Oracle® Database

Utilities

10*g* Release 1 (10.1)

Part No. B10825-01

December 2003

ORACLE®

Oracle Database Utilities, 10*g* Release 1 (10.1)

Part No. B10825-01

Primary Author: Kathy Rich

Contributors: Lee Barton, Ellen Batbouta, Janet Blowney, George Claborn, Jay Davison, Steve DiPirro, Bill Fisher, Dean Gagne, John Galanes, John Kalogeropoulos, Jonathan Klein, Cindy Lim, Eric Magrath, Brian McCarthy, Rod Payne, Ray Pfau, Rich Phillips, Paul Reilly, Mike Sakayeda, Francisco Sanchez, Marilyn Saunders, Jim Stenoish, Carol Tagliaferri

Contents

2 Data Pump Export

3 Data Pump Import

7 SQL*Loader Command-Line Reference

8 SQL*Loader Control File Reference

9 Field List Reference

10 Loading Objects, LOBs, and Collections

11 Conventional and Direct Path Loads

12 SQL*Loader Case Studies

13 External Tables Concepts

14 The ORACLE_LOADER Access Driver

16 DBVERIFY: Offline Database Verification Utility

17 DBNEWID Utility

18 Using the Metadata API

19 Using LogMiner to Analyze Redo Log Files

20 Original Export and Import

List of Examples

List of Figures

List of Tables

Send Us Your Comments

Oracle Database Utilities, 10*g* Release 1 (10.1)

Part No. B10825-01

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this publication. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most about this manual?

If you find any errors or have any other suggestions for improvement, please indicate the title and part number of the documentation 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: Oracle Database Utilities Documentation
- Postal service:
  Oracle Corporation
  Oracle 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 electronic mail address (optional).

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

Preface

This document describes how to use the Oracle Database utilities for data transfer, data maintenance, and database administration. The preface contains these topics:

Audience

Oracle Database Utilities is intended 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 (both the original versions and the Data Pump versions)
- Load data into Oracle tables from operating system files using SQL*Loader or from external sources using the external tables feature
- Perform a physical data structure integrity check on an offline database, using the DBVERIFY utility
- Maintain the internal database identifier (DBID) and the database name (DBNAME) for an operational database, using the DBNEWID utility

- Extract and manipulate complete representations of the metadata for database objects, using the Metadata API
- Query and analyze redo log files (through a SQL interface) using the LogMiner utility

To use this manual, you need a working knowledge of SQL and Oracle fundamentals, information that is contained in *Oracle 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 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, "Oracle Data Pump"

Chapter 1, "Overview of Oracle Data Pump"

This chapter provides an overview of Oracle Data Pump technology, which enables very high-speed movement of data and metadata from one database to another.

Chapter 2, "Data Pump Export"

This chapter describes the Oracle Data Pump Export utility, which is used to unload data and metadata into a set of operating system files called a dump file set.

Chapter 3, "Data Pump Import"

This chapter describes the Oracle Data Pump Import utility, which is used to load an export dump file set into a target system. It also describes how to perform a network import to load a target database directly from a source database with no intervening files.

Chapter 4, "Data Pump Performance"

This chapter discusses why the performance of Data Pump Export and Import is better than that of original Export and Import. It also suggests specific steps you can take to enhance performance of export and import operations.

Chapter 5, "The Data Pump API"

This chapter describes how the Data Pump API, `DBMS_DATAPUMP`, works.

Part II, "SQL*Loader"

Chapter 6, "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 7, "SQL*Loader Command-Line Reference"

This chapter describes the command-line syntax used by SQL*Loader. It discusses invoking SQL*Loader, command-line arguments, and exit codes.

Chapter 8, "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 9, "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 10, "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 11, "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 12, "SQL*Loader Case Studies"

This chapter presents case studies that illustrate some of the features of SQL*Loader. Some of the features demonstrated are the loading of variable-length data, fixed-format records, free-format files, multiple physical records as one logical record, and multiple tables, as well as performing direct path loads, and loading objects, collections, and REF columns.

Part III, "External Tables"

Chapter 13, "External Tables Concepts"

This chapter describes basic concepts about external tables.

Chapter 14, "The ORACLE_LOADER Access Driver"

This chapter describes the access parameters used to interface with the ORACLE_LOADER access driver.

Chapter 15, "The ORACLE_DATAPUMP Access Driver"

This chapter describes the ORACLE_DATAPUMP access driver, including its parameters, and information about loading and unloading supported datatypes.

Part IV, "Other Utilities"

Chapter 16, "DBVERIFY: Offline Database Verification Utility"

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

Chapter 17, "DBNEWID Utility"

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

Chapter 18, "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.

Chapter 19, "Using LogMiner to Analyze Redo Log Files"

This chapter describes the LogMiner utility, which you can use to query redo logs through a SQL interface.

Chapter 20, "Original Export and Import"

This chapter describes how to use the original Export and Import utilities to write data from an Oracle database into transportable files, and then to read that data into an Oracle database. It discusses guidelines, export and import modes, and available parameters. It also provides examples of various types of export and import operations.

Part V, "Appendixes"

Appendix A, "SQL*Loader Syntax Diagrams"

This appendix provides diagrams of the SQL*Loader syntax.

Appendix B, "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 these Oracle resources:

The Oracle Database documentation set, especially:

- *Oracle Database Concepts*
- *Oracle Database SQL Reference*
- *Oracle Database Administrator's Guide*
- *PL/SQL Packages and Types Reference*

Some of the examples in this book use the sample schemas of the seed database, which is installed by default when you install Oracle Database. Refer to *Oracle Database Sample Schemas* for information about how these schemas were created and how you can use them yourself.

Oracle error message documentation is only available in HTML. If you only have access to the Oracle Database Documentation CD, you can browse the error messages by range. Once you find the specific range, use your browser's "find in page" feature to locate the specific message. When connected to the Internet, you can search for a specific error message using the error message search feature of the Oracle online documentation.

Printed documentation is available for sale in the Oracle Store at

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

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/membership/`

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/documentation/`

Conventions

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

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.	*Oracle 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.
`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 italic monospace 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]
...	Horizontal ellipsis points indicate either: ■ That we have omitted parts of the code that are not directly related to the example ■ That you can repeat a portion of the code	CREATE TABLE ... AS *subquery*; 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;

Convention	Meaning	Example
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 Oracle Database 10*g* utilities and provides pointers to additional information. For information about features that were introduced in earlier releases of Oracle Database, refer to the documentation for those releases.

New Features in Oracle Database 10*g*

Data Pump Technology

Oracle Database 10*g* introduces the new Oracle Data Pump technology, which enables very high-speed movement of data and metadata from one database to another. This technology is the basis for Oracle's new data movement utilities, Data Pump Export and Data Pump Import.

See Chapter 1, "Overview of Oracle Data Pump" for more information.

Data Pump Export

Data Pump Export is a utility that makes use of Oracle Data Pump technology to unload data and metadata at high speeds into a set of operating system files called a dump file set. The dump file set can be moved to another system and loaded by the Data Pump Import utility.

Although the functionality of Data Pump Export (invoked with the `expdp` command) is similar to that of the original Export utility (`exp`), they are completely separate utilities.

See Chapter 2, "Data Pump Export" for more information.

Data Pump Import

Data Pump Import is a utility for loading a Data Pump Export dump file set into a target system.

Although the functionality of Data Pump Import (invoked with the `impdp` command) is similar to that of the original Import utility (`imp`), they are completely separate utilities.

See Chapter 3, "Data Pump Import" for more information.

Data Pump API

The Data Pump API provides a high-speed mechanism to move all or part of the data and metadata from one database to another. The Data Pump Export and Data Pump Import utilities are based on the Data Pump API.

The Data Pump API is implemented through a PL/SQL package, `DBMS_DATAPUMP`, that provides programmatic access to Data Pump data and metadata movement capabilities.

See Chapter 5, "The Data Pump API" for more information.

Metadata API

The following features have been added or updated for Oracle Database 10*g*.

- You can now use remap parameters, which enable you to modify an object by changing specific old attribute values to new values. For example, when you are importing data into a database, you can use the `REMAP_SCHEMA` parameter to change occurrences of schema name `scott` in a dump file set to schema name `blake`.
- All dictionary objects needed for a full export are supported.
- You can request that a heterogeneous collection of objects be returned in creation order.
- In addition to retrieving metadata as XML and creation DDL, you can now submit the XML to re-create the object.

