader to Generate Data for Input on page 6-54.

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

A filler field, specified by FILLER, is a datafile mapped field that does not correspond to a database column. Filler fields are assigned values from the data fields to which they are mapped.

Keep the following in mind with regard to filler fields:

- 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 FILLER.
- Filler fields have names but they are not loaded into the table.
- Filler fields can be used as arguments to init_specs (for example, NULLIF and DEFAULTIF).
- Filler fields can be used as arguments to directives (for example, SID, OID, REF, BFILE).
- 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 or DEFAULTIF clause.
- Filler fields are initialized to NULL if TRAILING NULLCOLS is specified and applicable. If another field references a nullified filler field, an error is generated.
- Filler fields can occur anyplace in the datafile, including inside the field list for an object or inside the definition of a VARRAY.
- SQL strings cannot be specified as part of a filler field specification because no space is allocated for fillers in the bind array.

Note: The information in this section also applies to specifying bound fillers by using BOUNDFILLER. The only exception is that with bound fillers, SQL strings *can* be specified as part of the field because space is allocated for them in the bind array.

A sample filler field specification looks as follows:

```
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

The datatype specification of a field 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 can contain any character data.

Only *one* datatype can be specified for each field; if a datatype is not specified, CHAR is assumed.

SQL*Loader Datatypes on page 6-7 describes how SQL*Loader datatypes are converted into Oracle datatypes and gives detailed information on each SQL*Loader datatype.

Before you specify the datatype, you must specify the position of the field.

SQL*Loader Datatypes

SQL*Loader datatypes can be grouped into portable and nonportable datatypes. Within each of these two groups, the datatypes are subgrouped into value datatypes and length-value datatypes.

Portable versus nonportable refers to whether or not the datatype is platform dependent. Platform dependency can exist for a number of reasons, including differences in the byte ordering schemes of different platforms (big endian versus little endian), differences in the number of bits in a platform (16-bit, 32-bit, 64-bit), differences in signed number representation schemes (2’s complement versus 1’s complement), and so on. In some cases, such as with byte ordering schemes and platform word length, SQL*Loader provides mechanisms to help overcome platform dependencies. These mechanisms are discussed in the descriptions of the appropriate datatypes.

Both portable and nonportable datatypes can be values or length-values. Value datatypes assume that a data field has a single part. Length-value datatypes require that the data field consist of two subfields where the length subfield specifies how long the value subfield can be.

Nonportable Datatypes

Nonportable datatypes are grouped into value datatypes and length-value datatypes. The nonportable value datatypes are as follows:

- `INTEGER(`*n*`)`
- `SMALLINT`
- `FLOAT`
- `DOUBLE`
- `BYTEINT`
- `ZONED`
- (packed) `DECIMAL`

The nonportable length-value datatypes are as follows:

- `VARGRAPHIC`
- `VARCHAR`
- `VARRAW`
- `LONG VARRAW`

The syntax for the nonportable datatypes is shown in the syntax diagram for datatype_spec on page A-9.

INTEGER(*n*)

The data is a full-word binary integer, where *n* is an optionally supplied length of 1, 2, 4, or 8. If no length specification is given, then the length, in bytes, is based on the size of a `LONG INT` in the C programming language on your particular platform.

`INTEGER`s are not portable because their byte size, their byte order, and the representation of signed values may be different between systems. However, if the representation of signed values is the same between systems, SQL*Loader may be able to access `INTEGER` data with correct results. If `INTEGER` is specified with a length specification (*n*), and the appropriate technique is used (if necessary) to indicate the byte order of the data, then SQL*Loader can access the data with correct results between systems. If `INTEGER` is specified without a length specification, then SQL*Loader can access the data with correct results only if the size of a LONG INT in the C programming language is the same length in bytes on both systems. In that case, the appropriate technique must still be used (if necessary) to indicated the byte order of the data.

Specifying an explicit length for binary integers is useful in situations where the input data was created on a platform whose word length differs from that on which SQL*Loader is running. For instance, input data containing binary integers might be created on a 64-bit platform and loaded into a database using SQL*Loader on a 32-bit platform. In this case, use INTEGER(8) to instruct SQL*Loader to process the integers as 8-byte quantities, not as 4-byte quantities.

By default, INTEGER is treated as a SIGNED quantity. If you want SQL*Loader to treat it as an unsigned quantity, specify UNSIGNED. To return to the default behavior, specify SIGNED.

> **See Also:** Loading Data Across Different Platforms on page 6-36

SMALLINT

The data is a half-word binary integer. The length of the field is the length of a half-word integer on your system. By default, it is treated as a SIGNED quantity. If you want SQL*Loader to treat it as an unsigned quantity, specify UNSIGNED. To return to the default behavior, specify SIGNED.

SMALLINT can be loaded with correct results only between systems where a SHORT INT has the same length in bytes. If the byte order is different between the systems, use the appropriate technique to indicate the byte order of the data. See Byte Ordering on page 6-37.

> **Note:** 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, *end* is ignored. The length of the field is the length of a single-precision, floating-point binary number on your system. (The datatype is FLOAT in C.) This length cannot be overridden in the control file.

FLOAT can be loaded with correct results only between systems where the representation of a FLOAT is compatible and of the same length. If the byte order is different between the two systems, use the appropriate technique to indicate the byte order of the data. See Byte Ordering on page 6-37.

DOUBLE

The data is a double-precision, floating-point binary number. If you specify *end* in the POSITION clause, *end* is ignored. The length of the field is the length of a double-precision, floating-point binary number on your system. (The datatype is DOUBLE or LONG FLOAT in C.) This length cannot be overridden in the control file.

DOUBLE can be loaded with correct results only between systems where the representation of a DOUBLE is compatible and of the same length. If the byte order is different between the two systems, use the appropriate technique to indicate the byte order of the data. See Byte Ordering on page 6-37.

BYTEINT

The decimal value of the binary representation of the byte is loaded. For example, the input character x"1C" is loaded as 28. The length of a BYTEINT field is always 1 byte. If POSITION*(start:end)* is specified, *end* is ignored. (The datatype is UNSIGNED CHAR in C.)

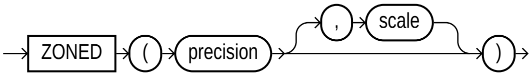
An example of the syntax for this datatype is:

```
(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 the ZONED datatype is:



In this syntax, *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. The following example specifies an 8-digit integer starting at position 32:

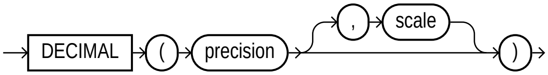
```
sal   POSITION(32)   ZONED(8),
```

The Oracle database server uses the VAX/VMS zoned decimal format when the zoned data is generated on an ASCII-based platform. It is also possible to load zoned decimal data that is generated on an EBCDIC-based platform. In this case, Oracle uses the IBM format as specified in the ESA/390 Principles of Operations, version 8.1 manual. The format that is used depends on the character set encoding of the input datafile. See CHARACTERSET Parameter on page 5-19 for more information.

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 DECIMAL datatype is:



The *precision* parameter is the number of digits in a value. The length of the field in bytes, as computed from digits, is (N+1)/2 rounded up.

The *scale* parameter is the scaling factor, or number of digits to the right of the decimal point. The default is zero (indicating an integer). The scaling factor can be greater than the number of digits but cannot be negative.

An example is:

```
sal DECIMAL (7,2)
```

This example 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 1 is added for the sign.)

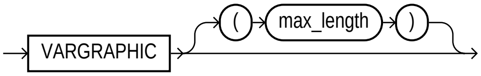
VARGRAPHIC

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). The Oracle database server does not support DBCS; however, SQL*Loader reads DBCS as single bytes and loads it as `RAW` data. Like `RAW` data, `VARGRAPHIC` fields are stored without modification in whichever column you specify.

> **Note:** 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 6-9 for more information.

`VARGRAPHIC` data can be loaded with correct results only between systems where a `SHORT INT` has the same length in bytes. If the byte order is different between the systems, use the appropriate technique to indicate the byte order of the length subfield. See Byte Ordering on page 6-37.

The syntax for the `VARGRAPHIC` datatype is:



The length of the current field is given in the first 2 bytes. A maximum length specified for the `VARGRAPHIC` datatype does *not* include the size of the length subfield. The maximum length specifies the number of graphic (double-byte) characters. It is multiplied by 2 to determine the maximum length of the field in bytes.

The default maximum field length is 2 KB graphic characters, or 4 KB (2 * 2KB). To minimize memory requirements, specify a maximum length for such fields whenever possible.

If a position specification is specified (using `pos_spec`) before the `VARGRAPHIC` statement, it provides the location of the length subfield, not of the first graphic character. If you specify `pos_spec`*`(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 the position specification.

If a `VARGRAPHIC` field is truncated by the end of the logical record before its full length is read, a warning is issued. Because the length of a `VARGRAPHIC` field is embedded in every occurrence of the input data for that field, it is assumed to be accurate.

VARGRAPHIC data cannot be delimited.

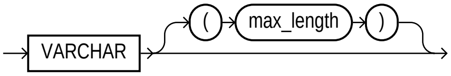
VARCHAR

A VARCHAR field is a length-value datatype. It consists of a binary length subfield followed by a character string of the specified length. The length is in bytes unless character-length semantics are used for the datafile. In that case, the length is in characters. See Character-Length Semantics on page 5-22.

VARCHAR fields can be loaded with correct results only between systems where a SHORT data field INT has the same length in bytes. If the byte order is different between the systems, or if the VARCHAR field contains data in the UTF16 character set, use the appropriate technique to indicate the byte order of the length subfield and of the data. The byte order of the data is only an issue for the UTF16 character set. See Byte Ordering on page 6-37.

> **Note:** 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 6-9 for more information.

The syntax for the VARCHAR 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 for a VARCHAR datatype, then a buffer of that size, in bytes, is allocated for these fields. However, if character-length semantics are used for the datafile, the buffer size in bytes is the max_length times the size in bytes of the largest possible character in the character set. See Character-Length Semantics on page 5-22.

The default maximum 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.

The POSITION clause, if used, gives the location, in bytes, 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 the length of a VARCHAR field 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 2-byte binary length subfield followed by a RAW string value subfield.

VARRAW results in a VARRAW with a 2-byte length subfield and a maximum size of 4 KB (that is, the default). VARRAW(65000) results in a VARRAW with a length subfield of 2 bytes and a maximum size of 65000 bytes.

VARRAW fields can be loaded between systems with different byte orders if the appropriate technique is used to indicate the byte order of the length subfield. See Byte Ordering on page 6-37.

LONG VARRAW

LONG VARRAW is a VARRAW with a 4-byte length subfield instead of a 2-byte length subfield.

LONG VARRAW results in a VARRAW with 4-byte length subfield and a maximum size of 4 KB (that is, the default). LONG VARRAW(300000) results in a VARRAW with a length subfield of 4 bytes and a maximum size of 300000 bytes.

LONG VARRAW fields can be loaded between systems with different byte orders if the appropriate technique is used to indicate the byte order of the length subfield. See Byte Ordering on page 6-37.

Portable Datatypes

The portable datatypes are grouped into value datatypes and length-value datatypes. The portable value datatypes are as follows:

- CHAR
- Datetime and Interval
- GRAPHIC

- GRAPHIC EXTERNAL
- Numeric EXTERNAL (INTEGER, FLOAT, DECIMAL, ZONED)
- RAW

The portable length-value datatypes are as follows:

- VARCHARC
- VARRAWC

The syntax for these datatypes is shown in the diagram for datatype_spec on page A-9.

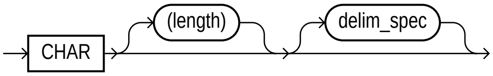
The character datatypes are CHAR, DATE, and the numeric EXTERNAL datatypes. These fields can be delimited and can have lengths (or maximum lengths) specified in the control file.

CHAR

The data field contains character data. The length, which is optional, is a maximum length. Note the following with regard to length:

- If a length is not specified, it is derived from the POSITION specification.
- If a length *is* specified, it overrides the length in the POSITION specification.
- If no length is given and there is no position specification, CHAR data is assumed to have a length of 1, unless the field is delimited:
  - For a delimited CHAR field, if a length is specified, that length is used as a maximum.
  - For a delimited CHAR field for which no length is specified, the default is 255 bytes.
  - For a delimited CHAR field that is greater than 255 bytes, you must specify a maximum length. Otherwise you will receive an error stating that the field in the datafile exceeds maximum length.

The syntax for the CHAR datatype is:



See Also: Specifying Delimiters on page 6-24

Datetime and Interval Datatypes

The datetime datatypes are:

- DATE
- TIME
- TIMESTAMP
- TIME WITH TIME ZONE
- TIMESTAMP WITH TIME ZONE

Values of datetime datatypes are sometimes called datetimes.

The interval datatypes are:

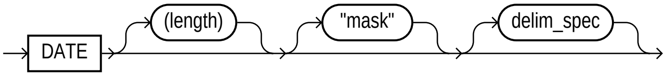
- INTERVAL YEAR TO MONTH
- INTERVAL DAY TO SECOND

Values of interval datatypes are sometimes called intervals.

Both datetimes and intervals are made up of fields. The values of these fields determine the value of the datatype.

> **See Also:** *Oracle9i SQL Reference* for more detailed information about datetime and interval datatypes

DATE The DATE field contains character data that should be converted to an Oracle date using the specified date mask. The syntax for the DATE field 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
```

Whitespace is ignored and dates are parsed from left to right unless delimiters are present. (A DATE field that consists entirely of whitespace is loaded as a NULL field.)

The length specification is optional, unless a varying-length date mask is specified. The length is in bytes unless character-length semantics are used for the datafile. In that case, the length is in characters. See Character-Length Semantics on page 5-22.

In the preceding example, the date mask, "DD-Mon-YYYY" contains 11 bytes, with byte-length semantics. Therefore, SQL*Loader expects a maximum of 11 bytes in the field, so the specification works properly. But, suppose a specification such as the following is given:

```
DATE "Month dd, YYYY"
```

In this case, the date mask contains 14 bytes. If a value with a length longer than 14 bytes is specified, such as "September 30, 1991", 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 bytes 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 Study 3: Loading a Delimited, Free-Format File on page 10-11 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 6-24.

TIME The TIME datatype stores hour, minute, and second values. For example:

```
09:26:50
```

TIMESTAMP The TIMESTAMP datatype is an extension of the DATE datatype. It stores the year, month, and day of the DATE datatype, plus the hour, minute, and second values of the TIME datatype. An example TIMESTAMP is as follows:

```
TIMESTAMP '1999-01-31 09:26:50'
```

If you specify a date value without a time component, the default time is 12:00:00 AM (midnight).

TIME WITH TIME ZONE The TIME WITH TIME ZONE datatype is a variant of TIME that includes a time zone displacement in its value. The time zone displacement is the difference (in hours and minutes) between local time and UTC (coordinated universal time, formerly Greenwich mean time).

If the LOCAL option is specified, then data stored in the database is normalized to the database time zone, and time zone displacement is not stored as part of the column data. When the data is retrieved, it is returned in the user's local session time zone.

TIMESTAMP WITH TIME ZONE The TIMESTAMP WITH TIME ZONE datatype is a variant of TIMESTAMP that includes a time zone displacement in its value. The time zone displacement is the difference (in hours and minutes) between local time and UTC (coordinated universal time, formerly Greenwich mean time).

If the LOCAL option is specified, then data stored in the database is normalized to the database time zone, and time zone displacement is not stored as part of the column data. When the data is retrieved, it is returned in the user's local session time zone.

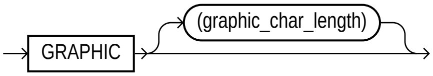
INTERVAL YEAR TO MONTH The INTERVAL YEAR TO MONTH datatype stores a period of time using the YEAR and MONTH datetime fields.

INTERVAL DAY TO SECOND The INTERVAL DAY TO SECOND datatype stores a period of time using the DAY and SECOND datetime fields.

GRAPHIC

The data is a string of double-byte characters (DBCS). The Oracle database server 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 the GRAPHIC datatype is:



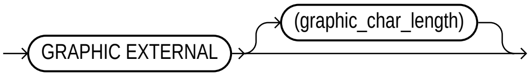
For GRAPHIC and GRAPHIC EXTERNAL, specifying POSITION*(start:end)* gives the exact location of the field in the logical record.

If you specify a length for the GRAPHIC (EXTERNAL) datatype, 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 data field 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 the GRAPHIC EXTERNAL datatype is:



GRAPHIC indicates that the data is double-byte characters. EXTERNAL indicates that the first and last characters are ignored. The graphic_char_length value specifies the length in DBCS (see GRAPHIC on page 6-18).

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

The numeric EXTERNAL datatypes are the numeric datatypes (INTEGER, FLOAT, DECIMAL, and ZONED) specified as EXTERNAL, with optional length and delimiter specifications. The length is in bytes unless character-length semantics are used for the datafile. In that case, the length is in characters. See Character-Length Semantics on page 5-22.

These datatypes are the human-readable, character form of numeric data. The same rules that apply to CHAR data with regard to length, position, and delimiters apply to numeric EXTERNAL data. See CHAR on page 6-15 for a complete description of these rules.

The syntax for the numeric EXTERNAL datatypes is shown as part of datatype_spec on page A-9.

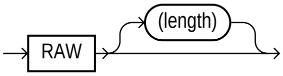
Note: The data is a number in character form, not binary representation. Therefore, these datatypes are identical to CHAR and are treated identically, *except for the use of DEFAULTIF.* If you want the default to be null, use CHAR; if you want it to be zero, use EXTERNAL. See Using the WHEN, NULLIF, and DEFAULTIF Clauses on page 6-32.

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

When raw, binary data is loaded "as is" into a RAW database column, it is not converted by the Oracle database server. If it is loaded into a CHAR column, the Oracle database server converts it to hexadecimal. It cannot be loaded into a DATE or number column.

The syntax for the RAW datatype is as follows:



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. The length is always in bytes, even if character-length semantics are used for the datafile. RAW data fields cannot be delimited.

VARCHARC

The datatype VARCHARC consists of a character length subfield followed by a character string value-subfield.

The declaration for VARCHARC specifies the length of the length subfield, optionally followed by the maximum size of any string. If byte-length semantics are in use for the datafile, then the length and the maximum size are both in bytes. If character-length semantics are in use for the datafile, then the length and maximum size are in characters. If a maximum size is not specified, 4 KB is the default regardless of whether byte-length semantics or character-length semantics are in use.

For example:

- VARCHARC results in an error because you must at least specify a value for the length subfield.
- VARCHARC(7) results in a VARCHARC whose length subfield is 7 bytes long and whose maximum size is 4 KB (the default) if byte-length semantics are used for the datafile. If character-length semantics are used, it results in a VARCHARC with a length subfield that is 7 characters long and a maximum size of 4 KB (the default). Remember that when a maximum size is not specified, the default of 4 KB is always used, regardless of whether byte-length or character-length semantics are in use.
- VARCHARC(3,500) results in a VARCHARC whose length subfield is 3 bytes long and whose maximum size is 500 bytes if byte-length semantics are used for the datafile. If character-length semantics are used, it results in a VARCHARC with a length subfield that is 3 characters long and a maximum size of 500 characters.

See Character-Length Semantics on page 5-22.

VARRAWC

The datatype VARRAWC consists of a RAW string value subfield.

For example:

- VARRAWC results in an error.
- VARRAWC(7) results in a VARRAWC whose length subfield is 7 bytes long and whose maximum size is 4 KB (that is, the default).
- VARRAWC(3,500) results in a VARRAWC whose length subfield is 3 bytes long and whose maximum size is 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 SMALLINT, FLOAT, and DOUBLE data is fixed, regardless of the number of bytes specified in the POSITION clause.

2. If the length specified (or precision) of a DECIMAL, INTEGER, 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 character or VARGRAPHIC field conflicts with the size calculated from a POSITION*(start:end)* specification, then the specified maximum is used.

For example, assume that the native datatype INTEGER is 4 bytes long and the following field specification is given:

```
column1 POSITION(1:6) INTEGER
```

In this case, a warning is issued, and the proper length (4) is used. 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
```

Field Lengths for Length-Value Datatypes

A control file can specify a maximum length for the following length-value datatypes: VARCHAR, VARCHARC, VARGRAPHIC, VARRAW, and VARRAWC. The specified maximum length is in bytes if byte-length semantics are used for the field, and in characters if character-length semantics are used for the field. If no length is specified, the maximum length defaults to 4096 bytes. If the length of the field exceeds the maximum length, the record is rejected with the following error:

```
Variable length field exceed maximum length
```

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).

SQL*Loader or the server does any necessary data conversion to store the data in the proper internal format. This includes converting data from the datafile character set to the database character set when they differ.

The datatype of the data in the file does not need to be the same as the datatype of the column in the Oracle table. The Oracle database server 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. The SQL*Loader 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 the Oracle database server is capable of converting may be loaded into these or other database columns.

Datatype Conversions for Datetime and Interval Datatypes

Table 6–2 shows which conversions between Oracle database datatypes and SQL*Loader control file datetime and interval datatypes are supported and which are not.

In the table, the abbreviations for the Oracle Database Datatypes are as follows:

N = NUMBER

C = CHAR or VARCHAR2

D = DATE

T = TIME and TIME WITH TIME ZONE

TS = TIMESTAMP and TIMESTAMP WITH TIME ZONE

YM = INTERVAL YEAR TO MONTH

DS = INTERVAL DAY TO SECOND

For the SQL*Loader datatypes, the definitions for the abbreviations in the table are the same for D, T, TS, YM, and DS. However, as noted in the previous section, SQL*Loader does *not* contain datatype specifications for Oracle internal datatypes such as NUMBER, CHAR, and VARCHAR2. However, any data that the Oracle database server is capable of converting can be loaded into these or other database columns.

For an example of how to read this table, look at the row for the SQL*Loader datatype DATE (abbreviated as D). Reading across the row, you can see that datatype conversion is supported for the Oracle database datatypes of CHAR, VARCHAR2, DATE, TIMESTAMP, and TIMESTAMP WITH TIMEZONE datatypes. However, conversion is not supported for the Oracle database datatypes NUMBER, TIME, TIME WITH TIME ZONE, INTERVAL YEAR TO MONTH, or INTERVAL DAY TO SECOND datatypes.

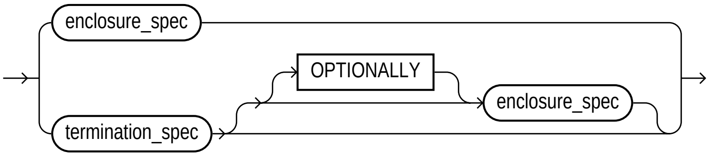
Table 6–2 Datatype Conversions for Datetime and Interval Datatypes

SQL*Loader Datatype	Oracle Database Datatype (Conversion Support)
N	**N** (Yes), **C** (Yes), **D** (No), **T** (No), **TS** (No), **YM** (No), **DS** (No)
C	**N** (Yes), **C** (Yes), **D** (Yes), **T** (Yes), **TS** (Yes), **YM** (Yes), **DS** (Yes)
D	**N** (No), **C** (Yes), **D** (Yes), **T** (No), **TS** (Yes), **YM** (No), **DS** (No)
T	**N** (No), **C** (Yes), **D** (No), **T** (Yes), **TS** (Yes), **YM** (No), **DS** (No)
TS	**N** (No), **C** (Yes), **D** (Yes), **T** (Yes), **TS** (Yes), **YM** (No), **DS** (No)
YM	**N** (No), **C** (Yes), **D** (No), **T** (No), **TS** (No), **YM** (Yes), **DS** (No)
DS	**N** (No), **C** (Yes), **D** (No), **T** (No), **TS** (No), **YM** (No), **DS** (Yes)

Specifying Delimiters

The boundaries of CHAR, datetime, interval, or numeric EXTERNAL fields may also be marked by specific delimiter characters contained in the input data record. The RAW datatype may also be marked by delimiters, but only if it is in an input LOBFILE, and only if the delimiter is TERMINATED BY EOF (end of file). You indicate how the field is delimited by using a delimiter specification after specifying the datatype.

Delimited data can be terminated or enclosed, as shown in the following syntax:



You can specify a TERMINATED BY clause, an ENCLOSED BY clause, or both. If both are used, the TERMINATED BY clause must come first.

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.

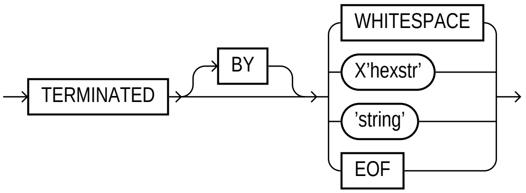
If TERMINATED BY WHITESPACE is specified, data is read until the first occurrence of a whitespace character (spaces, tabs, blanks, line feeds, form feeds, or carriage returns). Then the current position is advanced until no more adjacent whitespace characters are found. This allows field values to be delimited by varying amounts of whitespace. For more information about the syntax, see Syntax for Termination and Enclosure Specification on page 6-25.

ENCLOSED Fields

ENCLOSED fields are read by skipping whitespace until a nonwhitespace 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. For more information about the syntax, see Syntax for Termination and Enclosure Specification on page 6-25.

Syntax for Termination and Enclosure Specification



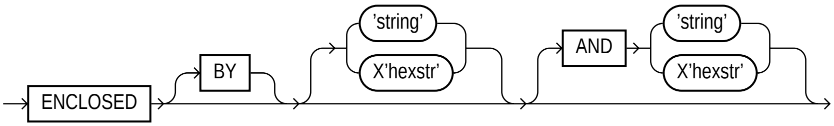


Table 6–3 describes the syntax for the termination and enclosure specification.

Table 6–3 Parameters for Termination and Enclosure Specification

Parameter	Description
TERMINATED	Data is read until the first occurrence of a delimiter.
BY	An optional word to increase readability.
WHITESPACE	Delimiter is any whitespace character including spaces, tabs, blanks, line feeds, form feeds, or carriage returns. (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 it 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.
string	The delimiter is a string.
X'hexstr'	The delimiter is a string that has the value specified by *X'hexstr'* in the character encoding scheme, such as X'1F' (equivalent to 31 decimal). "X"can be either lowercase or uppercase.
AND	Specifies a trailing enclosure delimiter that may be different from the initial enclosure delimiter. If AND is not present, then the initial and trailing delimiters are assumed to be the same.
EOF	Indicates that the entire file has been loaded into the LOB. This is valid only when data is loaded from a LOB file. Fields terminated by EOF cannot be enclosed.

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 punctuation mark that is a delimiter must also 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 parentheses, (, and right parentheses, ).
```

For this reason, problems can arise when adjacent fields use the same delimiters. For example, with 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. Therefore, delimited fields can require significant amounts of storage for the bind array. A good policy is to specify the smallest possible maximum value if the fields are shorter than 255 bytes. If the fields are longer than 255 bytes, then you must specify a maximum length for the field, either with a length specifier or with the `POSITION` clause.

Loading Trailing Blanks with Delimiters

Trailing blanks are not loaded with nondelimited datatypes unless you specify PRESERVE BLANKS. If a data field is 9 characters long and contains the value `DANIEL`*bbb*, where *bbb* is three blanks, it is loaded into the Oracle database 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. You could also specify

PRESERVE BLANKS without the TERMINATED BY clause and obtain the same results.

See Also:

- Trimming Whitespace on page 6-42
- Preserving Whitespace on page 6-49

Conflicting Field Lengths for Character Datatypes

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, as follows:

```
POSITION(1:10) CHAR(15)
```

In this example, 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 positions, then that length is the *maximum* length of the field. The specified maximum length is in bytes if byte-length semantics are used for the field, and in characters if character-length semantics are used for the field. If no length is specified or can be calculated from the start and end positions, the maximum length defaults to 255 bytes. The actual length can vary up to that maximum, based on the presence of the delimiter.

If starting and ending positions are specified for the field, as well as delimiters, then only the position specification has any effect. Any enclosure or termination delimiters are ignored.

If the expected delimiter is absent, 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 produces a field that is longer than the maximum, SQL*Loader rejects the record and returns and 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, assume the mask is specified as follows:

```
"Month dd, yyyy"
```

Then "May 3, 1991" would occupy 11 bytes in the record (with byte-length semantics), 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.

> **See Also:** Datetime and Interval Datatypes on page 6-16 for more information on the `DATE` field

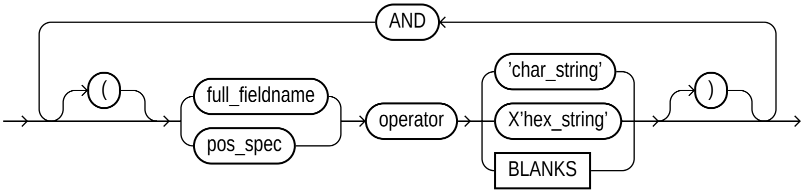
Specifying Field Conditions

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 can specify either a position in the logical record or the name of a field in the datafile (including filler fields).

> **Note:** A field condition cannot be based on fields in a secondary datafile (SDF).

The syntax for the `field_condition` clause is as follows:



The syntax for the pos_spec clause is as follows:

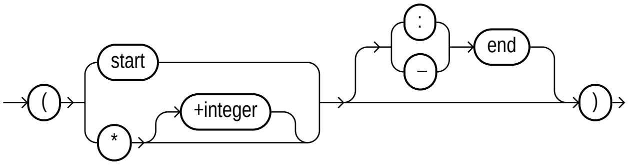


Table 6–4 describes the parameters used for the field condition clause. For a full description of the position specification parameters, see Table 6–1.

Table 6–4 Parameters for the Field Condition Clause

Parameter	Description
pos_spec	Specifies the starting and ending position of the comparison field in the logical record. It must be surrounded by parentheses. Either *start-end* or *start:end* is acceptable. The starting location can be specified as a column number, or as * (next column), or as *+*n* (next column plus an offset). If you omit an ending position, 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 zeros.
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.

Table 6–4 (Cont.) Parameters for the Field Condition Clause

Parameter	Description
full_fieldname	*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 field name referencing or naming the same entity can be different, because the 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 quotation marks that is compared to the comparison field. If the comparison is true, the current record is inserted into the table.
X'hex_string'	A string of hexadecimal digits, where each pair of digits corresponds to one byte in the field. It is enclosed in single or double quotation marks. If the comparison is true, the current record is inserted into the table.
BLANKS	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 multibyte character set that has multiple blanks.

Comparing Fields to BLANKS

The BLANKS parameter makes it possible to determine if a field of unknown length is blank.

For example, use the following clause to load a blank field as null:

```
full_fieldname ... NULLIF column_name=BLANKS
```

The BLANKS parameter recognizes only 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 parameter 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="  "
```

There can be more than one blank in a multibyte character set. It is a good idea to use the BLANKS parameter 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 parameter will match combinations of different blank characters. For more information on multibyte character sets, see Multibyte (Asian) Character Sets on page 5-17.

Comparing Fields to Literals

When a data field is compared to a literal string that is shorter than the data field, the string is padded. Character strings are padded with blanks, for example:

```
NULLIF (1:4)=" "
```

This example 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 zeros, as in the following clause:

```
NULLIF (1:4)=X’FF’
```

This clause compares position 1:4 to hexadecimal ’FF000000’.

Using the WHEN, NULLIF, and DEFAULTIF Clauses

The following information applies to scalar fields. For nonscalar fields (column objects, LOBs, and collections), the WHEN, NULLIF, and DEFAULTIF clauses are processed differently because nonscalar fields are more complex.

The results of a WHEN, NULLIF, or DEFAULTIF clause can be different depending on whether the clause specifies a field name or a position.

If the WHEN, NULLIF, or DEFAULTIF clause specifies a field name, SQL*Loader compares the clause to the evaluated value of the field. The evaluated value takes trimmed whitespace into consideration. See Trimming Whitespace on page 6-42 for information about trimming blanks and tabs.

If the WHEN, NULLIF, or DEFAULTIF clause specifies a position, SQL*Loader compares the clause to the original logical record in the datafile. No whitespace trimming is done on the logical record in that case.

Different results are more likely if the field has whitespace that is trimmed, or if the WHEN, NULLIF, or DEFAULTIF clause contains blanks or tabs or uses the BLANKS parameter. If you require the same results for a field specified by name and for the same field specified by position, use the PRESERVE BLANKS option. The PRESERVE BLANKS option instructs SQL*Loader not to trim whitespace when it evaluates the values of the fields.

The results of a WHEN, NULLIF, or DEFAULTIF clause are also affected by the order in which SQL*Loader operates, as described in the following steps. SQL*Loader performs these steps in order, but it does not always perform all of them. Once a field is set, any remaining steps in the process are ignored. For example, if the field is set in step 5, SQL*Loader does not move on to step 6.

1. SQL*Loader evaluates the value of each field for the input record and trims any whitespace that should be trimmed (according to existing guidelines for trimming blanks and tabs).
2. For each record, SQL*Loader evaluates any WHEN clauses for the table.
3. If the record satisfies the WHEN clauses for the table, or no WHEN clauses are specified, SQL*Loader checks each field for a NULLIF clause.
4. If a NULLIF clause exists, SQL*Loader evaluates it.
5. If the NULLIF clause is satisfied, SQL*Loader sets the field to NULL.
6. If the NULLIF clause is not satisfied, or if there is no NULLIF clause, SQL*Loader checks the length of the field from field evaluation. If the field has a length of 0 from field evaluation (for example, it was a null field, or whitespace trimming resulted in a null field), SQL*Loader sets the field to NULL. In this case, any DEFAULTIF clause specified for the field is not evaluated.
7. If any specified NULLIF clause is false or there is no NULLIF clause, and if the field does not have a length of 0 from field evaluation, SQL*Loader checks the field for a DEFAULTIF clause.
8. If a DEFAULTIF clause exists, SQL*Loader evaluates it.
9. If the DEFAULTIF clause is satisfied, then the field is set to 0 if the field in the datafile is a numeric field. It is set to NULL if the field is not a numeric field. The following fields are numeric fields and will be set to 0 if they satisfy the DEFAULTIF clause:
   - BYTEINT
   - SMALLINT
   - INTEGER

- FLOAT
- DOUBLE
- ZONED
- (packed) DECIMAL
- Numeric EXTERNAL (INTEGER, FLOAT, DECIMAL, and ZONED)

10. If the DEFAULTIF clause is not satisfied, or if there is no DEFAULTIF clause, SQL*Loader sets the field with the evaluated value from step 1.

The order in which SQL*Loader operates could cause results that you do not expect. For example, the DEFAULTIF clause may look like it is setting a numeric field to NULL rather than to 0.

Example 6–2 through Example 6–5 clarify the results for different situations. In the examples, a blank or space is indicated with a period (.). Assume that col1 and col2 are VARCHAR2(5) columns in the database.

Example 6–2 DEFAULTIF Clause Is Not Evaluated

The control file specifies:

```
(col1 POSITION (1:5),
 col2 POSITION (6:8) CHAR INTEGER EXTERNAL DEFAULTIF col1 = 'aname')
```

The datafile contains:

```
aname...
```

In Example 6–2, col1 for the row evaluates to aname. col2 evaluates to NULL with a length of 0 (it is "..." but the trailing blanks are trimmed for a positional field).

When SQL*Loader determines the final loaded value for col2, it finds no WHEN clause and no NULLIF clause. It then checks the length of the field, which is 0 from field evaluation. Therefore, SQL*Loader sets the final value for col2 to NULL. The DEFAULTIF clause is not evaluated, and the row is loaded as aname for col1 and NULL for col2.

Example 6–3 DEFAULTIF Clause Is Evaluated

The control file specifies:

```
.
.
.
```

```
PRESERVE BLANKS
.
.
.
(col1 POSITION (1:5),
 col2 POSITION (6:8) INTEGER EXTERNAL DEFAULTIF col1 = 'aname'
```

The datafile contains:

```
aname...
```

In Example 6–3, col1 for the row again evaluates to 'aname'. col2 evaluates to '...' because trailing blanks are not trimmed when PRESERVE BLANKS is specified.

When SQL*Loader determines the final loaded value for col2, it finds no WHEN clause and no NULLIF clause. It then checks the length of the field from field evaluation, which is 3, not 0.

Then SQL*Loader evaluates the DEFAULTIF clause, which evaluates to true because col1 is 'aname', which is the same as 'aname'.

Because col2 is a numeric field, SQL*Loader sets the final value for col2 to '0'. The row is loaded as 'aname' for col1 and as '0' for col2.

Example 6–4 DEFAULTIF Clause Specifies a Position

The control file specifies:

```
(col1 POSITION (1:5),
 col2 POSITION (6:8) INTEGER EXTERNAL DEFAULTIF (1:5) = BLANKS)
```

The datafile contains:

```
.....123
```

In Example 6–4, col1 for the row evaluates to NULL with a length of 0 (it is ..... but the trailing blanks are trimmed). col2 evaluates to 123.

When SQL*Loader sets the final loaded value for col2, it finds no WHEN clause and no NULLIF clause. It then checks the length of the field from field evaluation, which is 3, not 0.

Then SQL*Loader evaluates the DEFAULTIF clause. It compares (1:5) which is ..... to BLANKS, which evaluates to true. Therefore, because col2 is a numeric field (integer EXTERNAL is numeric), SQL*Loader sets the final value for col2 to 0. The row is loaded as NULL for col1 and 0 for col2.

Example 6–5 DEFAULTIF Clause Specifies a Field Name

The control file specifies:

```
(col1 POSITION (1:5),
 col2 POSITION(6:8) INTEGER EXTERNAL DEFAULTIF col1 = BLANKS)
```

The datafile contains:

```
.....123
```

In Example 6–5, col1 for the row evaluates to NULL with a length of 0 (it is ....., but the trailing blanks are trimmed). col2 evaluates to 123.

When SQL*Loader determines the final value for col2, it finds no WHEN clause and no NULLIF clause. It then checks the length of the field from field evaluation, which is 3, not 0.

Then SQL*Loader evaluates the DEFAULTIF clause. As part of the evaluation, it checks to see that col1 is NULL from field evaluation. It is NULL, so the DEFAULTIF clause evaluates to false. Therefore, SQL*Loader sets the final value for col2 to 123, its original value from field evaluation. The row is loaded as NULL for col1 and 123 for col2.

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, the target system cannot directly read data generated on the source system.

The best solution is to load data across an Oracle Net database link, taking advantage of the automatic conversion of datatypes. This is the recommended approach, whenever feasible, and means that SQL*Loader must be run on the source system.

Problems with interplatform 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 the reverse). Note, however, that incompatible datatype implementation may prevent this.

If you cannot use an Oracle Net database link and the datafile must be accessed by SQL*Loader running on the target system, it is advisable to use only the portable SQL*Loader datatypes (for example, CHAR, DATE, VARCHARC, and numeric

EXTERNAL). Datafiles written using these datatypes may be longer than those written with native datatypes. They may take more time to load, but they transport more readily across platforms.

If you know in advance that the byte ordering schemes or native integer lengths differ between the platform on which the input data will be created and the platform on which SQL*loader will be run, then investigate the possible use of the appropriate technique to indicate the byte order of the data or the length of the native integer. Possible techniques for indicating the byte order are to use the BYTEORDER parameter or to place a byte-order mark (BOM) in the file. Both methods are described in Byte Ordering on page 6-37. It may then be possible to eliminate the incompatibilities and achieve a successful cross-platform data load. If the byte order is different from the SQL*Loader default, then you must indicate a byte order.

Byte Ordering

Note: The information in this section is only applicable if you are planning to create input data on a system that has a different byte-ordering scheme than the system on which SQL*Loader will be run. Otherwise, you can skip this section.

SQL*Loader can load data from a datafile that was created on a system whose byte ordering is different from the byte ordering on the system where SQL*Loader is running, even if the datafile contains certain nonportable datatypes.

By default, SQL*Loader uses the byte order of the system where it is running as the byte order for all datafiles. For example, on a Sun Solaris system, SQL*Loader uses big endian byte order. On an Intel or an Intel-compatible PC, SQL*Loader uses little endian byte order.

Byte order affects the results when data is written and read an even number of bytes at a time (typically 2 bytes, 4 bytes, or 8 bytes). The following are some examples of this:

- The 2-byte integer value 1 is written as 0x0001 on a big endian system and as 0x0100 on a little endian system.
- The 4-byte integer 66051 is written as 0x00010203 on a big endian system and as 0x03020100 on a little endian system.

Byte order also affects character data in the UTF16 character set if it is written and read as 2-byte entities. For example, the character 'a' (0x61 in ASCII) is written as 0x0061 in UTF16 on a big endian system, but as 0x6100 on a little endian system.

All Oracle-supported character sets, except UTF16, are written one byte at a time. So, even for multibyte character sets such as UTF8, the characters are written and read the same way on all systems, regardless of the byte order of the system. Therefore, data in the UTF16 character set is nonportable because it is byte-order dependent. Data in all other Oracle-supported character sets is portable.