See Chapter 18, "Using the Metadata API" for full descriptions of these features.

External Tables

A new access driver, `ORACLE_DATAPUMP`, is now available. See Chapter 15, "The ORACLE_DATAPUMP Access Driver" for more information.

LogMiner

The LogMiner utility, previously documented in the *Oracle9i Database Administrator's Guide*, is now documented in this guide. The new and changed LogMiner features for Oracle Database 10*g* are as follows:

- The new DBMS_LOGMNR.REMOVE_LOGFILE() procedure removes log files from the list of those being analyzed. This subprogram replaces the REMOVEFILE option to the DBMS_LOGMNR.ADD_LOGFILE() procedure.
- The new NO_ROWID_IN_STMT option for DBMS_LOGMNR.START_LOGMNR procedure lets you filter out the ROWID clause from reconstructed SQL_REDO and SQL_UNDO statements.
- Supplemental logging is enhanced as follows:
  - At the database level, there are two new options for identification key logging:
    - FOREIGN KEY

      Supplementally logs all other columns of a row's foreign key if any column in the foreign key is modified.
    - ALL

      Supplementally logs all the columns in a row (except for LOBs LONGs, and ADTs) if any column value is modified.
  - At the table level, there are these new features:
    - Identification key logging is now supported (PRIMARY KEY, FOREIGN KEY, UNIQUE INDEX, and ALL).
    - The NO LOG option provides a way to prevent a column in a user-defined log group from being supplementally logged.

See Chapter 19, "Using LogMiner to Analyze Redo Log Files" for more information.

Part I

Oracle Data Pump

This part contains the following chapters:

- Chapter 1, "Overview of Oracle Data Pump"

  This chapter provides an overview of Oracle Data Pump technology, which enables very high-speed movement of data and metadata from one database to another.

- Chapter 2, "Data Pump Export"

  This chapter describes the Oracle Data Pump Export utility, which is used to unload data and metadata into a set of operating system files called a dump file set.

- Chapter 3, "Data Pump Import"

  This chapter describes the Oracle Data Pump Import utility, which is used to load an export dump file set into a target system. It also describes how to perform a network import to load a target database directly from a source database with no intervening files.

- Chapter 4, "Data Pump Performance"

  This chapter discusses why the performance of Data Pump Export and Import is better than that of original Export and Import. It also suggests specific steps you can take to enhance performance of export and import operations.

- Chapter 5, "The Data Pump API"

  This chapter describes how the Data Pump API, `DBMS_DATAPUMP`, works.

1

Overview of Oracle Data Pump

Oracle Data Pump technology enables very high-speed movement of data and metadata from one database to another.

This chapter discusses the following topics:

Data Pump Components

Oracle Data Pump is made up of three distinct parts:

- The command-line clients, `expdp` and `impdp`
- The `DBMS_DATAPUMP` PL/SQL package (also known as the Data Pump API)
- The `DBMS_METADATA` PL/SQL package (also known as the Metadata API)

The Data Pump clients, `expdp` and `impdp`, invoke the Data Pump Export utility and Data Pump Import utility, respectively. They provide a user interface that closely resembles the original export (`exp`) and import (`imp`) utilities.

The expdp and impdp clients use the procedures provided in the DBMS_DATAPUMP PL/SQL package to execute export and import commands, using the parameters entered at the command-line. These parameters enable the exporting and importing of data and metadata for a complete database or subsets of a database.

Note: All Data Pump Export and Import processing, including the reading and writing of dump files, is done on the server. **This means that the data base administrator (DBA) must create directory objects.** See Default Locations for Dump, Log, and SQL Files on page 1-13 for more information about directory objects.

When data is moved, Data Pump automatically uses either direct path load (or unload) or the external tables mechanism, or a combination of both. When metadata is moved, Data Pump uses functionality provided by the DBMS_METADATA PL/SQL package. The DBMS_METADATA package provides a centralized facility for the extraction, manipulation, and resubmission of dictionary metadata.

The DBMS_DATAPUMP and DBMS_METADATA PL/SQL packages can be used independently of the Data Pump clients.

See Also: *PL/SQL Packages and Types Reference* for descriptions of the DBMS_DATAPUMP and DBMS_METADATA packages

What New Features Do Data Pump Export and Import Provide?

The new Data Pump Export and Import utilities (invoked with the expdp and impdp commands, respectively) have a similar look and feel to the original Export (exp) and Import (imp) utilities, but they are completely separate. Dump files generated by the new Data Pump Export utility are not compatible with dump files generated by the original Export utility. Therefore, files generated by the original Export (exp) utility cannot be imported with the Data Pump Import (impdp) utility.

Oracle recommends that you use the new Data Pump Export and Import utilities because they support all Oracle Database 10*g* features, except for XML schemas. Original Export and Import support the full set of Oracle database release 9.2 features. Also, the design of Data Pump Export and Import results in greatly enhanced data movement performance over the original Export and Import utilities.

> **Note:** See Chapter 20, "Original Export and Import" for information about situations in which you should still use the original Export and Import utilities.

The following are the major new features that provide this increased performance, as well as enhanced ease of use:

- The ability to specify the maximum number of threads of active execution operating on behalf of the Data Pump job. This enables you to adjust resource consumption versus elapsed time. See PARALLEL on page 2-25 for information about using this parameter in export. See PARALLEL on page 3-23 for information about using this parameter in import. (This feature is available only in the Enterprise Edition of Oracle Database 10*g*.)
- The ability to restart Data Pump jobs. See START_JOB on page 2-41 for information about restarting export jobs. See START_JOB on page 3-48 for information about restarting import jobs.
- The ability to detach from and reattach to long-running jobs without affecting the job itself. This allows DBAs and other operations personnel to monitor jobs from multiple locations. The Data Pump Export and Import utilities can be attached to only one job at a time; however, you can have multiple clients or jobs running at one time. (If you are using the Data Pump API, the restriction on attaching to only one job at a time does not apply.) You can also have multiple clients attached to the same job. See ATTACH on page 2-9 for information about using this parameter in export. See ATTACH on page 3-8 for information about using this parameter in import.
- Support for export and import operations over the network, in which the source of each operation is a remote instance. See NETWORK_LINK on page 2-23 for information about using this parameter in export. See NETWORK_LINK on page 3-22 for information about using this parameter in import.
- The ability, in an import job, to change the name of the source datafile to a different name in all DDL statements where the source datafile is referenced. See REMAP_DATAFILE on page 3-27.
- Enhanced support for remapping tablespaces during an import operation. See REMAP_TABLESPACE on page 3-29.
- Support for filtering the metadata that is exported and imported, based upon objects and object types. For information about filtering metadata during an export operation, see INCLUDE on page 2-20 and EXCLUDE on page 2-15. For

information about filtering metadata during an import operation, see INCLUDE on page 3-18 and EXCLUDE on page 3-13.

- Support for an interactive-command mode that allows monitoring of and interaction with ongoing jobs. See Commands Available in Export's Interactive-Command Mode on page 2-37 and Commands Available in Import's Interactive-Command Mode on page 3-44.
- The ability to estimate how much space an export job would consume, without actually performing the export. See ESTIMATE_ONLY on page 2-14.
- The ability to specify the version of database objects to be moved. In export jobs, `VERSION` applies to the version of the database objects to be exported. See VERSION on page 2-34 for more information about using this parameter in export.

  In import jobs, `VERSION` applies only to operations over the network. This means that `VERSION` applies to the version of database objects to be extracted from the source database. See VERSION on page 3-41 for more information about using this parameter in import.
- Most Data Pump export and import operations occur on the Oracle database server. (This contrasts with original export and import, which were primarily client-based.) See Default Locations for Dump, Log, and SQL Files on page 1-13 for information about some of the implications of server-based operations.

The remainder of this chapter discusses Data Pump technology as it is implemented in the Data Pump Export and Import utilities. To make full use of Data Pump technology, you must be a privileged user. Privileged users have the `EXP_FULL_DATABASE` and `IMP_FULL_DATABASE` roles. Nonprivileged users have neither.

Privileged users can do the following:

- Export and import database objects owned by others
- Export and import nonschema-based objects such as tablespace and schema definitions, system privilege grants, resource plans, and so forth
- Attach to, monitor, and control Data Pump jobs initiated by others
- Perform remapping operations on schemas and database datafiles

How Does Data Pump Access Data?

Data Pump supports two access methods to load and unload table row data: direct path and external tables. Because both methods support the same external data

representation, data that is unloaded with one method can be loaded using the other method. Data Pump automatically chooses the fastest method appropriate for each table.

Data Pump also uses functionality provided in the DBMS_METADATA PL/SQL package to handle all operations involving metadata, including complete extraction, transformation, and re-creation of all database object definitions.

Direct Path Loads and Unloads

The Oracle database has provided direct path unload capability for export operations since Oracle release 7.3 and a direct path API for OCI since Oracle8*i*. Data Pump technology enhances direct path technology in the following ways:

- Support of a direct path, proprietary format unload.
- Improved performance through elimination of unnecessary conversions. This is possible because the direct path internal stream format is used as the format stored in the Data Pump dump files.
- Support of additional datatypes and transformations.

Data Pump uses direct path load and unload when the structure of a table allows it.

In the following circumstances, Data Pump cannot use direct path loading:

- A global index on multipartition tables exists during a single-partition load. This includes object tables that are partitioned.
- A domain index exists for a LOB column.
- A table is in a cluster.
- A table has an active trigger.
- A table has fine-grained access control enabled in insert mode.
- A table contains BFILE columns or columns of opaque types.
- A referential integrity constraint is present.
- A table contains VARRAY columns with an embedded opaque type.

If any of these conditions exist for a table, Data Pump uses external tables rather than direct path to move the data for that table.

External Tables

The Oracle database has provided an external tables capability since Oracle9*i* that allows reading of data sources external to the database. As of Oracle Database 10*g*, the external tables feature also supports writing database data to destinations external to the database. Data Pump provides an external tables access driver (ORACLE_DATAPUMP) that reads and writes files. The format of the files is the same format used with the direct path method. This allows for high-speed loading and unloading of database tables. Data Pump uses external tables as the data access mechanism in the following situations:

- Loading and unloading very large tables and partitions in situations where parallel SQL can be used to advantage
- Loading tables with global or domain indexes defined on them, including partitioned object tables
- Loading tables with active triggers or clustered tables
- Loading and unloading tables with encrypted columns
- Loading tables with fine-grained access control enabled for inserts
- Loading tables that are partitioned differently at load time and unload time

See Also: Chapter 15, "The ORACLE_DATAPUMP Access Driver"

Accessing Data Over a Network

You can perform Data Pump exports and imports over the network, rather than locally.

When you perform an import over the network, the source is another database, not a dump file set.

When you perform an export over the network, the source can be a read-only database on another system. Dump files are written out on the local system just as they are with a local (non-networked) export.

See Also:

- NETWORK_LINK on page 2-23 for information about performing exports over the network
- NETWORK_LINK on page 3-22 for information about performing imports over the network

What Happens During Execution of a Data Pump Job?

Data Pump jobs use a master table, a master process, and worker processes to perform the work and keep track of progress.

Coordination of a Job

For every Data Pump Export job and Data Pump Import job, a master process is created. The master process controls the entire job, including communicating with the clients, creating and controlling a pool of worker processes, and performing logging operations.

Tracking Progress Within a Job

While the data and metadata are being transferred, a master table is used to track the progress within a job. The master table is implemented as a user table within the database. The specific function of the master table for export and import jobs is as follows:

- For export jobs, the master table records the location of database objects within a dump file set. Export builds and maintains the master table for the duration of the job. At the end of an export job, the content of the master table is written to a file in the dump file set.
- For import jobs, the master table is loaded from the dump file set and is used to control the sequence of operations for locating objects that need to be imported into the target database.

The master table is created in the schema of the current user performing the export or import operation. Therefore, that user must have sufficient tablespace quota for its creation. The name of the master table is the same as the name of the job that created it. Therefore, you cannot explicitly give a Data Pump job the same name as a preexisting table or view.

For all operations, the information in the master table is used to restart a job.

The master table is either retained or dropped, depending on the circumstances, as follows:

- Upon successful job completion, the master table is dropped.
- If a job is stopped using the `STOP_JOB` interactive command, the master table is retained for use in restarting the job.

- If a job is killed using the KILL_JOB interactive command, the master table is dropped and the job cannot be restarted.
- If a job terminates unexpectedly, the master table is retained. You can delete it if you do not intend to restart the job.

See Also: JOB_NAME on page 2-22 for more information about how job names are formed.

Filtering Data During a Job

Within the master table, specific objects are assigned attributes such as name or owning schema. Objects also belong to a class of objects (such as TABLE, INDEX, or DIRECTORY). The class of an object is called its object type. You can use the EXCLUDE and INCLUDE parameters to restrict the types of objects that are exported and imported. The objects can be based upon the name of the object or the name of the schema that owns the object.

Transforming Metadata During a Job

When you are moving data from one database to another, it is often useful to perform transformations on the metadata for remapping storage between tablespaces or redefining the owner of a particular set of objects. This is done using the following Data Pump Import parameters: REMAP_DATAFILE, REMAP_SCHEMA, REMAP_TABLESPACE, and TRANSFORM.

See Also:

- REMAP_DATAFILE on page 3-27
- REMAP_SCHEMA on page 3-27
- REMAP_TABLESPACE on page 3-29
- TRANSFORM on page 3-37

Maximizing Job Performance

To improve throughput of a job, you can use the PARALLEL parameter to set a degree of parallelism that takes maximum advantage of current conditions. For example, to limit the effect of a job on a production system, the database administrator (DBA) might wish to restrict the parallelism. The degree of parallelism can be reset at any time during a job. For example, PARALLEL could be set to 2 during production hours to restrict a particular job to only two degrees of

parallelism, and during nonproduction hours it could be reset to 8. The parallelism setting is enforced by a master process, which allocates work to be executed to a set of worker processes that perform the data and metadata processing within an operation. These worker processes operate in parallel.

Note: The ability to adjust the degree of parallelism is available only in the Enterprise Edition of Oracle Database.

Loading and Unloading of Data

The worker processes are the ones that actually unload and load metadata and table data in parallel. The number of worker processes created is equal to the value supplied for the PARALLEL command-line parameter. The number of worker processes can be reset throughout the life of a job.

Note: The value of PARALLEL is restricted to 1 in the Standard Edition of Oracle Database 10*g*.

When a worker process is assigned the task of loading or unloading a very large table or partition, it may choose to use the external tables access method to make maximum use of parallel execution. In such a case, the worker process becomes a parallel execution coordinator. The actual loading and unloading work is divided among some number of parallel I/O execution processes (sometimes called slaves) allocated from the instancewide pool of parallel I/O execution processes.

Monitoring Job Status

During the execution of a job, a log file will be optionally written. The log file summarizes the progress of the job and any errors that were encountered along the way. Whereas the log file records the completion status of the job, real-time status can be obtained by using the STATUS command in interactive mode of Data Pump Export or Import. Cumulative status for the job is returned, along with a description of the current operation. In addition, an estimate for the completion percentage of the current job is also returned. If the job is done, the state will be listed as Stopped or Completed.

See Also:

- STATUS on page 2-41 for information about the STATUS command in interactive mode of Data Pump Export
- STATUS on page 3-48 for information about the STATUS command in interactive mode of Data Pump Import

An alternative way to determine job status or to get other information about Data Pump jobs, would be to query the DBA_DATAPUMP_JOBS, USER_DATAPUMP_JOBS, or DBA_DATAPUMP_SESSIONS views.

The DBA_DATAPUMP_JOBS and USER_DATAPUMP_JOBS Views

The DBA_DATAPUMP_JOBS and USER_DATAPUMP_JOBS views identify all active Data Pump jobs, regardless of their state, on an instance (or on all instances for Real Application Clusters). They also show all Data Pump master tables not currently associated with an active job. You can use the job information to attach to an active job. Once you are attached to the job, you can stop it, change its parallelism, or monitor its progress. You can use the master table information to restart a stopped job or to remove any master tables that are no longer needed.

Table 1–1 describes the columns in the DBA_DATAPUMP_JOBS view and the USER_DATAPUMP_JOBS view.