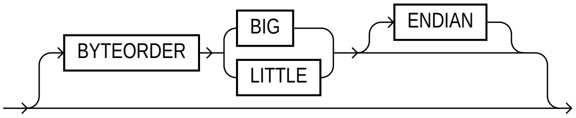
Byte order in a datafile is only an issue if the datafile that contains the byte-order-dependent data is created on a system that has a different byte order from the system on which SQL*Loader is running. If SQL*Loader knows the byte order of the data, it swaps the bytes as necessary to ensure that the data is loaded correctly in the target database. Byte swapping means that data in big endian format is converted to little endian format, or the reverse.

To indicate byte order of the data to SQL*Loader, you can use the `BYTEORDER` parameter, or you can place a byte-order mark (BOM) in the file. If you do not use one of these techniques, SQL*Loader will not correctly load the data into the datafile.

> **See Also:** Case Study 11: Loading Data in the Unicode Character Set on page 10-47 for an example of how SQL*Loader handles byte swapping

Specifying Byte Order

To specify the byte order of data in the input datafiles, use the following syntax in the SQL*Loader control file:



The `BYTEORDER` parameter has the following characteristics:

- `BYTEORDER` is placed after the `LENGTH` parameter in the SQL*Loader control file.
- It is possible to specify a different byte order for different datafiles. However, the `BYTEORDER` specification before the `INFILE` parameters applies to the entire list of primary datafiles.

- The BYTEORDER specification for the primary datafiles is also used as the default for LOBFILEs and SDFs. To override this default, specify BYTEORDER with the LOBFILE or SDF specification.
- The BYTEORDER parameter is not applicable to data contained within the control file itself.
- The BYTEORDER parameter applies to the following:
  - Binary INTEGER and SMALLINT data
  - Binary lengths in varying-length fields (that is, for the VARCHAR, VARGRAPHIC, VARRAW, and LONG VARRAW datatypes)
  - Character data for datafiles in the UTF16 character set
  - FLOAT and DOUBLE datatypes, if the system where the data was written has a compatible floating-point representation with that on the system where SQL*Loader is running
- The BYTEORDER parameter does not apply to any of the following:
  - Raw datatypes (RAW, VARRAW, or VARRAWC)
  - Graphic datatypes (GRAPHIC, VARGRAPHIC, or GRAPHIC EXTERNAL)
  - Character data for datafiles in any character set other than UTF16
  - ZONED or (packed) DECIMAL datatypes

Using Byte Order Marks (BOMs)

Datafiles that use a Unicode encoding (UTF-16 or UTF-8) may contain a byte-order mark (BOM) in the first few bytes of the file. For a datafile that uses the character set UTF16, the value 0xFEFF in the first two bytes of the file is the BOM indicating that the file contains big endian data. A value of 0xFFFE is the BOM indicating that the file contains little endian data.

If the first primary datafile uses the UTF16 character set and it also begins with a BOM, that mark is read and interpreted to determine the byte order for all primary datafiles. SQL*Loader reads and interprets the BOM, skips it, and begins processing data with the byte immediately after the BOM. The BOM setting overrides any BYTEORDER specification for the first primary datafile. BOMs in datafiles other than the first primary datafile are read and used for checking for byte-order conflicts only. They do not change the byte-order setting that SQL*Loader uses in processing the datafile.

In summary, the precedence of the byte-order indicators for the first primary datafile is as follows:

- BOM in the first primary datafile, if the datafile uses a Unicode character set that is byte-order dependent (UTF16) and a BOM is present
- `BYTEORDER` parameter value, if specified before the `INFILE` parameters
- The byte order of the system where SQL*Loader is running

For a datafile that uses a UTF8 character set, a BOM of 0xEFBBBF in the first 3 bytes indicates that the file contains UTF8 data. It does not indicate the byte order of the data, because data in UTF8 is not byte-order dependent. If SQL*Loader detects a UTF8 BOM, it skips it but does not change any byte-order settings for processing the datafiles.

SQL*Loader first establishes a byte-order setting for the first primary datafile using the precedence order just defined. This byte-order setting is used for all primary datafiles. If another primary datafile uses the character set UTF16 and also contains a BOM, the BOM value is compared to the byte-order setting established for the first primary datafile. If the BOM value matches the byte-order setting of the first primary datafile, SQL*Loader skips the BOM, and uses that byte-order setting to begin processing data with the byte immediately after the BOM. If the BOM value does not match the byte-order setting established for the first primary datafile, then SQL*Loader issues an error message and stops processing.

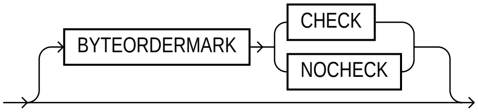
If any LOBFILEs or secondary datafiles are specified in the control file, SQL*Loader establishes a byte-order setting for each LOBFILE and secondary datafile (SDF) when it is ready to process the file. The default byte-order setting for LOBFILEs and SDFs is the byte-order setting established for the first primary datafile. This is overridden if the `BYTEORDER` parameter is specified with a LOBFILE or SDF. In either case, if the LOBFILE or SDF uses the UTF16 character set and contains a BOM, the BOM value is compared to the byte-order setting for the file. If the BOM value matches the byte-order setting for the file, SQL*Loader skips the BOM, and uses that byte-order setting to begin processing data with the byte immediately after the BOM. If the BOM value does not match, then SQL*Loader issues an error message and stops processing.

In summary, the precedence of the byte-order indicators for LOBFILEs and SDFs is as follows:

- `BYTEORDER` parameter value specified with the LOBFILE or SDF
- The byte-order setting established for the first primary datafile

Suppressing Checks for BOMs

A datafile in a Unicode character set may contain binary data that matches the BOM in the first bytes of the file. For example the integer(2) value 0xFEFF = 65279 decimal matches the big endian BOM in UTF16. In that case, you can tell SQL*Loader to read the first bytes of the datafile as data and not check for a BOM by specifying the BYTEORDERMARK parameter with the value NOCHECK. The syntax for the BYTEORDERMARK parameter is:



BYTEORDERMARK NOCHECK indicates that SQL*Loader should not check for a BOM and should read all the data in the datafile as data.

BYTEORDERMARK CHECK tells SQL*Loader to check for a BOM. This is the default behavior for a datafile in a Unicode character set. But this specification may be used in the control file for clarification. It is an error to specify BYTEORDERMARK CHECK for a datafile that uses a non-Unicode character set.

The BYTEORDERMARK parameter has the following characteristics:

- It is placed after the optional BYTEORDER parameter in the SQL*Loader control file.
- It applies to the syntax specification for primary datafiles, as well as to LOBFILEs and secondary datafiles (SDFs).
- It is possible to specify a different BYTEORDERMARK value for different datafiles; however, the BYTEORDERMARK specification before the INFILE parameters applies to the entire list of primary datafiles.
- The BYTEORDERMARK specification for the primary datafiles is also used as the default for LOBFILEs and SDFs, except that the value CHECK is ignored in this case if the LOBFILE or SDF uses a non-Unicode character set. This default setting for LOBFILEs and secondary datafiles can be overridden by specifying BYTEORDERMARK with the LOBFILE or SDF specification.

Loading All-Blank Fields

Fields that are numeric or totally blank cause the record to be rejected. To load one of these fields as NULL, use the NULLIF clause with the BLANKS parameter.

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.

A DATE field that consists entirely of blanks is loaded as a NULL field.

See Also:

- Case Study 6: Loading Data Using the Direct Path Load Method on page 10-24 for an example of how to load all-blank fields as NULL with the NULLIF clause
- Trimming Whitespace on page 6-42
- Preserving Whitespace on page 6-49

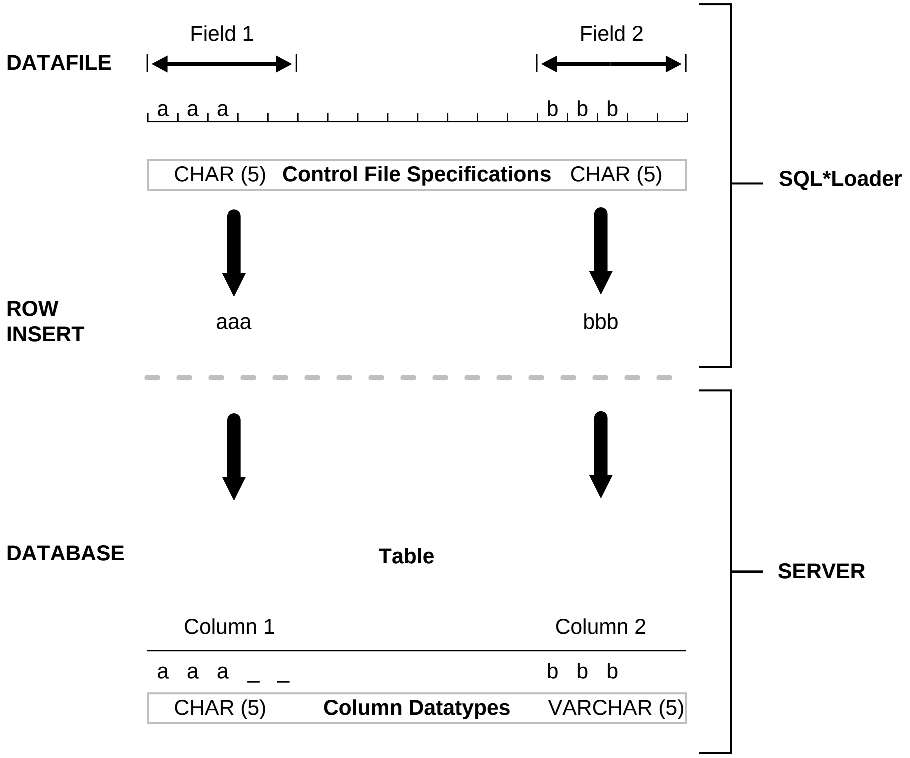
Trimming Whitespace

Blanks, tabs, and other nonprinting characters (such as carriage returns and line feeds) constitute whitespace. Leading whitespace occurs at the beginning of a field. Trailing whitespace occurs at the end of a field. Depending on how the field is specified, whitespace may or may not be included when the field is inserted into the database. This is illustrated in Figure 6–1 where two CHAR fields are defined for a data record.

The field specifications are contained in the control file. 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 or DATE column in the database, with the Oracle database server handling any necessary conversions.

By default, SQL*Loader removes trailing spaces from CHAR data before passing it to the database. So, in Figure 6–1, both field 1 and field 2 are passed to the database as 3-byte fields. However, when the data is inserted into the table, there is a difference.

Figure 6–1 Example of Field Conversion



Column 1 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 5 bytes wide. The extra space on the right is padded with blanks. Column 2, however, is defined as a varying-length field with a *maximum* length of 5 bytes. The data for that column (bbb) is left-justified as well, but the length remains 3 bytes.

Table 6–5 summarizes when and how whitespace is removed from input data fields when PRESERVE BLANKS is not specified. See Preserving Whitespace on page 6-49 for details on how to prevent whitespace trimming.

Table 6–5 Behavior Summary for Trimming Whitespace

Specification	Data	Result	Leading Whitespace Present[1]	Trailing Whitespace Present[1]
Predetermined size	__aa__	__aa	Yes	No
Terminated	__aa__,	__aa__	Yes	Yes[2]
Enclosed	"__aa__"	__aa__	Yes	Yes
Terminated and enclosed	"__aa__",	__aa__	Yes	Yes
Optional enclosure (present)	"__aa__",	__aa__	Yes	Yes
Optional enclosure (absent)	__aa__,	aa__	No	Yes
Previous field terminated by whitespace	__aa__	aa[3]	No	[3]

[1] When an all-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.

The rest of this section discusses the following topics with regard to trimming whitespace:

- Datatypes for Which Whitespace Can Be Trimmed
- Field Length Specifications for Datatypes for Which Whitespace Can Be Trimmed
- Relative Positioning of Fields
- Leading Whitespace
- Trailing Whitespace
- Enclosed Fields

Datatypes for Which Whitespace Can Be Trimmed

The information in this section applies only to fields specified with one of the character-data datatypes:

- CHAR datatype
- Datetime and interval datatypes
- Numeric EXTERNAL datatypes:
  - INTEGER EXTERNAL
  - FLOAT EXTERNAL
  - (packed) DECIMAL EXTERNAL
  - ZONED (decimal) EXTERNAL

Note: Although VARCHAR and VARCHARC fields also contain character data, these fields are never trimmed. These fields include all whitespace that is part of the field in the datafile.

Field Length Specifications for Datatypes for Which Whitespace Can Be Trimmed

There are two ways to specify field length. If a field has a constant length that is defined in the control file with a position specification or the datatype and length, 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, using either enclosure or termination delimiters.

If a position specification with start and end values is defined for a field that also has enclosure or termination delimiters defined, only the position specification has any effect. The enclosure and termination delimiters are ignored.

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 exact position of the field is not specified, the length of the field is predetermined.

Delimited Fields

Delimiters are characters that demarcate field boundaries.

Enclosure delimiters surround a field, like the quotation marks in the following example, where "__" represents blanks or tabs:

```
"__aa__"
```

Termination delimiters signal the end of a field, like the comma in the following example:

```
__aa__,
```

Delimiters are specified with the control clauses TERMINATED BY and ENCLOSED BY, as shown in the following example:

```
loc TERMINATED BY "." OPTIONALLY ENCLOSED BY '|'
```

Relative Positioning of Fields

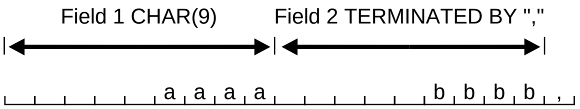
This section describes how SQL*Loader determines the starting position of a field in the following situations:

- No Start Position Specified for a Field
- Previous Field Terminated by a Delimiter
- Previous Field Has Both Enclosure and Termination Delimiters

No Start Position Specified for a Field

When a starting position is not specified for a field, it begins immediately after the end of the previous field. Figure 6–2 illustrates this situation when the previous field (Field 1) has a predetermined size.

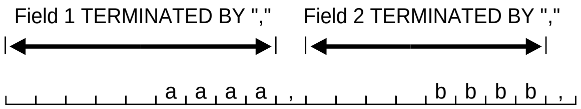
Figure 6–2 Relative Positioning After a Fixed Field



Previous Field Terminated by a Delimiter

If the previous field (Field 1) is terminated by a delimiter, then the next field begins immediately after the delimiter, as shown in Figure 6–3.

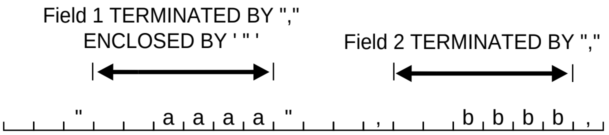
Figure 6–3 Relative Positioning After a Delimited Field



Previous Field Has Both Enclosure and Termination Delimiters

When a field is specified with both enclosure delimiters and a termination delimiter, then the next field starts after the termination delimiter, as shown in Figure 6–4. If a nonwhitespace character is found after the enclosure delimiter, but before the terminator, then SQL*Loader generates an error.

Figure 6–4 Relative Positioning After Enclosure Delimiters



Leading Whitespace

In Figure 6–4, both fields are stored with leading whitespace. Fields do *not* include leading whitespace in the following cases:

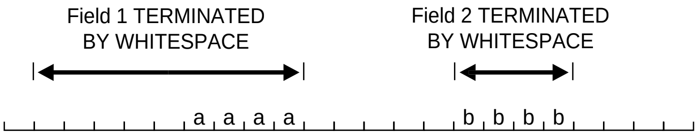
- When the previous field is terminated by whitespace, and no starting position is specified for the current field
- When optional enclosure delimiters are specified for the field, and the enclosure delimiters are *not* present

These cases are illustrated in the following sections.

Previous Field Terminated by Whitespace

If the previous field is `TERMINATED BY WHITESPACE`, then all whitespace after the field acts as the delimiter. The next field starts at the next nonwhitespace character. Figure 6–5 illustrates this case.

Figure 6–5 Fields Terminated by Whitespace



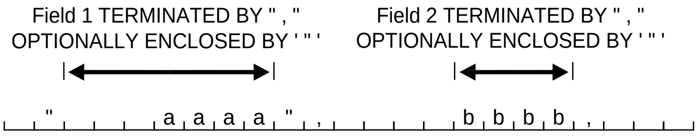
This situation occurs when the previous field is explicitly specified with the `TERMINATED BY WHITESPACE` clause, as shown in the example. It also occurs when you use the global `FIELDS TERMINATED BY WHITESPACE` clause.

Optional Enclosure Delimiters

Leading whitespace is also removed from a field when optional enclosure delimiters are specified but not present.

Whenever optional enclosure delimiters are specified, SQL*Loader scans forward, looking for the first enclosure delimiter. If an enclosure delimiter is not found, SQL*Loader skips over whitespace, eliminating it from the field. The first nonwhitespace character signals the start of the field. This situation is shown in Field 2 in Figure 6–6. (In Field 1 the whitespace is included because SQL*Loader found enclosure delimiters for the field.)

Figure 6–6 Fields Terminated by Optional Enclosure Delimiters


Field 1 TERMINATED BY " , "
OPTIONALLY ENCLOSED BY ' " '
Field 2 TERMINATED BY " , "
OPTIONALLY ENCLOSED BY ' " '
" a a a a " ,
b b b b ,

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 quotation mark in Field 1, Figure 6–6.

Trailing Whitespace

Trailing whitespace is always trimmed from character-data fields that have a predetermined size. These are the only fields for which trailing whitespace is always trimmed.

Enclosed Fields

If a field is enclosed, or terminated and enclosed, like the first field shown in Figure 6–6, then any whitespace outside the enclosure delimiters is not part of the field. Any whitespace between the enclosure delimiters belongs to the field, whether it is leading or trailing whitespace.

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 Trimming Whitespace on page 6-42.

PRESERVE BLANKS Option

The PRESERVE BLANKS option:

- Retains leading whitespace when optional enclosure delimiters are not present
- Leaves trailing whitespace intact when fields are specified with a predetermined size

For example, consider the following field, where underscores represent blanks:

```
__aa__,
```

If this field is loaded with the following control clause, then both the leading whitespace and the trailing whitespace are retained if PRESERVE BLANKS is specified. Otherwise, the leading whitespace is trimmed.

```
TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
```

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 nonblank, nontab character.

Applying SQL Operators to Fields

A wide variety of SQL operators can be applied to field data with the SQL string. This string can contain any combination of SQL expressions that are recognized by the Oracle database server as valid for the VALUES clause of an INSERT statement. In general, any SQL function that returns a single value that is compatible with the target column’s datatype can be used. SQL strings can be applied to simple scalar column types as well as to user-defined complex types such as column object and collections. See the information about expressions in the *Oracle9i SQL Reference*.

The column name and the name of the column in the SQL string must match exactly, including the quotation marks, as in the following 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 following requirements and restrictions apply when you are using SQL strings:

- The SQL string appears after any other specifications for a given column.
- The SQL string must be enclosed in double quotation marks.
- To enclose a column name in quotation marks within a SQL string, you must use escape characters.

  In the preceding example, Last is enclosed in double quotation marks to preserve the mixed case, and the double quotation marks necessitate the use of the backslash (escape) character.

- If the SQL string contains a column name that references a column object attribute, then the full field name must be used and it must be enclosed in quotation marks.
- The SQL string is evaluated after any `NULLIF` or `DEFAULTIF` clauses, but before a date mask.
- If the Oracle database server 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.
- SQL strings are required when using the `EXPRESSION` parameter in a field specification.
- If the SQL string contains a bind variable, the bind variable cannot be longer than 4000 bytes or the record will be rejected.
- The SQL string cannot reference fields that are loaded using `OID`, `SID`, `REF`, or `BFILE`. Also, it cannot reference filler fields.
- In direct path mode, a SQL string cannot reference a `VARRAY`, nested table, or `LOB` column. This also includes a `VARRAY`, nested table, or `LOB` column that is an attribute of a column object.
- The SQL string cannot be used on `RECNUM`, `SEQUENCE`, `CONSTANT`, or `SYSDATE` fields.
- The SQL string cannot be used on LOBs, `BFILE`s, `XML` columns, or a file that is an element of a collection.
- In direct path mode, the final result that is returned after evaluation of the expression in the SQL string must be a scalar datatype. That is, the expression may not return an object or collection datatype when performing a direct path load.

Referencing Fields

To refer to fields in the record, precede the field name with a colon (:). Field values from the current record are substituted. A field name preceded by a colon (:) in a SQL string is also referred to as a bind variable. The following example illustrates how a reference is made to both the current field and to other fields in the control file:

```
LOAD DATA
INFILE *
APPEND INTO TABLE YYY
(
 field1  POSITION(1:6) CHAR "LOWER(:field1)"
 field2  CHAR TERMINATED BY ','
         NULLIF ((1) = 'a') DEFAULTIF ((1)= 'b')
         "RTRIM(:field2)"
 field3  CHAR(7) "TRANSLATE(:field3, ':field1', ':1')",
 field4  COLUMN OBJECT
 (
  attr1  CHAR(3)  "UPPER(":\"FIELD4.ATTR3\")",
  attr2  CHAR(2),
  attr3  CHAR(3)  ":\"FIELD4.ATTR1\" + 1"
 ),
 field5  EXPRESSION "MYFUNC(:FIELD4, SYSDATE)"
)
BEGINDATA
ABCDEF1234511  ,:field1500YYabc
abcDEF67890    ,:field2600ZZghl
```

Note the following about the preceding example:

- Only the `:field1` that is not in single quotation marks is interpreted as a bind variable; `':field1'` and `':1'` are text literals that are passed unchanged to the TRANSLATE function. For more information on the use of quotation marks inside quoted strings, see Specifying Filenames and Object Names on page 5-5.
- For each input record read, the value of the field referenced by the bind variable will be substituted for the bind variable. For example, the value ABCDEF in the first record is mapped to the first field `:field1`. This value is then passed as an argument to the LOWER function.
- When a bind variable is a reference to an attribute of a column object, the full field name must be in uppercase and enclosed in quotation marks. For instance, `:\"FIELD4.ATTR1\"` and `:\"FIELD4.ATTR3\"`.

- A bind variable in a SQL string need not reference the current field. In the preceding example, the bind variable in the SQL string for field FIELD4.ATTR1 references field FIELD4.ATTR3. The field FIELD4.ATTR1 is still mapped to the values 500 and 600 in the input records, but the final values stored in its corresponding columns are ABC and GHL.
- field5 is not mapped to any field in the input record. The value that is stored in the target column is the result of executing the MYFUNC PL/SQL function, which takes two arguments. The use of the EXPRESSION parameter requires that a SQL string be used to compute the final value of the column because no input data is mapped to the field.

Common Uses of SQL Operators in Field Specifications

SQL operators are commonly used for the following tasks:

- 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 SQL 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 CHAR(10)
       "NVL( LTRIM(RTRIM(:field1)), 'unknown' )"
```

Using SQL Strings with a Date Mask

When a SQL string is used with a date mask, the date mask is evaluated after the SQL string. Consider a field specified as follows:

```
field1 DATE "dd-mon-yy" "RTRIM(:field1)"
```

SQL*Loader internally generates and inserts the following:

```
TO_DATE(RTRIM(<field1_value>), 'dd-mon-yyyy')
```

Note that when using the DATE field datatype, it is not possible to have a SQL string without a date mask. This is because SQL*Loader assumes that the first quoted string it finds after the DATE parameter is a date mask. For instance, the following field specification would result in an error (ORA-01821: date format not recognized):

```
field1 DATE "RTRIM(TO_DATE(:field1, 'dd-mon-yyyy'))"
```

In this case, a simple workaround is to use the CHAR datatype.

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')"
```

This example 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 Study 7: Extracting Data from a Formatted Report on page 10-28 to load data from a formatted report.

Using SQL*Loader to Generate Data for Input

The parameters described in this section provide the means for SQL*Loader to generate the data stored in the database record, rather than reading it from a datafile. The following parameters are described:

- CONSTANT Parameter
- EXPRESSION Parameter
- RECNUM Parameter
- SYSDATE Parameter
- SEQUENCE Parameter

Loading Data Without Files

It is possible to use SQL*Loader to generate data by specifying only sequences, record numbers, system dates, constants, and SQL string expressions as field specifications.

SQL*Loader inserts as many records as are specified by the LOAD statement. The SKIP parameter is not permitted in this situation.

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 or between loads.

CONSTANT Parameter

To set a column to a constant value, use 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 you must do so if it contains whitespace or reserved words. Be sure to specify a legal value for the target column. If the value is bad, every record is rejected.

Numeric values larger than 2^32 - 1 (4,294,967,295) must be enclosed in quotation marks.

Note: Do not use the CONSTANT parameter 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 record. The combination of CONSTANT and a value is a complete column specification.

Setting a Column to an Expression Value

Use the EXPRESSION parameter after a column name to set that column to the value returned by a SQL operator or specially written PL/SQL function. The operator or function is indicated in a SQL string that follows the EXPRESSION parameter. Any arbitrary expression may be used in this context provided that any parameters required for the operator or function are correctly specified and that the result returned by the operator or function is compatible with the datatype of the column being loaded.

EXPRESSION Parameter

The combination of column name, EXPRESSION parameter, and a SQL string is a complete field specification.

```
column_name EXPRESSION "SQL string"
```

Setting a Column to the Datafile Record Number

Use the RECNUM parameter after a column name to set that column to the number of the logical record from which that record 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 Parameter

The combination of column name and RECNUM 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 parameter. See the section on the DATE datatype in *Oracle9i SQL Reference*.

SYSDATE Parameter

The combination of column name and the SYSDATE parameter 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 loaded only in that form. If the system date is loaded into a DATE column, then it can be loaded 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 parameter 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 Parameter

The combination of column name and the SEQUENCE parameter is a complete column specification.

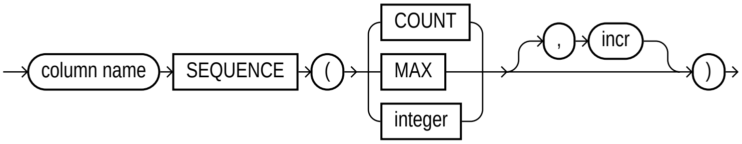


Table 6–6 describes the parameters used for column specification.

Table 6–6 Parameters Used for Column Specification

Parameter	Description
column_name	The name of the column in the database to which to assign the sequence
SEQUENCE	Use the SEQUENCE parameter to specify the value for a column
COUNT	The sequence starts with the number of records already in the table plus the increment
MAX	The sequence starts with the current maximum value for the column plus the increment
integer	Specifies the specific sequence number to begin with

Table 6–6 (Cont.) Parameters Used for Column Specification

Parameter	Description
incr	The value that the sequence number is to increment after a record is loaded or rejected. This is optional. The default is 1.

If a record 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, and 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 Study 3: Loading a Delimited, Free-Format File on page 10-11 provides an example of the SEQUENCE parameter.

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.

Sometimes, however, 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. 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. Use the number of table-inserts per record as the sequence increment and start the sequence numbers for each insert with successive numbers.

Example: Generating Different Sequence Numbers for Each Insert

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.

7

Loading Objects, LOBs, and Collections

This chapter discusses the following topics:

Loading Column Objects

Column objects in the control file are described in terms of their attributes. If the object type on which the column object is based is declared to be nonfinal, then the column object in the control file may be described in terms of the attributes, both derived and declared, of any subtype derived from the base object type. In the datafile, the data corresponding to each of the attributes of a column object is in a data field similar to that corresponding to a simple relational column.

Note: 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`, and so on), 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 you specify `fld2` in a clause such as `NULLIF`, you must use the full field name `fld1.fld2`.

The following sections show examples of loading column objects:

- Loading Column Objects in Stream Record Format
- Loading Column Objects in Variable Record Format
- Loading Nested Column Objects
- Loading Column Objects with a Derived Subtype
- Specifying Null Values for Objects
- Loading Column Objects with User-Defined Constructors

Loading Column Objects in Stream Record Format

Example 7–1 shows a case in which the data is in predetermined size fields. The newline character marks the end of a physical record. You can also mark the end of a physical record by using a custom record separator in the operating system file-processing clause (`os_file_proc_clause`).

Example 7–1 Loading Column Objects in Stream Record Format

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) )
```

Datafile (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 to describe nested column objects.

Loading Column Objects in Variable Record Format

Example 7–2 shows a case in which the data is in delimited fields.

Example 7–2 Loading Column Objects in Variable Record Format

Control File Contents

```
LOAD DATA
1 INFILE 'sample.dat' "var 6"
INTO TABLE departments
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
2  (dept_no
   dept_name,
   dept_mgr       COLUMN OBJECT
      (name       CHAR(30),
      age         INTEGER EXTERNAL(5),
      emp_id      INTEGER EXTERNAL(5)) )
```

Datafile (sample.dat)

```
3  000034101,Mathematics,Johny Q.,30,1024,
   000039237,Physics,"Albert Einstein",65,0000,
```

Notes

1. The "var" string includes the number of bytes in the length field at the beginning of each record (in this example, the number is 6). If no value is specified, the default is 5 bytes. The maximum size of a variable record is 2^32-1. Specifying larger values will result in an error.

2. 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 an omitted type specification defaults to CHAR of length 255.
3. The first six bytes (italicized) specify the length of the forthcoming record. These length specifications include the newline characters, which are ignored thanks to the terminators after the emp_id field.

Loading Nested Column Objects

Example 7–3 shows a control file describing nested column objects (one column object nested in another column object).

Example 7–3 Loading Nested Column Objects

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))))
```

Datafile (sample.dat)

```
101,Mathematics,Johny Q.,30,1024,"Barbie",650-251-0010,
237,Physics,"Albert Einstein",65,0000,Wife Einstein,654-3210,
```

Notes

1. This entry specifies a column object nested within a column object.

Loading Column Objects with a Derived Subtype

Example 7–4 shows a case in which a nonfinal base object type has been extended to create a new derived subtype. Although the column object in the table definition is declared to be of the base object type, SQL*Loader allows any subtype to be loaded into the column object, provided that the subtype is derived from the base object type.

Example 7–4 Loading Column Objects with a Subtype

Object Type Definitions

```
CREATE TYPE person_type AS OBJECT
  (name     VARCHAR(30),
   ssn      NUMBER(9)) not final;

CREATE TYPE employee_type UNDER person_type
  (empid    NUMBER(5));

CREATE TABLE personnel
  (deptno   NUMBER(3),
   deptname VARCHAR(30),
   person   person_type);
```

Control File Contents

```
LOAD DATA
INFILE ’sample.dat’
INTO TABLE personnel
FIELDS TERMINATED BY ’,’ OPTIONALLY ENCLOSED BY ’"’
   (deptno        INTEGER EXTERNAL(3),
    deptname      CHAR,
1   person        COLUMN OBJECT TREAT AS employee_type
      (name       CHAR,
       ssn        INTEGER EXTERNAL(9),
2      empid      INTEGER EXTERNAL(5)))
```

Datafile (sample.dat)

```
101,Mathematics,Johny Q.,301189453,10249,
237,Physics,"Albert Einstein",128606590,10030,
```

Notes

1. The TREAT AS clause indicates that SQL*Loader should treat the column object `person` as if it were declared to be of the derived type `employee_type`, instead of its actual declared type, `person_type`.

2. The empid attribute is allowed here because it is an attribute of the employee_type. If the TREAT AS clause had not been specified, this attribute would have resulted in an error, because it is not an attribute of the column's declared type.

Specifying Null Values for Objects

Specifying null values for nonscalar 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 column objects, you can use the NULLIF clause to specify the field conditions under which a particular attribute should be initialized to NULL. Example 7–5 demonstrates this.

Example 7–5 Specifying Attribute Nulls Using the NULLIF Clause

Control File Contents

```
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))
```

Datafile (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 a 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 Example 7–5, 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 preceding 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.

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 field in the field condition of the NULLIF clause of the particular object. This is shown in Example 7–6.

Example 7–6 Loading Data Using Filler 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)
```

Datafile (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 the is_null field is blank.

See Also: Specifying Filler Fields on page 6-6

Loading Column Objects with User-Defined Constructors

The Oracle9*i* database server automatically supplies a default constructor for every object type. This constructor requires that all attributes of the type be specified as arguments in a call to the constructor. When a new instance of the object is created, its attributes take on the corresponding values in the argument list. This constructor is known as the attribute-value constructor. SQL*Loader uses the attribute-value constructor by default when loading column objects.

It is possible to override the attribute-value constructor by creating one or more user-defined constructors. When you create a user-defined constructor, you must supply a type body that performs the user-defined logic whenever a new instance of the object is created. A user-defined constructor may have the same argument list as the attribute-value constructor but differ in the logic that its type body implements.

When the argument list of a user-defined constructor function matches the argument list of the attribute-value constructor, there is a difference in behavior between conventional and direct path SQL*Loader. Conventional path mode results in a call to the user-defined constructor. Direct path mode results in a call to the attribute-value constructor. Example 7–7 illustrates this difference.

Example 7–7 Loading a Column Object with a User-Defined Constructor That Matches the Attribute-Value Constructor

Object Type Definitions

```
CREATE TYPE person_type AS OBJECT
     (name     VARCHAR(30),
      ssn      NUMBER(9)) not final;

   CREATE TYPE employee_type UNDER person_type
     (empid    NUMBER(5),
   -- User-defined constructor that looks like an attribute-value constructor
      CONSTRUCTOR FUNCTION
     employee_type (name VARCHAR2, ssn NUMBER, empid NUMBER)
     RETURN SELF AS RESULT);

CREATE TYPE BODY employee_type AS
  CONSTRUCTOR FUNCTION
     employee_type (name VARCHAR2, ssn NUMBER, empid NUMBER)
   RETURN SELF AS RESULT AS
-- This UDC makes sure that the name attribute is in uppercase.
   BEGIN
     SELF.name  := UPPER(name);
     SELF.ssn   := ssn;
     SELF.empid := empid;
     RETURN;
   END;

CREATE TABLE personnel
  (deptno   NUMBER(3),
   deptname VARCHAR(30),
   employee employee_type);
```

Control File Contents

```
LOAD DATA
   INFILE *
   REPLACE
   INTO TABLE personnel
   FIELDS TERMINATED BY ’,’ OPTIONALLY ENCLOSED BY ’"’
      (deptno        INTEGER EXTERNAL(3),
       deptname      CHAR,
       employee      COLUMN OBJECT
         (name       CHAR,
          ssn        INTEGER EXTERNAL(9),
          empid      INTEGER EXTERNAL(5)))

   BEGINDATA
1  101,Mathematics,Johny Q.,301189453,10249,
   237,Physics,"Albert Einstein",128606590,10030,
```

Notes

1. When this control file is run in conventional path mode, the name fields, `Johny Q.` and `Albert Einstein`, are both loaded in uppercase. This is because the user-defined constructor is called in this mode. In contrast, when this control file is run in direct path mode, the name fields are loaded exactly as they appear

in the input data. This is because the attribute-value constructor is called in this mode.

It is possible to create a user-defined constructor whose argument list does not match that of the attribute-value constructor. In this case, both conventional and direct path modes will result in a call to the attribute-value constructor. Consider the definitions in Example 7–8.

Example 7–8 Loading a Column Object with a User-Defined Constructor That Does Not Match the Attribute-Value Constructor

Object Type Definitions

```
CREATE SEQUENCE employee_ids
    START     WITH  1000
    INCREMENT BY    1;

   CREATE TYPE person_type AS OBJECT
     (name     VARCHAR(30),
      ssn      NUMBER(9)) not final;

   CREATE TYPE employee_type UNDER person_type
     (empid    NUMBER(5),
   -- User-defined constructor that does not look like an attribute-value
   -- constructor
      CONSTRUCTOR FUNCTION
        employee_type (name VARCHAR2, ssn NUMBER)
        RETURN SELF AS RESULT);

   CREATE TYPE BODY employee_type AS
     CONSTRUCTOR FUNCTION
        employee_type (name VARCHAR2, ssn NUMBER)
      RETURN SELF AS RESULT AS
   -- This UDC makes sure that the name attribute is in lowercase and
   -- assigns the employee identifier based on a sequence.
        nextid     NUMBER;
        stmt       VARCHAR2(64);
      BEGIN

        stmt := 'SELECT employee_ids.nextval FROM DUAL';
        EXECUTE IMMEDIATE stmt INTO nextid;

        SELF.name  := LOWER(name);
        SELF.ssn   := ssn;
        SELF.empid := nextid;
        RETURN;
      END;

   CREATE TABLE personnel
     (deptno   NUMBER(3),
      deptname VARCHAR(30),
      employee employee_type);
```

If the control file described in Example 7–7 is used with these definitions, then the name fields are loaded exactly as they appear in the input data (that is, in mixed case). This is because the attribute-value constructor is called in both conventional and direct path modes.

It is still possible to load this table using conventional path mode by explicitly making reference to the user-defined constructor in a SQL expression. Example 7–9 shows how this can be done.

Example 7–9   Loading a Column Object with a User-Defined Constructor That Does Not Match the Attribute-Value Constructor by Using a SQL Expression

Control File Contents

```
LOAD DATA
   INFILE *
   REPLACE
   INTO TABLE personnel
   FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
      (deptno        INTEGER EXTERNAL(3),
       deptname      CHAR,
       name          BOUNDFILLER CHAR,
       ssn           BOUNDFILLER INTEGER EXTERNAL(9),
       employee      EXPRESSION "employee_type(:NAME, :SSN)")

   BEGINDATA
1  101,Mathematics,Johny Q.,301189453,
   237,Physics,"Albert Einstein",128606590,
```

Notes

1. The employee column object is now loaded using a SQL expression. This expression invokes the user-defined constructor with the correct number of arguments. The name fields, `Johny Q.` and `Albert Einstein`, will both be loaded in lowercase. In addition, the employee identifiers for each row's employee column object will have taken their values from the `employee_ids` sequence.

If the control file in Example 7–9 is used in direct path mode, the following error is reported:

```
SQL*Loader-951: Error calling once/load initialization
ORA-26052: Unsupported type 121 for SQL expression on column EMPLOYEE.
```

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 7–10 demonstrates loading an object table with primary key object identifiers (OIDs).

Example 7–10 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))
```

Datafile (sample.dat)

```
Johny Quest, 18, 007,
Speed Racer, 16, 000,
```

By looking only at the preceding 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.