Table 1–1 DBA_DATAPUMP_JOBS View and USER_DATAPUMP_JOBS View

Column	Datatype	Description
OWNER_NAME	VARCHAR2(30)	User who initiated the job (valid only for DBA_DATAPUMP_JOBS)
JOB_NAME	VARCHAR2(30)	User-supplied name for the job (or the default name generated by the server)
OPERATION	VARCHAR2(30)	Type of job
JOB_MODE	VARCHAR2(30)	Mode of job
STATE	VARCHAR2(30)	State of the job
DEGREE	NUMBER	Number of worker processes performing the operation
ATTACHED_SESSIONS	NUMBER	Number of sessions attached to the job

Note: The information returned is obtained from dynamic performance views associated with the executing jobs and from the database schema information concerning the master tables. A query on these views can return multiple rows for a single Data Pump job (same owner and job name) if the query is executed while the job is transitioning between an Executing state and the Not Running state.

The DBA_DATAPUMP_SESSIONS View

The DBA_DATAPUMP_SESSIONS view identifies the user sessions that are attached to a job. The information in this view is useful for determining why a stopped operation has not gone away.

Table 1–2 describes the columns in the DBA_DATAPUMP_SESSIONS view.

Table 1–2 The DBA_DATAPUMP_SESSIONS View

Column	Datatype	Description
OWNER_NAME	VARCHAR2(30)	User who initiated the job.
JOB_NAME	VARCHAR2(30)	User-supplied name for the job (or the default name generated by the server).
SADDR	RAW(4) (RAW(8) on 64-bit systems)	Address of session attached to the job. Can be used with V$SESSION view.

Monitoring the Progress of Executing Jobs

Data Pump operations that transfer table data (export and import) maintain an entry in the V$SESSION_LONGOPS dynamic performance view indicating the job progress (in megabytes of table data transferred). The entry contains the estimated transfer size and is periodically updated to reflect the actual amount of data transferred.

Note: The usefulness of the estimate value for export operations depends on the type of estimation requested when the operation was initiated, and it is updated as required if exceeded by the actual transfer amount. The estimate value for import operations is exact.

The V$SESSION_LONGOPS columns that are relevant to a Data Pump job are as follows:

- USERNAME - job owner
- OPNAME - job name
- TARGET_DESC - job operation
- SOFAR - megabytes (MB) transferred thus far during the job
- TOTALWORK - estimated number of megabytes (MB) in the job
- UNITS - 'MB'
- MESSAGE - a formatted status message of the form:

```
'<job_name>: <operation_name> : nnn out of mmm MB done'
```

File Allocation

There are three types of files managed by Data Pump jobs:

- Dump files to contain the data and metadata that is being moved
- Log files to record the messages associated with an operation
- SQL files to record the output of a SQLFILE operation. A SQLFILE operation is invoked using the Data Pump Import SQLFILE parameter and results in all of the SQL DDL that Import will be executing based on other parameters, being written to a SQL file. See SQLFILE on page 3-32 for more information.

An understanding of how Data Pump allocates and handles these files will help you to use Export and Import to their fullest advantage.

Specifying Files and Adding Additional Dump Files

For export operations, you can specify dump files at the time the job is defined, as well as at a later time during the operation. For example, if you discover that space is running low during an export operation, you can add additional dump files by using the Data Pump Export ADD_FILE command in interactive mode.

For import operations, all dump files must be specified at the time the job is defined.

Log files and SQL files will overwrite previously existing files. Dump files will never overwrite previously existing files. Instead, an error will be generated.

Default Locations for Dump, Log, and SQL Files

Because Data Pump is server-based, rather than client-based, dump files, log files, and SQL files are accessed relative to server-based directory paths. Data Pump requires you to specify directory paths as directory objects. A directory object maps a name to a directory path on the file system.

For example, the following SQL statement creates a directory object named `dpump_dir1` that is mapped to a directory located at `/usr/apps/datafiles`.

```
SQL> CREATE DIRECTORY dpump_dir1 AS '/usr/apps/datafiles';
```

The reason that a directory object is required is to ensure data security and integrity. For example:

- If you were allowed to specify a directory path location for an input file, you might be able to read data that the server has access to, but to which you should not.
- If you were allowed to specify a directory path location for an output file, the server might overwrite a file that you might not normally have privileges to delete.

Before you can run Data Pump Export or Data Pump Import, a directory object must be created by a database administrator (DBA) or by any user with the `CREATE ANY DIRECTORY` privilege. Then, when you are using Export or Import, you specify the directory object with the `DIRECTORY` parameter.

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 Oracle database to read and write files on behalf of user `hr` in the directory named by `dpump_dir1`, the DBA must execute the following command:

```
SQL> GRANT READ, WRITE ON DIRECTORY dpump_dir1 TO hr;
```

Note that `READ` or `WRITE` permission to a directory object only means that the Oracle database will read or write that file on your behalf. You are not given direct access to those files outside of the Oracle database unless you have the appropriate operating system privileges. Similarly, the Oracle database requires permission from the operating system to read and write files in the directories.

Data Pump Export and Import use the following order of precedence to determine a file's location:

1. If a directory object is specified as part of the file specification, then the location specified by that directory object is used. (The directory object must be separated from the filename by a colon.)
2. If a directory object is not specified for a file, then the directory object named by the DIRECTORY parameter is used.
3. If a directory object is not specified, and if no directory object was named by the DIRECTORY parameter, then the value of the environment variable, DATA_PUMP_DIR, is used. This environment variable is defined using operating system commands on the client system where the Data Pump Export and Import utilities are run. The value assigned to this client-based environment variable must be the name of a server-based directory object, which must first be created on the server system by a DBA. For example, the following SQL statement creates a directory object on the server system. The name of the directory object is DUMP_FILES1, and it is located at '/usr/apps/dumpfiles1'.

   ```
   SQL> CREATE DIRECTORY DUMP_FILES1 AS '/usr/apps/dumpfiles1';
   ```

   Then, a user on a UNIX-based client system using csh can assign the value DUMP_FILES1 to the environment variable DATA_PUMP_DIR. The DIRECTORY parameter can then be omitted from the command line. The dump file employees.dmp, as well as the log file export.log, will be written to '/usr/apps/dumpfiles1'.

   ```
   %setenv DATA_PUMP_DIR DUMP_FILES1
   %expdp hr/hr TABLES=employees DUMPFILE=employees.dmp
   ```

4. If none of the previous three conditions yields a directory object and you are a privileged user, then Data Pump attempts to use the value of the default server-based directory object, DATA_PUMP_DIR. It is important to understand that Data Pump does *not* create the DATA_PUMP_DIR directory object; it merely attempts to use its value when a privileged user has not provided a directory object using any of the mechanisms previously described. This default directory object must first be created by a DBA. Do not confuse this with the client-based environment variable of the same name.

Using Directory Objects When Automatic Storage Management Is Enabled

If you use Data Pump Export or Import with Automatic Storage Management (ASM) enabled, you must define the directory object used for the dump file so that the ASM disk-group name is used (instead of an operating system directory path). A separate directory object, which points to an operating system directory path,

should be used for the log file. For example, you would create a directory object for the ASM dump file as follows:

```
SQL> CREATE or REPLACE DIRECTORY dpump_dir as '+DATAFILES/';
```

Then you would create a separate directory object for the log file:

```
SQL> CREATE or REPLACE DIRECTORY dpump_log as '/homedir/user1/';
```

To enable user hr to have access to these directory objects, you would assign the necessary privileges, for example:

```
SQL> GRANT READ, WRITE ON DIRECTORY dpump_dir TO hr;
SQL> GRANT READ, WRITE ON DIRECTORY dpump_log TO hr;
```

You would then use the following Data Pump Export command:

```
> expdp hr/hr DIRECTORY=dpump_dir DUMPFILE=hr.dmp LOGFILE=dpump_log:hr.log
```

See Also:

- DIRECTORY on page 2-10 for information about using this parameter in Data Pump Export
- DIRECTORY on page 3-10 for information about using this parameter in Data Pump Import
- *Oracle Database SQL Reference* for information about the CREATE DIRECTORY command
- *Oracle Database Administrator's Guide* for more information about Automatic Storage Management (ASM)

Setting Parallelism

For export and import operations, the parallelism setting (specified with the PARALLEL parameter) should be less than or equal to the number of dump files in the dump file set. If there are not enough dump files, the performance will not be optimal because multiple threads of execution will be trying to access the same dump file.

The PARALLEL parameter is valid only in the Enterprise Edition of Oracle Database 10*g*.

Using Substitution Variables

Instead of, or in addition to, listing specific filenames, you can use the DUMPFILE parameter during export operations to specify multiple dump files, by using a substitution variable (%U) in the filename. This is called a dump file template. The new dump files are created as they are needed, beginning with 01 for %U, then using 02, 03, and so on. Enough dump files are created to allow all processes specified by the current setting of the PARALLEL parameter to be active. If one of the dump files becomes full because its size has reached the maximum size specified by the FILESIZE parameter, it is closed, and a new dump file (with a new generated name) is created to take its place.

If multiple dump file templates are provided, they are used to generate dump files in a round-robin fashion. For example, if expa%U, expb%U, and expc%U were all specified for a job having a parallelism of 6, the initial dump files created would be expa01.dmp, expb01.dmp, expc01.dmp, expa02.dmp, expb02.dmp, and expc02.dmp.

For import and SQLFILE operations, if dump file specifications expa%U, expb%U, and expc%U are specified, then the operation will begin by attempting to open the dump files expa01.dmp, expb01.dmp, and expc01.dmp. If the dump file containing the master table is not found in this set, the operation expands its search for dump files by incrementing the substitution variable and looking up the new filenames (for example, expa02.dmp, expb02.dmp, and expc02.dmp). The search continues until the dump file containing the master table is located. If a dump file does not exist, the operation stops incrementing the substitution variable for the dump file specification that was in error. For example, if expb01.dmp and expb02.dmp are found but expb03.dmp is not found, then no more files are searched for using the expb%U specification. Once the master table is found, it is used to determine whether all dump files in the dump file set have been located.

Original Export and Import Versus Data Pump Export and Import

If you are familiar with the original Export (exp) and Import (imp) utilities, it is important to understand that many of the concepts behind them do not apply to Data Pump Export (expdp) and Data Pump Import (impdp). In particular:

- Data Pump Export and Import operate on a group of files called a dump file set rather than on a single sequential dump file.
- Data Pump Export and Import access files on the server rather than on the client. This results in improved performance. It also means that directory objects are required when you specify file locations.

- Data Pump Export and Import use parallel execution rather than a single stream of execution, for improved performance. This means that the order of data within dump file sets is more variable.
- Data Pump Export and Import represent metadata in the dump file set as XML documents rather than as DDL commands. This provides improved flexibility for transforming the metadata at import time.
- Data Pump Export and Import are self-tuning utilities. Tuning parameters that were used in original Export and Import, such as `BUFFER` and `RECORDLENGTH`, are neither required nor supported by Data Pump Export and Import.
- At import time there is no option to perform interim commits during the restoration of a partition. This was provided by the `COMMIT` parameter in original Import.
- There is no option to merge extents when you re-create tables. In original Import, this was provided by the `COMPRESS` parameter. Instead, extents are reallocated according to storage parameters for the target table.
- Sequential media, such as tapes and pipes, are not supported.
- When you are importing data into an existing table using either `APPEND` or `TRUNCATE`, if any row violates an active constraint, the load is discontinued and no data is loaded. This is different from original Import, which logs any rows that are in violation and continues with the load.

  See Also: For a comparison of Data Pump Export and Import parameters to the parameters of original Export and Import, see the following:

  - How Data Pump Export Parameters Map to Those of the Original Export Utility on page 2-35
  - How Data Pump Import Parameters Map to Those of the Original Import Utility on page 3-42

2

Data Pump Export

This chapter describes the Oracle Data Pump Export utility. The following topics are discussed:

What Is Data Pump Export?

> **Note:** Data Pump Export (invoked with the `expdp` command) is a new utility as of Oracle Database 10*g*. Although its functionality and its parameters are similar to those of the original Export utility (`exp`), they are completely separate utilities and their files are not compatible. See Chapter 20, "Original Export and Import" for a description of the original Export utility.

Data Pump Export (hereinafter referred to as Export for ease of reading) is a utility for unloading data and metadata into a set of operating system files called a dump file set. The dump file set can be imported only by the Data Pump Import utility.

The dump file set can be imported on the same system or it can be moved to another system and loaded there.

The dump file set is made up of one or more disk files that contain table data, database object metadata, and control information. The files are written in a proprietary, binary format. During an import operation, the Data Pump Import utility uses these files to locate each database object in the dump file set.

Because the dump files are written by the server, rather than by the client, the data base administrator (DBA) must create directory objects. See Default Locations for Dump, Log, and SQL Files on page 1-13 for more information about directory objects.

Data Pump Export enables you to specify that a job should move a subset of the data and metadata, as determined by the export mode. This is done using data filters and metadata filters, which are specified through Export parameters. See Filtering During Export Operations on page 2-6.

To see some examples of the various ways in which you can use Data Pump Export, refer to Examples of Using Data Pump Export on page 2-43.

Invoking Data Pump Export

The Data Pump Export utility is invoked using the `expdp` command. The characteristics of the export operation are determined by the Export parameters you specify. These parameters can be specified either on the command line or in a parameter file.

Note: Do not invoke Export as `SYSDBA`, except at the request of Oracle technical support. `SYSDBA` is used internally and has specialized functions; its behavior is not the same as for general users.

The following sections contain more information about invoking Export:

- Data Pump Export Interfaces on page 2-3
- Data Pump Export Modes on page 2-3
- Network Considerations on page 2-5

Data Pump Export Interfaces

You can interact with Data Pump Export by using a command line, a parameter file, or an interactive-command mode.

- Command-Line Interface: Enables you to specify most of the Export parameters directly on the command line. For a complete description of the parameters available in the command-line interface, see Parameters Available in Export's Command-Line Mode on page 2-8.
- Parameter File Interface: Enables you to specify command-line parameters in a parameter file. The only exception is the `PARFILE` parameter, because parameter files cannot be nested.
- Interactive-Command Interface: Stops logging to the terminal and displays the Export prompt, from which you can enter various commands, some of which are specific to interactive-command mode. This mode is enabled by pressing Ctrl+C during an export operation started with the command-line interface or the parameter file interface. Interactive-command mode is also enabled when you attach to an executing or stopped job.

  For a complete description of the commands available in interactive-command mode, see Commands Available in Export's Interactive-Command Mode on page 2-37.

Data Pump Export Modes

One of the most significant characteristics of an export operation is its mode, because the mode largely determines what is exported. Export provides different modes for unloading different portions of the database. The mode is specified on the command line, using the appropriate parameter. The available modes are as follows:

- Full Export Mode on page 2-4
- Schema Mode on page 2-4
- Table Mode on page 2-4
- Tablespace Mode on page 2-4
- Transportable Tablespace Mode on page 2-5

See Also: Examples of Using Data Pump Export on page 2-43

Full Export Mode

A full export is specified using the FULL parameter. In a full database export, the entire database is unloaded. This mode requires that you have the EXP_FULL_DATABASE role.

See Also: FULL on page 2-19

Schema Mode

A schema export is specified using the SCHEMAS parameter. This is the default export mode. If you have the EXP_FULL_DATABASE role, then you can specify a list of schemas and optionally include the schema definitions themselves, as well as system privilege grants to those schemas. If you do not have the EXP_FULL_DATABASE role, you can export only your own schema.

Cross-schema references are not exported unless the referenced schema is also specified in the list of schemas to be exported. For example, a trigger defined on a table within one of the specified schemas, but that resides in a schema not explicitly specified, is not exported. This is also true for external type definitions upon which tables in the specified schemas depend. In such a case, it is expected that the type definitions already exist in the target instance at import time.

See Also: SCHEMAS on page 2-29

Table Mode

A table export is specified using the TABLES parameter. In table mode, only a specified set of tables, partitions, and their dependent objects are unloaded. You must have the EXP_FULL_DATABASE role to specify tables that are not in your own schema, and only one schema can be specified. Note that type definitions for columns are *not* exported in table mode. It is expected that the type definitions already exist in the target instance at import time. Also, as in schema exports, cross-schema references are not exported.