You may want to load data that already contains real OIDs and 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)
```

In this clause, *`fieldname`* is the name of one of the fields (typically a filler field) from the field specification list that is mapped to a data field 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, to ensure uniqueness, you should use the Oracle OID generator to generate the OIDs to be loaded.

The OID clause can only be used for system-generated OIDs, not primary key OIDs.

Example 7–11 demonstrates loading real OIDs with the row-objects.

Example 7–11 Loading OIDs

Control File Contents

```
   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))
```

Datafile (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. 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 a 32-digit hexadecimal number. The 32-digit hexadecimal number is later converted into a 16-byte `RAW` and stored in the object table.

Loading Object Tables with a Subtype

If an object table’s row object is based on a nonfinal type, SQL*Loader allows for any derived subtype to be loaded into the object table. As previously mentioned, the syntax required to load an object table with a derived subtype is almost identical to that used for a typical relational table. However, in this case, the actual subtype to be used must be named, so that SQL*Loader can determine if it is a valid subtype for the object table. This concept is illustrated in Example 7–12.

Example 7–12 Loading an Object Table with a Subtype

Object Type Definitions

```
CREATE TYPE employees_type AS OBJECT
  (name     VARCHAR2(30),
   age      NUMBER(3),
   emp_id   NUMBER(5)) not final;

CREATE TYPE hourly_emps_type UNDER employees_type
  (hours    NUMBER(3));

CREATE TABLE employees_v3 of employees_type;
```

Control File Contents

```
   LOAD DATA

   INFILE 'sample.dat'
   INTO TABLE employees_v3
1  TREAT AS hourly_emps_type
   FIELDS TERMINATED BY ','
     (name     CHAR(30),
      age      INTEGER EXTERNAL(3),
      emp_id   INTEGER EXTERNAL(5),
2     hours    INTEGER EXTERNAL(2))
```

Datafile (sample.dat)

```
   Johny Quest, 18, 007, 32,
   Speed Racer, 16, 000, 20,
```

Notes

1. The TREAT AS clause indicates that SQL*Loader should treat the object table as if it were declared to be of type `hourly_emps_type`, instead of its actual declared type, `employee_type`.

2. The `hours` attribute is allowed here because it is an attribute of the `hourly_emps_type`. If the TREAT AS clause had not been specified, this attribute would have resulted in an error, because it is not an attribute of the object table's declared type.

Loading REF Columns

SQL*Loader can load real REF columns (REFs containing real OIDs of the referenced objects), primary key REF columns, and unscoped REF columns that allow primary keys.

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 clause.

The REF clause 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 A-7 for the appropriate syntax. Example 7–13 demonstrates real REF loading.

Example 7–13 Loading Real REF Columns

Control File Contents

```
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))
```

Datafile (sample.dat)

```
22345, QuestWorld, 21E978406D3E41FCE03400400B403BC3, EMPLOYEES_V2,
23423, Geography, 21E978406D4441FCE03400400B403BC3, EMPLOYEES_V2,
```

Notes

1. If the specified table does not exist, the record is rejected. 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 clause. The REF clause takes for arguments a comma-separated list of field names and 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 A-7 for the appropriate syntax.

SQL*Loader assumes that the ordering of the arguments matches the relative ordering of the columns making up the primary key OID in the referenced table. Example 7–14 demonstrates loading primary key REF columns.

Example 7–14 Loading Primary Key REF Columns

Control File Contents

```
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))
```

Datafile (sample.dat)

```
22345, QuestWorld, 007,
23423, Geography, 000,
```

Unscoped REF Columns That Allow Primary Keys

An unscoped REF column that allows primary keys can reference both system-generated and primary key REFs. The syntax for loading into such a REF column is the same as if you were loading into a real REF column or into a primary key REF column. See Example 7–13, "Loading Real REF Columns" and Example 7–14, "Loading Primary Key REF Columns".

The following restrictions apply when loading into an unscoped REF column that allows primary keys:

- Only one type of REF can be referenced by this column during a single-table load, either system-generated or primary key, but not both. If you try to reference both types, the data row will be rejected with an error message indicating that the referenced table name is invalid.

- If you are loading unscoped primary key REFs to this column, only one object table can be referenced during a single-table load. That is, if you want to load unscoped primary key REFs, some pointing to object table X and some pointing to object table Y, you would have to do one of the following:
  - perform two single-table loads
  - perform a single load using multiple INTO TABLE clauses for which the WHEN clause keys off some aspect of the data, such as the object table name for the unscoped primary key REF. For example:

```
LOAD DATA
INFILE 'data.dat'

INTO TABLE orders_apk
APPEND
when CUST_TBL = "CUSTOMERS_PK"
fields terminated by ","
(
  order_no   position(1)  char,
  cust_tbl FILLER     char,
  cust_no  FILLER     char,
  cust   REF (cust_tbl, cust_no) NULLIF order_no='0'
)

INTO TABLE orders_apk
APPEND
when CUST_TBL = "CUSTOMERS_PK2"
fields terminated by ","
(
  order_no  position(1)  char,
  cust_tbl FILLER     char,
  cust_no  FILLER     char,
  cust   REF (cust_tbl, cust_no) NULLIF order_no='0'
)
```

  If you do not use either of these methods, the data row will be rejected with an error message indicating that the referenced table name is invalid.

- Unscoped primary key REFs in collections are not supported by SQL*Loader.
- If you are loading system-generated REFs into this REF column, any limitations described in Real REF Columns on page 7-15 also apply here.

- If you are loading primary key REFs into this REF column, any limitations described in Primary Key REF Columns on page 7-16 also apply here.

Note: For an unscoped REF column that allows primary keys, SQL*Loader takes the first valid object table parsed (either from the REF directive or from the data rows) and uses that object table's OID type to determine the REF type that can be referenced in that single-table load.

Loading LOBs

A LOB is a *large object type*. SQL*Loader supports the following types of LOBs:

- BLOB: an internal LOB containing unstructured binary data
- CLOB: an internal LOB containing character data
- NCLOB: an internal LOB containing characters from a national character set
- BFILE: a BLOB stored outside of the database tablespaces in a server-side operating system 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."

XML columns are columns declared to be of type SYS.XMLTYPE. SQL*Loader treats XML columns as if they were CLOBs. All of the methods described in the following sections for loading LOB data from the primary datafile or from LOBFILEs are applicable to loading XML columns.

Note: You cannot specify a SQL string for LOB fields. This is true even if you specify LOBFILE_spec.

Because LOBs can be quite large, SQL*Loader is able to load LOB data from either a primary datafile (in line with the rest of the data) or from LOBFILEs. This section addresses the following topics:

- Loading LOB Data from a Primary Datafile
- Loading LOB Data from an External LOBFILE (BFILE)
- Loading LOB Data from LOBFILEs

Loading LOB Data from a Primary Datafile

To load internal LOBs (BLOBs, CLOBs, and NCLOBs) or XML columns from a primary datafile, you can use the following standard SQL*Loader formats:

- Predetermined size fields
- Delimited fields
- Length-value pair fields

Each of these formats is described in the following sections.

LOB Data in Predetermined Size Fields

This is a very fast and conceptually simple format in which to load LOBs, as shown in Example 7–15.

> **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 data field.

To load LOBs using this format, you should use either CHAR or RAW as the loading datatype.

Example 7–15 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)
```

Datafile (sample.dat)

```
Johny Quest      Johny Quest
             500 Oracle Parkway
             jquest@us.oracle.com ...
```

Notes

1. If the data field 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. 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 a problem. However, this added flexibility can affect performance, because SQL*Loader must scan through the data, looking for the delimiter string.

As with single-character delimiters, when you specify string delimiters, you should consider 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 notation (that is, *X'hexadecimal string'*). If the delimiters are specified in hexadecimal notation, the specification must consist of characters that are valid in the character set of the input datafile. In contrast, if hexadecimal notation is not used, the delimiter specification is considered to be in the client's (that is, the control file's) character set. In this case, the delimiter is converted into the datafile's character set before SQL*Loader searches for the delimiter in the datafile.

Note the following:

- Stutter syntax is supported with string delimiters (that is, the closing enclosure delimiter can be stuttered).
- Leading whitespaces in the initial multicharacter enclosure delimiter are not allowed.
- If a field is terminated by WHITESPACE, the leading whitespaces are trimmed.

Example 7–16 shows an example of loading LOB data in delimited fields.

Example 7–16 Loading LOB Data in Delimited Fields

Control File Contents

```
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>')
```

Datafile (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. With the default byte-length semantics, the maximum length for a LOB that can be read using `CHAR(507)` is 507 bytes. If character-length semantics were used, the maximum would be 507 characters. See Character-Length Semantics on page 5-22.

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 hexadecimal notation, `X'7C0A'`).

LOB Data in Length-Value Pair Fields

You can use VARCHAR, VARCHARC, or VARRAW datatypes to load LOB data organized in length-value pair fields. 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 7–17 demonstrates loading LOB data in length-value pair fields.

Example 7–17 Loading LOB Data in Length-Value Pair Fields

Control File Contents

```
  LOAD DATA
1 INFILE 'sample.dat' "str '<endrec>\n'"
  INTO TABLE person_table
  FIELDS TERMINATED BY ','
     (name       CHAR(25),
2    "RESUME"    VARCHARC(3,500))
```

Datafile (sample.dat)

```
  Johny Quest,479                Johny Quest
                             500 Oracle Parkway
                             jquest@us.oracle.com
                                    ... <endrec>
3    Speed Racer,000<endrec>
```

Notes

1. If the backslash escape character is not supported, the string used as a record separator in the example could be expressed in hexadecimal notation.
2. "RESUME" is a field that corresponds to a CLOB column. In the control file, it is a VARCHARC, whose length field is 3 bytes long and whose maximum size is 500 bytes (with byte-length semantics). If character-length semantics were used, the length would be 3 characters and the maximum size would be 500 characters. See Character-Length Semantics on page 5-22.
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 from an External LOBFILE (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. The file 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 file system). For more information, see the *Oracle9i Application Developer's Guide - Large Objects (LOBs)*.

A control file field corresponding to a BFILE column consists of a column name followed by the BFILE clause. The BFILE clause takes as arguments a DIRECTORY OBJECT (the server_directory alias) name followed by a BFILE name. Both arguments can be provided as string constants, or they can be dynamically loaded through some other field. See the *Oracle9i SQL Reference* for more information.

In the next two examples of loading BFILEs, Example 7–18 has only the filename specified dynamically, while Example 7–19 demonstrates specifying both the BFILE and the DIRECTORY OBJECT dynamically.

Example 7–18 Loading Data Using BFILEs: Only Filename Specified Dynamically

Control File Contents

```
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))
```

Datafile (sample.dat)

```
1,Mercury,mercury.jpeg,
2,Venus,venus.jpeg,
3,Earth,earth.jpeg,
```

Notes

1. The directory name is quoted; therefore, the string is used as is and is not capitalized.

Example 7–19 Loading Data Using BFILEs: Filename and Directory Name Specified Dynamically

Control File Contents

```
LOAD DATA
INFILE sample.dat
INTO TABLE planets
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
   (pl_id    NUMBER(4),
   pl_name   CHAR(20),
   fname     FILLER CHAR(30),
1  dname     FILLER CHAR(20),
   pl_pict   BFILE(dname, fname) )
```

Datafile (sample.dat)

```
1, Mercury, mercury.jpeg, scott_dir1,
2, Venus, venus.jpeg, scott_dir1,
3, Earth, earth.jpeg, scott_dir2,
```

Notes

1. dname is mapped to the datafile field containing the directory name corresponding to the file being loaded.

Loading LOB Data from LOBFILEs

LOB data can be lengthy enough so that it makes sense to load it from a LOBFILE instead of from a primary datafile. 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.

There is no requirement that a LOB from a LOBFILE fit in memory. SQL*Loader reads LOBFILEs in 64 KB chunks.

In LOBFILEs the data can be in any of the following types of fields:

- A single LOB field into which the entire contents of a file can be read
- Predetermined size fields (fixed-length fields)
- Delimited fields (that is, `TERMINATED BY` or `ENCLOSED BY`)

  The clause `PRESERVE BLANKS` is not applicable to fields read from a LOBFILE.
- Length-value pair fields (variable-length fields)—to load data from this type of field, use the `VARRAW`, `VARCHAR`, or `VARCHARC` SQL*Loader datatypes

See Examples of Loading LOB Data from LOBFILEs on page 7-24 for examples of using each of these field types. All of the previously mentioned field types can be used to load `XML` columns.

See lobfile_spec on page A-8 for LOBFILE syntax.

Dynamic Versus Static LOBFILE Specifications

You can specify LOBFILEs either statically (the name of the file is specified in the control file) or dynamically (a `FILLER` field is used as the source of the filename). In either case, if the LOBFILE is *not* terminated by EOF, then when the end of the LOBFILE is reached, the file is closed and further attempts to read data from that file produce results equivalent to reading data from an empty field.

However, if you have a LOBFILE that *is* terminated by EOF, then the entire file is always returned on each attempt to read data from that file.

You should not specify the same LOBFILE as the source of two different fields. If you do so, typically, the two fields will read the data independently.

Examples of Loading LOB Data from LOBFILEs

This section contains examples of loading data from different types of fields in LOBFILEs.

One LOB per File In Example 7–20, each LOBFILE is the source of a single LOB. To load LOB data that is organized in this way, you follow the column or field name with the LOBFILE datatype specifications.

Example 7–20   Loading LOB DATA with One LOB per LOBFILE

Control File Contents

```
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)
```

Datafile (sample.dat)

```
Johny Quest,jqresume.txt,
Speed Racer,'/private/sracer/srresume.txt',
```

Secondary Datafile (jqresume.txt)

```
             Johny Quest
         500 Oracle Parkway
            ...
```

Secondary Datafile (srresume.txt)

```
         Speed Racer
     400 Oracle Parkway
        ...
```

Notes

1. The filler field is mapped to the 40-byte data field, which is read using the SQL*Loader `CHAR` datatype. This assumes the use of default byte-length semantics. If character-length semantics were used, the field would be mapped to a 40-character data field.
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. If no existing LOBFILE is specified, the `"RESUME"` field is initialized to empty.

Predetermined Size LOBs In Example 7–21, 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. However, it is often difficult to guarantee that all LOBs are the same size.

Example 7–21 Loading LOB Data Using Predetermined Size LOBs

Control File Contents

```
LOAD DATA
INFILE 'sample.dat'
INTO TABLE person_table
FIELDS TERMINATED BY ','
   (name     CHAR(20),
1  "RESUME"    LOBFILE(CONSTANT '/usr/private/jquest/jqresume.txt')
               CHAR(2000))
```

Datafile (sample.dat)

```
Johny Quest,
Speed Racer,
```

Secondary Datafile (jqresume.txt)

```
             Johny Quest
         500 Oracle Parkway
            ...
             Speed Racer
         400 Oracle Parkway
            ...
```

Notes

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. This assumes the use of the default byte-length semantics. If character-length semantics were used, SQL*Loader would load 2000 characters of data, starting from the first character after the last-loaded character. See Character-Length Semantics on page 5-22.

Delimited LOBs In Example 7–22, the LOB data instances in the LOBFILE are delimited. In this format, loading different size LOBs into the same column is not a problem. However, this added flexibility can affect performance, because SQL*Loader must scan through the data, looking for the delimiter string.

Example 7–22 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")
```

Datafile (sample.dat)

```
Johny Quest,
Speed Racer,
```

Secondary Datafile (jqresume.txt)

```
             Johny Quest
         500 Oracle Parkway
            ... <endlob>
             Speed Racer
         400 Oracle Parkway
            ... <endlob>
```

Notes

1. Because a maximum length of 2000 is specified for CHAR, SQL*Loader knows what to expect as the maximum length of the field, which can result in memory usage optimization. *If you choose to specify a maximum 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 Example 7–23 each LOB in the LOBFILE is preceded by its length. You could use VARCHAR, VARCHARC, or VARRAW datatypes to load LOB data organized in this way.

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 7–23 Loading LOB Data Using Length-Value Pair Specified LOBs

Control File Contents

```
LOAD DATA
INFILE 'sample.dat'
```

```
INTO TABLE person_table
FIELDS TERMINATED BY ','
   (name           CHAR(20),
1  "RESUME"       LOBFILE(CONSTANT 'jqresume') VARCHARC(4,2000))
```

Datafile (sample.dat)

```
Johny Quest,
Speed Racer,
```

Secondary Datafile (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 the first 4 bytes should be interpreted as the length. The value of `2000` tells SQL*Loader that the maximum size of the field is 2000 bytes. This assumes the use of the default byte-length semantics. If character-length semantics were used, the first 4 characters would be interpreted as the length in characters. The maximum size of the field would be 2000 characters. See Character-Length Semantics on page 5-22.

2. The entry `0501` preceding `Johny Quest` tells SQL*Loader that the LOB consists of the next 501 characters.

3. This entry specifies an empty (not null) LOB.

Considerations When Loading LOBs from LOBFILEs

Keep in mind the following when you load data using LOBFILEs:

- Only LOBs and `XML` columns can be loaded from LOBFILEs.
- The failure to load a particular LOB does not result in the rejection of the record containing that LOB. Instead, you will have a record that contains an empty LOB. In the case of an `XML` column, a null value will be inserted if there is a failure loading the LOB.
- It is not necessary to specify the maximum length of a field corresponding to a `LOB` type column; nevertheless, if a maximum length is specified, SQL*Loader uses it as a hint to optimize memory usage. Therefore, it is important that the maximum length specification does not understate the true maximum length.

- You cannot supply a position specification (pos_spec) when loading data from a LOBFILE.
- NULLIF or DEFAULTIF field conditions cannot be based on fields read from LOBFILEs.
- If a nonexistent LOBFILE is specified as a data source for a particular field, that field is initialized to empty. If the concept of empty does not apply to the particular field type, the field is initialized to null.
- Table-level delimiters are not inherited by fields that are read from a LOBFILE.
- When loading an XML column or referencing a LOB column in a SQL expression in conventional path mode, SQL*Loader must process the LOB data as a temporary LOB. To ensure the best load performance possible in these cases, refer to the guidelines concerning temporary LOB performance in *Oracle9i Application Developer's Guide - Large Objects (LOBs)*.

Loading Collections (Nested Tables and VARRAYs)

Like LOBs, collections can be loaded either from a primary datafile (data inline) or from secondary datafiles (data out of line). See Secondary Datafiles (SDFs) on page 7-31 for details about SDFs.

When you load 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:

- To specify the number of rows or elements that are to be loaded into each nested table or VARRAY instance, use the DDL COUNT function. The value specified for COUNT must either be a number or a character string containing a number, and it must be previously described in the control file before the COUNT clause itself. This positional dependency is specific to the COUNT clause. COUNT(0) or COUNT(cnt_field), where cnt_field is 0 for the current row, results in a empty collection (not null), unless overridden by a NULLIF clause. See count_spec on page A-12.
- Use the TERMINATED BY and ENCLOSED BY clauses to specify a unique collection delimiter. This method cannot be used if an SDF clause is used.

In the control file, collections are described similarly to column objects. See Loading Column Objects on page 7-1. There are some differences:

- Collection descriptions employ the two mechanisms discussed in the preceding list.

- Collection descriptions can include a secondary datafile (SDF) specification.
- A NULLIF or DEFAULTIF clause cannot refer to a field in an SDF unless the clause is on a field in the same SDF.
- Clauses 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.
- The field list must contain only one nonfiller field and any number of filler fields. If the VARRAY is a VARRAY of column objects, then the attributes of each column object will be in a nested field list.

Restrictions in Nested Tables and VARRAYs

The following restrictions exist for nested tables and VARRAYs:

- A field_list cannot contain a collection_fld_spec.
- 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 VARRAY parameter.

Example 7–24 demonstrates loading a VARRAY and a nested table.

Example 7–24 Loading a VARRAY and a Nested Table

Control File Contents

```
   LOAD DATA
   INFILE ‘sample.dat’ “str ‘\n’ ”
   INTO TABLE dept
   REPLACE
   FIELDS TERMINATED BY ‘,’ OPTIONALLY ENCLOSED BY ‘”’
   (
     dept_no       CHAR(3),
     dname         CHAR(25) NULLIF dname=BLANKS,
1    emps          VARRAY TERMINATED BY ':'
     (
       emps        COLUMN OBJECT
       (
         name      CHAR(30),
         age       INTEGER EXTERNAL(3),
2        emp_id    CHAR(7) NULLIF emps.emps.emp_id=BLANKS
     )
   ),
3   proj_cnt      FILLER CHAR(3),
4   projects      NESTED TABLE SDF (CONSTANT "pr.txt" "fix 57") COUNT (proj_cnt)
  (
    projects    COLUMN OBJECT
    (
      project_id        POSITION (1:5) INTEGER EXTERNAL(5),
      project_name      POSITION (7:30) CHAR
                        NULLIF projects.projects.project_name = BLANKS
    )
  )
)
```

Datafile (sample.dat)

```
 101,MATH,"Napier",28,2828,"Euclid", 123,9999:0
 210,"Topological Transforms",:2
```

Secondary Datafile (SDF) (pr.txt)

```
21034 Topological Transforms
77777 Impossible Proof
```

Notes

1. The TERMINATED BY clause specifies the VARRAY instance terminator (note that no COUNT clause is used).
2. Full name field references (using dot notation) resolve the field name conflict created by the presence of this filler field.
3. proj_cnt is a filler field used as an argument to the COUNT clause.
4. This entry specifies the following:
   - An SDF called pr.txt as the source of data. It also specifies a fixed-record format within the SDF.
   - 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. The main field name corresponding to the nested table field description is the same as the field name of its nested nonfiller-field, specifically, the name of the column object field description.

Secondary Datafiles (SDFs)

Secondary datafiles (SDFs) are similar in concept to 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. They are useful when you load large nested tables and VARRAYs.

> **Note:** Only a `collection_fld_spec` can name an SDF as its data source.

SDFs are specified using the `SDF` parameter. The `SDF` parameter can be followed by either the file specification string, or a `FILLER` field that is mapped to a data field containing one or more file specification strings.

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.
- The record size.
- The character set for an SDF can be specified using the `CHARACTERSET` clause (see Handling Different Character Encoding Schemes on page 5-16).
- A default delimiter (using the delimiter specification) for the fields that inherit a particular SDF specification (all member fields or attributes of the collection that contain the SDF specification, with exception of the fields containing their own LOBFILE specification).

Also note the following with regard to SDFs:

- If a nonexistent SDF is specified as a data source for a particular field, that field is initialized to empty. If the concept of empty does not apply to the particular field type, the field is initialized to null.
- Table-level delimiters are not inherited by fields that are read from an SDF.
- To load SDFs larger than 64 KB, 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` clause.

  See Also:

  - READSIZE (read buffer size) on page 4-10
  - OPTIONS Clause on page 5-4
  - sdf_spec on page A-11

Dynamic Versus Static SDF Specifications

You can specify SDFs 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 an SDF is reached, the file is closed and further attempts at reading data from that particular file produce results equivalent to reading data from an empty field.

In a 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.

The dynamic switching of the data source files has a resetting effect. For example, when SQL*Loader switches 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 as the source of two different fields. If you do so, typically, the two fields will read the data independently.

Loading a Parent Table Separately from Its Child Table

When you load a table that contains a nested table column, it may be possible to load the parent table separately from the child table. You can load the parent and child tables independently if the SIDs (system-generated or user-defined) are already known at the time of the load (that is, the SIDs are in the datafile with the data).

Example 7–25 and Example 7–26 illustrate how to load parent and child tables with user-provided SIDs.

Example 7–25 Loading a Parent Table with User-Provided SIDs

Control File Contents

```
   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))
```

Datafile (sample.dat)

```
101,Math,21E978407D4441FCE03400400B403BC3,|
210,”Topology”,21E978408D4441FCE03400400B403BC3,|
```

Notes

1. `mysid` is a filler field that is mapped to a datafile field containing the actual set-ids and is supplied as an argument to the `SID` clause.

Example 7–26 Loading a Child Table (the Nested Table Storage Table) with User-Provided SIDs

Control File Contents

```
   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))
```

Datafile (sample.dat)

```
21034, "Topological Transforms", 21E978407D4441FCE03400400B403BC3,
77777, "Impossible Proof", 21E978408D4441FCE03400400B403BC3,
```

Notes

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 that is the source of the real set-ids.

Memory Issues When Loading VARRAY Columns

The following list describes some issues to keep in mind when you load VARRAY columns:

- VARRAYs are created in the client’s memory before they are loaded into the database. Each element of a VARRAY requires 4 bytes of client memory before it can be loaded into the database. Therefore, when you load a VARRAY with a thousand elements, you will require at least 4000 bytes of client memory for each VARRAY instance before you can load the VARRAYs into the database. In many cases, SQL*Loader requires two to three times that amount of memory to successfully construct and load a VARRAY.

- The BINDSIZE parameter specifies the amount of memory allocated by SQL*Loader for loading records. Given the value specified for BINDSIZE, SQL*Loader takes into consideration the size of each field being loaded, and determines the number of rows it can load in one transaction. The larger the number of rows, the fewer transactions, resulting in better performance.

  But if the amount of memory on your system is limited, then at the expense of performance, you can specify a lower value for ROWS than SQL*Loader calculated.

- 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, specify a smaller value for BINDSIZE or ROWS and retry the load.

8

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 are records of successful SQL*Loader execution. However, errors can also cause log file entries. For example, errors found during parsing of the control file appear in the log file.

This chapter describes the following sections of a SQL*Loader log file:

- Header Information
- Global Information
- Table Information
- Datafile Information
- Table Load Information
- Summary Statistics
- Additional Summary Statistics for Direct Path Loads and Multithreading
- Log File Created When EXTERNAL_TABLE=GENERATE_ONLY

Header Information

The Header Section contains the following entries:

- Date of the run
- Software version number

For example:

```
SQL*Loader: Release 9.2.0.1.0 - Production on Wed Feb 27 11:07:28 2002
```

```
(c) Copyright 2002 Oracle Corporation.  All rights reserved.
```

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 datafile 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 records were loaded or only those meeting criteria specified in the WHEN clause.
- INSERT, APPEND, or REPLACE specification
- The following column information:
  - Column name
  - If found in a datafile, the position, length, delimiter, and datatype. See Column Information on page 8-3 for a description of these columns.

- If specified, RECNUM, SEQUENCE, CONSTANT, or EXPRESSION
- 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
```

Column Information

This section contains a more detailed description of the column information that is provided in the Table Information Section of the SQL*Loader log file.

Position

The following are the possibilities for the Position column:

- If a position is specified, the position values are in bytes, starting with byte position 1, regardless of whether byte-length semantics or character-length semantics are used.
- If both a start and end position are specified, they are separated by a colon.
- If only a start position is specified, then only that position is displayed.
- If no start or end position is specified, then FIRST is displayed for the first field and NEXT is displayed for other fields.
- If the start position is derived from other information, then DERIVED is displayed.

Length

The length, in bytes, is displayed under the heading `Len.` It gives the maximum size of the field, including the size of any embedded length fields. The size will be different with byte-length semantics versus character-length semantics. For example, for `VARCHAR (2,10)` with byte-length semantics, the length is 2 (the size of the length field) plus 10 (maximum size of the field itself), which equals 12 bytes. For `VARCHAR (2,10)` with character-length semantics, the length is calculated using the maximum size, in bytes, of a character in the datafile character set.

For fields that do not have a specified maximum length, an asterisk (*) is written in the Length column.

Delimiter

The delimiters are displayed under the headings, `Term` (for terminated by) and `Encl` (for enclosed by). If the delimiter is optional, it is preceded by `O` and is displayed within parentheses.

Datatype

The datatype is displayed as specified in the control file.

If the SQL*Loader control file contains any directives for loading datetime and interval datatypes, then the log file contains the parameter `DATE`, `DATETIME,` or `INTERVAL` under the Datatype heading. If applicable, the parameter `DATE`, `DATETIME,` or `INTERVAL` is followed by the corresponding mask. For example:

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

   Column Name                  Position   Len  Term Encl Datatype
------------------------------------------------------   ---  ---- ---- ---------
   col1                             NEXT   *              DATETIME HH.MI.SSXFF AM
```

Datafile Information

The Datafile Information Section appears only for datafiles with data errors, and provides the following entries:

- SQL*Loader and Oracle data record 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 did not meet the specified criteria for the `WHEN` clause
- Number of rows whose relevant fields were all null
- Date cache statistics, if applicable

For example:

```
Table EMP:
25000 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.

Date Cache:
 Max Size: 2000
 Entries:  1000
 Hits:     11000
 Misses:       0
```

> **See Also:** Specifying a Value for the Date Cache on page 9-22 for information on how you can improve performance by adjusting the maximum size of the date cache

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 datafiles, the number of records that were:
  - Skipped
  - Read
  - Rejected
  - Discarded
- Beginning and 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 Wed Feb 27 10:46:53 1990
Run ended on Wed Feb 27 10:47:17 1990

Elapsed time was:     00:00:15.62
CPU time was:         00:00:07.76
```

Oracle Statistics That Are Logged

The statistics that are reported to the log file vary, depending on the load type.

- For conventional loads and direct loads of a nonpartitioned table, statistics reporting is unchanged from Oracle7.
- For direct loads of a partitioned table, a per-partition statistics section is provided after the table-level statistics section.

- For a single-partition load, the partition name will be included in the table-level statistics section.

Information About Single-Partition Loads

The following information is logged when a single partition is loaded:

- The table column description includes the partition name.
- Error messages include the partition name.
- Statistics listings include the partition name.

Statistics for Loading a Table

The following statistics are logged when a table is loaded:

- Direct path load of a partitioned table reports per-partition statistics.
- Conventional path load cannot report per-partition statistics.
- For loading a nonpartitioned table, statistics are unchanged from Oracle7.

For conventional loads and direct loads of a nonpartitioned table, statistics reporting is unchanged from Oracle7.

If you request logging, but media recovery is not enabled, the load is not logged.

Additional Summary Statistics for Direct Path Loads and Multithreading

For direct path loads, the log contains the following additional data (the numbers in your log file will be different):

```
Column array rows:      20000
Stream buffer bytes:   256000
```

See Specifying the Number of Column Array Rows and Size of Stream Buffers on page 9-21 for information about the origin of these statistics.

Direct path loads on multiple-CPU systems have the option of using multithreading. If multithreading is enabled (the default behavior), the following additional statistics are logged (the numbers in your log will be different):

```
Total stream buffers loaded by SQL*Loader main thread:   102
Total stream buffers loaded by SQL*Loader load thread:   200
```

See Optimizing Direct Path Loads on Multiple-CPU Systems on page 9-23 for more information about multithreading.

Log File Created When EXTERNAL_TABLE=GENERATE_ONLY

When you use the external tables feature, you can place all of the SQL commands needed to do the load, as described in the control file, in the SQL*Loader log file. To do this, set the EXTERNAL_TABLE parameter to GENERATE_ONLY. The actual load can be done later without the use of SQL*Loader by executing these statements in SQL*Plus.

To generate an example of the log file created when using EXTERNAL_TABLE=GENERATE_ONLY, execute the following command for case study 1 (Case Study 1: Loading Variable-Length Data on page 10-5):

```
sqlldr scott/tiger ulcase1 EXTERNAL_TABLE=GENERATE_ONLY
```

The resulting log file looks as follows:

```
SQL*Loader: Release 9.2.0.1.0 - Production on Wed Feb 27 11:07:28 2002

(c) Copyright 2002 Oracle Corporation.  All rights reserved.

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
Continuation:    none specified
Path used:      External Table

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

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



CREATE DIRECTORY statements needed for files
------------------------------------------------------------------------
```

```
CREATE DIRECTORY SYS_SQLLDR_XT_TMPDIR_00000 AS
'/private/adestore/krich/.ade/view_storage/krich_dev/rdbms/demo'


CREATE TABLE statement for external table:
------------------------------------------------------------------------
CREATE TABLE "SYS_SQLLDR_X_EXT_DEPT"
(
  DEPTNO NUMBER(2),
  DNAME VARCHAR2(14),
  LOC VARCHAR2(13)
)
ORGANIZATION external
(
  TYPE oracle_loader
  DEFAULT DIRECTORY SYS_SQLLDR_XT_TMPDIR_00000
  ACCESS PARAMETERS
  (
    RECORDS DELIMITED BY NEWLINE CHARACTERSET US7ASCII
    BADFILE 'SYS_SQLLDR_XT_TMPDIR_00000':'ulcase1.bad'
    LOGFILE 'ulcase1.log_xt'
    READSIZE 1048576
    SKIP 20
    FIELDS TERMINATED BY "," OPTIONALLY ENCLOSED BY '"' LDRTRIM
    REJECT ROWS WITH ALL NULL FIELDS
    (
      DEPTNO CHAR(255)
        TERMINATED BY "," OPTIONALLY ENCLOSED BY '"',
      DNAME CHAR(255)
        TERMINATED BY "," OPTIONALLY ENCLOSED BY '"',
      LOC CHAR(255)
        TERMINATED BY "," OPTIONALLY ENCLOSED BY '"'
    )
  )
  location
  (
    'ulcase1.ctl'
  )
)REJECT LIMIT UNLIMITED


INSERT statements used to load internal tables:
------------------------------------------------------------------------
INSERT /*+ append */ INTO DEPT
(
  DEPTNO,
  DNAME,
  LOC
)
SELECT
  DEPTNO,
  DNAME,
  LOC
FROM "SYS_SQLLDR_X_EXT_DEPT"


statements to cleanup objects created by previous statements:
------------------------------------------------------------------------
DROP TABLE "SYS_SQLLDR_X_EXT_DEPT"
DROP DIRECTORY SYS_SQLLDR_XT_TMPDIR_00000


Run began on Wed Feb 27 11:07:28 2002
Run ended on Wed Feb 27 11:07:34 2002

Elapsed time was:     00:00:06.13
CPU time was:         00:00:00.20
```

See Also:

- EXTERNAL_TABLE on page 4-7
- Part III, "External Tables"

9

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
- Conventional Path Load
- Direct Path Load
- Using Direct Path Load
- Optimizing Performance of Direct Path Loads
- Optimizing Direct Path Loads on Multiple-CPU Systems
- Avoiding Index Maintenance
- Direct Loads, Integrity Constraints, and Triggers
- Parallel Data Loading Models
- General Performance Improvement Hints

For an example of using the direct path load method, see Case Study 6: Loading Data Using the Direct Path Load Method on page 10-24. 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` statements to populate tables 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 does not compete with other users for database resources, so it can usually load data at near disk speed. Considerations inherent to direct path loads, such as restrictions, security, and backup implications, are discussed in this chapter.

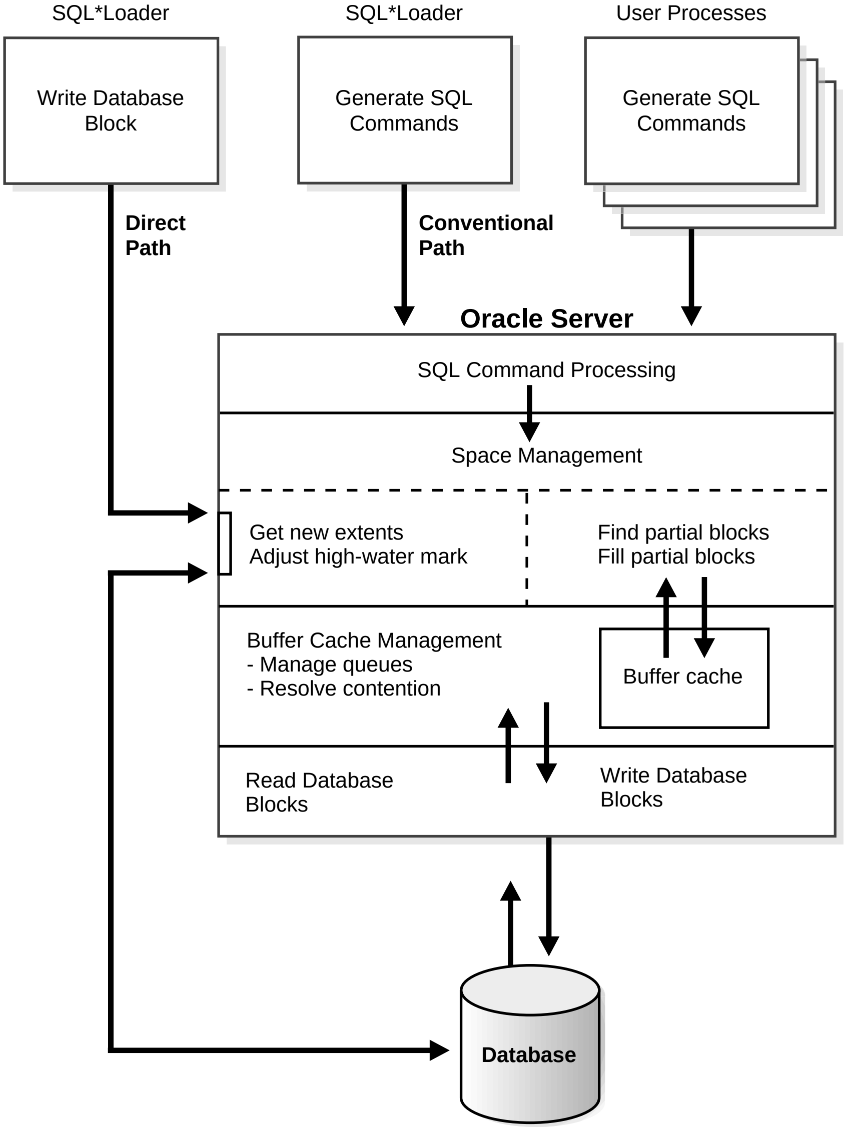
The tables to be loaded must already exist in the database. SQL*Loader never creates tables. It loads existing tables that either already contain data or are empty.

The following privileges are required for a load:

- You must have `INSERT` privileges on the table to be loaded.
- You must have `DELETE` privileges on the table to be loaded, when using the `REPLACE` or `TRUNCATE` option to empty old data from the table before loading the new data in its place.

Figure 9–1 shows how conventional and direct path loads perform database writes.

Figure 9–1 Database Writes on SQL*Loader Direct Path and Conventional Path



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.

The Oracle database server looks for partially filled blocks and attempts to fill them on each insert. Although appropriate during normal use, this can slow bulk loads dramatically.

See Also: Discontinued Conventional Path Loads on page 5-24

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 SQL*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 load. However, certain restrictions on direct path loads may require you to use a conventional path load. You should use a conventional path load in the following situations:

- When accessing an indexed table concurrently with the load, or when applying inserts or updates to a nonindexed table concurrently with the load

  To use a direct path load (with the exception of parallel loads), SQL*Loader must have exclusive write access to the table and exclusive read / write access to any indexes.

- 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 loading records and you want to ensure that a record is rejected under any of the following circumstances:

  – If the record, upon insertion, causes an Oracle error

  – If the record is formatted incorrectly, so that SQL*Loader cannot find field boundaries

  – If the record violates a constraint or tries to make a unique index non-unique

Direct Path Load

Instead of filling a bind array buffer and passing it to the Oracle database server with a SQL `INSERT` statement, a direct path load uses the direct path API to pass the data to be loaded to the load engine in the server. The load engine builds a column array structure from the data passed to it.

The direct path load engine uses the column array structure to format Oracle data blocks and build index keys. The newly formatted database blocks are 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.

See Also: Discontinued Direct Path Loads on page 5-24

Data Conversion During Direct Path Loads

During a direct path load, data conversion occurs on the client side rather than on the server side. This means that NLS parameters in the initialization parameter file (server-side language handle) will not be used. To override this behavior, you can specify a format mask in the SQL*Loader control file which is equivalent to the setting of the NLS parameter in the initialization parameter file, or set the appropriate environment variable. For example, to specify a date format for a field, you can either set the date format in the SQL*Loader control file as shown in Example 9–1 or set an NLS_DATE_FORMAT environment variable as shown in Example 9–2.

Example 9–1 Setting the Date Format in the SQL*Loader Control File

```
LOAD DATA
INFILE 'data.dat'
INSERT INTO TABLE emp
FIELDS TERMINATED BY "|"
(
EMPNO NUMBER(4) NOT NULL,
ENAME CHAR(10),
JOB CHAR(9),
MGR NUMBER(4),
HIREDATE DATE 'YYYYMMDD',
SAL NUMBER(7,2),
COMM NUMBER(7,2),
DEPTNO NUMBER(2)
)
```

Example 9–2 Setting an NLS_DATE_FORMAT Environment Variable

On UNIX bourne or korn shell:

```
% NLS_DATE_FORMAT='YYYYMMDD'
% export NLS_DATE_FORMAT
```

On UNIX csh:

```
%setenv NLS_DATE_FORMAT='YYYYMMDD'
```

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.

Note: If you are performing a direct path load into multiple partitions and a space error occurs, the load is rolled back to the last commit point. If there was no commit point, then the entire load is rolled back. This ensures that no data encountered after the space error is written out to a different partition.

You can use the `ROWS` parameter to specify the frequency of the commit points. If the `ROWS` parameter is not specified, the entire load is rolled back.

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 that do not map to the partition or subpartition specified in the SQL*Loader control file. Local index partitions that 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 either of the following forms:

```
LOAD INTO TABLE T PARTITION (P) VALUES ...

LOAD INTO TABLE T SUBPARTITION (P) VALUES ...
```

While you are loading a partition of a partitioned or subpartitioned table, you are also allowed to perform DML operations on, and direct path loads of, other partitions in the table.

Although a direct path load minimizes database processing, several calls to the Oracle database 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. The following operations occur during the load: index keys are built and put into a sort, and space management routines are used to get new extents when needed and to adjust the upper boundary (high-water mark) for a data savepoint. See Using Data Saves to Protect Against Data Loss on page 9-13 for information on adjusting the upper boundary.

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` statements; therefore, the processing load on the Oracle database is reduced.
- 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 multiblock 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. This minimizes contention with other Oracle users.
- The sorted indexes option available during direct path loads allows you to presort data using high-performance sort routines that are native to your system or installation.
- When a table to be loaded is empty, the presorting option eliminates the sort and merge phases of index-building. The index is 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` parameter is set to `Y`
  - The object being loaded has the `NOLOG` attribute set

  See Instance Recovery and Direct Path Loads on page 9-15.

Restrictions on Using Direct Path Loads

The following conditions must be satisfied for you 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 Oracle Enterprise Manager command `MONITOR TABLE` to find the object ID for the tables you want to load. Then use the command `MONITOR LOCK` to see if there are any locks on the tables.

- For versions of the Oracle database server prior to 9*i*, you can only perform a SQL*Loader direct path load when the client and server are the same version. This also means that you cannot perform a direct path load of Oracle9*i* data into a database of an earlier version. For example, you cannot use direct path load to load data from a release 9.0.1 database into a release 8.1.7 database.

  However, beginning with Oracle9*i*, you can perform a SQL*Loader direct path load between different versions as long as both the client and server are version 9*i* or later. For example, you can perform a direct path load from a release 9.0.1 database into a release 9.2 database.

The following features are not available with direct path load.