See Also: TABLES on page 2-30

Tablespace Mode

A tablespace export is specified using the TABLESPACES parameter. In tablespace mode, only the tables contained in a specified set of tablespaces are unloaded. If a table is unloaded, its dependent objects are also unloaded. Both object metadata and data are unloaded. In tablespace mode, if any part of a table resides in the specified set, then that table and all of its dependent objects are exported. You must have the EXP_FULL_DATABASE role to use tablespace mode.

See Also:

- TABLESPACES on page 2-31
- *Oracle Database Administrator's Guide* for detailed information about transporting tablespaces between databases

Transportable Tablespace Mode

A transportable tablespace export is specified using the TRANSPORT_TABLESPACES parameter. In transportable tablespace mode, only the metadata for the tables (and their dependent objects) within a specified set of tablespaces are unloaded. This allows the tablespace datafiles to then be copied to another Oracle database and incorporated using transportable tablespace import. This mode requires that you have the EXP_FULL_DATABASE role.

Unlike tablespace mode, transportable tablespace mode requires that the specified tables be completely self-contained. That is, the components of all objects in the set must also be in the set.

Transportable tablespace exports cannot be restarted once stopped. Also, they cannot have a degree of parallelism greater than 1.

See Also: TRANSPORT_TABLESPACES on page 2-33

Network Considerations

You can use SQL*Net connection strings or connection descriptors when you invoke the Data Pump Export utility. To do so, the listener must be running (`lsnrctl start`). The following example shows how to invoke Export using a SQL*Net connection:

```
expdp hr/hr@inst1 DIRECTORY=dpump_dir1 DUMPFILE=hr.dmp TABLES=employees
```

The `hr/hr@inst1` results in a SQL*Net connection. The *inst1* refers to a service name specified in the `tnsnames.ora` file. This means that the export client is being run remotely from the server to export the data to a dump file.

Do not confuse invoking the Export utility using a SQL*Net connection string with performing an export operation using the Export NETWORK_LINK command-line parameter.

The NETWORK_LINK parameter initiates a network export. This means that the `expdp` client initiates an export request, typically to the local server. That server contacts the remote database referenced by the database link in the NETWORK_LINK

parameter, retrieves data from it, and writes the data to a dump file set back on the local system.

See Also:

- NETWORK_LINK on page 2-23
- *Oracle Net Services Administrator's Guide*
- *Oracle Database Heterogeneous Connectivity Administrator's Guide*

Filtering During Export Operations

Data Pump Export provides much greater data and metadata filtering capability than was provided by the original Export utility.

Data Filters

Data filters specify restrictions on the rows that are to be exported. These restrictions can be based on partition names and on the results of subqueries.

Each data filter can be specified once per table within a job. If different filters using the same name are applied to both a particular table and to the whole job, the filter parameter supplied for the specific table will take precedence.

Metadata Filters

Metadata filtering is implemented through the `EXCLUDE` and `INCLUDE` parameters. The `EXCLUDE` and `INCLUDE` parameters are mutually exclusive.

Metadata filters identify a set of objects to be included or excluded from an Export or Import operation. For example, you could request a full export, but without Package Specifications or Package Bodies.

To use filters correctly and to get the results you expect, remember that *dependent objects of an identified object are processed along with the identified object*. For example, if a filter specifies that an index is to be included in an operation, then statistics from that index will also be included. Likewise, if a table is excluded by a filter, then indexes, constraints, grants, and triggers upon the table will also be excluded by the filter.

If multiple filters are specified for an object type, an implicit `AND` operation is applied to them. That is, objects pertaining to the job must pass *all* of the filters applied to their object types.

The same metadata filter name can be specified multiple times within a job.

To see which objects can be filtered, you can perform queries on the following views: DATABASE_EXPORT_OBJECTS, SCHEMA_EXPORT_OBJECTS, and TABLE_EXPORT_OBJECTS. For example, you could perform the following query:

```
SQL> SELECT OBJECT_PATH, COMMENTS FROM SCHEMA_EXPORT_OBJECTS
  2  WHERE OBJECT_PATH LIKE '%GRANT';
```

The output of this query looks similar to the following:

```
OBJECT_PATH
--------------------------------------------------------------------------------
COMMENTS
--------------------------------------------------------------------------------
GRANT
Grants on objects in the selected schemas

OBJECT_GRANT
Grants on objects in the selected schemas

PROCDEPOBJ_GRANT
Grants on instance procedural objects in the selected schemas


OBJECT_PATH
--------------------------------------------------------------------------------
COMMENTS
--------------------------------------------------------------------------------
PROCOBJ_GRANT
Schema procedural object grants in the selected schemas

ROLE_GRANT
Role grants to users associated with the selected schemas

SYSTEM_GRANT
System privileges granted to users associated with the selected schemas


6 rows selected.
```

See Also: EXCLUDE on page 2-15 and INCLUDE on page 2-20

Parameters Available in Export's Command-Line Mode

This section provides descriptions of the parameters available in the command-line mode of Data Pump Export. Many of the descriptions include an example of how to use the parameter.

Using the Export Parameter Examples

If you try running the examples that are provided for each parameter, be aware of the following requirements:

- Most of the examples use the sample schemas of the seed database, which is installed by default when you install Oracle Database. In particular, the human resources (`hr`) schema is often used.
- The examples assume that the directory objects, `dpump_dir1` and `dpump_dir2`, already exist and that `READ` and `WRITE` privileges have been granted to the `hr` schema for these directory objects. See Default Locations for Dump, Log, and SQL Files on page 1-13 for information about creating directory objects and assigning privileges to them.
- Some of the examples require the `EXP_FULL_DATABASE` and `IMP_FULL_DATABASE` roles. The examples assume that the `hr` schema has been granted these roles.

If necessary, ask your DBA for help in creating these directory objects and assigning the necessary privileges and roles.

Syntax diagrams of these parameters are provided in Syntax Diagrams for Data Pump Export on page 2-46.

Unless specifically noted, these parameters can also be specified in a parameter file.

Command-Line Escape Characters Used in Examples

Some of the examples in this chapter (such as for the `EXCLUDE` parameter) may show certain clauses enclosed in quotation marks and backslashes. This is because those clauses contain a blank, a situation for which most operating systems require that the entire 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, such as the backslash. If the backslashes were not present, the command-line parser that Export uses would not understand the quotation marks and would remove them. Some examples in this chapter may use escape characters to show you how they would be used. However, in general, Oracle recommends that you place such statements in a parameter file because escape characters are not necessary in parameter files.

See Also:

- Default Locations for Dump, Log, and SQL Files on page 1-13 for information about creating default directory objects
- Examples of Using Data Pump Export on page 2-43
- *Oracle Database Sample Schemas*

Note: If you are accustomed to using the original Export utility (exp), you may be wondering which Data Pump parameters are used to perform the operations you used to perform with original Export. For a comparison, see How Data Pump Export Parameters Map to Those of the Original Export Utility on page 2-35.

ATTACH

Default: job currently in the user's schema, if there is only one

Purpose

Attaches the client session to an existing export job and automatically places you in the interactive-command interface. Export displays a description of the job to which you are attached and also displays the Export prompt.

Syntax and Description

```
ATTACH [=[schema_name.]job_name]
```

The *schema_name* is optional. To specify a schema other than your own, you must have the EXP_FULL_DATABASE role.

The *job_name* is optional if only one export job is associated with your schema and the job is active. To attach to a stopped job, you must supply the job name. To see a list of Data Pump job names, you can query the DBA_DATAPUMP_JOBS view or the USER_DATAPUMP_JOBS view.

When you are attached to the job, Export displays a description of the job and then displays the Export prompt.

Restrictions

When you specify the ATTACH parameter, you cannot specify any other parameters except for the connection string (user/password).

You cannot attach to a job in another schema unless it is already running.

Example

The following is an example of using the ATTACH parameter. It assumes that the job, hr.export_job, already exists.

```
> expdp hr/hr ATTACH=hr.export_job
```

> **See Also:** Commands Available in Export's Interactive-Command Mode on page 2-37

CONTENT

Default: ALL

Purpose

Enables you to filter what Export unloads: data only, metadata only, or both.

Syntax and Description

```
CONTENT={ALL | DATA_ONLY | METADATA_ONLY}
```

- ALL unloads both data and metadata. This is the default.
- DATA_ONLY unloads only table row data; no database object definitions are unloaded.
- METADATA_ONLY unloads only database object definitions; no table row data is unloaded.

Example

The following is an example of using the CONTENT parameter:

```
> expdp hr/hr DIRECTORY=dpump_dir1 DUMPFILE=hr.dmp CONTENT=METADATA_ONLY
```

This command will execute a schema-mode export that will unload only the metadata associated with the hr schema. It defaults to a schema-mode export of the hr schema because no export mode is specified.

DIRECTORY

Default: none for nonprivileged users; DATA_PUMP_DIR for privileged users

Purpose

Specifies the location to which Export can write the dump file set and the log file.

Syntax and Description

```
DIRECTORY=directory_object
```

The *directory_object* is the name of a database directory object (*not the name of an actual directory*) that was previously created by the database administrator (DBA) using the SQL CREATE DIRECTORY command.

A directory object specified on the DUMPFILE or LOGFILE parameter overrides any directory object that you specify for the DIRECTORY parameter.

Example 1

The following is an example of using the DIRECTORY parameter:

```
> expdp hr/hr DIRECTORY=dpump_dir1 DUMPFILE=employees.dmp CONTENT=METADATA_ONLY
```

The dump file, employees.dmp, will be written to the path that is associated with the directory object dpump_dir1.

Example 2

The following is an example of using the default DATA_PUMP_DIR directory object available to privileged users. This example assumes that the EXP_FULL_DATABASE and IMP_FULL_DATABASE roles have been granted to the user hr. After the DATA_PUMP_DIR directory object has been created, a privileged user need not use the DIRECTORY parameter. Dump files, log files, and SQL files will be written to the path associated with DATA_PUMP_DIR.

```
SQL> CREATE DIRECTORY data_pump_dir AS '/usr/dba/dpumpfiles':

> expdp hr/hr DUMPFILE=emp.dmp LOGFILE=emp.log TABLES=hr.employees
```

The emp.dmp and emp.log files will be written to /usr/dba/dpumpfiles.

If the DATA_PUMP_DIR directory object had not first been created by a DBA, then the following error messages would have been displayed:

```
ORA-39002: invalid operation
ORA-39070: Unable to open the log file.
ORA-39087: directory name DATA_PUMP_DIR is invalid
```

Remember that the default DATA_PUMP_DIR directory object is not available to nonprivileged users. In the following example, user sh is a nonprivileged user. Therefore, because no directory object is specified, error messages are generated and the export is not performed.

```
> expdp sh/sh DUMPFILE=sales.dmp LOGFILE=sales.log TABLES=sh.sales

ORA-39002: invalid operation
ORA-39070: Unable to open the log file.
ORA-39145: directory object parameter must be specified and non-null
```

See Also:

- Default Locations for Dump, Log, and SQL Files on page 1-13 for more information about default directory objects
- *Oracle Database SQL Reference* for information about the CREATE DIRECTORY command

DUMPFILE

Default: expdat.dmp

Purpose

Specifies the names, and optionally, the directory objects of dump files for an export job.

Syntax and Description

```
DUMPFILE=[directory_object:]file_name [, ...]
```

The *directory_object* is optional if one has already been established by the DIRECTORY parameter. If you supply a value here, it must be a directory object that already exists. A database directory object that is specified as part of the DUMPFILE parameter overrides a value specified by the DIRECTORY parameter.

You can supply multiple *file_name* specifications as a comma-delimited list or in separate DUMPFILE parameter specifications. If no extension is given for the filename, then Export uses the default file extension of .dmp. The filenames can contain a substitution variable (%U), which implies that multiple files may be generated. The substitution variable is expanded in the resulting filenames into a 2-digit, fixed-width, incrementing integer starting at 01 and ending at 99. If a file specification contains two substitution variables, both are incremented at the same

time. For example, exp%Uaa%U.dmp would resolve to exp01aa01.dmp, exp02aa02.dmp, and so forth.

If the FILESIZE parameter is specified, each dump file will have a maximum of that size in bytes and be nonextensible. If more space is required for the dump file set and a template with a substitution variable (%U) was supplied, a new dump file is automatically created of the size specified by FILESIZE, if there is room on the device.

If templates with a substitution variable (%U) were specified along with the PARALLEL parameter, then one file for each template is initially created. More files are created from the templates as they are needed based on how much data is being exported and how many parallel processes are given work to perform during the job.

As each file specification or file template containing a substitution variable is defined, it is instantiated into one fully qualified filename and Export attempts to create it. The file specifications are processed in the order in which they are specified. If the job needs extra files because the maximum file size is reached, or to keep parallel workers active, then additional files are created if file templates with substitution variables were specified.

Restrictions

If there are preexisting files that match the resulting filenames, an error is generated. The existing dump files will not be overwritten.

Example

The following is an example of using the DUMPFILE parameter:

```
> expdp hr/hr SCHEMAS=hr DIRECTORY=dpump_dir1 DUMPFILE=dpump_dir2:exp1.dmp,
 exp2%U.dmp PARALLEL=3
```

The dump file, exp1.dmp, will be written to the path associated with the directory object dpump_dir2 because dpump_dir2 was specified as part of the dump file name, and therefore overrides the directory object specified with the DIRECTORY parameter. Because all three parallel processes will be given work to perform during this job, the exp201.dmp and exp202.dmp dump files will be created and they will be written to the path associated with the directory object, dpump_dir1, that was specified with the DIRECTORY parameter.

See Also:

- File Allocation on page 1-12

ESTIMATE

Default: BLOCKS

Purpose

Specifies the method that Export will use to estimate how much disk space each table in the export job will consume (in bytes). The estimate is printed in the log file and displayed on the client's standard output device. The estimate is for table row data only; it does not include metadata.

Syntax and Description

```
ESTIMATE={BLOCKS | STATISTICS}
```

- BLOCKS - The estimate is calculated by multiplying the number of database blocks used by the target objects times the appropriate block sizes.
- STATISTICS - The estimate is calculated using statistics for each table. For this method to be as accurate as possible, all tables should have been analyzed recently.

Example

The following example shows a use of the ESTIMATE parameter in which the estimate is calculated using statistics for the employees table:

```
> expdp hr/hr TABLES=employees ESTIMATE=STATISTICS DIRECTORY=dpump_dir1
 DUMPFILE=estimate_stat.dmp
```

ESTIMATE_ONLY

Default: n

Purpose

Instructs Export to estimate the space that a job would consume, without actually performing the export operation.

Syntax and Description

```
ESTIMATE_ONLY={y | n}
```

If ESTIMATE_ONLY=y, then Export estimates the space that would be consumed, but quits without actually performing the export operation.

Example

The following shows an example of using the ESTIMATE_ONLY parameter to determine how much space an export of the HR schema will take.

```
> expdp hr/hr ESTIMATE_ONLY=y NOLOGFILE=y
```

EXCLUDE

Default: none

Purpose

Enables you to filter the metadata that is exported by specifying objects and object types that you want excluded from the export operation.

Syntax and Description

```
EXCLUDE=object_type[:name_clause] [, ...]
```

All object types for the given mode of export will be included except those specified in an EXCLUDE statement. If an object is excluded, all of its dependent objects are also excluded. For example, excluding a table will also exclude all indexes and triggers on the table.

The *name_clause* is optional. It allows selection of specific objects within an object type. It is a SQL expression used as a filter on the type's object names. It consists of a SQL operator and the values against which the object names of the specified type are to be compared. The name clause applies only to object types whose instances have names (for example, it is applicable to TABLE, but not to GRANT). The name clause must be separated from the object type with a colon and enclosed in double quotation marks, because single-quotation marks are required to delimit the name strings. For example, you could set EXCLUDE=INDEX:"LIKE 'EMP%'" to exclude all indexes whose names start with emp. This is shown in the following example:

```
> expdp hr/hr EXCLUDE=INDEX:\"LIKE \'EMP%\'\" DUMPFILE=dpump_dir1:exp.dmp
NOLOGFILE=y
```

If no *name_clause* is provided, all objects of the specified type are excluded.

More than one EXCLUDE statement can be specified.

Oracle recommends that you place EXCLUDE clauses in a parameter file to avoid having to use escape characters on the command line. This example shows how the use of the backslash escape character would be necessary if you specified this EXCLUDE clause on the command line, rather than in a parameter file.