- Loading VARRAYs
- Loading a parent table together with a child table
- Loading `BFILE` columns

Restrictions on a Direct Path Load of a Single Partition

In addition to the previously listed restrictions, loading a single partition has the following restrictions:

- The table that the partition is a member of cannot have any global indexes defined on it.
- Enabled referential and check constraints on the table that the partition is a member of are not allowed.
- Enabled triggers are not allowed.

When to Use a Direct Path Load

If none of the previous 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 nonempty table.
- You want to load data in parallel for maximum performance. See Parallel Data Loading Models on page 9-30.

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 that 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 reenabled afterwards. If REENABLE is specified, SQL*Loader can reenable them automatically at the end of the load. When the constraints are reenabled, the entire table is checked. Any rows that fail this check are reported in the specified error log. See Direct Loads, Integrity Constraints, and Triggers on page 9-25.

Field Defaults on the Direct Path

Default column specifications defined in the database are not available when you use direct path loading. Fields for which default values are desired must be specified with the DEFAULTIF clause. 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 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.

Using Direct Path Load

This section explains how to use the SQL*Loader direct path load method.

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. You can run this script 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 path load mode, set the DIRECT parameter to true on the command line or in the parameter file, if used, in the format:

```
DIRECT=true
```

See Also:

- Case Study 6: Loading Data Using the Direct Path Load Method on page 10-24
- Optimizing Performance of Direct Path Loads on page 9-17 for information about parameters you can use to optimize performance of direct path loads
- Optimizing Direct Path Loads on Multiple-CPU Systems on page 9-23 if you are doing a direct path load on a multiple-CPU system or across systems

Building Indexes

You can improve performance of direct path loads by using temporary storage. After each block is formatted, the new index keys are put in 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.

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.

Improving Performance

To improve performance on systems with limited memory, use the SINGLEROW parameter. For more information, see SINGLEROW Option on page 5-38.

Note: If, during a direct load, you have specified that the data is to be presorted 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 Optimizing Performance of Direct Path Loads on page 9-17 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.

> **See Also:** *Oracle9i Database Administrator's Guide* for information on how to estimate index size and set storage parameters

Temporary Segment Storage Requirements

To estimate the amount of temporary segment space needed for storing the new index keys (in bytes), use the following formula:

```
1.3 * key_storage
```

In this formula, key storage is defined as follows:

```
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 would be 1.0. See Presorting Data for Faster Indexing on page 9-18 for more information.

Indexes Left in an Unusable State

SQL*Loader leaves indexes in an 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 an Index Unusable state returns an error. The following conditions cause a direct path load to leave an index or a partition of a partitioned index in an 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 savepoints are being used, and the load fails or is terminated by a keyboard interrupt after a data savepoint occurred.

To determine if an index is in an Index Unusable state, you can execute a simple query:

```
SELECT INDEX_NAME, STATUS
   FROM USER_INDEXES
   WHERE TABLE_NAME = 'tablename';
```

If you are not the owner of the table, then search ALL_INDEXES or DBA_INDEXES instead of USER_INDEXES.

To determine if an index partition is in an unusable state, you can execute the following query:

```
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_IND_PARTITIONS and DBA_IND_PARTITIONS instead of USER_IND_PARTITIONS.

Using Data Saves to Protect Against Data Loss

You can use data saves to protect against loss of data due to instance failure. All data loaded up to the last savepoint 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 parameter to skip those processed rows.

If there were any indexes on the table, drop them before continuing the load, then re-create them after the load. See Data Recovery During Direct Path Loads on page 9-15 for more information on media and instance recovery.

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 presorted data is loaded into an empty table, but these indexes are also unprotected.)

Using the ROWS Parameter

The `ROWS` parameter 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. 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 boundary (high-water mark) on the amount of work that is lost when an instance failure occurs during a long-running direct path load. Setting the value of `ROWS` to a small number adversely affects 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 as follows:

- A data save will make the rows visible to other users.
- Rows cannot be rolled back after a data save.

The major difference is that in a direct path load data save, the indexes will be unusable (in Index Unusable state) until the load completes.

Data Recovery During Direct Path Loads

SQL*Loader provides full support for data recovery when using the direct path load method. There are two main types of recovery:

- Media recovery - 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 - recovery from a system failure in which in-memory data was changed but lost due to the failure before it was written to disk. The Oracle database server can always recover from instance failures, even when redo logs are not archived

> **See Also:** *Oracle9i Database Administrator's Guide* for more information about recovery

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 Also:** *Oracle9i User-Managed Backup and Recovery Guide* for more information on the RECOVER command

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 an Index Unusable state. Indexes that are Unusable must be rebuilt before you can use the table or partition. See Indexes Left in an Unusable State on page 9-12 for information on how to determine if an index has been left in Index Unusable state.

Loading LONG Data Fields

Data that is longer than SQL*Loader's maximum buffer size can be loaded on the direct path by using LOBs. You can improve performance when doing this by using a large streamsize value.

See Also:

- Loading LOBs on page 7-18
- Specifying the Number of Column Array Rows and Size of Stream Buffers on page 9-21

You could also load data that is longer than the maximum buffer size by using the `PIECED` parameter, as described in the next section, but Oracle Corporation highly recommends that you use LOBs instead.

Loading Data As PIECED

The `PIECED` parameter can be used to load data in sections, if the data is in the last column of the logical record.

Declaring a column as `PIECED` informs the direct path loader that a `LONG` field might be split across multiple physical records (pieces). In such cases, SQL*Loader processes each piece of the `LONG` field as it is found in the physical record. All the pieces are read before the record is processed. SQL*Loader makes no attempt to materialize the `LONG` field before storing it; however, all the pieces are read before the record is processed.

The following restrictions apply when you declare 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 field 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.

Optimizing Performance of Direct Path Loads

You can control the time and temporary storage used during direct path loads.

To minimize time:

- Preallocate storage space
- Presort the data
- Perform infrequent data saves
- Minimize use of the redo log
- Specify the number of column array rows and the size of the stream buffer
- Specify a date cache value

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

Preallocating 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 information about managing database files in the *Oracle9i Database Administrator's Guide*. Then use the INITIAL or MINEXTENTS clause in the SQL CREATE TABLE statement to allocate the required space.

Another approach is to size extents large enough so that extent allocation is infrequent.

Presorting Data for Faster Indexing

You can improve the performance of direct path loads by presorting your data on indexed columns. Presorting minimizes temporary storage requirements during the load. Presorting also allows you to take advantage of high-performance sorting routines that are optimized for your operating system or application.

If the data is presorted and the existing index is not empty, then presorting 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 9-12.

If presorting is specified and the existing index is empty, then maximum efficiency is achieved. 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 Clause

The `SORTED INDEXES` clause identifies the indexes on which the data is presorted. This clause is allowed only for direct path loads. See Case Study 6: Loading Data Using the Direct Path Load Method on page 10-24 for an example.

Generally, you specify only one index in the `SORTED INDEXES` clause, 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 indexes can be specified at once.

All indexes listed in the `SORTED INDEXES` clause must be created before you start the direct path load.

Unsorted Data

If you specify an index in the `SORTED INDEXES` clause, and the data is not sorted for that index, then the index is left in an 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 that is left in an Index Unusable state must be rebuilt after the load.

Multiple-Column Indexes

If you specify a multiple-column index in the SORTED INDEXES clause, 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 presorting 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, 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.

Disabling Archiving

If archiving is disabled, direct path loads do not generate full image redo. Use the ARCHIVELOG and NOARCHIVELOG parameters to set the archiving mode. See the *Oracle9i Database Administrator's Guide* for more information about archiving.

Specifying the UNRECOVERABLE Parameter

To save time and space in the redo log file, use the UNRECOVERABLE parameter when you load data. An UNRECOVERABLE load does not record loaded data in the redo log file; instead, it generates invalidation redo.

The UNRECOVERABLE parameter 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 parameter, 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.

Setting the 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 that are not logged.

Specifying the Number of Column Array Rows and Size of Stream Buffers

The number of column array rows determines the number of rows loaded before the stream buffer is built. The STREAMSIZE parameter specifies the size (in bytes) of the data stream sent from the client to the server.

Use the COLUMNARRAYROWS parameter to specify a value for the number of column array rows.

Use the STREAMSIZE parameter to specify the size for direct path stream buffers.

The optimal values for these parameters vary, depending on the system, input datatypes, and Oracle column datatypes used. When you are using optimal values for your particular configuration, the elapsed time in the SQL*Loader log file should go down.

To see a list of default values for these and other parameters, invoke SQL*Loader without any parameters, as described in Invoking SQL*Loader on page 4-1.

Note: You should monitor process paging activity, because if paging becomes excessive, performance can be significantly degraded. You may need to lower the values for READSIZE, STREAMSIZE, and COLUMNARRAYROWS to avoid excessive paging.

It can be particularly useful to specify the number of column array rows and size of the steam buffer when you perform direct path loads on multiple-CPU systems. See Optimizing Direct Path Loads on Multiple-CPU Systems on page 9-23 for more information.

Specifying a Value for the Date Cache

If you are performing a direct path load in which the same date or timestamp values are loaded many times, a large percentage of total load time can end up being used for converting date and timestamp data. This is especially true if multiple date columns are being loaded. In such a case, it may be possible to improve performance by using the SQL*Loader date cache.

The date cache reduces the number of date conversions done when many duplicate values are present in the input data. It allows you to specify the number of unique dates anticipated during the load.

The date cache is enabled by default. To completely disable the date cache, set it to 0.

The default date cache size is 1000 elements. If the default is used and the number of unique input values loaded exceeds 1000, then the date cache is automatically disabled for that table. This prevents excessive and unnecessary lookup times that could affect performance. However, if instead of using the default, you specify a nonzero value for the date cache and it is exceeded, the date cache is *not* disabled. Instead, any input data that exceeded the maximum is explicitly converted using the appropriate conversion routines.

The date cache can be associated with only one table. No date cache sharing can take place across tables. A date cache is created for a table only if all of the following conditions are true:

- The DATE_CACHE parameter is not set to 0
- One or more date values, timestamp values, or both are being loaded that require datatype conversion in order to be stored in the table
- The load is a direct path load

Date cache statistics are written to the log file. You can use those statistics to improve direct path load performance as follows:

- If the number of cache entries is less than the cache size and there are no cache misses, then the cache size could safely be set to a smaller value.
- If the number of cache hits (entries for which there are duplicate values) is small and the number of cache misses is large, then the cache size should be increased. Be aware that if the cache size is increased too much, it may cause other problems such as excessive paging or too much memory usage.
- If most of the input date values are unique, the date cache will not enhance performance and therefore should not be used.

Note: Date cache statistics are *not* written to the SQL*Loader log file if the cache was active by default and disabled because the maximum was exceeded.

If increasing the cache size does not improve performance, revert to the default behavior or set the cache size to 0. The overall performance improvement also depends on the datatypes of the other columns being loaded. Improvement will be greater for cases in which the total number of date columns loaded is large compared to other types of data loaded.

See Also:

- DATE_CACHE on page 4-5
- Table Load Information on page 8-5 for an example of how date cache statistics are presented in the SQL*Loader log file

Optimizing Direct Path Loads on Multiple-CPU Systems

If you are performing direct path loads on a multiple-CPU system, SQL*Loader uses multithreading by default. A multiple-CPU system in this case is defined as a single system that has two or more CPUs.

Multithreaded loading means that, when possible, conversion of the column arrays to stream buffers and stream buffer loading are performed in parallel. This optimization works best when:

- Column arrays are large enough to generate multiple direct path stream buffers for loads
- Data conversions are required from input field datatypes to Oracle column datatypes

  The conversions are performed in parallel with stream buffer loading.

The status of this process is recorded in the SQL*Loader log file, as shown in the following sample portion of a log:

```
Total stream buffers loaded by SQL*Loader main thread:         47
Total stream buffers loaded by SQL*Loader load thread:        180
Column array rows:                                           1000
Stream buffer bytes:                                       256000
```

In this example, the SQL*Loader load thread has offloaded the SQL*Loader main thread, allowing the main thread to build the next stream buffer while the load thread loads the current stream on the server.

The goal is to have the load thread perform as many stream buffer loads as possible. This can be accomplished by increasing the number of column array rows, decreasing the stream buffer size, or both. You can monitor the elapsed time in the SQL*Loader log file to determine whether your changes are having the desired effect. See Specifying the Number of Column Array Rows and Size of Stream Buffers on page 9-21 for more information.

On single-CPU systems, optimization is turned off by default. When the server is on another system, performance may improve if you manually turn on multithreading.

To turn the multithreading option on or off, use the `MULTITHREADING` parameter at the SQL*Loader command line or specify it in your SQL*Loader control file.

> **See Also:** *Oracle Call Interface Programmer's Guide* for more information about the concepts of direct path loading

Avoiding Index Maintenance

For both the conventional path and the direct path, SQL*Loader maintains all existing indexes for a table.

To avoid index maintenance, use 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` parameter.
- Use the `SKIP_INDEX_MAINTENANCE` parameter (direct path only, use with caution).

By avoiding index maintenance, you minimize the amount of space required during a direct path load, in the following ways:

- 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 resort the indexes may be excessive. In such cases, it is usually better to use the conventional path load method, or to use the SINGLEROW parameter of SQL*Loader. For more information, see SINGLEROW Option on page 5-38.

Direct Loads, Integrity Constraints, and Triggers

With the conventional path load method, arrays of rows are inserted with standard SQL INSERT statements—integrity constraints and insert triggers are automatically applied. But when you load data with the direct path, SQL*Loader disables some integrity constraints and all database triggers. 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 information on maintaining data integrity in the *Oracle9i Application Developer's Guide - Fundamentals*.

Enabled Constraints

The constraints that remain in force are:

- NOT NULL
- UNIQUE
- PRIMARY KEY (unique-constraints on not-null columns)

NOT NULL constraints are checked at column array build time. Any row that violates the NOT NULL constraint is rejected.

UNIQUE constraints are verified when indexes are rebuilt at the end of the load. The index will be left in an Index Unusable state if a violation of a UNIQUE constraint is detected. See Indexes Left in an Unusable State on page 9-12.

Disabled Constraints

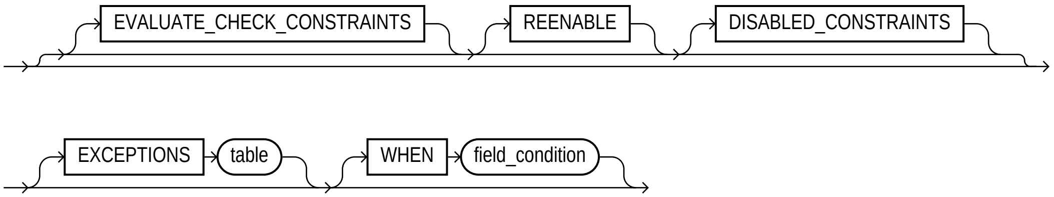
During a direct path load, the following constraints are automatically disabled by default:

- CHECK constraints
- Referential constraints (FOREIGN KEY)

You can override the disabling of CHECK constraints by specifying the EVALUATE_CHECK_CONSTRAINTS clause. SQL*Loader will then evaluate CHECK constraints during a direct path load. Any row that violates the CHECK constraint is rejected.

Reenable Constraints

When the load completes, the integrity constraints will be reenabled automatically if the REENABLE clause is specified. The syntax for the REENABLE clause is as follows:



The optional parameter 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 *Oracle9i SQL Reference* for instructions on how to create an exceptions table.

The SQL*Loader log file describes the constraints that were disabled, the ones that were reenabled, and what error, if any, prevented reenabling or validating of each constraint. It also contains the name of the exceptions table specified for each loaded table.

If the REENABLE clause is not used, then the constraints must be reenabled manually, at which time all rows in the table are verified. If the Oracle database server 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.

If the REENABLE clause is used, SQL*Loader automatically reenables the constraint and then verifies all new rows. If no errors are found in the new data, SQL*Loader automatically marks the constraint as validated. If any errors *are* found in the new data, error messages are written to the log file and SQL*Loader marks the status of the constraint as ENABLE NOVALIDATE. The names of violated constraints and the ROWIDs of the bad data are placed in an exceptions table, if one is specified.

Note: Normally, when a table constraint is left in an ENABLE NOVALIDATE state, new data can be inserted into the table but no new invalid data may be inserted. However, SQL*Loader direct path load does not enforce this rule. Thus, if subsequent direct path loads are performed with invalid data, the invalid data will be inserted but the same error reporting and exception table processing as described previously will take place. In this scenario the exception table may contain duplicate entries if it is not cleared out before each load. Duplicate entries can easily be filtered out by performing a query such as the following:

```
SELECT UNIQUE * FROM exceptions_table;
```

Note: 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 reenabled. The log file lists all triggers that were disabled for the load. There should not be any errors reenabling 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 1-byte or 1-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` parameter.

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.

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 as a delete flag, because an update trigger cannot modify the columns 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.

In 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, it can be used without exclusive access to the table.

1. Do the following to create a stored procedure that duplicates the effects of the insert trigger:
   - **a.** Declare a cursor for the table, selecting all new rows.
   - **b.** Open the cursor and fetch rows, one at a time, in a processing loop.
   - **c.** Perform the operations contained in the insert trigger.
   - **d.** If the operations succeed, change the "new data" flag to null.
   - **e.** 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. Reenable the insert trigger.

> **See Also:** *PL/SQL User's Guide and Reference* for more information about cursor management

Permanently Disabled Triggers and 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 reenable 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 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 reenable them. Any triggers or constraints that were not reenabled by SQL*Loader should be manually enabled with the ENABLE clause of the ALTER TABLE statement described in *Oracle9i SQL Reference*.

Increasing Performance with 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 using the standard DML execution logic.

Parallel Data Loading Models

This section discusses three basic models of concurrency that you can use to minimize the elapsed time required for data loading:

- Concurrent conventional path loads
- Intersegment concurrency with the direct path load method
- Intrasegment concurrency with the direct path load method

Concurrent Conventional Path Loads

Using multiple conventional path load sessions executing concurrently is discussed in Increasing Performance with Concurrent Conventional Path Loads on page 9-30. You can use this technique to load the same or different objects concurrently with no restrictions.

Intersegment Concurrency with Direct Path

Intersegment concurrency can be used for concurrent loading of different objects. You can apply this technique to concurrent direct path loading of different tables, or to concurrent direct path loading of different partitions of the same table.

When you direct path load a single partition, consider the following items:

- 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).

Intrasegment 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` parameters 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` parameter to `true` only allows multiple concurrent direct path load sessions.

Restrictions on Parallel Direct Path Loads

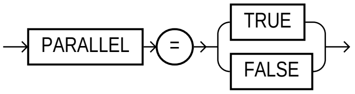
The following restrictions are enforced on parallel direct path loads:

- 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, you should partition the data first (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:

```
sqlldr USERID=scott/tiger CONTROL=load1.ctl DIRECT=TRUE PARALLEL=true
sqlldr USERID=scott/tiger CONTROL=load2.ctl DIRECT=TRUE PARALLEL=true
sqlldr 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.

Note: Indexes are not maintained during a parallel load. Any indexes must be created or re-created 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.

Parameters for Parallel Direct Path Loads

When you perform parallel direct path loads, there are options available for specifying attributes of the temporary segment to be allocated by the loader.

Specifying Temporary Segments

To allow for maximum I/O throughput, Oracle Corporation recommends that each concurrent direct path load session use files located on different disks. Use the `FILE` parameter of the `OPTIONS` clause to specify the filename of any valid datafile in the tablespace of the object (table or partition) being loaded.

For 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 could also specify the `FILE` parameter on the command line of each concurrent SQL*Loader session, but then it would apply globally to all objects being loaded with that session.

Using the FILE Parameter The `FILE` parameter in the Oracle database server has the following restrictions for parallel direct path loads:

- **For nonpartitioned tables:** The specified file must be in the tablespace of the table being loaded.
- **For partitioned tables, single-partition load:** The specified file must be in the tablespace of the partition being loaded.
- **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 Parameter You can use the STORAGE parameter to specify the storage attributes of the temporary segments allocated for a parallel direct path load. If the STORAGE parameter is not used, the storage attributes of the segment containing the object (table, partition) being loaded are used. Also, when the STORAGE parameter is not specified, SQL*Loader uses a default of 2 KB for EXTENTS.

```
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))
```

You can use the STORAGE parameter only in the control file, and not on the command line. Use of the STORAGE parameter 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.

Because each SQL*Loader session can attempt to reenable constraints on a table after a direct path load, there is a danger that one session may attempt to reenable a constraint before another session is finished loading data. In this case, the first session to complete the load will be unable to enable the constraint because the remaining sessions possess share locks on the table.

Because there is a danger that some constraints might not be reenabled after a direct path load, you should check the status of the constraint after completing the load to ensure that it was enabled properly.

PRIMARY KEY and UNIQUE KEY Constraints

PRIMARY KEY and UNIQUE KEY constraints create indexes on a table when they are enabled, and subsequently can take a significantly long time to enable after a direct path loading session if the table is very large. You should consider enabling these constraints manually after a load (and not specifying the automatic enable feature). This allows you to manually create the required indexes in parallel to save time before enabling the constraint.

See Also: *Oracle9i Database Performance Guide and Reference*

General Performance Improvement Hints

If you have control over the format of the data to be loaded, you can use the following hints to improve load performance:

- 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 identify physical record boundaries. Use the file processing option string "FIX nnn" or "VAR". If you use the default (stream mode) on most platforms (for example, UNIX and NT) the loader must scan each physical record for the record terminator (newline character).
- 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).
  - Do not trim whitespace if you do not need to (use PRESERVE BLANKS).
- Make conversions efficient. SQL*Loader performs character set conversion and datatype conversion for you. Of course, the quickest conversion is no conversion.
  - Use single-byte character sets if you can.

  - Avoid character set conversions if you can. The loader supports four character sets:

    - Client character set (NLS_LANG of the client sqlldr process)
    - Datafile character set (usually the same as the client character set)
    - Database server character set
    - Database 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 database server character set are the same. If the character sets are the same, character set conversion buffers are not allocated.

- Use direct path loads.
- Use the SORTED INDEXES clause.
- Avoid unnecessary NULLIF and DEFAULTIF clauses. Each clause must be evaluated on each column that has a clause associated with it for every row loaded.
- Use parallel direct path loads and parallel index creation when you can.
- Be aware of the effect on performance when you have large values for both the CONCATENATE clause and the COLUMNARRAYROWS clause. See Using CONCATENATE to Assemble Logical Records on page 5-27.

Additionally, the performance tips provided in Performance Hints When Using External Tables on page 11-6 also apply to SQL*Loader.

10

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.

> **Note:** The commands used in this chapter, such as `sqlldr,` are UNIX-specific invocations. Refer to your Oracle operating system-specific documentation for information about the correct commands to use on your operating system.

This chapter contains the following sections:

- Case Study 8: Loading Partitioned Tables
- Case Study 9: Loading LOBFILEs (CLOBs)
- Case Study 10: Loading REF Fields and VARRAYs
- Case Study 11: Loading Data in the Unicode Character Set

The Case Studies

This chapter contains the following case studies:

- Case Study 1: Loading Variable-Length Data on page 10-5: Loads stream format records in which the fields are terminated by commas and may be enclosed by quotation marks. The data is found at the end of the control file.
- Case Study 2: Loading Fixed-Format Fields on page 10-8: Loads data from a separate datafile.
- Case Study 3: Loading a Delimited, Free-Format File on page 10-11: Loads data from stream format records with delimited fields and sequence numbers. The data is found at the end of the control file.
- Case Study 4: Loading Combined Physical Records on page 10-14: Combines multiple physical records into one logical record corresponding to one database row.
- Case Study 5: Loading Data into Multiple Tables on page 10-18: Loads data into multiple tables in one run.
- Case Study 6: Loading Data Using the Direct Path Load Method on page 10-24: Loads data using the direct path load method.
- Case Study 7: Extracting Data from a Formatted Report on page 10-28: Extracts data from a formatted report.
- Case Study 8: Loading Partitioned Tables on page 10-34: Loads partitioned tables.
- Case Study 9: Loading LOBFILEs (CLOBs) on page 10-38: 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 Study 10: Loading REF Fields and VARRAYs on page 10-43: Loads a customer table that has a primary key as its OID and stores order items in a VARRAY. Loads an order table that has a reference to the customer table and the order items in a VARRAY.

- Case Study 11: Loading Data in the Unicode Character Set on page 10-47: Loads data in the Unicode character set, UTF16, in little endian byte order. This case study uses character-length semantics.

Case Study Files

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

- Control files (for example, `ulcase5.ctl`)
- Datafiles (for example, `ulcase5.dat`)
- Setup files (for example, `ulcase5.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 10–1 lists the files associated with each case.

Table 10–1 Case Studies and Their Related Files

Case	.ctl	.dat	.sql
1	Yes	No	Yes
2	Yes	Yes	No
3	Yes	No	Yes
4	Yes	Yes	Yes
5	Yes	Yes	Yes
6	Yes	Yes	Yes
7	Yes	Yes	Yes (S, E)
8	Yes	Yes	Yes
9	Yes	Yes	Yes
10	Yes	No	Yes
11	Yes	Yes	Yes

Note: 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 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))
```

Checking the Results of a Load

To check the results of a load, start SQL*Plus and perform a select operation from the table that was loaded in the case study. This is done, as follows:

1. Start SQL*Plus as scott/tiger by entering the following at the system prompt:

   ```
   sqlplus scott/tiger
   ```

   The SQL prompt is displayed.

2. At the SQL prompt, use the SELECT statement to select all rows from the table that the case study loaded. For example, if the table emp was loaded, enter:

   ```
   SQL> SELECT * FROM emp;
   ```

The contents of each row in the `emp` table will be displayed.

References and Notes

The summary at the beginning of each case study directs you to the sections of this guide that discuss the SQL*Loader feature being demonstrated.

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.

Case Study 1: Loading Variable-Length Data

Case 1 demonstrates:

- A simple control file identifying one table and three columns to be loaded.
- 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 5-10.
- Loading data in stream format, with both types of delimited fields: terminated and enclosed. See Field Length Specifications for Datatypes for Which Whitespace Can Be Trimmed on page 6-45.

Control File for Case Study 1

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. The names of columns to load are enclosed in parentheses. Because no datatype or length is specified, the default is type CHAR with a maximum length of 255.
6. BEGINDATA specifies the beginning of the data.

Running Case Study 1

Take the following steps to run the case study.

1. Start SQL*Plus as scott/tiger by entering the following at the system prompt:

   ```
   sqlplus scott/tiger
   ```

   The SQL prompt is displayed.

2. At the SQL prompt, execute the SQL script for this case study, as follows:

   ```
   SQL> @ulcase1
   ```

   This prepares and populates tables for the case study and then returns you to the system prompt.

3. At the system prompt, invoke SQL*Loader and run the case study, as follows:

   ```
   sqlldr USERID=scott/tiger CONTROL=ulcase1.ctl LOG=ulcase1.log
   ```

   SQL*Loader loads the dept table, creates the log file, and returns you to the system prompt. You can check the log file to see the results of running the case study.

Log File for Case Study 1

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 256000 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:                  49536 bytes(64 rows)
Read   buffer bytes: 1048576

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

Run began on Wed Feb 27 14:10:13 2002
Run ended on Wed Feb 27 14:10:14 2002
```

```
Elapsed time was:     00:00:01.53
CPU time was:         00:00:00.20
```

Notes:

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

Case Study 2: Loading Fixed-Format Fields

Case 2 demonstrates:

- A separate datafile. See Specifying Datafiles on page 5-7.
- Data conversions. See Datatype Conversions on page 6-22.

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

Control File for Case Study 2

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 `INFILE` parameter.
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 for Case Study 2

The following 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
7658 CHAN       ANALYST   7566  3450.00          20
7654 MARTIN     SALESMAN  7698  1312.50  1400.00 30
```

Running Case Study 2

Take the following steps to run the case study. If you have already run case study 1, you can skip to step 3 because the ulcase1.sql script handles both case 1 and case 2.

1. Start SQL*Plus as scott/tiger by entering the following at the system prompt:

   ```
   sqlplus scott/tiger
   ```

   The SQL prompt is displayed.

2. At the SQL prompt, execute the SQL script for this case study, as follows:

   ```
   SQL> @ulcase1
   ```

   This prepares and populates tables for the case study and then returns you to the system prompt.

3. At the system prompt, invoke SQL*Loader and run the case study, as follows:

   ```
   sqlldr USERID=scott/tiger CONTROL=ulcase2.ctl LOG=ulcase2.log
   ```

   SQL*Loader loads the table, creates the log file, and returns you to the system prompt. You can check the log file to see the results of running the case study.

Records loaded in this example from the emp table 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).

Log File for Case Study 2

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 256000 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:                   3840 bytes(64 rows)
```

```
Read   buffer bytes: 1048576

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

Run began on Wed Feb 27 14:17:39 2002
Run ended on Wed Feb 27 14:17:39 2002

Elapsed time was:     00:00:00.81
CPU time was:         00:00:00.15
```

Case Study 3: Loading a Delimited, Free-Format File

Case 3 demonstrates:

- Loading data (enclosed and terminated) in stream format. See Delimited Fields on page 6-45.
- Loading dates using the datatype DATE. See Datetime and Interval Datatypes on page 6-16.
- Using SEQUENCE numbers to generate unique keys for loaded data. See Setting a Column to a Unique Sequence Number on page 6-57.
- Using APPEND to indicate that the table need not be empty before inserting new records. See Table-Specific Loading Method on page 5-32.
- Using Comments in the control file set off by two hyphens. See Comments in the Control File on page 5-4.

Control File for Case Study 3

This control file loads the same table as in case 2, but it loads three additional columns (hiredate, projno, and loadseq). The demonstration table emp does not have columns projno and loadseq. 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 two hyphens that may appear anywhere on a line.
2. `INFILE *` specifies that the data is found at the end of the control file.
3. `APPEND` 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 specified to have a maximum of 20. The maximum length is in bytes, with default byte-length semantics. If character-length semantics were used instead, the length would be in characters. 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.

Running Case Study 3

Take the following steps to run the case study.

1. Start SQL*Plus as scott/tiger by entering the following at the system prompt:

   ```
   sqlplus scott/tiger
   ```

   The SQL prompt is displayed.
2. At the SQL prompt, execute the SQL script for this case study, as follows:

   ```
   SQL> @ulcase3
   ```

   This prepares and populates tables for the case study and then returns you to the system prompt.
3. At the system prompt, invoke SQL*Loader and run the case study, as follows:

   ```
   sqlldr USERID=scott/tiger CONTROL=ulcase3.ctl LOG=ulcase3.log
   ```

   SQL*Loader loads the table, creates the log file, and returns you to the system prompt. You can check the log file to see the results of running the case study.

Log File for Case Study 3

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 256000 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:                 134976 bytes(64 rows)
Read   buffer bytes: 1048576

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

Run began on Wed Feb 27 14:25:29 2002
Run ended on Wed Feb 27 14:25:30 2002

Elapsed time was:     00:00:00.81
CPU time was:         00:00:00.15
```

Case Study 4: Loading Combined Physical Records

Case 4 demonstrates:

- Combining multiple physical records to form one logical record with CONTINUEIF; see Using CONTINUEIF to Assemble Logical Records on page 5-27.
- Inserting negative numbers.
- Indicating with REPLACE that the table should be emptied before the new data is inserted; see Table-Specific Loading Method on page 5-32.
- Specifying a discard file in the control file using DISCARDFILE; see Specifying the Discard File on page 5-14.
- Specifying a maximum number of discards using DISCARDMAX; see Specifying the Discard File on page 5-14.
- Rejecting records due to duplicate values in a unique index or due to invalid data values; see Criteria for Rejected Records on page 5-13.

Control File for Case Study 4

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 from the logical record. Note that column 1 in each physical record should then contain either an asterisk or a nondata value.

Datafile for Case Study 4

The datafile for this case, ulcase4.dat, looks as follows. Asterisks are in the first position and, though not visible, a newline character is in position 20. 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    25 12-NOV-85
*7839 KING
PRESIDENT      5500.00           25 05-APR-83
*7934 MILLER
CLERK     7782 920.00            25 08-MAY-80
*7566 JONES
MANAGER   7839 3123.75           25 17-JUL-85
*7499 ALLEN
SALESMAN  7698 1600.00   300.00  25 3-JUN-84
*7654 MARTIN
SALESMAN  7698 1312.50  1400.00  25 21-DEC-85
*7658 CHAN
ANALYST   7566 3450.00           25 16-FEB-84
*     CHEN
ANALYST   7566 3450.00           25 16-FEB-84
*7658 CHIN
ANALYST   7566 3450.00           25 16-FEB-84
```

Rejected Records

The last two records are rejected, given two assumptions. If a unique index is 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.

Running Case Study 4

Take the following steps to run the case study.

1. Start SQL*Plus as scott/tiger by entering the following at the system prompt:

```
sqlplus scott/tiger
```

The SQL prompt is displayed.

2. At the SQL prompt, execute the SQL script for this case study, as follows:

```
SQL> @ulcase4
```

This prepares and populates tables for the case study and then returns you to the system prompt.

3. At the system prompt, invoke SQL*Loader and run the case study, as follows:

```
sqlldr USERID=scott/tiger CONTROL=ulcase4.ctl LOG=ulcase4.log
```

SQL*Loader loads the table, creates the log file, and returns you to the system prompt. You can check the log file to see the results of running the case study.

Log File for Case Study 4

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 256000 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:                  4608 bytes(64 rows)
Read   buffer bytes: 1048576

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

Run began on Wed Feb 27 14:28:53 2002
Run ended on Wed Feb 27 14:28:54 2002

Elapsed time was:     00:00:00.91
CPU time was:         00:00:00.13
```

Bad File for Case Study 4

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

```
*     CHEN         ANALYST
      7566         3450.00          25 16-FEB-84
*7658 CHIN         ANALYST
      7566         3450.00          25 16-FEB-84
```

Case Study 5: Loading Data into Multiple Tables

Case 5 demonstrates:

- Loading multiple tables. See Loading Data into Multiple Tables on page 5-43.
- Using SQL*Loader to break down repeating groups in a flat file and to load the data into normalized tables. In this way, one file record may generate multiple database rows.
- Deriving multiple logical records from each physical record. See Benefits of Using Multiple INTO TABLE Clauses on page 5-39.
- Using a WHEN clause. See Loading Records Based on a Condition on page 5-35.
- Loading the same field (empno) into multiple tables.

Control File for Case Study 5

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
2)   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 TABLE 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 nonblank 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.

Datafile for Case Study 5

```
1234 BAKER      10 9999 101 102 103
1234 JOKER      10 9999 777 888 999
2664 YOUNG      20 2893 425 abc 102
5321 OTOOLE     10 9999 321  55  40
2134 FARMER     20 4555 236 456
2414 LITTLE     20 5634 236 456  40
6542 LEE        10 4532 102 321  14
2849 EDDS       xx 4555     294  40
4532 PERKINS    10 9999  40
1244 HUNT       11 3452 665 133 456
123  DOOLITTLE  12 9940         132
1453 MACDONALD  25 5532     200
```

Running Case Study 5

Take the following steps to run the case study.

1. Start SQL*Plus as scott/tiger by entering the following at the system prompt:

   ```
   sqlplus scott/tiger
   ```

   The SQL prompt is displayed.
2. At the SQL prompt, execute the SQL script for this case study, as follows:

   ```
   SQL> @ulcase5
   ```

This prepares and populates tables for the case study and then returns you to the system prompt.

3. At the system prompt, invoke SQL*Loader and run the case study, as follows:

```
sqlldr USERID=scott/tiger CONTROL=ulcase5.ctl LOG=ulcase5.log
```

SQL*Loader loads the tables, creates the log file, and returns you to the system prompt. You can check the log file to see the results of running the case study.

Log File for Case Study 5

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 256000 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.
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:                   4096 bytes(64 rows)
Read   buffer bytes: 1048576

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

Run began on Wed Feb 27 14:34:33 2002
Run ended on Wed Feb 27 14:34:34 2002

Elapsed time was:     00:00:01.00
CPU time was:         00:00:00.22
```

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. Of the 9 records that met the `WHEN` clause criteria, two (`joker` and `young`) were rejected due to data errors.
4. Of the 10 records that met the `WHEN` clause criteria, three (`joker`, `young`, and `edds`) were rejected due to data errors.
5. Of the 9 records that met the `WHEN` clause criteria, three (`joker`, `young`, and `edds`) were rejected due to data errors.

Loaded Tables for Case Study 5

The following are sample SQL queries and their results:

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

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

SQL> SELECT * from PROJ order by EMPNO;

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

Case Study 6: Loading Data 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 9.

- How to specify the indexes for which the data is presorted. See Presorting Data for Faster Indexing on page 9-18.
- The NULLIF clause. See Using the WHEN, NULLIF, and DEFAULTIF Clauses on page 6-32.
- Loading all-blank numeric fields as NULL. See Loading All-Blank Fields on page 6-41.

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

Control File for Case Study 6

The control file is ulcase6.ctl.

```
   LOAD DATA
   INFILE 'ulcase6.dat'
   REPLACE
   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 statement identifies the indexes on which the data is sorted. This statement indicates that the datafile is sorted on the columns in the empix index. It 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 Using the WHEN, NULLIF, and DEFAULTIF Clauses on page 6-32.

Datafile for Case Study 6

```
7499 ALLEN      SALESMAN  7698  1600.00   300.00 30
7566 JONES      MANAGER   7839  3123.75          20
7654 MARTIN     SALESMAN  7698  1312.50  1400.00 30
7658 CHAN       ANALYST   7566  3450.00          20
7782 CLARK      MANAGER   7839  2572.50          10
```

```
7839 KING       PRESIDENT       5500.00          10
7934 MILLER     CLERK     7782   920.00          10
```

Running Case Study 6

Take the following steps to run the case study.

1. Start SQL*Plus as `scott/tiger` by entering the following at the system prompt:

   ```
   sqlplus scott/tiger
   ```

   The SQL prompt is displayed.

2. At the SQL prompt, execute the SQL script for this case study, as follows:

   ```
   SQL> @ulcase6
   ```

   This prepares and populates tables for the case study and then returns you to the system prompt.

3. At the system prompt, invoke SQL*Loader and run the case study, as follows. Be sure to specify `DIRECT=true.` Otherwise, conventional path is used as the default, which will result in failure of the case study.

   ```
   sqlldr USERID=scott/tiger CONTROL=ulcase6.ctl LOG=ulcase6.log DIRECT=true
   ```

   SQL*Loader loads the `emp` table using the direct path load method, creates the log file, and returns you to the system prompt. You can check the log file to see the results of running the case study.

Log File for Case Study 6

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.
Column array  rows :    5000
Stream buffer bytes:  256000
Read   buffer bytes: 1048576

Total logical records skipped:          0
Total logical records read:             7
Total logical records rejected:         0
Total logical records discarded:        0
Total stream buffers loaded by SQL*Loader main thread:        2
Total stream buffers loaded by SQL*Loader load thread:        0

Run began on Wed Feb 27 13:21:29 2002
Run ended on Wed Feb 27 13:21:32 2002

Elapsed time was:     00:00:02.96
CPU time was:         00:00:00.22
```

Case Study 7: Extracting Data from a Formatted Report

In this case study, SQL*Loader string-processing functions extract data from a formatted report. This example creates a trigger that uses the last value of unspecified fields. This case illustrates the following:

- Use of SQL*Loader with an INSERT trigger. See *Oracle9i Application Developer's Guide - Fundamentals* for more information on database triggers.
- Use of the SQL string to manipulate data; see Applying SQL Operators to Fields on page 6-50.
- Different initial and trailing delimiters. See Specifying Delimiters on page 6-24.
- Use of SYSDATE; see Setting a Column to the Current Date on page 6-56.
- Use of the TRAILING NULLCOLS clause; see TRAILING NULLCOLS Clause on page 5-37.
- Ambiguous field length warnings; see Conflicting Native Datatype Field Lengths on page 6-21 and Conflicting Field Lengths for Character Datatypes on page 6-28.
- Use of a discard file. See Specifying the Discard File in the Control File on page 5-14.

Creating a BEFORE INSERT Trigger

In this case study, a BEFORE INSERT trigger is required to fill in the 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 global variables package are created when you execute the ulcase7s.sql script.

The package defining the global variables looks as follows:

```
CREATE OR REPLACE PACKAGE uldemo7 AS   -- Global Package Variables
    last_deptno   NUMBER(2);
    last_job      VARCHAR2(9);
    last_mgr      NUMBER(4);
    END uldemo7;
/
```

The definition of the INSERT trigger looks as follows:

```
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 FOR EACH ROW clause is important. If it was not specified, the INSERT trigger would only execute once for each array of inserts, because SQL*Loader uses the array interface.

Be sure to execute the ulcase7e.sql script to drop the INSERT trigger and the global variables package before continuing with the rest of the case studies. See Running Case Study 7 on page 10-31.

Control File for Case Study 7

The control file is ulcase7.ctl.

```
   LOAD DATA
   INFILE 'ulcase7.dat'
   DISCARDFILE 'ulcase7.dis'
   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 seven fields are found instead of the expected eight.
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. See Log File for Case Study 7 on page 10-32. The length is in bytes with the default byte-length semantics. If character-length semantics were used instead, this length would be in characters.
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 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 WHITESPACE 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.

Datafile for Case Study 7

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/Commission
----  --------  --------  -----  --------  -----  -----------------
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
```

Running Case Study 7

Take the following steps to run the case study.

1. Start SQL*Plus as `scott/tiger` by entering the following at the system prompt:

   ```
   sqlplus scott/tiger
   ```

   The SQL prompt is displayed.

2. At the SQL prompt, execute the SQL script for this case study, as follows:

   ```
   SQL> @ulcase7s
   ```

   This prepares and populates tables for the case study and then returns you to the system prompt.

3. At the system prompt, invoke SQL*Loader and run the case study, as follows:

   ```
   sqlldr USERID=scott/tiger CONTROL=ulcase7.ctl LOG=ulcase7.log
   ```

SQL*Loader extracts data from the report, creates the log file, and returns you to the system prompt. You can check the log file to see the results of running the case study.

4. After running this case study, you *must* drop the insert triggers and global-variable package before you can continue with the rest of the case studies. To do this, execute the `ulcase7e.sql` script as follows:

```
SQL> @ulcase7e
```

Log File for Case Study 7

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: ulcase7.dis
 (Allow all discards)

Number to load: ALL
Number to skip: 0
Errors allowed: 50
Bind array:     64 rows, maximum of 256000 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:                  51584 bytes(64 rows)
Read   buffer bytes: 1048576

   Total logical records skipped:          0
   Total logical records read:            13
   Total logical records rejected:         0
2) Total logical records discarded:        7

Run began on Wed Feb 27 14:54:03 2002
Run ended on Wed Feb 27 14:54:04 2002

Elapsed time was:     00:00:00.99
CPU time was:         00:00:00.21
```

Notes:

1. A warning is generated by the difference between the specified length and the length derived from the position specification.
2. There are six header lines at the top of the report: 3 of them contain text and 3 of them are blank. All of them are rejected, as is the blank separator line in the middle.

Case Study 8: Loading Partitioned Tables

Case 8 demonstrates:

- Partitioning of data. See *Oracle9i Database Concepts* for more information on partitioned data concepts.
- Explicitly defined field positions and datatypes.
- Loading using the fixed record length option. See Input Data and Datafiles on page 3-4.

Control File for Case Study 8

The control file is `ulcase8.ctl`. It loads the `lineitem` table with fixed-length records, partitioning the data according to shipment date.

```
LOAD DATA
1)  INFILE 'ulcase8.dat' "fix 129"
BADFILE 'ulcase8.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 bytes in this example).

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 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
)
```

Datafile for Case Study 8

The datafile for this case, `ulcase8.dat`, looks as follows. Each record is 128 bytes in length. Five 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
```

Running Case Study 8

Take the following steps to run the case study.

1. Start SQL*Plus as `scott/tiger` by entering the following at the system prompt:

   ```
   sqlplus scott/tiger
   ```

   The SQL prompt is displayed.

2. At the SQL prompt, execute the SQL script for this case study, as follows:

   ```
   SQL> @ulcase8
   ```

   This prepares and populates tables for the case study and then returns you to the system prompt.

3. At the system prompt, invoke SQL*Loader and run the case study, as follows:

   ```
   sqlldr USERID=scott/tiger CONTROL=ulcase8.ctl LOG=ulcase8.log
   ```

   SQL*Loader partitions and loads the data, creates the log file, and returns you to the system prompt. You can check the log file to see the results of running the case study.

Log File for Case Study 8

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:     ulcase8.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 256000 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:                  11008 bytes(64 rows)
Read   buffer bytes: 1048576

Total logical records skipped:          0
Total logical records read:            10
Total logical records rejected:         7
Total logical records discarded:        0

Run began on Wed Feb 27 15:02:28 2002
Run ended on Wed Feb 27 15:02:29 2002

Elapsed time was:     00:00:01.37
CPU time was:         00:00:00.20
```

Case Study 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 for Case Study 9

The control file is ulcase9.ctl. It loads new records into emp, including a resume for each employee. Each resume is contained in a separate 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 data field to which it is mapped. See Specifying Filler Fields on page 6-6 for more information.
2. The resume column is loaded as a CLOB. The LOBFILE function specifies the field name in which the name of the file that contains data for the LOB field is provided. See Loading LOB Data from LOBFILEs on page 7-23 for more information.

Datafiles for Case Study 9

```
>>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 - Salesman 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
```

Running Case Study 9

Take the following steps to run the case study.

1. Start SQL*Plus as `scott/tiger` by entering the following at the system prompt:

   ```
   sqlplus scott/tiger
   ```

   The SQL prompt is displayed.

2. At the SQL prompt, execute the SQL script for this case study, as follows:

   ```
   SQL> @ulcase9
   ```

   This prepares and populates tables for the case study and then returns you to the system prompt.

3. At the system prompt, invoke SQL*Loader and run the case study, as follows:

   ```
   sqlldr USERID=scott/tiger CONTROL=ulcase9.ctl LOG=ulcase9.log
   ```

SQL*Loader loads the emp table, creates the log file, and returns you to the system prompt. You can check the log file to see the results of running the case study.

Log File for Case Study 9

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 256000 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     *  EOF      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:                 132096 bytes(64 rows)
Read   buffer bytes: 1048576

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

Run began on Wed Feb 27 15:06:49 2002
Run ended on Wed Feb 27 15:06:50 2002

Elapsed time was:     00:00:01.01
CPU time was:         00:00:00.20
```

Case Study 10: Loading REF Fields and VARRAYs

Case 10 demonstrates:

- Loading a customer table that has a primary key as its OID and stores order items in a VARRAY.
- Loading an order table that has a reference to the customer table and the order items in a VARRAY.

Note: Case study 10 requires that the COMPATIBILITY parameter be set to 8.1.0 or higher in your initialization parameter file. Otherwise, the table cannot be properly created and you will receive an error message. For more information on setting the COMPATIBILITY parameter, see *Oracle9i Database Migration*.

Control File for Case Study 10

```
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 data field to which it is mapped. See Specifying Filler Fields on page 6-6 for more information.

2. This field is created as a REF field. See Loading REF Columns on page 7-15 for more information.

3. `item_list` is stored in a VARRAY.

4. The second occurrence 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 parentheses. See Loading Column Objects on page 7-1 for more information.
6. The data is contained in the control file and is preceded by the BEGINDATA parameter.

Running Case Study 10

Take the following steps to run the case study.

1. Start SQL*Plus as scott/tiger by entering the following at the system prompt:

   ```
   sqlplus scott/tiger
   ```

   The SQL prompt is displayed.

2. At the SQL prompt, execute the SQL script for this case study, as follows:

   ```
   SQL> @ulcase10
   ```

   This prepares and populates tables for the case study and then returns you to the system prompt.

3. At the system prompt, invoke SQL*Loader and run the case study, as follows:

   ```
   sqlldr USERID=scott/tiger CONTROL=ulcase10.ctl LOG=ulcase10.log
   ```

   SQL*Loader loads the data, creates the log file, and returns you to the system prompt. You can check the log file to see the results of running the case study.

Log File for Case Study 10

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 256000 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:                 149120 bytes(64 rows)
Read   buffer bytes: 1048576

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

Run began on Wed Feb 27 14:05:29 2002
Run ended on Wed Feb 27 14:05:31 2002

Elapsed time was:     00:00:02.07
CPU time was:         00:00:00.20
```

Case Study 11: Loading Data in the Unicode Character Set

In this case study, SQL*Loader loads data from a datafile in a Unicode character set. This case study parallels case study 3, except that it uses the character set UTF16 and a maximum length is specified for the empno and deptno fields. The data must be in a separate datafile because the CHARACTERSET keyword is specified. This case study demonstrates the following:

- Using SQL*Loader to load data in the Unicode character set, UTF16.
- Using SQL*Loader to load data in a fixed-width multibyte character set.
- Using character-length semantics.
- Using SQL*Loader to load data in little endian byte order. SQL*Loader checks the byte order of the system on which it is running. If necessary, SQL*Loader

swaps the byte order of the data to ensure that any byte-order-dependent data is correctly loaded.

Control File for Case Study 11

The control file is `ulcase11.ctl`.

```
LOAD DATA
1) CHARACTERSET UTF16
2) BYTEORDER LITTLE
INFILE ulcase11.dat
REPLACE

INTO TABLE emp
3) FIELDS TERMINATED BY X’002c’ OPTIONALLY ENCLOSED BY X’0022’
4) (empno INTEGER EXTERNAL (5), ename, job, mgr,
 hiredate DATE(20) "DD-Month-YYYY",
 sal, comm,
5) deptno   CHAR(5) TERMINATED BY ":",
 projno,
 loadseq  SEQUENCE(MAX,1) )
```

Notes:

1. The character set specified with the `CHARACTERSET` keyword is `UTF16`. SQL*Loader will convert the data from the UTF16 character set to the datafile character set. This line also tells SQL*Loader to use character-length semantics for the load.
2. `BYTEORDER LITTLE` tells SQL*Loader that the data in the datafile is in little endian byte order. SQL*Loader checks the byte order of the system on which it is running to determine if any byte-swapping is necessary. In this example, all the character data in UTF16 is byte-order dependent.
3. The `TERMINATED BY` and `OPTIONALLY ENCLOSED BY` clauses both specify hexadecimal strings. The `X’002c’` is the encoding for a comma (,) in UTF-16 big endian format. The `X’0022’` is the encoding for a double quotation mark (") in big endian format. Because the datafile is in little endian format, SQL*Loader swaps the bytes before checking for a match.

   If these clauses were specified as character strings instead of hexadecimal strings, SQL*Loader would convert the strings to the datafile character set (UTF16) and byte-swap as needed before checking for a match.

4. Because character-length semantics are used, the maximum length for the `empno`, `hiredate`, and `deptno` fields is interpreted as characters, not bytes.
5. The `TERMINATED BY` clause for the `deptno` field is specified using the character string ":". SQL*Loader converts the string to the datafile character set (UTF16) and byte-swaps as needed before checking for a match.

   See Also:

   - Handling Different Character Encoding Schemes on page 5-16
   - Byte Ordering on page 6-37

Datafile for Case Study 11

```
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, 300.00, 30:103
7654, "Martin", "Salesman", 7698, 28-September-1981, 1312.50, 1400.00, 30:103
7658, "Chan", "Analyst", 7566, 03-May-1982, 3450,, 20:101
```

Running Case Study 11

Take the following steps to run the case study.

1. Start SQL*Plus as `scott/tiger` by entering the following at the system prompt:

   ```
   sqlplus scott/tiger
   ```

   The SQL prompt is displayed.
2. At the SQL prompt, execute the SQL script for this case study, as follows:

   ```
   SQL> @ulcase11
   ```

   This prepares the table `emp` for the case study and then returns you to the system prompt.
3. At the system prompt, invoke SQL*Loader and run the case study, as follows:

   ```
   sqlldr USERID=scott/tiger CONTROL=ulcase11.ctl LOG=ulcase11.log
   ```

SQL*Loader loads the table emp, creates the log file, and returns you to the system prompt. You can check the log file to see the results of running the case study.

Log File for Case Study 11

The following shows a portion of the log file for case study 11:

```
Control File:   ulcase11.ctl
Character Set utf16 specified for all input.
1) Using character length semantics.
2) Byteorder little endian specified.
Processing datafile as little endian.
3) SQL*Loader running on a big endian platform. Swapping bytes where needed.

Data File:      ulcase11.dat
  Bad File:     ulcase11.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 256000 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
------------------------------ ---------- ----- ---- ---- ---------------------
4) EMPNO                            FIRST   10   ,  O(") CHARACTER
ENAME                               NEXT     *   ,  O(") CHARACTER
JOB                                 NEXT     *   ,  O(") CHARACTER
MGR                                 NEXT     *   ,  O(") CHARACTER
4) HIREDATE                         NEXT    40   ,  O(") DATE DD-Month-YYYY
SAL                                 NEXT     *   ,  O(") CHARACTER
COMM                                NEXT     *   ,  O(") CHARACTER
DEPTNO                              NEXT    10   :  O(") CHARACTER
4) 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:                  104768 bytes(64 rows)
Read   buffer bytes: 1048576

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

Run began on Wed Feb 27 16:33:47 2002
Run ended on Wed Feb 27 16:33:49 2002

Elapsed time was:     00:00:01.74
CPU time was:         00:00:00.20
```

Notes:

1. SQL*Loader used character-length semantics for this load. This is the default if the character set is UTF16. This means that length checking for the maximum sizes is in characters (see item number 4 in this list).
2. `BYTEORDER LITTLE` was specified in the control file. This tells SQL*Loader that the byte order for the UTF16 character data in the datafile is little endian.
3. This message only appears when SQL*Loader is running on a system with the opposite byte order (in this case, big endian) from the datafile's byte order. It indicates that SQL*Loader detected that the byte order of the datafile is opposite from the byte order of the system on which SQL*Loader is running. Therefore, SQL*Loader had to byte-swap any byte-order-dependent data (in this case, all the UTF16 character data).
4. The maximum lengths under the `len` heading are in bytes even though character-length semantics were used. However, the maximum lengths are adjusted based on the maximum size, in bytes, of a character in UTF16. All characters in UTF16 are 2 bytes. Therefore, the sizes given for `empno` and `projno` (5) are multiplied by 2, resulting in a maximum size of 10 bytes.

   Similarly, the `hiredate` maximum size (20) is multiplied by 2, resulting in a maximum size of 40 bytes.

Loaded Tables for Case Study 11

To see the results of this execution of SQL*Loader, execute the following query at the SQL prompt:

```
SQL> SELECT * FROM emp;
```

The results of the query look as follows (the formatting may be slightly different on your display):

```
 EMPNO ENAME  JOB         MGR HIREDATE       SAL  COMM  DEPTNO PROJNO  LOADSEQ
------ ------ --------- ----- --------- -------- ----- ------- ------ --------
  7782 Clark  Manager    7839 09-JUN-81  2572.50            10    101        1


  7839 King   President       17-NOV-81  5500.00            10    102        2


  7934 Miller Clerk      7782 23-JAN-82   920.00            10    102        3


  7566 Jones  Manager    7839 02-APR-81  3123.75            20    101        4

 EMPNO ENAME  JOB         MGR HIREDATE       SAL  COMM  DEPTNO PROJNO  LOADSEQ
------ ------ --------- ----- --------- -------- ----- ------- ------ --------

  7499 Allen  Salesman   7698 20-FEB-81  1600.00   300      30    103        5


  7654 Martin Salesman   7698 28-SEP-81  1312.50  1400      30    103        6


  7658 Chan   Analyst    7566 03-MAY-82  3450.00            20    101        7


7 rows selected.
```

The output for the table is displayed in the character set US7ASCII, which is the normal default character set when the NLS_LANG parameter is not defined. SQL*Loader converts the output from the database character set, which normally defaults to WE8DEC, to the character set specified for your session by the NLS_LANG parameter.

Part III

External Tables

The chapters in this section describe the use of external tables.

Chapter 11, "External Tables Concepts"

This chapter describes basic concepts about external tables.

Chapter 12, "External Tables Access Parameters"

This chapter describes the access parameters used to interface with the external tables API.

11

External Tables Concepts

The Oracle9*i* external tables feature is a complement to existing SQL*Loader functionality. It allows you to access data in external sources as if it were in a table in the database.

External tables are read-only. No data manipulation language (DML) operations or index creation is allowed on an external table. Therefore, SQL*Loader may be the better choice in data loading situations that require additional indexing of the staging table. See Behavior Differences Between SQL*Loader and External Tables on page 11-7 for more information on how load behavior differs between SQL*Loader and external tables.

To use the external tables feature, you must have some knowledge of the file format and record format of the datafiles on your platform. You must also know enough about SQL to be able to create an external table and perform queries against it.

This chapter discusses the following topics:

- The Access Driver
- External Table Restrictions
- Location of Datafiles and Output Files
- Using External Tables to Load Data
- Parallel Access to External Tables
- Performance Hints When Using External Tables
- Behavior Differences Between SQL*Loader and External Tables

The Access Driver

An external table describes how the external table layer must present the data to the server. The access driver and the external table layer transform the data in the datafile to match the external table definition.

When you create an external table of a particular type, you provide access parameters that describe the external data source. If you do not specify a type for the external table, then the ORACLE_LOADER type is used as a default. For a description of the access parameters for the ORACLE_LOADER type, see Chapter 12.

The description of the data in the data source is separate from the definition of the external table. This means that:

- The source file can contain more or fewer fields than there are columns in the external table
- The datatypes for fields in the data source can be different from the columns in the external table

The access driver ensures that data from the data source is processed so that it matches the definition of the external table.

In the following example, a traditional table named emp is defined along with an external table named emp_load.

```
CREATE TABLE emp (emp_no CHAR(6), last_name CHAR(25), first_name CHAR(20), middle_initial
CHAR(1));

CREATE TABLE emp_load (employee_number CHAR(5), employee_last_name CHAR(20),
                       employee_first_name CHAR(15), employee_middle_name CHAR(15))
ORGANIZATION EXTERNAL (TYPE ORACLE_LOADER DEFAULT DIRECTORY ext_tab_dir
                      ACCESS PARAMETERS (RECORDS FIXED 62 FIELDS (employee_number INTEGER(2),
                                                                 employee_dob CHAR(20),
                                                                 employee_last_name CHAR(18),
                                                                 employee_first_name CHAR(11),
                                                                 employee_middle_name CHAR(11)))
                                          LOCATION ('foo.dat'));

INSERT INTO emp (emp_no, first_name, middle_initial, last_name)
(SELECT employee_number, employee_first_name,
        substr(employee_middle_name, 1, 1),
        employee_last_name
FROM emp_load);
```

Note the following in the preceding example:

- The employee_number field in the datafile is converted to a character string for the employee_number field in the external table.
- The datafile contains an employee_dob field that is not loaded into any field in the table.
- The substr function is used on the employee_middle_name column in the external table to generate the value for middle_initial in table emp.

External Table Restrictions

This section lists what external tables does *not* do and also describes some processing restrictions.

- An external table does not describe any data that is stored in the database.
- An external table does not describe how data is stored in the external source. This is the function of the access parameters.
- An external table is a read-only source. You cannot perform insert operations into an external table, nor can you update records in an external table.
- Column processing - external tables can return a different number of rows depending on what columns are queried. To minimize the amount of data conversion and data handling required to execute a query, the external tables access driver only processes columns that are referenced in the query. This means that a row that is rejected because a column in the row causes a datatype conversion error will not get rejected in a different query if the query does not reference that column.
- Handling of byte-order marks during a load - in an external table load for which the datafile character set is UTF8 or UTF16, it is not possible to suppress checking for byte-order marks. Suppression of byte-order mark checking is only necessary if the beginning of the datafile contains binary data that matches the byte-order mark encoding. (It is possible to suppress byte-order mark checking with SQL*Loader loads.) Note that checking for a byte-order mark does not mean that a byte-order mark must be present in the datafile. If no byte-order mark is present, the byte order of the server platform is used.

Location of Datafiles and Output Files

The access driver runs inside of the database server. This is different from SQL*Loader, which is a client program that sends the data to be loaded over to the server. This difference has the following implications:

- The server must have access to any files to be loaded by the access driver.
- The server must create and write the files created by the access driver: log file, bad file, and discard file.

The access driver does not allow you to specify random names for a file. This is because the server may have access to files that you do not, and allowing you to read this data would affect security. Similarly, you cannot specify a location for an output file, because the server could overwrite a file that you might not normally have privileges to delete.

Instead, you are required to specify directory objects as the locations from which to read files and write files. A directory object maps a name to a directory name on the file system. For example, the following statement creates a directory object named `load_src`.

```
create directory load_src as '/usr/apps/datafiles';
```

Directory objects can be created by DBAs or by any user with the `CREATE ANY DIRECTORY` privilege. After a directory is created, the user creating the directory object needs to grant `READ` or `WRITE` permission on the directory to other users. For example, to allow the server to read files on behalf of user `scott` in the directory named by `load_src`, the user who created the directory object must execute the following command:

```
GRANT READ ON DIRECTORY load_src TO scott;
```

The name of the directory object can appear in the following places in a `CREATE TABLE...ORGANIZATION EXTERNAL` statement:

- The default directory clause, which specifies the default directory to use for all input and output files that do not explicitly name a directory object.
- The `LOCATION` clause, which lists all of the datafiles for the external table. The files are named in the form *`directory:file`*. The *`directory`* portion is optional. If it is missing, the default directory is used as the directory for the file.
- The access parameters where output files are named. The files are named in the form *`directory:file`*. The *`directory`* portion is optional. If it is missing, the default directory is used as the directory for the file. Syntax in the access parameters allows you to indicate that a particular output file should not be created. This is useful if you do not care about the output files or if you do not have write access to any directory objects.

The `SYS` user is the only user that can own directory objects, but the `SYS` user can grant other users the privilege to create directory objects. Note that `READ` or `WRITE` permission to a directory object only means that the Oracle database server will read or write that file on your behalf. You are not given direct access to those files outside of the Oracle database server unless you have the appropriate operating system privileges. Similarly, the Oracle database server requires permission from the operating system to read and write files in the directories.

Using External Tables to Load Data

The main use for external tables is as a row source for loading data into a real table in the database. After you create an external table, you can issue a `CREATE TABLE AS SELECT` or `INSERT INTO... AS SELECT` statement using the external table as the source of the `SELECT` clause. Remember that external tables are read-only, so you cannot insert into them or update records in them.

When the external table is accessed through a SQL statement, the fields of the external table can be used just like any other field in a normal table. In particular, the fields can be used as arguments for any SQL built-in function, PL/SQL function, or Java function. This allows you to manipulate the data from the external source.

Although external tables cannot contain a column object, you can use constructor functions to build a column object from attributes in the external table. For example, assume a table in the database is defined as follows:

```
CREATE TYPE student_type AS object (
student_no CHAR(5),
name CHAR(20))
/

CREATE TABLE roster (
  student student_type,
  grade CHAR(2));
```

Also assume there is an external table defined as follows:

```
CREATE TABLE roster_data (
  student_no CHAR(5),
  name CHAR(20),
  grade CHAR(2))
  ORGANIZATION EXTERNAL (TYPE ORACLE_LOADER DEFAULT DIRECTORY ext_tab_dir
                         ACCESS PARAMETERS (FIELDS TERMINATED BY ',')
                         LOCATION ('foo.dat'));
```

To load table `roster` from `roster_data`, you would specify something similar to the following:

```
INSERT INTO roster (student, grade)
  (SELECT student_type(student_no, name), grade FROM roster_data);
```

Parallel Access to External Tables

To enable external table support of parallel processing on the datafiles, use the PARALLEL clause when you create the external table. The access driver attempts to divide large datafiles into chunks that can be processed separately.

The following file, record, and data characteristics make it impossible for a file to be processed in parallel:

- Sequential data sources (such as a tape drive or pipe)
- Data in any multibyte character set whose character boundaries cannot be determined starting at an arbitrary byte in the middle of a string

  This restriction does not apply to any datafile with a fixed number of bytes per record.
- Records with the VAR format

Performance Hints When Using External Tables

When you monitor performance, the most important measurement is the elapsed time for a load. Other important measurements are CPU usage, memory usage, and I/O rates.

You can alter performance by increasing or decreasing the degree of parallelism. The degree of parallelism indicates the number of access drivers that can be started to process the datafiles. The degree of parallelism allows you to choose on a scale between slower load with little resource usage and faster load with all resources utilized. The access driver cannot automatically tune itself, because it cannot determine how many resources you want to dedicate to the access driver.

Performance can also sometimes be increased with use of date cache functionality. By using the date cache to specify the number of unique dates anticipated during the load, you can reduce the number of date conversions done when many duplicate date or timestamp values are present in the input data. The date cache functionality provided by external tables is identical to the date cache functionality provided by SQL*Loader. See Specifying a Value for the Date Cache on page 9-22 for a detailed description.

In addition to changing the degree of parallelism and using the date cache to improve performance, consider the following information:

- Fixed-length records are processed faster than records terminated by a string.
- Fixed-length fields are processed faster than delimited fields.
- Single-byte character sets are the fastest to process.
- Fixed-width character sets are faster to process than varying-width character sets.
- Byte-length semantics for varying-width character sets are faster to process than character-length semantics.
- Single-character delimiters for record terminators and field delimiters are faster to process than multicharacter delimiters.
- Having the character set in the datafile match the character set of the database is faster than a character set conversion.
- Having datatypes in the datafile match the datatypes in the database is faster than datatype conversion.
- Not writing rejected rows to a reject file is faster because of the reduced overhead of not writing the rows.
- Condition clauses (including `WHEN`, `NULLIF`, and `DEFAULTIF`) slow down processing.

The access driver takes advantage of multithreading to streamline the work as much as possible.

Behavior Differences Between SQL*Loader and External Tables

This section describes important differences between loading data with external tables as opposed to loading data with SQL*Loader conventional and direct path loads.

Multiple Primary Input Datafiles

If there are multiple primary input datafiles with SQL*Loader loads, a bad file and a discard file are created for each input datafile. With external table loads, there is only one bad file and one discard file for all input datafiles. If parallel access drivers are used for the external table load, each access driver has its own bad file and discard file.

Syntax and Datatypes

The following are not supported with external table loads:

- Use of CONTINUEIF or CONCATENATE to combine multiple physical records into a single logical record.
- Loading of the following SQL*Loader datatypes: GRAPHIC, GRAPHIC EXTERNAL, and VARGRAPHIC
- Use of the following database column types: CLOBs, NCLOBs, BLOBs, LONGs, nested tables, VARRAYs, REFs, primary key REFs, and SIDs

Rejected Rows

With SQL*Loader, if the SEQUENCE parameter is used and there are rejected rows, the rejected row still updates the sequence number value. With external tables, if the SEQUENCE parameter is used, rejected rows do not update the sequence number value. For example, suppose you load 5 rows with sequence numbers beginning with 1 and incrementing by 1. In SQL*Loader, if rows 2 and 4 are rejected, the successfully loaded rows are assigned the sequence numbers 1, 3, and 5. In an external table load, the successfully loaded rows are assigned the sequence numbers 1, 2, and 3.

Byte-Order Marks

With SQL*Loader, if a primary datafile uses a Unicode character set (UTF8 or UTF16) and it also contains a byte-order mark (BOM), then the byte-order mark is written at the beginning of the corresponding bad and discard files. With external table loads, the byte-order mark is not written at the beginning of the bad and discard files.

Default Character Sets and Date Masks

For fields in a datafile, it is the *client's* NLS environment that determines the default character set and date masks. For fields in external tables, it is the *server's* NLS environment that determines the default character set and date masks.

12

External Tables Access Parameters

The access parameters described in this chapter provide the interface to the external table access driver. You specify access parameters when you create the external table. This chapter describes the syntax for the access parameters for the default access driver.

To use the information in this chapter, you must have some knowledge of the file format and record format (including character sets and field datatypes) of the datafiles on your platform. You must also know enough about SQL to be able to create an external table and perform queries against it.

You may find it helpful to use the EXTERNAL_TABLE=GENERATE_ONLY parameter in SQL*Loader to get the proper access parameters for a given SQL*Loader control file. When you specify GENERATE_ONLY, all the SQL statements needed to do the load using external tables, as described in the control file, are placed in the SQL*Loader log file. These SQL statements can be edited and customized. The actual load can be done later without the use of SQL*Loader by executing these statements in SQL*Plus.

See Also:

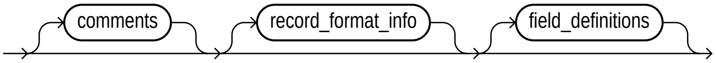
- EXTERNAL_TABLE on page 4-7
- Log File Created When EXTERNAL_TABLE=GENERATE_ONLY on page 8-8

Notes:

- It is sometimes difficult to describe syntax without using other syntax that is not documented until later in the chapter. If it is not clear what some syntax is supposed to do, you might want to skip ahead and read about that particular element.
- Many examples in this chapter show a `CREATE TABLE...ORGANIZATION EXTERNAL` statement followed by a sample of contents of the datafile for the external table. These contents are not part of the `CREATE TABLE` statement, but are shown to help complete the example.

access_parameters Clause

The access parameters clause contains comments, record formatting, and field formatting information. The syntax for the `access_parameters` clause is as follows:



comments

Comments are lines that begin with two dashes followed by text. Comments must be placed *before* any access parameters, for example:

```
--This is a comment
--This is another comment
RECORDS DELIMITED BY NEWLINE
```

All text to the right of the double hyphen is ignored, until the end of the line.

record_format_info

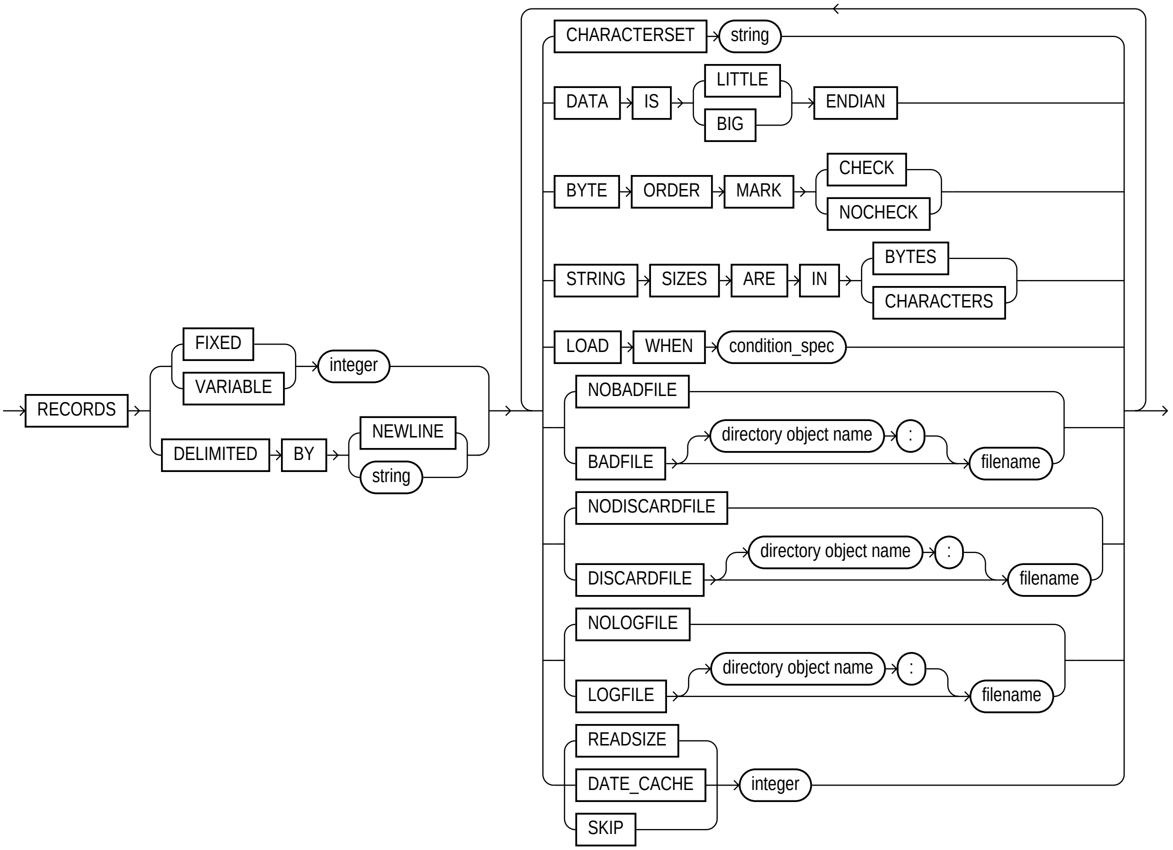
The `record_format_info` clause contains information about the record, such as its format, the character set of the data, and what rules are used to exclude records from being loaded. The `record_format_info` clause is optional. For a full description of the syntax, see record_format_info Clause on page 12-3.

field_definitions

The `field_definitions` clause is used to describe the fields in the datafile. If a datafile field has the same name as a column in the external table, then the data from the field is used for that column. For a full description of the syntax, see field_definitions Clause on page 12-15.

record_format_info Clause

The `record_format_info` clause contains information about the record, such as its format, the character set of the data, and what rules are used to exclude records from being loaded. The `record_format_info` clause is optional. If the clause is not specified, the default value is `RECORDS DELIMITED BY NEWLINE`. The syntax for the `record_format_info` clause is as follows:



FIXED length

The `FIXED` clause is used to identify the records as all having a fixed size of length bytes. The size specified for `FIXED` records must include any record termination characters, such as newlines. Compared to other record types, fixed-length fields in fixed-length records are the easiest field and record formats for the access driver to process.

The following is an example of using FIXED records. It assumes there is a 1-byte newline character at the end of each record in the datafile. It is followed by a sample of the datafile that can be used to load it.

```
CREATE TABLE emp_load (first_name CHAR(15), last_name CHAR(20), year_of_birth CHAR(4))
  ORGANIZATION EXTERNAL (TYPE ORACLE_LOADER DEFAULT DIRECTORY ext_tab_dir
                         ACCESS PARAMETERS (RECORDS FIXED 20 FIELDS (first_name CHAR(7),
                                                                     last_name CHAR(8),
                                                                     year_of_birth CHAR(4)))
                         LOCATION ('foo.dat'));

Alvin  Tolliver1976
KennethBaer    1963
Mary   Dube    1973
```

VARIABLE size

The VARIABLE clause is used to indicate that the records have a variable length and that each record is preceded by a character string containing a number with the count of bytes for the record. The length of the character string containing the count field is the size argument that follows the VARIABLE parameter. Note that size indicates a count of bytes, not characters. The count at the beginning of the record must include any record termination characters, but it does not include the size of the count field itself. The number of bytes in the record termination characters can vary depending on how the file is created and on what platform it is created.

The following is an example of using VARIABLE records. It assumes there is a 1-byte newline character at the end of each record in the datafile. It is followed by a sample of the datafile that can be used to load it.

```
CREATE TABLE emp_load (first_name CHAR(15), last_name CHAR(20), year_of_birth CHAR(4))
  ORGANIZATION EXTERNAL (TYPE ORACLE_LOADER DEFAULT DIRECTORY ext_tab_dir
                         ACCESS PARAMETERS (RECORDS VARIABLE 2 FIELDS TERMINATED BY ','
                                           (first_name CHAR(7),
                                            last_name CHAR(8),
                                            year_of_birth CHAR(4)))
                         LOCATION ('foo.dat'));

21Alvin,Tolliver,1976,
19Kenneth,Baer,1963,
16Mary,Dube,1973,
```

DELIMITED BY

The DELIMITED BY clause is used to indicate the characters that identify the end of a record.

If DELIMITED BY NEWLINE is specified, then the actual value used is platform-specific. On UNIX platforms, NEWLINE is assumed to be "\n". On Windows NT, NEWLINE is assumed to be "\r\n".

If DELIMITED BY *string* is specified, *string* can either be text or a series of hexadecimal digits. If it is text, then the text is converted to the character set of the datafile and the result is used for identifying record boundaries. See string on page 12-11.

If the following conditions are true, then you must use hexadecimal digits to identify the delimiter:

- The character set of the access parameters is different from the character set of the datafile
- Some characters in the delimiter string cannot be translated into the character set of the datafile

The hexadecimal digits are converted into bytes, and there is no character set translation performed on the hexadecimal string.

If the end of the file is found before the record terminator, the access driver proceeds as if a terminator was found, and all unprocessed data up to the end of the file is considered part of the record.

Caution: Do not include any binary data, including binary counts for VARCHAR and VARRAW, in a record that has delimiters. Doing so could cause errors or corruption, because the binary data will be interpreted as characters during the search for the delimiter.

The following is an example of using DELIMITED BY records.

```
CREATE TABLE emp_load (first_name CHAR(15), last_name CHAR(20), year_of_birth CHAR(4))
  ORGANIZATION EXTERNAL (TYPE ORACLE_LOADER DEFAULT DIRECTORY ext_tab_dir
                         ACCESS PARAMETERS (RECORDS DELIMITED BY '|' FIELDS TERMINATED BY ','
                                            (first_name CHAR(7),
                                             last_name CHAR(8),
                                             year_of_birth CHAR(4)))
                         LOCATION ('foo.dat'));
```

```
Alvin,Tolliver,1976|Kenneth,Baer,1963|Mary,Dube,1973
```

CHARACTERSET

The CHARACTERSET *string* clause identifies the character set of the datafile. If a character set is not specified, the data is assumed to be in the default character set for the database. See string on page 12-11.

> **Note:** Client-side NLS settings have no effect on the character set used for the database.

> **See Also:** *Oracle9i Database Globalization Support Guide* for a listing of Oracle-supported character sets

DATA IS...ENDIAN

The DATA IS...ENDIAN clause indicates the endianness of data whose byte order may vary depending on the platform that generated the datafile. Fields of the following types are affected by this clause:

- INTEGER
- UNSIGNED INTEGER
- FLOAT
- DOUBLE
- VARCHAR (numeric count only)
- VARRAW (numeric count only)
- Any character datatype in the UTF16 character set
- Any string specified by RECORDS DELIMITED BY *string* and in the UTF16 character set

Common platforms that generate little endian data include Windows 98 and Windows NT. Big endian platforms include Sun Solaris and IBM MVS. If the DATA IS...ENDIAN clause is not specified, then the data is assumed to have the same endianness as the platform where the access driver is running. UTF16 datafiles may have a mark at the beginning of the file indicating the endianness of the data. This mark will override the DATA IS...ENDIAN clause.

BYTE ORDER MARK (CHECK | NOCHECK)

The BYTE ORDER MARK clause is used to specify whether or not the datafile should be checked for the presence of a byte-order mark (BOM).

BYTE ORDER MARK NOCHECK indicates that the datafile should not be checked for a BOM and that all the data in the datafile should be read as data.

BYTE ORDER MARK CHECK indicates that the datafile should be checked for a BOM. This is the default behavior for a datafile in a Unicode character set.

The following are examples of some possible scenarios:

- If the data is specified as being little or big endian and CHECK is specified and it is determined that the specified endianness does not match the datafile, then an error is returned. For example, suppose you specify the following:

  ```
  DATA IS LITTLE ENDIAN
  BYTE ORDER MARK CHECK
  ```

  If the BOM is checked in the Unicode datafile and the data is actually big endian, an error is returned because you specified little endian.
- If a BOM is not found and no endianness is specified with the DATA IS...ENDIAN parameter, then the endianness of the platform is used.
- If BYTE ORDER MARK NOCHECK is specified and the DATA IS...ENDIAN parameter specified an endianness, then that value is used. Otherwise, the endianness of the platform is used.

See Also: Byte Ordering on page 6-37

STRING SIZES ARE IN

The STRING SIZES ARE IN clause is used to indicate whether the lengths specified for character strings are in bytes or characters. If not specified, the access driver uses the mode that the database uses. Character types with embedded lengths (such as VARCHAR) are also affected by this clause. If this clause is specified, the embedded lengths are a character count, not a byte count. Specifying STRING SIZES ARE IN CHARACTERS is needed only when loading multibyte character sets, such as UTF16.

LOAD WHEN

The LOAD WHEN *condition_spec* clause is used to identify the records that should be passed to the database. The evaluation method varies:

- If the *condition_spec* references a field in the record, the clause is evaluated only after all fields have been parsed from the record, but *before* any NULLIF or DEFAULTIF clauses have been evaluated.
- If the condition specification only references ranges (and no field names), then the clause is evaluated before the fields are parsed. This is useful for cases where the records in the file that are not to be loaded cannot be parsed into the current record definition without errors.

See condition_spec on page 12-12.

The following are some examples of using LOAD WHEN:

```
LOAD WHEN (empid != BLANKS)
LOAD WHEN ((dept_id = "SPORTING GOODS" OR dept_id = "SHOES") AND total_sales != 0)
```

BADFILE | NOBADFILE

The BADFILE clause names the file to which records are written when they cannot be loaded because of errors. For example, a record was written to the bad file because a field in the datafile could not be converted to the datatype of a column in the external table. Records that fail the LOAD WHEN clause are not written to the bad file but are written to the discard file instead. Also, any errors in using a record from an external table (such as a constraint violation when using INSERT INTO...AS SELECT... from an external table) will not cause the record to be written into the bad file.

The purpose of the bad file is to have one file where all rejected data can be examined and fixed so that it can be loaded. If you do not intend to fix the data, then you can use the NOBADFILE option to prevent creation of a bad file, even if there are bad records.

If you specify BADFILE, you must specify a filename or you will receive an error.

If neither BADFILE nor NOBADFILE is specified, the default is to create a bad file if at least one record is rejected. The name of the file will be the table name followed by _%p.

See [directory object name:] filename on page 12-12.

DISCARDFILE | NODISCARDFILE

The DISCARDFILE clause names the file to which records are written that fail the condition in the LOAD WHEN clause. The discard file is created when the first record to be discarded is encountered. If the same external table is accessed multiple times, then the discard file is rewritten each time. If there is no need to save the discarded records in a separate file, then use NODISCARDFILE.

If you specify DISCARDFILE, you must specify a filename or you will receive an error.

If neither DISCARDFILE nor NODISCARDFILE is specified, the default is to create a discard file if at least one record fails the LOAD WHEN clause. The name of the file will be the table name followed by _%p.

See [directory object name:] filename on page 12-12.

LOG FILE | NOLOGFILE

The LOGFILE clause names the file that contains messages generated by the external tables utility while it was accessing data in the datafile. If a log file already exists by the same name, the access driver reopens that log file and appends new log information to the end. This is different from bad files and discard files, which overwrite any existing file. NOLOGFILE is used to prevent creation of a log file.

If you specify LOGFILE, you must specify a filename or you will receive an error.

If neither LOGFILE nor NOLOGFILE is specified, the default is to create a log file. The name of the file will be the table name followed by _%p.

See [directory object name:] filename on page 12-12.

SKIP

Skips the specified number of records in the datafile before loading. SKIP can be specified only when nonparallel access is being made to the data.

READSIZE

The READSIZE parameter specifies the size of the read buffer. The size of the read buffer is a limit on the size of the largest record the access driver can handle. The size is specified with an integer indicating the number of bytes. The default value is 512KB (524288 bytes). You must specify a larger value if any of the records in the datafile are larger than 512KB. There is no limit on how large READSIZE can be, but practically, it is limited by the largest amount of memory that can be allocated by the access driver. Also, note that multiple buffers are allocated, so the amount of memory available for allocation is also another limit.

DATE_CACHE

By default, the date cache feature is enabled (for 1000 elements). To completely disable the date cache feature, set it to `0`.

`DATE_CACHE` specifies the date cache size (in entries). For example, `DATE_CACHE=5000` specifies that each date cache created can contain a maximum of 5000 unique date entries. Every table has its own date cache, if one is needed. A date cache is created only if at least one date or timestamp value is loaded that requires datatype conversion in order to be stored in the table.

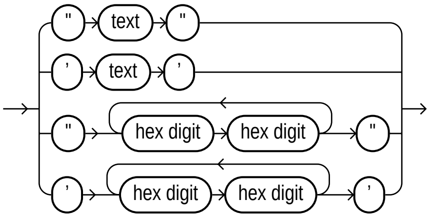
The date cache feature is only available for direct path loads. It is enabled by default. The default date cache size is 1000 elements. If the default size is used and the number of unique input values loaded exceeds 1000, then the date cache feature is automatically disabled for that table. However, if you override the default and specify a nonzero date cache size and that size is exceeded, then the cache is not disabled.

You can use the date cache statistics (entries, hits, and misses) contained in the log file to tune the size of the cache for future similar loads.

See Also: Specifying a Value for the Date Cache on page 9-22

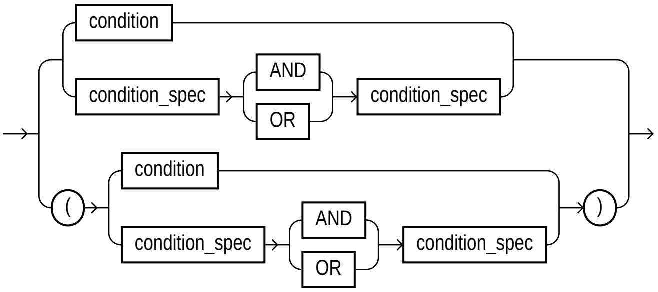
string

A `string` is a quoted series of characters or hexadecimal digits. There must be an even number of hexadecimal digits. All text will be converted to the character set of the datafile. Hexadecimal digits are converted into their binary translation, and the translation is treated as a character string. The access driver does not translate that string, but assumes it is in the character set of the datafile. The syntax for a `string` is as follows:



condition_spec

The condition_spec is an expression that evaluates to either true or false. It specifies one or more conditions that are joined by Boolean operators. The conditions and Boolean operators are evaluated from left to right. (Boolean operators are applied after the conditions are evaluated.) Parentheses can be used to override the default order of evaluation of Boolean operators. The evaluation of condition_spec clauses slows record processing, so these clauses should be used sparingly. The syntax for condition_spec is as follows:



Note that if the condition specification contains any conditions that reference field names, then the condition specifications are evaluated only after all fields have been found in the record and after blank trimming has been done. It is not useful to compare a field to BLANKS if blanks have been trimmed from the field.

The following are some examples of using condition_spec:

```
empid = BLANKS OR last_name = BLANKS
(dept_id = SPORTING GOODS OR dept_id = SHOES) AND total_sales != 0
```

See Also: condition on page 12-13

[directory object name:] filename

This clause is used to specify the name of an output file (BADFILE, DISCARDFILE, or LOGFILE). The directory object name is the name of a directory object where the user accessing the external table has privileges to write. If the directory object name is omitted, then the value specified for the DEFAULT DIRECTORY clause in the CREATE TABLE AS EXTERNAL statement is used.

The filename parameter is the name of the file to create in the directory object. The access driver does some symbol substitution to help make filenames unique in parallel loads. The symbol substitutions supported for UNIX and Windows NT are as follows (other platforms may have different symbols):

- %p is replaced by the process ID of the current process. For example, if the process ID of the access driver is 12345, then exttab_%p.log becomes exttab_12345.log.
- %a is replaced by the agent number of the current process. The agent number is the unique number assigned to each parallel process accessing the external table. This number is padded to the left with zeros to fill three characters. For example, if the third parallel agent is creating a file and bad_data_%a.bad was specified as the filename, then the agent would create a file named bad_data_003.bad.
- %% is replaced by %. If there is a need to have a percent sign in the filename, then this symbol substitution is used.

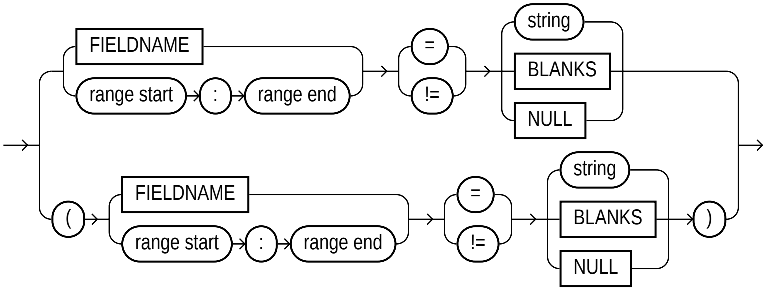
If the % character is encountered followed by anything other than one of the preceding characters, then an error is returned.

If %p or %a is not used to create unique filenames for output files and an external table is being accessed in parallel, then there may be problems with corrupted output files or with agents not being able to write to the files.

If you specify BADFILE (or DISCARDFILE or LOGFILE), you must specify a filename for it or you will receive an error. However, if you do not specify BADFILE (or DISCARDFILE or LOGFILE), then the access driver uses the name of the table followed by _%p as the name of the file. If no extension is supplied for the file, a default extension will be used. For bad files, the default extension is .bad; for discard files, the default is .dsc; and for log files, the default is .log.

condition

A condition compares a range of bytes or a field from the record against a constant string. The source of the comparison can be either a field in the record or a byte range in the record. The comparison is done on a byte-by-byte basis. If a string is specified as the target of the comparison, it will be translated into the character set of the datafile. If the field has a noncharacter datatype, no datatype conversion is performed on either the field value or the string. The syntax for a condition is as follows:



range start : range end

This clause describes a range of bytes or characters in the record to use for a condition. The value used for the `STRING SIZES ARE` clause determines whether `range` refers to bytes or characters. The `range start` and `range end` are byte or character offsets into the record. The `range start` must be less than or equal to the `range end`. Finding ranges of characters is faster for data in fixed-width character sets than it is for data in varying-width character sets. If the range refers to parts of the record that do not exist, then the record is rejected when an attempt is made to reference the range.

> **Note:** The datafile should not mix binary data (including datatypes with binary counts, such as VARCHAR) and character data that is in a varying-width character set or more than one byte wide. In these cases, the access driver may not find the correct start for the field, because it treats the binary data as character data when trying to find the start.

If a field is `NULL`, then any comparison of that field to any value other than `NULL` will return `FALSE`.

The following are some examples of using `condition`:

```
empid != BLANKS
10:13 = 0x00000830
PRODUCT_COUNT = "MISSING"
```

field_definitions Clause

The `field_definitions` clause names the fields in the datafile and specifies how to find them in records.

If the `field_definitions` clause is omitted, then:

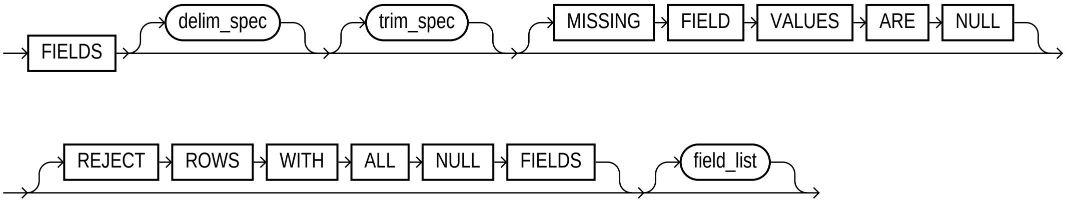
- The fields are assumed to be delimited by ','
- The fields are assumed to be character type
- The maximum length of the field is assumed to be 255
- The order of the fields in the datafile is the order in which the fields were defined in the external table
- No blanks are trimmed from the field

The following is an example of an external table created without any access parameters. It is followed by a sample of the datafile that can be used to load it.

```
CREATE TABLE emp_load (first_name CHAR(15), last_name CHAR(20), year_of_birth CHAR(4))
  ORGANIZATION EXTERNAL (TYPE ORACLE_LOADER DEFAULT DIRECTORY ext_tab_dir LOCATION ('foo.dat'));

Alvin,Tolliver,1976
Kenneth,Baer,1963
```

The syntax for the `field_definitions` clause is as follows:



delim_spec Clause

The `delim_spec` clause is used to identify how all fields are terminated in the record. The `delim_spec` specified for all fields can be overridden for a particular field as part of the `field_list` clause. For a full description of the syntax, see delim_spec on page 12-16.

trim_spec Clause

The trim_spec clause specifies the type of whitespace trimming to be performed by default on all character fields. The trim_spec clause specified for all fields can be overridden for individual fields by specifying a trim_spec clause for those fields. For a full description of the syntax, see trim_spec on page 12-19.

MISSING FIELD VALUES ARE NULL

MISSING FIELD VALUES ARE NULL indicates that if there is not enough data in a record for all fields, then those fields with missing data values are set to NULL. For a full description of the syntax, see MISSING FIELD VALUES ARE NULL on page 12-20.

REJECT ROWS WITH ALL NULL FIELDS

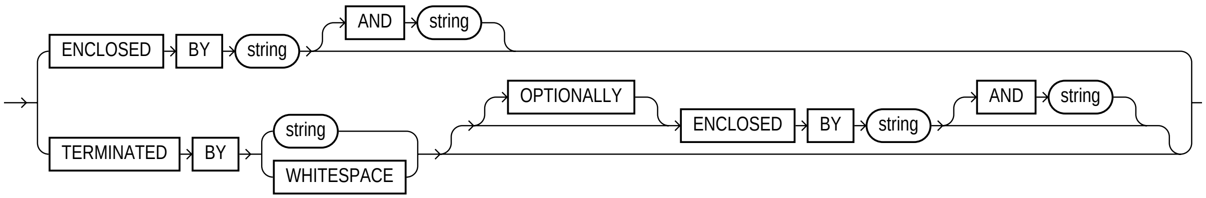
REJECT ROWS WITH ALL NULL FIELDS indicates that a row will not be loaded into the external table if all referenced fields in the row are null. If this parameter is not specified, the default value is to accept rows with all null fields. The setting of this parameter is written to the log file either as "reject rows with all null fields" or as "rows with all null fields are accepted."

field_list Clause

The field_list clause identifies the fields in the datafile and their datatypes. For a full description of the syntax, see field_list on page 12-21.

delim_spec

The delim_spec clause is used to find the end (and if ENCLOSED BY is specified, the start) of a field. Its syntax is as follows:



If ENCLOSED BY is specified, the access driver starts at the current position in the record and skips over all whitespace looking for the first delimiter. All whitespace between the current position and the first delimiter is ignored. Next, the access driver looks for the second enclosure delimiter (or looks for the first one again if a second one is not specified). Everything between those two delimiters is considered part of the field.

If TERMINATED BY *string* is specified with the ENCLOSED BY clause, then the terminator string must immediately follow the second enclosure delimiter. Any whitespace between the second enclosure delimiter and the terminating delimiter is skipped. If anything other than whitespace is found between the two delimiters, then the row is rejected for being incorrectly formatted.

If TERMINATED BY is specified without the ENCLOSED BY clause, then everything between the current position in the record and the next occurrence of the termination string is considered part of the field.

If OPTIONALLY is specified, then TERMINATED BY must also be specified. The OPTIONALLY parameter means the ENCLOSED BY delimiters can either both be present or both be absent. The terminating delimiter must be present regardless of whether the ENCLOSED BY delimiters are present. If OPTIONALLY is specified, then the access driver skips over all whitespace, looking for the first nonblank character. Once the first nonblank character is found, the access driver checks to see if the current position contains the first enclosure delimiter. If it does, then the access driver finds the second enclosure string and everything between the first and second enclosure delimiters is considered part of the field. The terminating delimiter must immediately follow the second enclosure delimiter (with optional whitespace allowed between the second enclosure delimiter and the terminating delimiter). If the first enclosure string is not found at the first nonblank character, then the access driver looks for the terminating delimiter. In this case, all characters from the beginning (including the leading blanks) to the terminating delimiter are considered part of the field.

After the delimiters have been found, the current position in the record is set to after the last delimiter for the field. If TERMINATED BY WHITESPACE was specified, then the current position in the record is set to after all whitespace following the field.

A missing terminator for the last field in the record is not an error. The access driver proceeds as if the terminator was found. It is an error if the second enclosure delimiter is missing.

The string used for the second enclosure can be included in the data field by including the second enclosure twice. For example, if a field is enclosed by single quotation marks, a data field could contain a single quotation mark by doing something like the following:

```
'I don''t like green eggs and ham'
```

There is no way to quote a terminator string in the field data without using enclosing delimiters. Because the field parser does not look for the terminating delimiter until after it has found the enclosing delimiters, the field can contain the terminating delimiter.

In general, specifying single characters for the strings is faster than multiple characters. Also, searching data in fixed-width character sets is usually faster than searching data in varying-width character sets.

The following are some examples of using `delim_spec`:

```
TERMINATED BY "|"
ENCLOSED BY "\" TERMINATED BY ","
ENCLOSED BY "START MESSAGE" AND "END MESSAGE"
```

Example: External Table with Terminating Delimiters

The following is an example of an external table that uses terminating delimiters. It is followed by a sample of the datafile that can be used to load it.

```
CREATE TABLE emp_load (first_name CHAR(15), last_name CHAR(20), year_of_birth CHAR(4))
  ORGANIZATION EXTERNAL (TYPE ORACLE_LOADER DEFAULT DIRECTORY ext_tab_dir
                         ACCESS PARAMETERS (FIELDS TERMINATED BY WHITESPACE)
                         LOCATION ('foo.dat'));

Alvin Tolliver 1976
Kenneth Baer 1963
Mary Dube 1973
```

Example: External Table with Enclosure and Terminator Delimiters

The following is an example of an external table that uses both enclosure and terminator delimiters. Remember that all whitespace between a terminating string and the first enclosure string is ignored, as is all whitespace between a second enclosing delimiter and the terminator. The example is followed by a sample of the datafile that can be used to load it.

```
CREATE TABLE emp_load (first_name CHAR(15), last_name CHAR(20), year_of_birth CHAR(4))
  ORGANIZATION EXTERNAL (TYPE ORACLE_LOADER DEFAULT DIRECTORY ext_tab_dir
                        ACCESS PARAMETERS (FIELDS TERMINATED BY "," ENCLOSED BY "("  AND ")")
                        LOCATION ('foo.dat'));

(Alvin) ,   (Tolliver),(1976)
(Kenneth),  (Baer) ,(1963)
(Mary),(Dube) ,   (1973)
```

Example: External Table with Optional Enclosure Delimiters

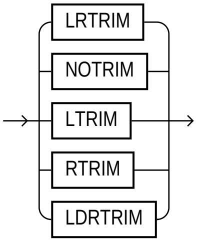
The following is an example of an external table that uses optional enclosure delimiters. Note that LRTRIM is used to trim leading and trailing blanks from fields. The example is followed by a sample of the datafile that can be used to load it.

```
CREATE TABLE emp_load (first_name CHAR(15), last_name CHAR(20), year_of_birth CHAR(4))
  ORGANIZATION EXTERNAL (TYPE ORACLE_LOADER DEFAULT DIRECTORY ext_tab_dir
                         ACCESS PARAMETERS (FIELDS TERMINATED BY ','
                                            OPTIONALLY ENCLOSED BY '(' and ')'
                                            LRTRIM)
                         LOCATION ('foo.dat'));

Alvin ,   Tolliver , 1976
(Kenneth),  (Baer), (1963)
( Mary ), Dube ,    (1973)
```

trim_spec

The trim_spec clause is used to specify that spaces should be trimmed from the beginning of a text field, the end of a text field, or both. Spaces include blanks and other nonprinting characters such as tabs, line feeds, and carriage returns. The syntax for the trim_spec clause is as follows:



NOTRIM indicates that no characters will be trimmed from the field.

LRTRIM, LTRIM, and RTRIM are used to indicate that characters should be trimmed from the field. LRTRIM means that both leading and trailing spaces are trimmed. LTRIM means that leading spaces will be trimmed. RTRIM means trailing spaces are trimmed.

LDRTRIM is used to provide compatibility with SQL*Loader trim features. It is the same as NOTRIM except in the following cases:

- If the field is not a delimited field, then spaces will be trimmed from the right.

- If the field is a delimited field with OPTIONALLY ENCLOSED BY specified, and the optional enclosures are missing for a particular instance, then spaces will be trimmed from the left.

The default is LDRTRIM. Specifying NOTRIM yields the fastest performance.

The trim_spec clause can be specified before the field list to set the default trimming for all fields. If trim_spec is omitted before the field list, then LDRTRIM is the default trim setting. The default trimming can be overridden for an individual field as part of the datatype_spec.

If trimming is specified for a field that is all spaces, then the field will be set to NULL.

In the following example, all data is fixed-length; however, the character data will not be loaded with leading spaces. The example is followed by a sample of the datafile that can be used to load it.

```
CREATE TABLE emp_load (first_name CHAR(15), last_name CHAR(20),
year_of_birth CHAR(4))
  ORGANIZATION EXTERNAL (TYPE ORACLE_LOADER DEFAULT DIRECTORY ext_tab_dir
                         ACCESS PARAMETERS (FIELDS LTRIM)
                         LOCATION ('foo.dat'));

Alvin,           Tolliver,1976
Kenneth,         Baer,    1963
Mary,            Dube,    1973
```

MISSING FIELD VALUES ARE NULL

MISSING FIELD VALUES ARE NULL indicates that if there is not enough data in a record for all fields, then those fields with missing data values are set to NULL. If MISSING FIELD VALUES ARE NULL is not specified, and there is not enough data in the record for all fields, then the row is rejected.

In the following example, the second record is stored with a NULL set for the year_of_birth column, even though the data for the year of birth is missing from the datafile. If the MISSING FIELD VALUES ARE NULL clause was omitted from the access parameters, then the second row would be rejected because it did not have a value for the year_of_birth column. The example is followed by a sample of the datafile that can be used to load it.

```
CREATE TABLE emp_load (first_name CHAR(15), last_name CHAR(20), year_of_birth INT)
  ORGANIZATION EXTERNAL (TYPE ORACLE_LOADER DEFAULT DIRECTORY ext_tab_dir
                         ACCESS PARAMETERS (FIELDS TERMINATED BY ","
                                            MISSING FIELD VALUES ARE NULL)
                         LOCATION ('foo.dat'));

Alvin,Tolliver,1976
Baer,Kenneth
Mary,Dube,1973
```

field_list

The field_list clause identifies the fields in the datafile and their datatypes. Evaluation criteria for the field_list clause are as follows:

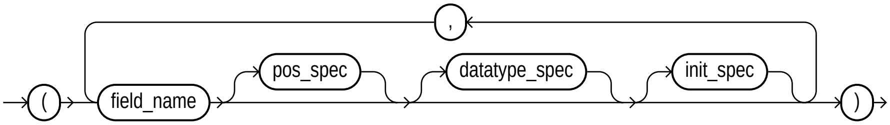
- If no datatype is specified for a field, it is assumed to be CHAR(1) for a nondelimited field, and CHAR(255)for a delimited field.
- If no field list is specified, then the fields in the datafile are assumed to be in the same order as the fields in the external table. The datatype for all fields is CHAR(255).
- If no field list is specified and no delim_spec clause is specified, then the fields in the datafile are assumed to be in the same order as fields in the external table. All fields are assumed to be CHAR(255) and terminated by a comma.

This example shows the definition for an external table with no field_list and a delim_spec. It is followed by a sample of the datafile that can be used to load it.

```
CREATE TABLE emp_load (first_name CHAR(15), last_name CHAR(20), year_of_birth INT)
  ORGANIZATION EXTERNAL (TYPE ORACLE_LOADER DEFAULT DIRECTORY ext_tab_dir
                         ACCESS PARAMETERS (FIELDS TERMINATED BY "|")
                         LOCATION ('foo.dat'));

Alvin|Tolliver|1976
Kenneth|Baer|1963
Mary|Dube|1973
```

The syntax for the field_list clause is as follows:



field_name

The field_name is a string identifying the name of a field in the datafile. If the string is not within quotation marks, the name is uppercased when matching field names with column names in the external table.

If field_name matches the name of a column in the external table that is referenced in the query, then the field value is used for the value of that external table column. If the name does not match any referenced name in the external table, then the field is not loaded but can be used for clause evaluation (for example WHEN or NULLIF).

pos_spec

The pos_spec clause indicates the position of the column within the record. For a full description of the syntax, see pos_spec Clause on page 12-22.

datatype_spec

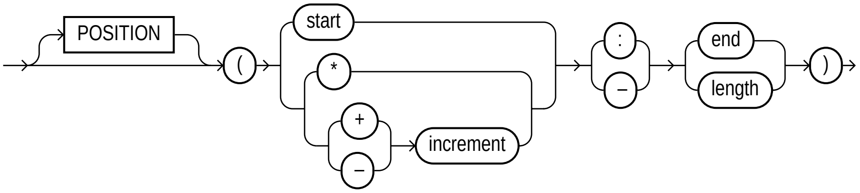
The datatype_spec clause indicates the datatype of the field. If datatype_spec is omitted, the access driver assumes the datatype is CHAR(255). For a full description of the syntax, see datatype_spec Clause on page 12-24.

init_spec

The init_spec clause indicates when a field is NULL or has a default value. For a full description of the syntax, see init_spec Clause on page 12-31.

pos_spec Clause

The pos_spec clause indicates the position of the column within the record. The setting of the STRING SIZES ARE IN clause determines whether pos_spec refers to byte positions or character positions. Using character positions with varying-width character sets takes significantly longer than using character positions with fixed-width character sets. Binary and multibyte character data should not be present in the same datafile when pos_spec is used for character positions. If they are, then the results are unpredictable. The syntax for the pos_spec clause is as follows:



start

The `start` parameter is the number of bytes or characters from the beginning of the record to where the field begins. It positions the start of the field at an absolute spot in the record rather than relative to the position of the previous field.

*

The `*` parameter indicates that the field begins at the first byte or character after the end of the previous field. This is useful if you have a varying-length field followed by a fixed-length field. This option cannot be used for the first field in the record.

increment

The `increment` parameter positions the start of the field at a fixed number of bytes or characters from the end of the previous field. Use `*-increment` to indicate that the start of the field starts before the current position in the record (this is a costly operation for multibyte character sets). Use `*+increment` to move the start after the current position.

end

The `end` parameter indicates the absolute byte or character offset into the record for the last byte of the field. If `start` is specified along with `end`, then `end` cannot be less than `start`. If `*` or `increment` is specified along with `end`, and the start evaluates to an offset larger than the `end` for a particular record, then that record will be rejected.

length

The `length` parameter indicates that the end of the field is a fixed number of bytes or characters from the start. It is useful for fixed-length fields when the start is specified with `*`.

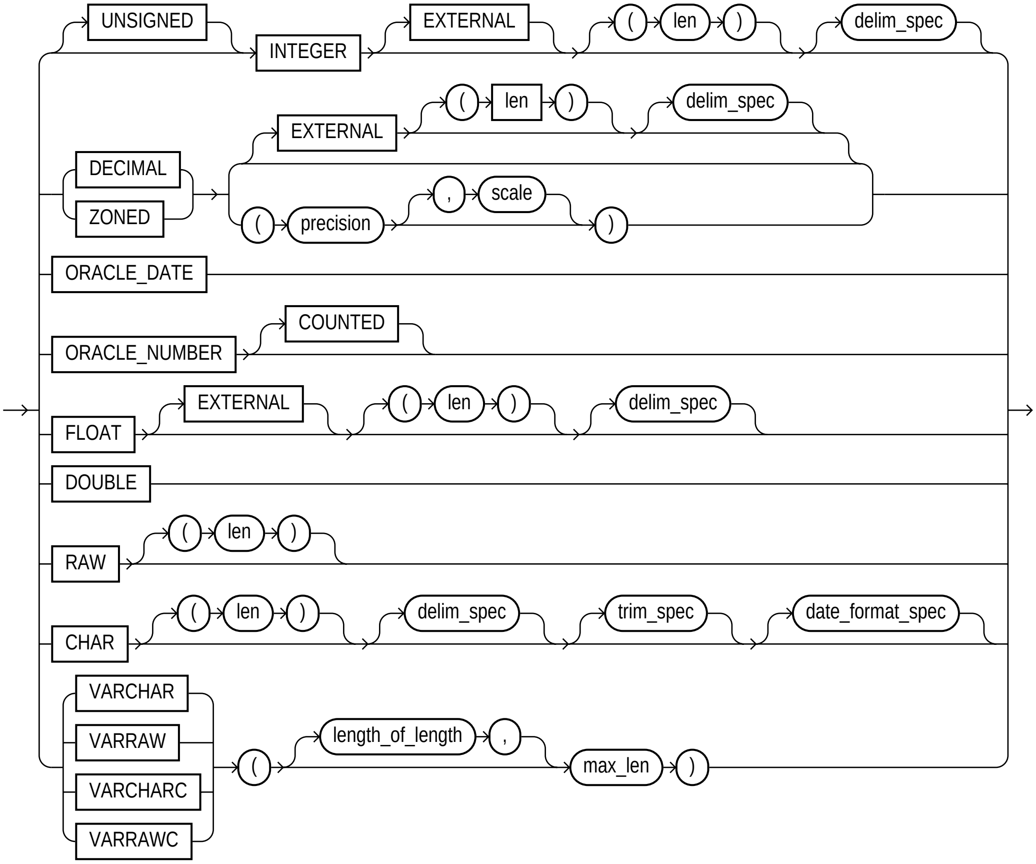
The following example shows various ways of using `pos_spec`. It is followed by a sample of the datafile that can be used to load it.

```
CREATE TABLE emp_load (first_name CHAR(15),
                      last_name CHAR(20),
                      year_of_birth INT,
                      phone CHAR(12),
                      area_code CHAR(3),
                      exchange CHAR(3),
                      extension CHAR(4))
  ORGANIZATION EXTERNAL
  (TYPE ORACLE_LOADER
   DEFAULT DIRECTORY ext_tab_dir
   ACCESS PARAMETERS
     (FIELDS RTRIM
            (first_name (1:15) CHAR(15),
             last_name (*:+20),
             year_of_birth (36:39),
             phone (40:52),
             area_code (*-12: +3),
             exchange (*+1: +3),
             extension (*+1: +4)))
   LOCATION ('foo.dat'));

Alvin          Tolliver            1976415-922-1982
Kenneth        Baer                1963212-341-7912
Mary           Dube                1973309-672-2341
```

datatype_spec Clause

The `datatype_spec` clause is used to describe the datatype of a field in the datafile if the datatype is different than the default. The datatype of the field can be different than the datatype of a corresponding column in the external table. The access driver handles the necessary conversions. The syntax for the `datatype_spec` clause is as follows:



If the number of bytes or characters in any field is 0, then the field is assumed to be NULL. The optional DEFAULTIF clause specifies when the field is set to its default value. Also, the optional NULLIF clause specifies other conditions for when the column associated with the field is set to NULL. If the DEFAULTIF or NULLIF clause is true, then the actions of those clauses override whatever values are read from the datafile.

See Also: init_spec Clause on page 12-31 for more information about NULLIF and DEFAULTIF

[UNSIGNED] INTEGER [EXTERNAL] [(len)]

This clause defines a field as an integer. If EXTERNAL is specified, the number is a character string. If EXTERNAL is not specified, the number is a binary field. The valid values for *len* in binary integer fields are 1, 2, 4, and 8. If *len* is omitted for binary integers, the default value is whatever the value of *sizeof(int)* is on the platform where the access driver is running. Use of the DATA IS {BIG | LITTLE} ENDIAN clause may cause the data to be byte-swapped before it is stored.

If EXTERNAL is specified, then the value of *len* is the number of bytes or characters in the number (depending on the setting of the STRING SIZES ARE IN BYTES or CHARACTERS clause). If no length is specified, the default value is 255.

DECIMAL [EXTERNAL] and ZONED [EXTERNAL]

The DECIMAL clause is used to indicate that the field is a packed decimal number. The ZONED clause is used to indicate that the field is a zoned decimal number. The *precision* field indicates the number of digits in the number. The *scale* field is used to specify the location of the decimal point in the number. It is the number of digits to the right of the decimal point. If *scale* is omitted, a value of 0 is assumed.

Note that there are different encoding formats of zoned decimal numbers depending on whether the character set being used is EBCDIC-based or ASCII-based. If the language of the source data is EBCDIC, then the zoned decimal numbers in that file must match the EBCDIC encoding. If the language is ASCII-based, then the numbers must match the ASCII encoding.

If the EXTERNAL parameter is specified, then the data field is a character string whose length matches the precision of the field.

ORACLE_DATE

ORACLE_DATE is a field containing a date in the Oracle binary date format. This is the format used by the DTYDAT datatype in OCI programs. The field is a fixed length of 7.

ORACLE_NUMBER

ORACLE_NUMBER is a field containing a number in the Oracle number format. The field is a fixed length (the maximum size of an Oracle number field) unless COUNTED is specified, in which case the first byte of the field contains the number of bytes in the rest of the field.

ORACLE_NUMBER is a fixed-length 22-byte field. The length of an ORACLE_NUMBER COUNTED field is one for the count byte, plus the number of bytes specified in the count byte.

DOUBLE [EXTERNAL]

The DOUBLE clause indicates that the field is the same format as the C language DOUBLE datatype on the platform where the access driver is executing. Use of the DATA IS {BIG | LITTLE} ENDIAN clause may cause the data to be byte-swapped before it is stored. This datatype may not be portable between certain platforms.

If the EXTERNAL parameter is specified, then the field is a character string whose maximum length is 255.

FLOAT [EXTERNAL]

The FLOAT clause indicates that the field is the same format as the C language FLOAT datatype on the platform where the access driver is executing. Use of the DATA IS {BIG | LITTLE} ENDIAN clause may cause the data to be byte-swapped before it is stored. This datatype may not be portable between certain platforms.

If the EXTERNAL parameter is specified, then the field is a character string whose maximum length is 255.

RAW

The RAW clause is used to indicate that the source data is binary data. The *len* for RAW fields is always in number of bytes. When a RAW field is loaded in a character column, the data that is written into the column is the hexadecimal representation of the bytes in the RAW field.

CHAR

The CHAR clause is used to indicate that a field is a character datatype. The length (*len)* for CHAR fields specifies the largest number of bytes or characters in the field. The *len* is in bytes or characters, depending on the setting of the STRING SIZES ARE IN clause.

If no length is specified for a field of datatype CHAR, then the size of the field is assumed to be 1, unless the field is delimited:

- For a delimited CHAR field, if a length is specified, that length is used as a maximum.

- For a delimited CHAR field for which no length is specified, the default is 255 bytes.
- For a delimited CHAR field that is greater than 255 bytes, you must specify a maximum length. Otherwise you will receive an error stating that the field in the datafile exceeds maximum length.

The date_format_spec clause is used to indicate that the field contains a date or time in the specified format.

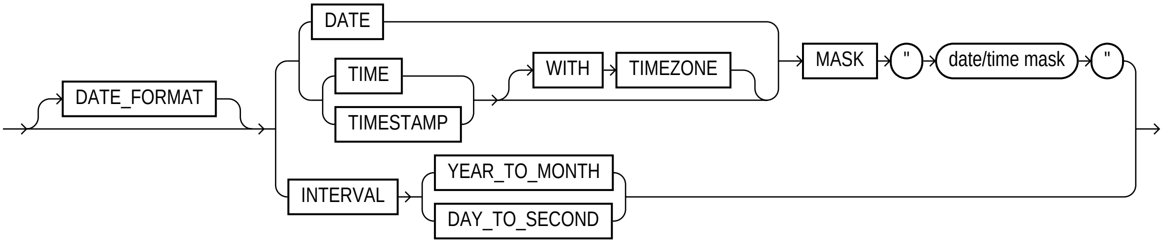
The following example shows the use of the CHAR clause. It is followed by a sample of the datafile that can be used to load it.

```
CREATE TABLE emp_load
 (first_name CHAR(15),
  last_name CHAR(20),
  hire_date CHAR(10),
  resume_file CHAR(500))
  ORGANIZATION EXTERNAL
   (TYPE ORACLE_LOADER
    DEFAULT DIRECTORY ext_tab_dir
    ACCESS PARAMETERS (FIELDS TERMINATED BY ","
                        (first_name,
                         last_name,
                         hire_date CHAR(10) DATE_FORMAT DATE MASK "mm/dd/yyyy",
                         resume_file))
    LOCATION ('foo.dat'));

Alvin,Tolliver,12/2/1995,tolliver_resume.ps
Kenneth,Baer,6/6/1997,KB_resume.ps
Mary,Dube,1/18/2000,dube_resume.ps
```

date_format_spec

The date_format_spec clause is used to indicate that a character string field contains date data, time data, or both, in a specific format. This information is used only when a character field is converted to a date or time datatype and only when a character string field is mapped into a date column. The syntax for the date_format_spec clause is as follows:



DATE The DATE clause indicates that the string contains a date.

MASK The MASK clause is used to override the default globalization format mask for the datatype. If a date mask is not specified, then the NLS session's setting (*not* the client settings) for the appropriate globalization parameter for the datatype is used.

- NLS_DATE_FORMAT for DATE datatypes
- NLS_TIME_FORMAT for TIME datatypes
- NLS_TIMESTAMP_FORMAT for TIMESTAMP datatypes
- NLS_TIME_WITH_TIMEZONE_FORMAT for TIME WITH TIME ZONE datatypes
- NLS_TIMESTAMP_WITH_TIMEZONE_FORMAT for TIMESTAMP WITH TIME ZONE datatypes

TIME The TIME clause indicates that a field contains a formatted time string.

TIMESTAMP The TIMESTAMP clause indicates that a field contains a formatted timestamp.

INTERVAL The INTERVAL clause indicates that a field contains a formatted interval. The type of interval can be either YEAR TO MONTH or DAY TO SECOND.

VARCHAR and VARRAW

The VARCHAR datatype has a binary count field followed by character data. The value in the binary count field is either the number of bytes in the field or the number of characters. See STRING SIZES ARE IN on page 12-8 for information on how to specify whether the count is interpreted as a count of characters or count of bytes.

The VARRAW datatype has a binary count field followed by binary data. The value in the binary count field is the number of bytes of binary data. The data in the VARRAW field is not affected by the DATA IS...ENDIAN clause.

The optional *length_of_length* field in the specification is the number of bytes in the count field. Valid values for *length_of_length* for VARCHAR are 1, 2, 4, and 8. If *length_of_length* is not specified, a value of 2 is used. The count field has the same endianness as specified by the DATA IS...ENDIAN clause.

The *max_len* field is used to indicate the largest size of any instance of the field in the datafile. For VARRAW fields, *max_len* is number of bytes. For VARCHAR fields, *max_len* is either number of characters or number of bytes depending on the STRING SIZES ARE IN clause.

The following example shows various uses of VARCHAR and VARRAW.The binary values for the count bytes and value for raw data are shown in the datafile in italics, with 2 characters per binary byte.

```
CREATE TABLE emp_load
             (first_name CHAR(15),
              last_name CHAR(20),
              resume CHAR(2000),
              picture RAW(2000))
  ORGANIZATION EXTERNAL
  (TYPE ORACLE_LOADER
   DEFAULT DIRECTORY ext_tab_dir
   ACCESS PARAMETERS
     (FIELDS (first_name VARCHAR(2,12),
              last_name VARCHAR(2,20),
              resume VARCHAR(4,10000),
              picture VARRAW(4,100000)))
    LOCATION ('foo.dat'));
```

```
0005Alvin0008Tolliver0000001DAlvin Tolliver’s Resume etc. 0000001013f4690a30bc29d7e40023ab4599ffff
```

VARCHARC and VARRAWC

The VARCHARC datatype has a character count field followed by character data. The value in the count field is either the number of bytes in the field or the number of characters. See STRING SIZES ARE IN on page 12-8 for information on how to specify whether the count is interpreted as a count of characters or count of bytes. The optional *length_of_length* is either the number of bytes or the number of characters in the count field for VARCHARC, depending on whether lengths are being interpreted as characters or bytes.

The maximum value for *length_of_lengths* for VARCHARC is 10 if string sizes are in characters, and 20 if string sizes are in bytes. The default value for *length_of_length* is 5.

The VARRAWC datatype has a character count field followed by binary data. The value in the count field is the number of bytes of binary data. The *length_of_length* is the number of bytes in the count field.

The *max_len* field is used to indicate the largest size of any instance of the field in the datafile. For VARRAWC fields, *max_len* is number of bytes. For VARCHARC fields, *max_len* is either number of characters or number of bytes depending on the STRING SIZES ARE IN clause.

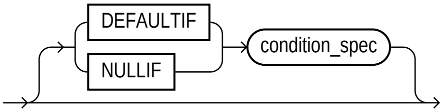
The following example shows various uses of VARCHARC and VARRAWC. The length of the picture field is 0, which means the field is set to NULL.

```
CREATE TABLE emp_load
             (first_name CHAR(15),
              last_name CHAR(20),
              resume CHAR(2000),
              picture RAW (2000))
  ORGANIZATION EXTERNAL
  (TYPE ORACLE_LOADER
    DEFAULT DIRECTORY ext_tab_dir
    ACCESS PARAMETERS
      (FIELDS (first_name VARCHARC(5,12),
               last_name VARCHARC(2,20),
               resume VARCHARC(4,10000),
               picture VARRAWC(4,100000)))
  LOCATION ('foo.dat'));

00007William05Ricca0035Resume for William Ricca is missing0000
```

init_spec Clause

The init_spec clause is used to specify when a field should be set to NULL or when it should be set to a default value. The syntax for the init_spec clause is as follows:



Only one `NULLIF` clause and only one `DEFAULTIF` clause can be specified for any field. These clauses behave as follows:

- If `NULLIF` *`condition_spec`* is specified and it evaluates to `true`, the field is set to `NULL`.
- If `DEFAULTIF` *`condition_spec`* is specified and it evaluates to `true`, the value of the field is set to a default value. The default value depends on the datatype of the field, as follows:
  - For a character datatype, the default value is an empty string.
  - For a numeric datatype, the default value is a 0.
  - For a date datatype, the default value is `NULL`.
- If a `NULLIF` clause and a `DEFAULTIF` clause are both specified for a field, then the `NULLIF` clause is evaluated first and the `DEFAULTIF` clause is evaluated only if the `NULLIF` clause evaluates to `false`.

Part IV

Other Utilities

This section contains the following chapters:

13

DBVERIFY: Offline Database Verification Utility

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 ensure 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.

DBVERIFY checks are limited to cache-managed blocks (that is, data blocks). Because DBVERIFY is only for use with datafiles, it will not work against control files or redo logs.

There are two command-line interfaces to DBVERIFY. With the first interface, you specify disk blocks of a single datafile for checking. With the second interface, you specify a segment for checking. The following sections provide descriptions of these interfaces:

- Using DBVERIFY to Validate Disk Blocks of a Single Datafile
- Using DBVERIFY to Validate a Segment

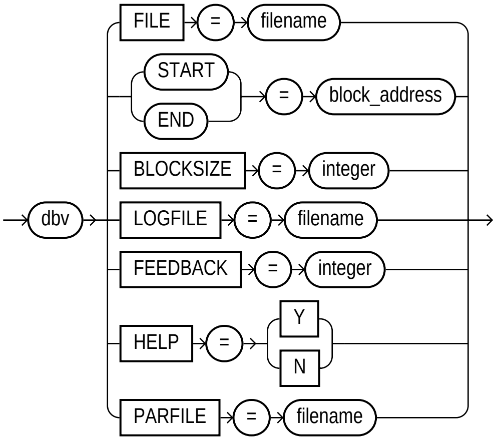
Note: The command used to invoke DBVERIFY is dependent on your operating system (for example, on Sun/Sequent systems, the command is `dbv`). See your Oracle operating system-specific documentation.

Using DBVERIFY to Validate Disk Blocks of a Single Datafile

In this mode, DBVERIFY scans one or more disk blocks of a single datafile and performs page checks.

Syntax

The syntax for `DBVERIFY` when you want to validate disk blocks of a single datafile is as follows:



Parameters

Descriptions of the parameters are as follows:

Parameter	Description
`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.

Parameter	Description
`BLOCKSIZE`	`BLOCKSIZE` is required only if the file to be verified does not have a block size of 2 KB. If the file does not have block size of 2 KB and you do not specify `BLOCKSIZE`, you will receive the error DBV-00103.
`LOGFILE`	Specifies the file to which logging information should be written. The default sends output to the terminal display.
`FEEDBACK`	Causes DBVERIFY to send a progress display to the terminal in the form of a single period (.) for *n* number of pages verified during the DBVERIFY run. If *n* = 0, there is no progress display.
`HELP`	Provides online help.
`PARFILE`	Specifies the name of the parameter file to use. You can store various values for DBVERIFY parameters in flat files. This allows you to customize parameter files to handle different types of datafiles and to perform specific types of integrity checks on datafiles.

Command-Line Interface

The following example shows a sample use of the command-line interface to this mode of DBVERIFY.

```
% dbv FILE=t_db1.dbf FEEDBACK=100
```

Sample DBVERIFY Output

The following example is sample output of verification for the file `t_db1.dbf.` The feedback parameter has been given the value 100 to display one period (.) for every 100 pages processed:

```
% dbv FILE=t_db1.dbf FEEDBACK=100

DBVERIFY: Release 9.2.0.1.0 - Production on Wed Feb 27 13:55:26 2002

(c) Copyright 2002 Oracle Corporation.  All rights reserved.

DBVERIFY - Verification starting : FILE = t_db1.dbf

................................................................................


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
```

Notes:

- Pages = Blocks
- Total Pages Examined = number of blocks in the file
- Total Pages Processed = number of blocks that were verified (formatted blocks)
- Total Pages Failing (Data) = number of blocks that failed the data block checking routine
- Total Pages Failing (Index) = number of blocks that failed the index block checking routine
- Total Pages Marked Corrupt = number of blocks for which the cache header is invalid, thereby making it impossible for DBVERIFY to identify the block type
- Total Pages Influx = number of blocks that are being read and written to at the same time. If the database is open when DBVERIFY is run, DBVERIFY reads blocks multiple times to get a consistent image. But because the database is open, there may be blocks that are being read and written to at the same time (INFLUX). DBVERIFY cannot get a consistent image of pages that are in flux.

Using DBVERIFY to Validate a Segment

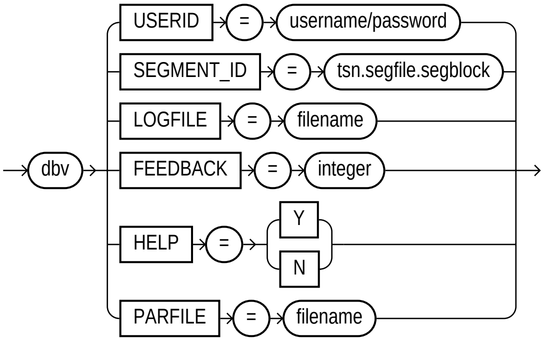
In this mode, DBVERIFY allows you to specify a table segment or index segment for verification. It checks to make sure that a row chain pointer is within the segment being verified.

This mode requires that you specify a segment (data or index) to be validated. It also requires that you log on to the database with SYSDBA privileges, because information about the segment must be retrieved from the database.

During this mode, the segment is locked. If the specified segment is an index, the parent table is locked. Note that some indexes, such as IOTs, do not have parent tables.

Syntax

The syntax for DBVERIFY when you want to validate a segment is as follows:



Parameters

Descriptions of the parameters are as follows:

Parameter	Description
USERID	Specifies your username and password.
SEGMENT_ID	Specifies the segment that you want to verify. You can identify the tsn, segfile, and segblock by joining and querying the appropriate data dictionary tables, for example, USER_TABLES and USER_SEGMENTS.
LOGFILE	Specifies the file to which logging information should be written. The default sends output to the terminal display.
FEEDBACK	Causes DBVERIFY to send a progress display to the terminal in the form of a single period (.) for *n* number of pages verified during the DBVERIFY run. If *n* = 0, there is no progress display.
HELP	Provides online help.

Parameter	Description
PARFILE	Specifies the name of the parameter file to use. You can store various values for DBVERIFY parameters in flat files. This allows you to customize parameter files to handle different types of datafiles and to perform specific types of integrity checks on datafiles.

Command-Line Interface

The following example shows a sample use of the command-line interface to this mode of DBVERIFY.

```
dbv USERID=username/password SEGMENT_ID=tsn.segfile.segblock
```

14

DBNEWID Utility

DBNEWID is a database utility that can change the internal database identifier (DBID) and the database name (DBNAME) for an operational database.

This chapter contains the following sections:

- What Is the DBNEWID Utility?
- Ramifications of Changing the DBID and DBNAME
- Changing the DBID and DBNAME of a Database
- DBNEWID Syntax

What Is the DBNEWID Utility?

Prior to the introduction of the DBNEWID utility, you could manually create a copy of a database and give it a new database name (DBNAME) by re-creating the control file. However, you could not give the database a new identifier (DBID). The DBID is an internal, unique identifier for a database. Because Recovery Manager (RMAN) distinguishes databases by DBID, you could not register a seed database and a manually copied database together in the same RMAN repository. The DBNEWID utility solves this problem by allowing you to change any of the following:

- Only the DBID of a database
- Only the DBNAME of a database
- Both the DBNAME and DBID of a database

Ramifications of Changing the DBID and DBNAME

Changing the DBID of a database is a serious procedure. When the DBID of a database is changed, all previous backups and archived logs of the database become unusable. After you change the DBID, you must open the database with the RESETLOGS option, which re-creates the online redo logs and resets their sequence to 1 (see the *Oracle9i Database Administrator's Guide)*. Consequently, you should make a backup of the whole database immediately after changing the DBID.

Changing the DBNAME without changing the DBID does not require you to open with the RESETLOGS option, so database backups and archived logs are not invalidated. However, changing the DBNAME does have consequences. You must change the DB_NAME initialization parameter after a database name change to reflect the new name. Also, you may have to re-create the Oracle password file. If you restore an old backup of the control file (before the name change), then you should use the initialization parameter file and password file from before the database name change.

Changing the DBID and DBNAME of a Database

This section contains these topics:

- Changing the DBID and Database Name
- Changing Only the Database Name
- Troubleshooting a DBID Change Operation
- Troubleshooting a Database Name Change Operation

Changing the DBID and Database Name

The following steps describe how to change the DBID of a database. Optionally, you can change the database name as well.

1. Ensure that you have a recoverable whole database backup.
2. Ensure that the target database is mounted but not open, and that it was shut down consistently prior to mounting. For example:

   ```
   SHUTDOWN IMMEDIATE
   STARTUP MOUNT
   ```

3. Invoke the DBNEWID utility on the command line, specifying a valid user with the SYSDBA privilege. For example:

```
% nid TARGET=SYS/oracle@test_db
```

To change the database name in addition to the DBID, specify the `DBNAME` parameter. This example changes the name to `test_db2`:

```
% nid TARGET=SYS/oracle@test DBNAME=test_db2
```

The DBNEWID utility performs validations in the headers of the datafiles and control files before attempting I/O to the files. If validation is successful, then DBNEWID prompts you to confirm the operation (unless you specify a log file, in which case it does not prompt), changes the DBID for each datafile (including offline normal and read-only datafiles), and then exits. The database is left mounted but is not yet usable. For example:

```
DBNEWID: Release 9.2.0.1.0

(c) Copyright 2002 Oracle Corporation.  All rights reserved.

Connected to database TEST_DB (DBID=3942195360)

Control Files in database:
    /oracle/dbs/cf1.f
    /oracle/dbs/cf2.f

Change database id of database SOLARIS? (Y/[N]) => y

Proceeding with operation
    Datafile /oracle/dbs/tbs_01.f - changed
    Datafile /oracle/dbs/tbs_02.f - changed
    Datafile /oracle/dbs/tbs_11.f - changed
    Datafile /oracle/dbs/tbs_12.f - changed
    Datafile /oracle/dbs/tbs_21.f - changed

New DBID for database TEST_DB is 3942196782.
All previous backups and archived redo logs for this database are unusable
Proceed to shutdown database and open with RESETLOGS option.
DBNEWID - Database changed.
```

If validation is not successful, then DBNEWID terminates and leaves the target database intact. You can open the database, fix the error, and then either resume the DBNEWID operation or continue using the database without changing its DBID.

4. After DBNEWID successfully changes the DBID, shut down the database:

```
SHUTDOWN IMMEDIATE
```

5. Mount the database. For example:

   ```
   STARTUP MOUNT
   ```

6. Open the database in RESETLOGS mode and resume normal use. For example:

   ```
   ALTER DATABASE OPEN RESETLOGS;
   ```

   Make a new database backup. Because you reset the online redo logs, the old backups and archived logs are no longer usable in the current incarnation of the database.

Changing Only the Database Name

The following steps describe how to change the database name without changing the DBID.

1. Ensure that you have a recoverable whole database backup.
2. Ensure that the target database is mounted but not open, and that it was shut down consistently prior to mounting. For example:

   ```
   SHUTDOWN IMMEDIATE
   STARTUP MOUNT
   ```

3. Invoke the utility on the command line, specifying a valid user with the SYSDBA privilege. You must specify both the DBNAME and SETNAME parameters. This example changes the name to `test_db2`:

   ```
   % nid TARGET=SYS/oracle@test_db DBNAME=test_db2 SETNAME=YES
   ```

   DBNEWID performs validations in the headers of the control files (*not* the datafiles) before attempting I/O to the files. If validation is successful, then DBNEWID prompts for confirmation, changes the database name in the control files, and exits. After DBNEWID completes successfully, the database is left mounted but is not yet usable.

   ```
   DBNEWID: Release 9.2.0.1.0

   (c) Copyright 2002 Oracle Corporation.  All rights reserved.


   Connected to database TEST_DB (DBID=3942196782)
   ```

```
Control Files in database:
    /oracle/dbs/cf1.f
    /oracle/dbs/cf2.f

Change database name of database TEST_DB to TEST_DB2? (Y/[N]) => Y

Proceeding with operation

Database name changed from TEST_DB to TEST_DB2 - database needs to be
shutdown.
Modify parameter file and generate a new password file before restarting.

DBNEWID - Successfully changed database name
```

If validation is not successful, then DBNEWID terminates and leaves the target database intact. You can open the database, fix the error, and then either resume the DBNEWID operation or continue using the database without changing the database name.

4. Shut down the database. For example:

   ```
   SHUTDOWN IMMEDIATE
   ```

5. Set the DB_NAME initialization parameter in the initialization parameter file to the new database name.
6. Create a new password file.
7. Start up the database and resume normal use. For example:

   ```
   STARTUP
   ```

Troubleshooting a DBID Change Operation

If the DBNEWID utility succeeds in its validation stage but detects an error while changing the DBID, then the utility stops and leaves the database in the middle of the change. In this case, you cannot open the database until the DBNEWID operation is either completed or reverted. DBNEWID displays messages indicating the status of the operation.

Before continuing or reverting, fix the underlying cause of the error. Sometimes the only solution is to restore the whole database from a recent backup and perform recovery to the point in time before DBNEWID was started. This underscores the importance of having a recent backup available before running DBNEWID.

If you choose to continue the DBID change operation rather than revert it, reexecute your original command. The DBNEWID utility resumes and attempts to continue the change until all datafiles and control files have the new DBID. At this point, the database is left mounted. You should shut it down and then mount it again prior to opening it with the `RESETLOGS` option.

If you choose to revert a DBNEWID operation, and if the reversion succeeds, then DBNEWID reverts all performed changes and leaves the database in a mounted state.

To revert a stalled DBID change operation, run the DBNEWID utility again, specifying the `REVERT` keyword. For example:

```
% nid TARGET=SYS/oracle REVERT=YES LOGFILE=$HOME/nid.log
```

Troubleshooting a Database Name Change Operation

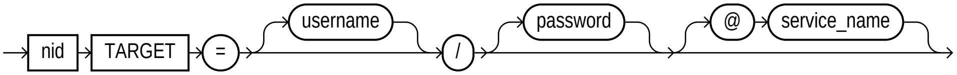
If you specify that only the database name should be changed (and not the DBID), then the validation process is the same as for a DBID change except that DBNEWID checks only the control files. It does not read the datafiles. If the validation encounters a problem, then the database is left mounted.

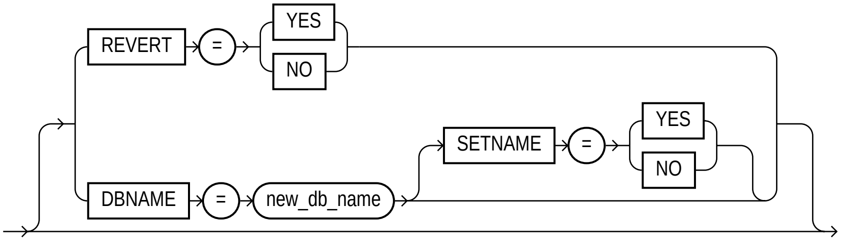
It is possible for validation to succeed, but for the actual database name change to fail. The possible failure scenarios depend on how many control files are in the database, as follows:

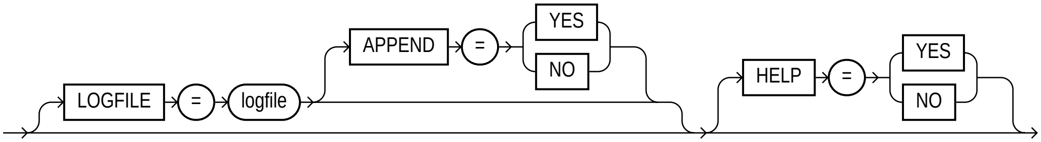
- If you have one or more control files and DBNEWID fails on the first control file, then the database name is not changed in the control file. You can either try the operation again or open the database and resume normal database use.
- If you have more than one control file and DBNEWID fails on the second control file or on any one thereafter, then some control files will have the old DBNAME and some will have the new DBNAME. In this case, you must either manually copy the first changed control file to all `CONTROL_FILES` locations, or revert by copying the unchanged control files to all `CONTROL_FILES` locations.

DBNEWID Syntax

The following diagrams show the syntax for the DBNEWID utility.







Parameters

Table 14–1 describes the parameters in the DBNEWID syntax.

Table 14–1 Parameters for the DBNEWID Utility

Parameter	Description
TARGET	Specifies the username and password used to connect to the database. The user must have the SYSDBA privilege. If you are using operating system authentication, then you can connect with the slash (/). If the $ORACLE_HOME and $ORACLE_SID variables are not set correctly in the environment, then you can specify a secure (IPC or BEQ) service to connect to the target database. A target database must be specified in all invocations of the DBNEWID utility.
REVERT	Specify YES to indicate that a failed change of DBID should be reverted (default is NO). The utility signals an error if no change DBID operation is in progress on the target database. A successfully completed change of DBID cannot be reverted. REVERT=YES is only valid when a DBID change failed.
DBNAME=*new_db_name*	Changes the database name of the database. You can change the DBID and the DBNAME of a database at the same time. To change only the DBNAME, also specify the SETNAME parameter.
SETNAME	Specify YES to indicate that DBNEWID should change the database name of the database but should not change the DBID (default is NO). When you specify SETNAME=YES, the utility only writes to the target database control files.
LOGFILE=*logfile*	Specifies that DBNEWID should write its messages to the specified file. By default the utility overwrites the previous log. If you specify a log file, then DBNEWID does not prompt for confirmation.
APPEND	Specify YES to append log output to the existing log file (default is NO).
HELP	Specify YES to print a list of the DBNEWID syntax options (default is NO).

Restrictions and Usage Notes

The DBNEWID utility has the following restrictions:

- The utility is available only on the UNIX and Windows NT operating systems.
- The nid executable file should be owned and run by the Oracle owner because it needs direct access to the datafiles and control files. If another user runs the utility, then set the user ID to the owner of the datafiles and control files.
- The DBNEWID utility must access the datafiles of the database directly through a local connection. Although DBNEWID can accept a net service name, it cannot change the DBID of a nonlocal database.
- To change the DBID of a database, the database must be mounted and must have been shut down consistently prior to mounting. In the case of an Oracle Real Application Clusters database, the database must be mounted in NOPARALLEL mode.

- You must open the database with the RESETLOGS option after changing the DBID. Note that you do not have to open with the RESETLOGS option after changing only the database name.
- No other process should be running against the database when DBNEWID is executing. If another session shuts down and starts the database, then DBNEWID aborts.
- All online datafiles should be consistent without needing recovery.
- Normal offline datafiles should be accessible and writable. If this is not the case, you must drop these files before invoking the DBNEWID utility.
- All read-only tablespaces must be accessible and made writable at the operating system level prior to invoking DBNEWID. If these tablespaces cannot be made writable (for example, they are on a CD-ROM), then you must unplug the tablespaces using the transportable tablespace feature and then plug them back in the database before invoking the DBNEWID utility (see the *Oracle9i Database Administrator's Guide*).
- You can only specify REVERT when changing only the DBID.

Examples of Using DBNEWID

Changing Only the DBID

The following example connects with operating system authentication and changes only the DBID:

```
% nid TARGET=/
```

Changing the DBID and Database Name

The following example connects as user SYS and changes the DBID and also changes the database name to test2:

```
% nid TARGET=SYS/oracle@test1 DBNAME=test2
```

Changing Only the Database Name

The following example connects as user SYSTEM and changes only the database name, and also specifies a log file for the output:

```
% nid TARGET=SYSTEM/manager@test2 DBNAME=test3 SETNAME=YES LOGFILE=dbid.out
```

15

Using the Metadata API

This chapter describes the Metadata application programming interface (API), which you can use to extract and manipulate complete representations of the metadata for database objects. The following topics are discussed in this chapter:

Introduction to the Metadata API

The Metadata API provides a centralized, simple, and flexible means for performing the following tasks:

- Extracting complete definitions of database objects (metadata) as either XML or creation DDL
- Transforming the metadata through industry-standard XSLT (Extensible Stylesheet Language Transformation).
- Generating SQL DDL to re-create the database objects

The Metadata API is available as of Oracle9*i* release 9.0.1, whenever the instance is operational. It is not available in Oracle Lite.

Previous Methods Used to Extract Metadata

An object's metadata is distributed in normalized fashion across the database dictionary. In prior releases, you first had to understand how and where your object's metadata was represented in the dictionary, then you had to issue multiple queries to extract the object's full representation. Once the metadata was extracted, you would typically perform the following tasks:

1. Transform it in some way, such as changing the object's tablespace, changing a column datatype, changing an object's owner, and so on.
2. Convert it to SQL DDL text for execution on the source or some other database.

Before Oracle9*i* release 9.0.1, there was no assistance for either of these steps.

Metadata API Components

Underlying the Metadata API is an object model of the Oracle database dictionary consisting of a series of user-defined types (UDTs) and corresponding object views. The UDTs provide the aggregation of each object class's metadata, and the object views map the UDTs' attributes onto the appropriate base relational tables in the dictionary. The Metadata API generates queries against these object views to retrieve aggregated database object definitions.

The results of these queries are converted into XML documents by the XML SQL Utility (XSU), which was also introduced in Oracle9*i* release 9.0.1. When the caller requests DDL output, the Metadata API uses the appropriate implementation of the Oracle server's integral XML parser and XSL processor to convert the XML documents into creation DDL.

Metadata API Features

The Metadata API has the following features:

- Provides a powerful PL/SQL interface for detailed programmatic control or casual browsing.
- Supports retrieval of complete, aggregated database definitions for all dictionary objects that can be created with a SQL `CREATE` statement. For a full list of object types, see the `DBMS_METADATA` chapter in *Oracle9i Supplied PL/SQL Packages and Types Reference*.
- Provides only complete representations of objects.

> **Note:** Subsetting of object attributes is not supported in this release except through XSLT transformation.

- Provides database object metadata in an XML format that is easily transformable through XSLT by downstream processes.
- Provides complete Oracle-specific creation DDL for all supported objects.
- Provides flexible object selection. Can return multiple objects per query.
- Supports daisy-chained transforms where the output of the first becomes the input to the second and so on.
- Supports customization of DDL output through object type-specific transform parameters.

Internet Computing

The Metadata API uses two internet standards, XML and XSLT, for encoding and transforming object metadata. Use of an industry-standard format for metadata encoding (rather than a proprietary format) allows you to use standard tools to parse and transform the output.

There is currently no industry-standard XML model for database metadata, so the Metadata API uses a model optimized for generating Oracle DDL. Document element names are derived directly from attributes of the UDTs in the Oracle database dictionary model. As standard models emerge, the Metadata API will support the ability to plug them in. Older documents can be converted to alternate models with XSLT.

How Is the Metadata API Implemented?

The Metadata API is implemented using the PL/SQL DBMS_METADATA package. The DBMS_METADATA package allows you to retrieve metadata from the database dictionary. It provides a flexible and extensible means for object selection. You can use DBMS_METADATA to extract database object metadata in XML and DDL.

The DBMS_METADATA package has two types of interface:

- DBMS_METADATA Programmatic Interface
- DBMS_METADATA Browsing Interface

> **Note:** A description of the types and public interface defined by the Metadata API is in the following location:
>
> `$ORACLE_HOME/rdbms/admin/dbmsmeta.sql`

DBMS_METADATA and Security

The object views of the Oracle metadata model implement security as follows:

- Nonprivileged users can see the metadata of only their own objects.
- SYS and users with SELECT_CATALOG_ROLE can see all objects.
- Nonprivileged users can also retrieve object and system privileges granted to them or by them to others. This also includes privileges granted to PUBLIC.
- If callers request objects they are not privileged to retrieve, no exception is raised; the object is simply not retrieved.
- If nonprivileged users are granted some form of access to an object in someone else's schema, they will be able to retrieve the grant specification through the Metadata API, but not the object's actual metadata.

DBMS_METADATA Programmatic Interface

The DBMS_METADATA programmatic interface is for fine-grained, detailed control:

- The following procedures are provided: OPEN, SET_FILTER, SET_PARSE_ITEM, SET_COUNT, ADD_TRANSFORM, SET_TRANSFORM_PARAM, FETCH_*xxx*, CLOSE
- Metadata is expressed as XML. This allows industry-standard metadata transformations using XSLT.

- You can ask DBMS_METADATA to return metadata as DDL. The API uses XSL scripts internally to transparently perform the conversion.
- You can invoke an XSL script, using either the Oracle XML parser or some third-party tool, to do an offline conversion of the XML representation.

Table 15–1 lists the procedures provided by the DBMS_METADATA programmatic interface and provides a brief description of each one. For more detailed descriptions, including syntax, see *Oracle9i Supplied PL/SQL Packages and Types Reference*.

Table 15–1 Procedures for the DBMS_METADATA Programmatic Interface

PL/SQL Procedure	Description
DBMS_METADATA.OPEN()	Specifies type of object to be retrieved, version of its metadata, and object model. Return value is an opaque context handle for the set of objects to be used in subsequent calls.
BMS_METADATA.SET_FILTER()	Specifies restrictions on objects to be retrieved, such as, object name or schema. Allows specification of base objects for dependent objects such as indexes and triggers.
DBMS_METADATA.SET_COUNT()	Specifies number of objects to be retrieved in a single FETCH_xxx call. By default, each call to FETCH_xxx returns one object.
DBMS_METADATA.GET_QUERY()	Returns text of query (or queries) used by FETCH_xxx. This text is provided to assist in debugging.
DBMS_METADATA.SET_PARSE_ITEM()	Enables output parsing and specifies an object attribute to be parsed and returned. This frees the caller from having to parse SQL DDL for key attributes.
DBMS_METADATA.ADD_TRANSFORM()	Specifies a transform that FETCH_xxx applies to the XML representation of retrieved objects. You can add more than one transform. By default (with no transforms added), objects are returned as XML documents. Call the ADD_TRANSFORM procedure to specify an XSLT script to transform the returned documents. If ’DDL’ is specified, the objects’ creation DDL is returned from subsequent FETCH_xxx calls. The ADD_TRANSFORM procedure returns an opaque transform handle different from that returned by OPEN.

Table 15–1 (Cont.) Procedures for the DBMS_METADATA Programmatic Interface

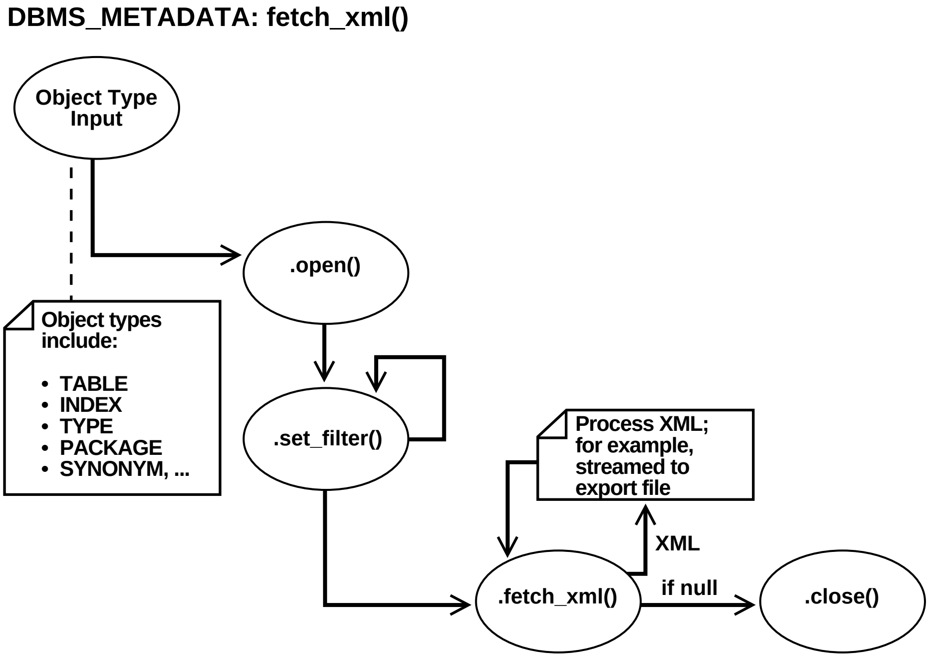
PL/SQL Procedure	Description
DBMS_METADATA.SET_TRANSFORM_PARAM()	Specifies parameters to the XSLT stylesheet identified by the *transform_handle* returned from the ADD_TRANSFORM procedure. For the DDL transform, these parameters alter the form of the DDL. For example, constraints may be requested as column constraints or ALTER TABLE statements.
DBMS_METADATA.FETCH_xxx()	The FETCH_xxx routines return metadata for objects meeting the criteria established by the OPEN, SET_FILTER, SET_COUNT, and ADD_TRANSFORM procedures. FETCH_XML and FETCH_DDL return the metadata as XML and SQL DDL, respectively. The FETCH_CLOB routines return either XML or DDL as denoted by the transforms specified. The types used by these routines are described in *Oracle9i Supplied PL/SQL Packages and Types Reference.*
DBMS_METADATA.CLOSE()	Invalidates the handle returned by the OPEN procedure and cleans up the associated state.

Using the DBMS_METADATA.FETCH_XML Procedure

Figure 15–1 illustrates the steps in DBMS_METADATA.FETCH_XML() usage:

1. Open the object type using the DBMS_METADATA.OPEN() procedure. Object types that you can open include, but are not limited to, tables, indexes, types, packages, and synonyms.
2. Specify which objects to retrieve using the DBMS_METADATA.SET_FILTER()procedure.
3. Fetch the metadata of each qualifying object as an XML document using the DBMS_METADATA.FETCH_XML() procedure. The XML is processed; for example, it might be streamed to an export file.
4. If the result of this operation is NULL, then call the DBMS_METADATA.CLOSE()procedure.

Figure 15–1 Using DBMS_METADATA.FETCH_XML()



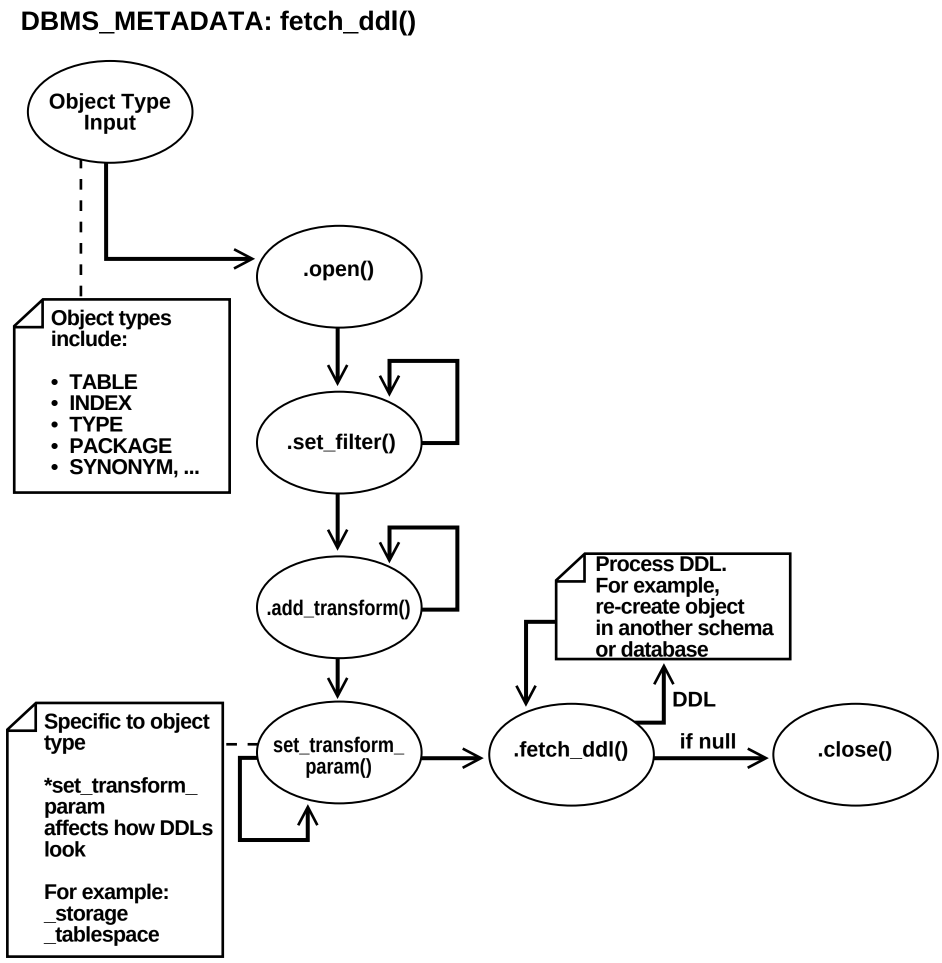
Using the DBMS_METADATA.FETCH_DDL Procedure

Figure 15–2 illustrates the steps in `DBMS_METADATA.FETCH_DDL()` usage:

1. Open the object type using the `DBMS_METADATA.OPEN()` procedure. Object types that you can open include, but are not limited to, tables, indexes, types, packages, and synonyms.
2. Specify which objects to retrieve using the `DBMS_METADATA.SET_FILTER()` procedure.
3. Specify what transforms are to be invoked on the output. Use the `DBMS_METADATA.ADD_TRANSFORM()` procedure to add a transform. The last transform added must be the "DDL" transform.
4. Use the `DBMS_METADATA.SET_TRANSFORM_PARAM()` procedure to customize the DDL. For example, you could use it to exclude storage clauses on table definitions. Transform parameters are specific to the object type chosen.

5. Fetch the DDL using the `DBMS_METADATA.FETCH_DDL()` procedure. An example of the DDL processing is re-creating objects in another schema or database.
6. If the result of this operation is `NULL`, then call the `DBMS_METADATA.CLOSE()` procedure.

Figure 15–2 Using DBMS_METADATA.FETCH_DDL()



Performance Tips for the Programmatic Interface of the Metadata API

This section describes how to enhance performance when using the programmatic interface of the Metadata API.

1. Fetch all of one type of object before fetching the next. For example, if you are retrieving the definitions of all objects in your schema, first fetch all the tables, then all the indexes, then all the triggers, and so on. This will be much faster

than nesting OPEN contexts; that is, fetch one table then all of its indexes, grants, and triggers, then the next table and all of its indexes, grants, and triggers, and so on. The Metadata API Example on page 15-11 reflects this second, less efficient means, but its purpose is to demonstrate most of the programmatic calls, which are best shown by this method.

2. Use the SET_COUNT procedure to retrieve more than one object at a time. This minimizes server round trips and eliminates many redundant function calls.
3. Use the procedure rather than function form of FETCH_CLOB. The procedure form returns the output CLOB by reference through the IN OUT NOCOPY specifier. The function form returns the output CLOB by value requiring an extra LOB copy.
4. When writing a PL/SQL package that calls the Metadata API, declare LOB variables and objects that contain LOBs (such as SYS.KU$_DDLS) at package scope rather than within individual functions. This eliminates the creation and deletion of LOB duration structures upon function entrance and exit, which are very expensive operations.

See Also: *Oracle9i Application Developer's Guide - Large Objects (LOBs)*

DBMS_METADATA Browsing Interface

The DBMS_METADATA browsing interface is for casual use within SQL clients such as SQL*Plus. It is provided by the GET_xxx, GET_DEPENDENT_xxx, and GET_GRANTED_xxx functions.

- The GET_DDL and GET_XML functions return metadata for a single named object. For example, the following query will show the DDL for all tables in the current user's schema:

  ```
  SQL> SELECT dbms_metadata.get_ddl('TABLE', table_name) FROM user_tables;
  ```

- The GET_DEPENDENT_XML, GET_DEPENDENT_DDL, GET_GRANTED_XML, and GET_GRANTED_DDL functions return metadata for one or more dependent or granted objects.

Table 15–2 lists the procedures provided by the DBMS_METADATA browsing interface and provides a brief description of each one. For more detailed descriptions, including syntax, see *Oracle9i Supplied PL/SQL Packages and Types Reference*.

Table 15–2 Procedures for the DBMS_METADATA Browsing Interface

PL/SQL Procedure Name	Description
DBMS_METADATA.GET_xxx()	Provides a way to return metadata for a single object. Each GET_xxx call consists of an OPEN procedure, one or two SET_FILTER calls, optionally an ADD_TRANSFORM procedure, a FETCH_xxx call, and a CLOSE procedure. The *object_type* parameter has the same semantics as in the OPEN procedure. *schema* and *name* are used for filtering. If a transform is specified, session-level transform flags are inherited.
DBMS_METADATA.GET_DEPENDENT_xxx()	Returns the metadata for one or more dependent objects, specified as XML or DDL.
DBMS_METADATA.GET_GRANTED_xxx()	Returns the metadata for one or more granted objects, specified as XML or DDL.

Example: Using the DBMS_METADATA Browsing Interface

The following SQL*Plus query will display the creation DDL for all tables in the current user’s schema. To generate complete, uninterrupted output, set the PAGESIZE to 0 and set LONG to some large number, as shown, before executing your query.

```
SQL> SET PAGESIZE 0
SQL> SET LONG 90000
SQL> SELECT dbms_metadata.get_ddl(’TABLE’, table_name) FROM user_tables;
```

Metadata API Example

The detailed Metadata API programming example in this section, PAYROLL_DEMO, retrieves the DDL for all tables in the MDDEMO schema that start with 'PAYROLL'. It then fetches the DDL for grants, indexes, and triggers defined on those tables. This script can be found in the file rdbms/demo/mddemo.sql in your Oracle home directory.

mddemo.sql

```
-- This script demonstrates how to use the Metadata API. It first
-- establishes a schema (MDDEMO) and some payroll users, then creates three
-- payroll-like tables within it along with associated indexes, triggers
-- and grants.
-- It then creates a package PAYROLL_DEMO that shows common usage of the
-- Metadata API. The procedure GET_PAYROLL_TABLES retrieves the DDL for the
-- two tables in this schema that start with ’PAYROLL’ then for each one,
-- retrieves the DDL for its associate dependent objects; indexes, grants
-- and triggers. All the DDL is written to a table named "MDDEMO"."DDL".

-- First, Install the demo. cd to rdbms/demo:
-- > sqlplus system/manager
-- SQL> @mddemo

-- Then, run it.
-- > sqlplus mddemo/mddemo
-- SQL> set long 40000
-- SQL> set pages 0
-- SQL> call payroll_demo.get_payroll_tables();
-- SQL> select ddl from DDL order by seqno;

Rem Set up schema for demo pkg. PAYROLL_DEMO.

connect system/manager
drop user mddemo cascade;
drop user mddemo_clerk cascade;
drop user mddemo_mgr cascade;

create user mddemo identified by mddemo;
GRANT resource, connect, create session
     , create table
     , create procedure
     , create sequence
     , create trigger
     , create view
     , create synonym
     , alter session
TO mddemo;

create user mddemo_clerk identified by clerk;
create user mddemo_mgr identified by mgr;

connect mddemo/mddemo

Rem Create some payroll-like tables...

create table payroll_emps
( lastname varchar2(60) not null,
  firstname varchar2(20) not null,
 mi varchar2(2),
 suffix varchar2(10),
 DOB date not null,
 badge_no number(6) primary key,
 exempt varchar(1) not null,
 salary number (9,2),
 hourly_rate number (7,2)
)
/
create table payroll_timecards
 badge_no number(6) references payroll_emps (badge_no),
 week number(2),
job_id number(5),
hours_worked number(4,2)
)
/
-- This is a dummy table used only to show that tables NOT starting with
-- 'PAYROLL' are NOT retrieved by payroll_demo.get_payroll_tables

create table audit_trail
(action_time DATE,
lastname VARCHAR2(60),
action LONG
)
/

Rem Then, create some grants...

grant update (salary,hourly_rate) on payroll_emps to mddemo_clerk;
grant ALL on payroll_emps to mddemo_mgr with grant option;

grant insert,update on payroll_timecards to mddemo_clerk;
grant ALL on payroll_timecards to mddemo_mgr with grant option;

Rem Then, create some indexes...

create index i_payroll_emps_name on payroll_emps(lastname);
create index i_payroll_emps_dob on payroll_emps(DOB);

create index i_payroll_timecards_badge on payroll_timecards(badge_no);

Rem Then, create some triggers (and required procedure)...

create or replace procedure check_sal( salary in number) as
begin
```

```
  return;  -- Fairly loose security here...
end;
/

create or replace trigger salary_trigger before insert or update of salary on
payroll_emps
for each row when (new.salary > 150000)
call check_sal(:new.salary)
/

create or replace trigger hourly_trigger before update of hourly_rate on
payroll_emps
for each row
begin :new.hourly_rate:=:old.hourly_rate;end;
/

--
-- Set up a table to hold the generated DDL
--
CREATE TABLE ddl (ddl CLOB, seqno NUMBER);

Rem Finally, create the PAYROLL_DEMO package itself.

CREATE OR REPLACE PACKAGE payroll_demo AS

   PROCEDURE get_payroll_tables;
END;
/
CREATE OR REPLACE PACKAGE BODY payroll_demo AS

-- GET_PAYROLL_TABLES: Fetch DDL for payroll tables and their dependent objects

PROCEDURE  get_payroll_tables IS

tableOpenHandle NUMBER;
depObjOpenHandle NUMBER;
tableTransHandle  NUMBER;
indexTransHandle NUMBER;
schemaName VARCHAR2(30);
tableName VARCHAR2(30);
tableDDLs sys.ku$_ddls;
tableDDL sys.ku$_ddl;
parsedItems   sys.ku$_parsed_items;
depObjDDL CLOB;
seqNo NUMBER := 1;
```

```
TYPE obj_array_t IS VARRAY(3) OF VARCHAR2(30);

-- Load this array with the dependent object classes to be retrieved...
obj_array obj_array_t := obj_array_t('OBJECT_GRANT', 'INDEX', 'TRIGGER');

BEGIN

-- Open a handle for tables in the current schema.
  tableOpenHandle := dbms_metadata.open('TABLE');

-- Tell mdAPI to retrieve one table at a time. This call is not actually
-- necessary since 1 is the default... just showing the call.
  dbms_metadata.set_count(tableOpenHandle, 1);

-- Retrieve tables whose name starts with 'PAYROLL'. When the filter is
-- 'NAME_EXPR', the filter value string must include the SQL operator. This
-- gives the caller flexibility to use LIKE, IN, NOT IN, subqueries, etc.
  dbms_metadata.set_filter(tableOpenHandle, 'NAME_EXPR', 'LIKE ''PAYROLL%''');

-- Tell the mdAPI to parse out each table's schema and name separately so we
-- can use them to set up the calls to retrieve its dependent objects.
  dbms_metadata.set_parse_item(tableOpenHandle, 'SCHEMA');
  dbms_metadata.set_parse_item(tableOpenHandle, 'NAME');

-- Add the DDL transform so we get SQL creation DDL
  tableTransHandle := dbms_metadata.add_transform(tableOpenHandle, 'DDL');

-- Tell the XSL stylesheet we don't want physical storage information (storage,
-- tablespace, etc), and that we want a SQL terminator on each DDL. Notice that
-- these calls use the transform handle, not the open handle.
  dbms_metadata.set_transform_param(tableTransHandle,
     'SEGMENT_ATTRIBUTES', FALSE);
  dbms_metadata.set_transform_param(tableTransHandle,
    'SQLTERMINATOR', TRUE);

-- Ready to start fetching tables. We use the FETCH_DDL interface (rather than
-- FETCH_XML or FETCH_CLOB). This interface returns a SYS.KU$_DDLS; a table of
-- SYS.KU$_DDL objects. This is a table because some object types return
-- multiple DDL statements (like types / pkgs which have create header and
-- body statements). Each KU$_DDL has a CLOB containing the 'CREATE foo'
-- statement plus a nested table of the parse items specified. In our case,
-- we asked for two parse items; Schema and Name. (NOTE: See admin/dbmsmeta.sql
-- for a more detailed description of these types)
```

```
  LOOP
    tableDDLs := dbms_metadata.fetch_ddl(tableOpenHandle);
    EXIT WHEN tableDDLs IS NULL;  -- Get out when no more payroll tables

-- In our case, we know there is only one row in tableDDLs (a KU$_DDLS tbl obj)
-- for the current table. Sometimes tables have multiple DDL statements;
-- eg, if constraints are applied as ALTER TABLE statements, but we didn't ask
-- for that option. So, rather than writing code to loop through tableDDLs,
-- we'll just work with the 1st row.
--
-- First, write the CREATE TABLE text to our output table then retrieve the
-- parsed schema and table names.
    tableDDL := tableDDLs(1);
    INSERT INTO ddl VALUES(tableDDL.ddltext, seqNo);
    seqNo := seqNo + 1;
    parsedItems := tableDDL.parsedItems;

-- Must check the name of the returned parse items as ordering isn't guaranteed
    FOR i IN 1..2 LOOP
      IF parsedItems(i).item = 'SCHEMA'
      THEN
        schemaName := parsedItems(i).value;
      ELSE
        tableName  := parsedItems(i).value;
      END IF;
    END LOOP;

-- Now, we want to retrieve all the dependent objects defined on the current
-- table: indexes, triggers and grants. Since all 'dependent' object types
-- have BASE_OBJECT_NAME and BASE_OBJECT_SCHEMA in common as filter criteria,
-- we'll set up a loop to get all objects of the 3 types, just changing the
-- OPEN context in each pass through the loop. Transform parameters are
-- different for each object type, so we'll only use one that's common to all;
-- SQLTERMINATOR.

    FOR i IN 1..3 LOOP
      depObjOpenHandle := dbms_metadata.open(obj_array(i));
      dbms_metadata.set_filter(depObjOpenHandle,'BASE_OBJECT_SCHEMA',
       schemaName);
      dbms_metadata.set_filter(depObjOpenHandle,'BASE_OBJECT_NAME',tableName);

-- Add the DDL transform and say we want a SQL terminator
      indexTransHandle := dbms_metadata.add_transform(depObjOpenHandle, 'DDL');
      dbms_metadata.set_transform_param(indexTransHandle,
        'SQLTERMINATOR', TRUE);
```

```
-- Retrieve dependent object DDLs as CLOBs and write them to table DDL.
      LOOP
        depObjDDL := dbms_metadata.fetch_clob(depObjOpenHandle);
        EXIT WHEN depObjDDL IS NULL;
        INSERT INTO ddl VALUES(depObjDDL, seqNo);
        seqNo := seqNo + 1;
      END LOOP;

-- Free resources allocated for current dependent object stream.
      dbms_metadata.close(depObjOpenHandle);

    END LOOP; -- End of fetch dependent objects loop

  END LOOP;   -- End of fetch table loop

-- Free resources allocated for table stream and close output file.
  dbms_metadata.close(tableOpenHandle);
  RETURN;

END;  -- of procedure get_payroll_tables

END payroll_demo;
/
```

PAYROLL_DEMO Output

This is the output obtained from executing the procedure, `mddemo.payroll_demo.get_payroll_tables`. The output is obtained by executing the following query as user `mddemo`:

```
SQL> SELECT ddl FROM ddl ORDER BY seqno;

CREATE TABLE "MDDEMO"."PAYROLL_EMPS"
   (    "LASTNAME" VARCHAR2(60) NOT NULL ENABLE,
        "FIRSTNAME" VARCHAR2(20) NOT NULL ENABLE,
        "MI" VARCHAR2(2),
        "SUFFIX" VARCHAR2(10),
        "DOB" DATE NOT NULL ENABLE,
        "BADGE_NO" NUMBER(6,0),
        "EXEMPT" VARCHAR2(1) NOT NULL ENABLE,
        "SALARY" NUMBER(9,2),
        "HOURLY_RATE" NUMBER(7,2),
 PRIMARY KEY ("BADGE_NO") ENABLE
   ) ;
```

```
  GRANT UPDATE ("SALARY") ON "MDDEMO"."PAYROLL_EMPS" TO "MDDEMO_CLERK";
  GRANT UPDATE ("HOURLY_RATE") ON "MDDEMO"."PAYROLL_EMPS" TO "MDDEMO_CLERK";
  GRANT ALTER ON "MDDEMO"."PAYROLL_EMPS" TO "MDDEMO_MGR" WITH GRANT OPTION;
  GRANT DELETE ON "MDDEMO"."PAYROLL_EMPS" TO "MDDEMO_MGR" WITH GRANT OPTION;
  GRANT INDEX ON "MDDEMO"."PAYROLL_EMPS" TO "MDDEMO_MGR" WITH GRANT OPTION;
  GRANT INSERT ON "MDDEMO"."PAYROLL_EMPS" TO "MDDEMO_MGR" WITH GRANT OPTION;
  GRANT SELECT ON "MDDEMO"."PAYROLL_EMPS" TO "MDDEMO_MGR" WITH GRANT OPTION;
  GRANT UPDATE ON "MDDEMO"."PAYROLL_EMPS" TO "MDDEMO_MGR" WITH GRANT OPTION;
  GRANT REFERENCES ON "MDDEMO"."PAYROLL_EMPS" TO "MDDEMO_MGR" WITH GRANT OPTION;
  GRANT ON COMMIT REFRESH ON "MDDEMO"."PAYROLL_EMPS" TO "MDDEMO_MGR" WITH GRANT OPTION;
  GRANT QUERY REWRITE ON "MDDEMO"."PAYROLL_EMPS" TO "MDDEMO_MGR" WITH GRANT OPTION;

  CREATE INDEX "MDDEMO"."I_PAYROLL_EMPS_DOB" ON "MDDEMO"."PAYROLL_EMPS" ("DOB")
  PCTFREE 10 INITRANS 2 MAXTRANS 255
  STORAGE(INITIAL 10240 NEXT 10240 MINEXTENTS 1 MAXEXTENTS 121 PCTINCREASE 50
  FREELISTS 1 FREELIST GROUPS 1 BUFFER_POOL DEFAULT) TABLESPACE "SYSTEM" ;


  CREATE INDEX "MDDEMO"."I_PAYROLL_EMPS_NAME" ON "MDDEMO"."PAYROLL_EMPS" ("LASTNAME")
  PCTFREE 10 INITRANS 2 MAXTRANS 255
  STORAGE(INITIAL 10240 NEXT 10240 MINEXTENTS 1 MAXEXTENTS 121 PCTINCREASE 50
  FREELISTS 1 FREELIST GROUPS 1 BUFFER_POOL DEFAULT) TABLESPACE "SYSTEM" ;

  CREATE OR REPLACE TRIGGER hourly_trigger before update of hourly_rate on payroll_emps
for each row
begin :new.hourly_rate:=:old.hourly_rate;end;
/
ALTER TRIGGER "MDDEMO"."HOURLY_TRIGGER" ENABLE;

  CREATE OR REPLACE TRIGGER salary_trigger before insert or update of salary on payroll_emps
for each row
WHEN (new.salary > 150000)  CALL check_sal(:new.salary)
/
ALTER TRIGGER "MDDEMO"."SALARY_TRIGGER" ENABLE;


CREATE TABLE "MDDEMO"."PAYROLL_TIMECARDS"
   (    "BADGE_NO" NUMBER(6,0),
        "WEEK" NUMBER(2,0),
        "JOB_ID" NUMBER(5,0),
        "HOURS_WORKED" NUMBER(4,2),
 FOREIGN KEY ("BADGE_NO")
  REFERENCES "MDDEMO"."PAYROLL_EMPS" ("BADGE_NO") ENABLE
   ) ;
```

```
GRANT INSERT ON "MDDEMO"."PAYROLL_TIMECARDS" TO "MDDEMO_CLERK";
GRANT UPDATE ON "MDDEMO"."PAYROLL_TIMECARDS" TO "MDDEMO_CLERK";
GRANT ALTER ON "MDDEMO"."PAYROLL_TIMECARDS" TO "MDDEMO_MGR" WITH GRANT OPTION;
GRANT DELETE ON "MDDEMO"."PAYROLL_TIMECARDS" TO "MDDEMO_MGR" WITH GRANT OPTION;
GRANT INDEX ON "MDDEMO"."PAYROLL_TIMECARDS" TO "MDDEMO_MGR" WITH GRANT OPTION;
GRANT INSERT ON "MDDEMO"."PAYROLL_TIMECARDS" TO "MDDEMO_MGR" WITH GRANT OPTION;
GRANT SELECT ON "MDDEMO"."PAYROLL_TIMECARDS" TO "MDDEMO_MGR" WITH GRANT OPTION;
GRANT UPDATE ON "MDDEMO"."PAYROLL_TIMECARDS" TO "MDDEMO_MGR" WITH GRANT OPTION;
GRANT REFERENCES ON "MDDEMO"."PAYROLL_TIMECARDS" TO "MDDEMO_MGR" WITH GRANT OPTION;
GRANT ON COMMIT REFRESH ON "MDDEMO"."PAYROLL_TIMECARDS" TO "MDDEMO_MGR" WITH GRANT OPTION;
GRANT QUERY REWRITE ON "MDDEMO"."PAYROLL_TIMECARDS" TO "MDDEMO_MGR" WITH GRANT OPTION;

CREATE INDEX "MDDEMO"."I_PAYROLL_TIMECARDS_BADGE" ON "MDDEMO"."PAYROLL_TIMECARDS" ("BADGE_NO")
PCTFREE 10 INITRANS 2 MAXTRANS 255
STORAGE(INITIAL 10240 NEXT 10240 MINEXTENTS 1 MAXEXTENTS 121 PCTINCREASE 50
FREELISTS 1 FREELIST GROUPS 1 BUFFER_POOL DEFAULT) TABLESPACE "SYSTEM" ;
```

Part V

Appendixes

This section contains the following appendixes:

Appendix A, "SQL*Loader Syntax Diagrams"

This appendix provides diagrams of the SQL*Loader syntax.

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.

Appendix C, "Backus-Naur Form Syntax"

This appendix explains the symbols and conventions of the BNF variant used in text descriptions of the syntax diagrams.

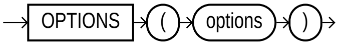
A

SQL*Loader Syntax Diagrams

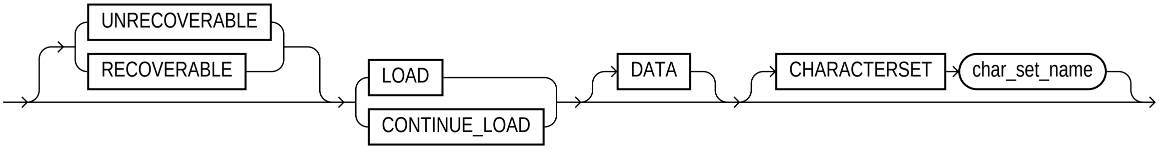
The SQL*Loader DDL diagrams (sometimes called railroad diagrams) use standard SQL syntax notation. For more information about the syntax notation used in this appendix, see the *PL/SQL User's Guide and Reference* and the *Oracle9i SQL Reference*.

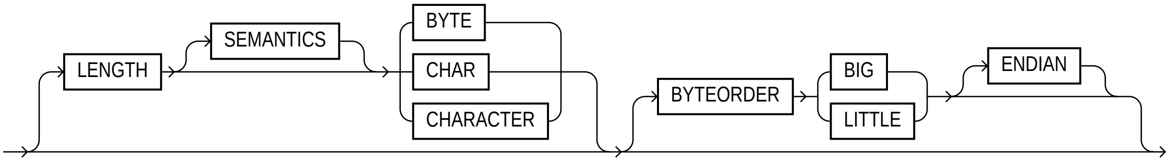
The following diagrams of DDL syntax are shown with certain clauses collapsed (such as pos_spec). These diagrams are expanded and explained further along in the appendix.

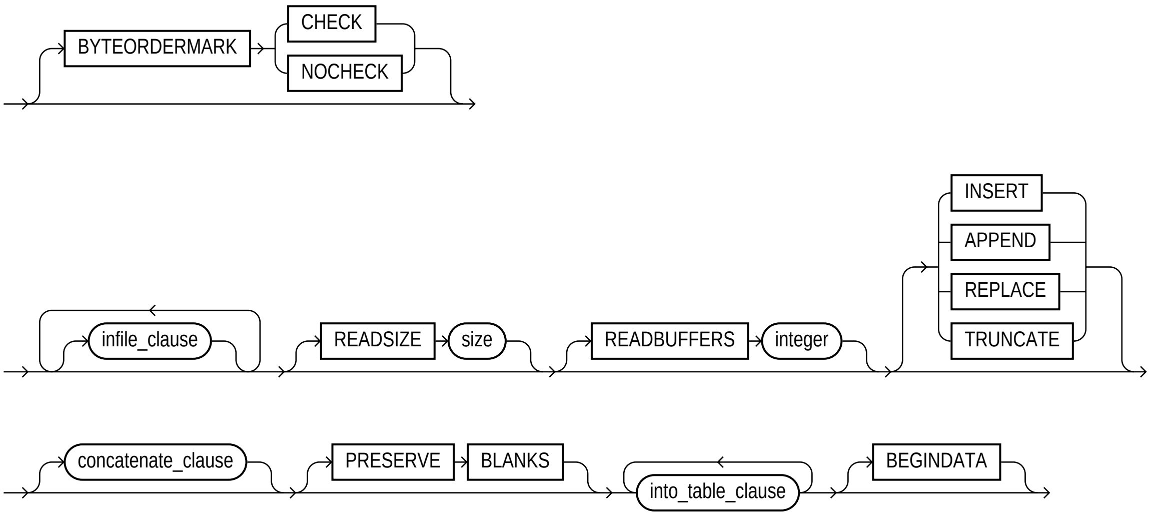
Options Clause



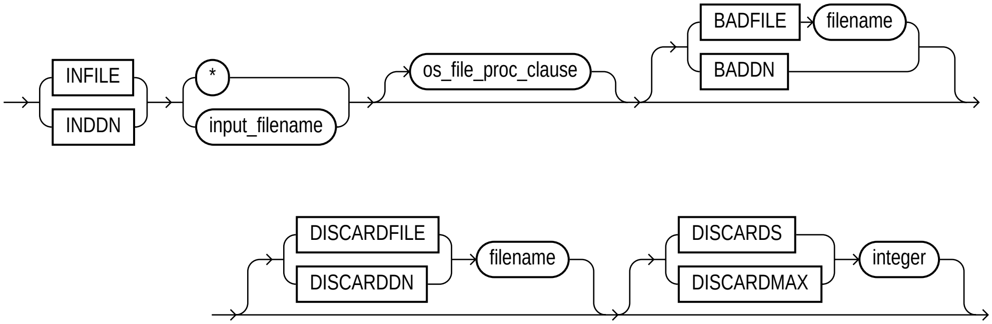
Load Statement



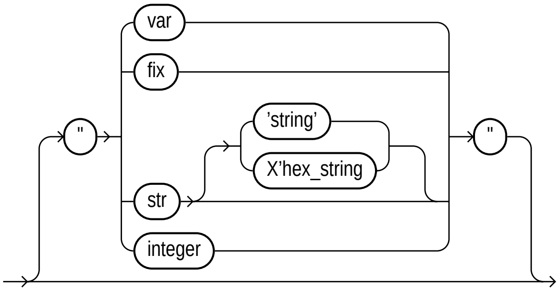




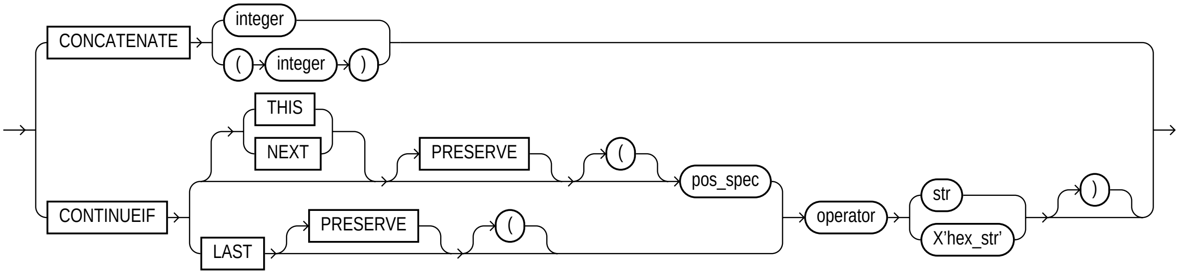
infile_clause



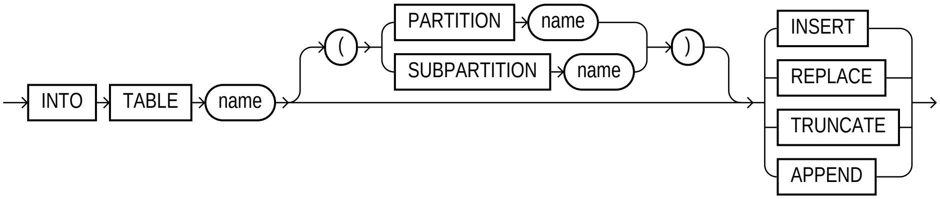
os_file_proc_clause

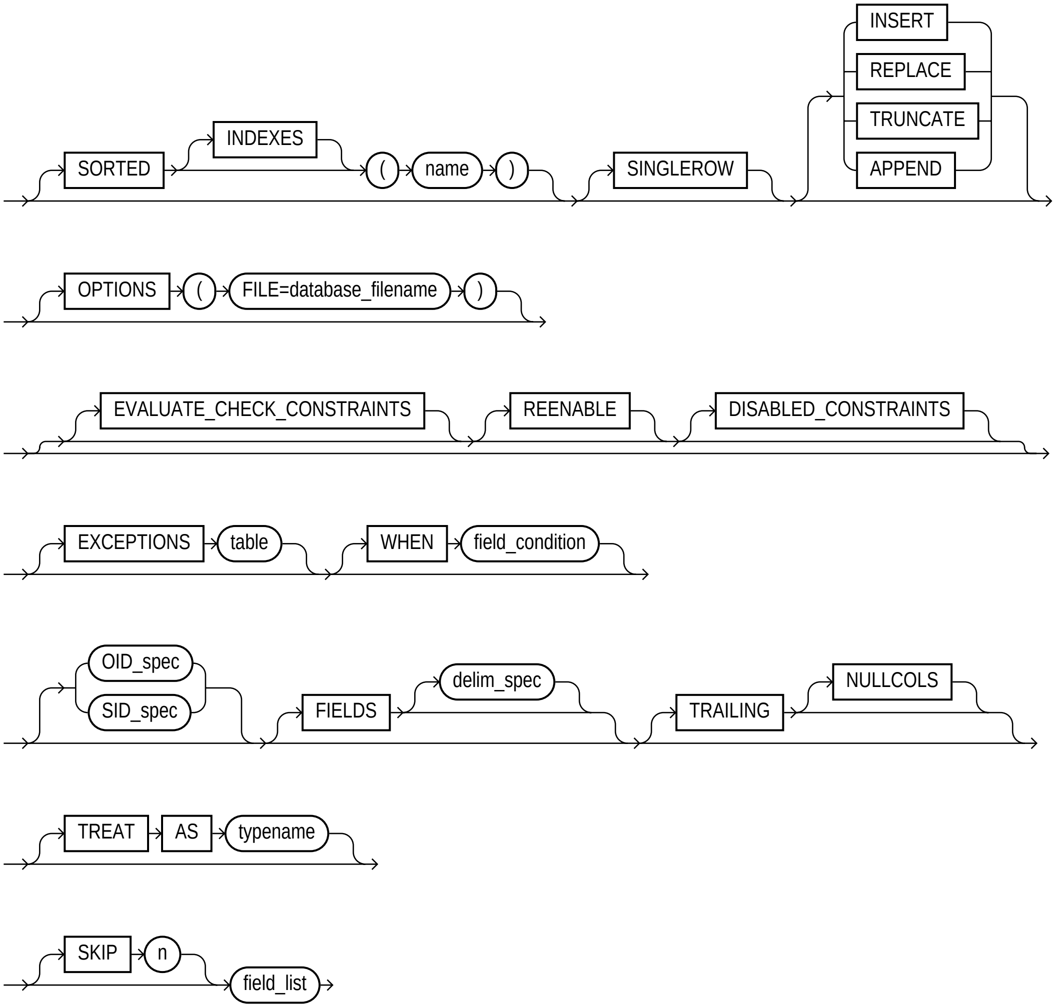


concatenate_clause

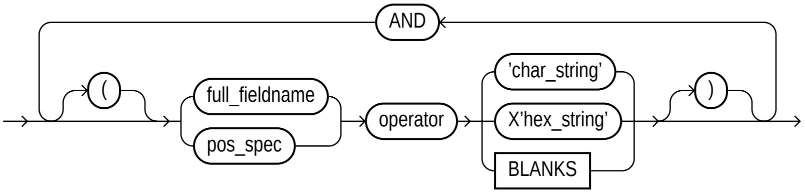


into_table_clause

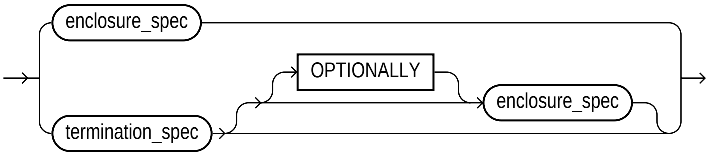



SORTED
INDEXES
(
name
)
SINGLEROW
INSERT
REPLACE
TRUNCATE
APPEND
OPTIONS
(
FILE=database_filename
)
EVALUATE_CHECK_CONSTRAINTS
REENABLE
DISABLED_CONSTRAINTS
EXCEPTIONS
table
WHEN
field_condition
OID_spec
SID_spec
FIELDS
delim_spec
TRAILING
NULLCOLS
TREAT
AS
typename
SKIP
n
field_list

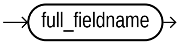
field_condition



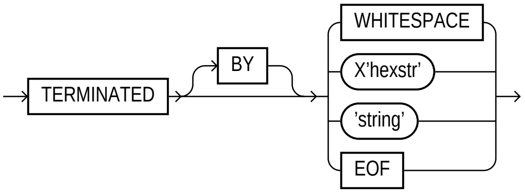
delim_spec



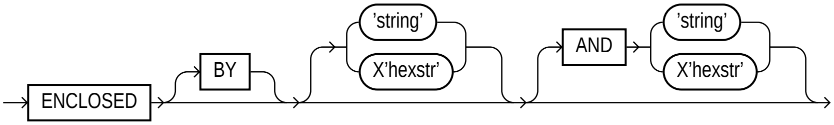
full_fieldname



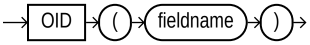
termination_spec



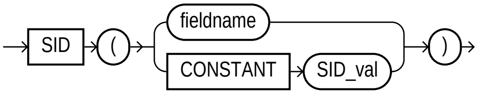
enclosure_spec



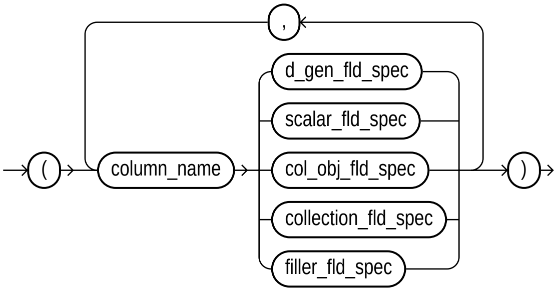
oid_spec



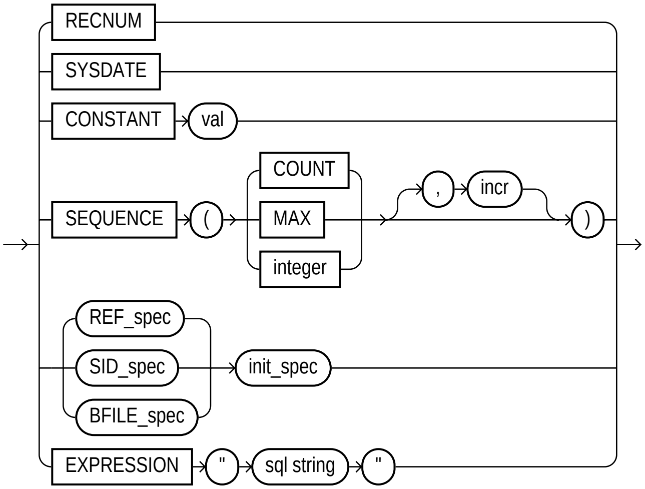
sid_spec



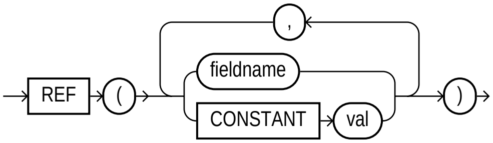
field_list



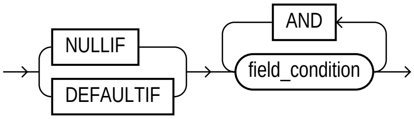
d_gen_fld_spec



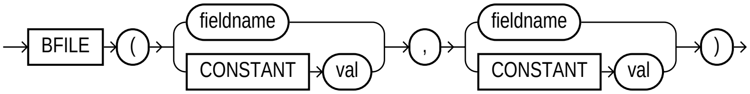
ref_spec



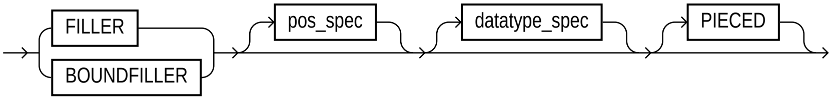
init_spec



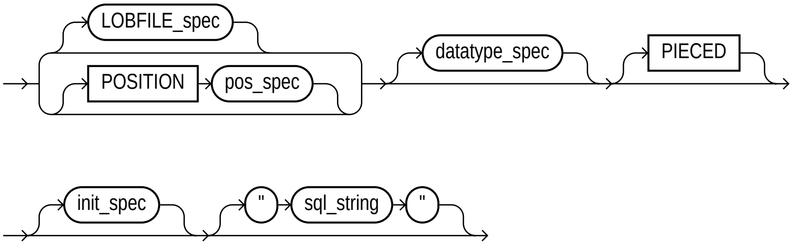
bfile_spec



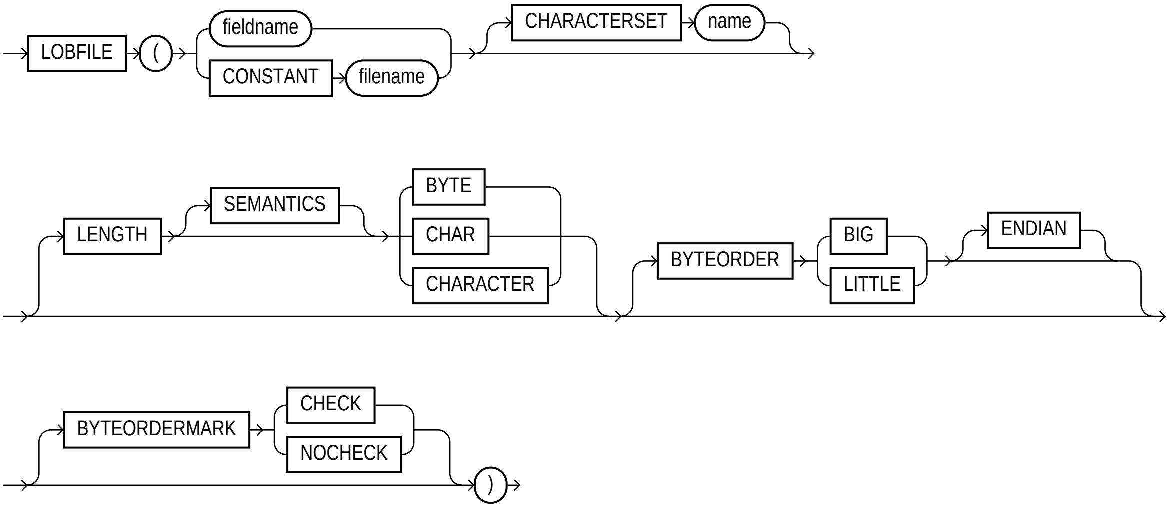
filler_fld_spec



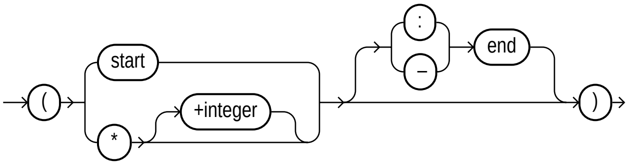
scalar_fld_spec



lobfile_spec

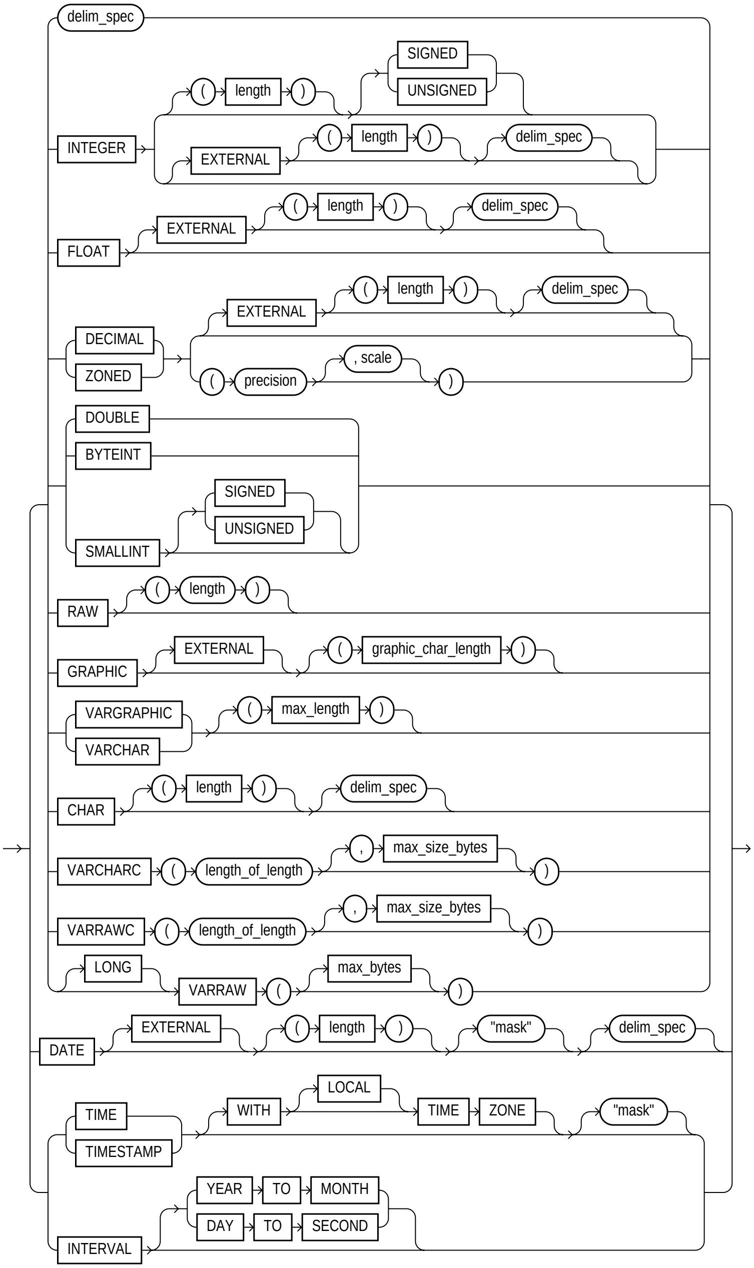


pos_spec

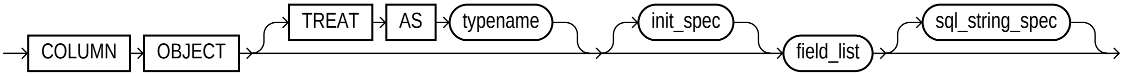


datatype_spec

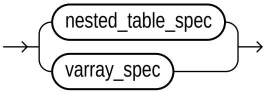
The syntax for `datatype_spec` is as follows:


delim_spec
INTEGER
(
length
)
SIGNED
UNSIGNED
EXTERNAL
(
length
)
delim_spec
FLOAT
EXTERNAL
(
length
)
delim_spec
DECIMAL
ZONED
EXTERNAL
(
length
)
delim_spec
(
precision
, scale
)
DOUBLE
BYTEINT
SMALLINT
SIGNED
UNSIGNED
RAW
(
length
)
GRAPHIC
EXTERNAL
(
graphic_char_length
)
VARGRAPHIC
VARCHAR
(
max_length
)
CHAR
(
length
)
delim_spec
VARCHARC
(
length_of_length
,
max_size_bytes
)
VARRAWC
(
length_of_length
,
max_size_bytes
)
LONG
VARRAW
(
max_bytes
)
DATE
EXTERNAL
(
length
)
"mask"
delim_spec
TIME
TIMESTAMP
WITH
LOCAL
TIME
ZONE
"mask"
INTERVAL
YEAR
TO
MONTH
DAY
TO
SECOND

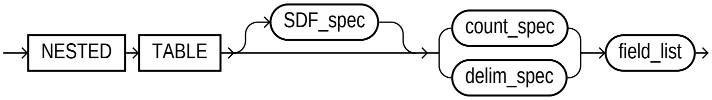
col_obj_fld_spec



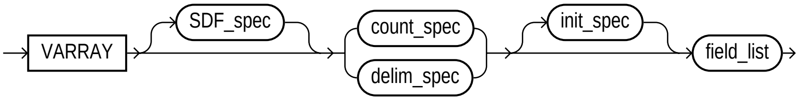
collection_fld_spec



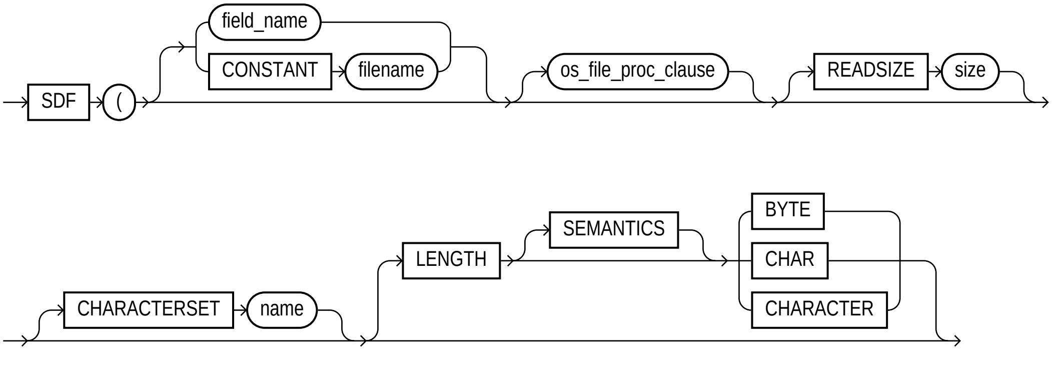
nested_table_spec

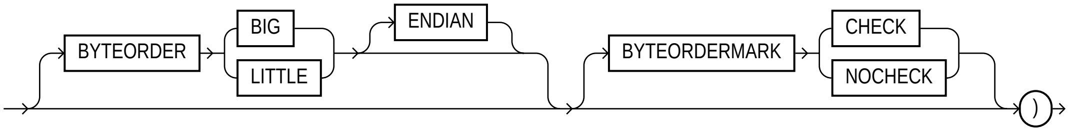


varray_spec

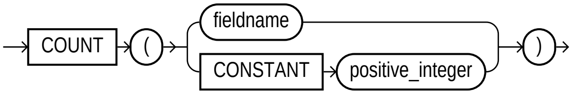


sdf_spec




BYTEORDER
BIG
LITTLE
ENDIAN
BYTEORDERMARK
CHECK
NOCHECK
)

count_spec


COUNT
(
fieldname
CONSTANT
positive_integer
)

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

If the tables you are loading already contain data, you have three choices (shown in Table B–1) for the disposition of that data.

*Table B–1 DB2 Functions and Equivalent SQL*Loader Options*

DB2	SQL*Loader Options	Result
RESUME NO or no RESUME clause	INSERT	Data is loaded only if the 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.

The DB2 syntax for the RESUME clause is as follows:

```
RESUME  { YES | NO [ REPLACE ] }
```

Instead of the DB2 syntax for RESUME, you may prefer to use the equivalent SQL*Loader options.

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 the Oracle database server. The syntactical elements described in the following section 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 you are loading a single DB2-compatible file, these clauses can be in their old position—between the RESUME and RECLEN clauses. Note that while the 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 or from DB2 UNLOAD files. SQL*Loader gives an error upon encountering the DB2 Load Utility commands described in the following sections.

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 the FORMAT statement 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. If the `SQL/DS` option appears in this statement, SQL*Loader will terminate with an error.

DBCS Graphic Strings

Because the Oracle database server 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
{ [ RECNUM
| SYSDATE | CONSTANT value
| SEQUENCE ( { n | MAX | COUNT } [ , increment ] )
| [[ POSITION ( { start [ {:|-} end ] | * [+n] } ) ]
[ datatype_spec ]
[ NULLIF field_condition ]
[ DEFAULTIF field_condition ]
[ "sql string" ] ] ]  }
[ , column_name ] ...)
[ INTO TABLE ] ... [ BEGINDATA ]
[ BEGINDATA]
```

C

Backus-Naur Form Syntax

Each graphic syntax diagram in this book is followed by a link to a text description of the graphic. The text descriptions are a simple variant of Backus-Naur Form (BNF) syntax that includes the symbols and conventions explained in Table C–1.

Table C–1 Symbols and Conventions for Backus-Naur Form Syntax

Symbol or Convention	Meaning
[ ]	Brackets enclose one or more optional items.
{ }	Braces enclose two or more items, one of which is required.
|	A vertical bar separates alternatives within brackets or braces.
...	Ellipsis points show that the preceding syntactic element can be repeated.
delimiters	Delimiters other than brackets, braces, vertical bars, and ellipses must be entered as shown.
boldface	Words appearing in boldface are keywords. They must be typed as shown. (Keywords are case-sensitive in some, but not all, operating systems.) Words that are not in boldface are placeholders for which you must substitute a name or value.

Index