See Also:

- INCLUDE on page 2-20 for an example of using a parameter file
- Command-Line Escape Characters Used in Examples on page 2-8

If the *object_type* you specify is CONSTRAINT, GRANT, or USER, you should be aware of the effects this will have, as described in the following paragraphs.

Excluding Constraints

The following constraints cannot be excluded:

- NOT NULL constraints
- Constraints needed for the table to be created and loaded successfully; for example, primary key constraints for index-organized tables, or REF SCOPE and WITH ROWID constraints for tables with REF columns

This means that the following EXCLUDE statements will be interpreted as follows:

- EXCLUDE=CONSTRAINT will exclude all (nonreferential) constraints, except for NOT NULL constraints and any constraints needed for successful table creation and loading.
- EXCLUDE=REF_CONSTRAINT will exclude referential integrity (foreign key) constraints.

Excluding Grants and Users

Specifying EXCLUDE=GRANT excludes object grants on all object types and system privilege grants.

Specifying EXCLUDE=USER excludes only the definitions of users, not the objects contained within users' schemas.

To exclude a specific user and all objects of that user, specify a filter such as the following (where hr is the schema name of the user you want to exclude):

```
EXCLUDE=SCHEMA:"='HR'"
```

If you try to exclude a user by using a statement such as EXCLUDE=USER:"='HR'", then only CREATE USER hr DDL statements will be excluded, and you may not get the results you expect.

Restrictions

The EXCLUDE and INCLUDE parameters are mutually exclusive.

Neither EXCLUDE nor INCLUDE can be used if the CONTENT=DATA_ONLY parameter is specified, because that implies export of table row data only.

Example

The following is an example of using the EXCLUDE statement.

```
> expdp hr/hr DIRECTORY=dpump_dir1 DUMPFILE=hr_exclude.dmp EXCLUDE=VIEW,
PACKAGE, FUNCTION
```

This will result in a schema-mode export in which all of the hr schema will be exported except its views, packages, and functions.

See Also:

- Filtering During Export Operations on page 2-6 for more information about the effects of using the EXCLUDE parameter

FILESIZE

Default: 0 (unlimited)

Purpose

Specifies the maximum size of each export dump file. If the size is reached for any member of the dump file set, that file is closed and an attempt is made to create a new file, if the file specification contains a substitution variable. If there is insufficient space on the device to write a file of the specified size, the export operation will stop. It can be restarted after the situation is corrected.

Syntax and Description

```
FILESIZE=integer[B | K | M | G]
```

The *integer* can be followed by B, K, M, or G (indicating bytes, kilobytes, megabytes, and gigabytes respectively). Bytes is the default. The actual size of the resulting file may be rounded down slightly to match the size of the internal blocks used in dump files.

Restrictions

The minimum size for a file is ten times the default Data Pump block size, which is 4 kilobytes.

Example

The following shows an example in which the size of the dump file is set to 3 megabytes:

```
> expdp hr/hr DIRECTORY=dpump_dir1 DUMPFILE=hr_3m.dmp FILESIZE=3M
```

FLASHBACK_SCN

Default: none

Purpose

Specifies the system change number (SCN) that Export will use to enable the Flashback utility. The export operation is performed with data that is consistent as of this SCN. If the NETWORK_LINK parameter is specified, the SCN refers to the SCN of the source database.

Syntax and Description

```
FLASHBACK_SCN=scn_value
```

Restrictions

FLASHBACK_SCN and FLASHBACK_TIME are mutually exclusive.

Example

The following example assumes that an existing SCN value of 384632 exists. It exports the hr schema up to SCN 384632.

```
> expdp hr/hr DIRECTORY=dpump_dir1 DUMPFILE=hr_scn.dmp FLASHBACK_SCN=384632
```

Note: If you are on a logical standby system, the FLASHBACK_SCN parameter is ignored because SCNs are selected by logical standby. See *Oracle Data Guard Concepts and Administration* for information about logical standby databases.

FLASHBACK_TIME

Default: none

Purpose

The SCN that most closely matches the specified time is found, and this SCN is used to enable the Flashback utility. The export operation is performed with data that is consistent as of this SCN.

Syntax and Description

```
FLASHBACK_TIME="TO_TIMESTAMP(time-value)"
```

Restrictions

FLASHBACK_TIME and FLASHBACK_SCN are mutually exclusive.

Example

You can specify the time in any format that the DBMS_FLASHBACK.ENABLE_AT_TIME procedure accepts, for example:

```
> expdp hr/hr DIRECTORY=dpump_dir1 DUMPFILE=hr_time.dmp
FLASHBACK_TIME="TO_TIMESTAMP('25-08-2003 14:35:00', 'DD-MM-YYYY HH24:MI:SS')"
```

This example may require the use of escape characters, depending on your operating system. See Command-Line Escape Characters Used in Examples on page 2-8.

> **See Also:** *Oracle Database Application Developer's Guide - Fundamentals* for information about using flashback

FULL

Default: n

Purpose

Specifies that you want to perform a full database mode export.

Syntax and Description

```
FULL={y | n}
```

If FULL=y, all data and metadata are exported.

To perform a full export, you must have the EXP_FULL_DATABASE role.

Restrictions

The following system schemas are not exported as part of a Full export because the metadata they contain is exported as part of other objects in the dump file set: SYS, ORDSYS, ORDPLUGINS, CTXSYS, MDSYS, LBACSYS, and XDB. Also excluded are any tables that are registered in the SYS.KU_NOEXP_TAB dictionary table.

Example

The following is an example of using the FULL parameter. The dump file, expfull.dmp is written to the dpump_dir2 directory.

```
> expdp hr/hr DIRECTORY=dpump_dir2 DUMPFILE=expfull.dmp FULL=y NOLOGFILE=y
```

HELP

Default: N

Purpose

Displays online help for the Export utility.

Syntax and Description

```
HELP = {y | n}
```

If HELP=y is specified, Export displays a summary of all Export command-line parameters and interactive commands.

Example

```
> expdp HELP = y
```

This example will display a brief description of all Export parameters and commands.

INCLUDE

Default: none

Purpose

Enables you to filter the metadata that is exported by specifying objects and object types for the current export mode. The specified objects and all their dependent objects are exported. Grants on these objects are also exported.

Syntax and Description

```
INCLUDE = object_type[:name_clause] [, ...]
```

Only object types explicitly specified in INCLUDE statements are exported. No other object types, including the schema definition information that is normally part of a schema-mode export when you have the EXP_FULL_DATABASE role, are exported.

To see a list of valid object type path names for use with the INCLUDE parameter, you can query the following views: DATABASE_EXPORT_OBJECTS, SCHEMA_EXPORT_OBJECTS, and TABLE_EXPORT_OBJECTS.

The *name_clause* is optional. It allows fine-grained selection of specific objects within an object type. It is a SQL expression used as a filter on the object names of the type. It consists of a SQL operator and the values against which the object names of the specified type are to be compared. The name clause applies only to object types whose instances have names (for example, it is applicable to TABLE, but not to GRANT). The optional name clause must be separated from the object type with a colon and enclosed in double quotation marks, because single-quotation marks are required to delimit the name strings.

Oracle recommends that INCLUDE statements be placed in a parameter file so that you can avoid having to use operating system-specific escape characters on the command line. For example, suppose you have a parameter file named hr.par with the following content:

```
SCHEMAS=HR
DUMPFILE=expinclude.dmp
DIRECTORY=dpump_dir1
LOGFILE=expinclude.log
INCLUDE=TABLE:"IN ('EMPLOYEES', 'DEPARTMENTS')"
INCLUDE=PROCEDURE
INCLUDE=INDEX:"LIKE 'EMP%'"
```

You could then use the hr.par file to start an export operation, without having to enter any other parameters on the command line:

```
> expdp hr/hr parfile=hr.par
```

Restrictions

The INCLUDE and EXCLUDE parameters are mutually exclusive. Additionally, neither one can be used if the CONTENT=DATA_ONLY parameter is specified, because that implies export of table rows only.

Example

The following example performs an export of all tables (and their dependent objects) in the hr schema:

```
> expdp hr/hr INCLUDE=TABLE DUMPFILE=dpump_dir1:exp_inc.dmp NOLOGFILE=y
```

JOB_NAME

Default: system-generated name of the form SYS_<operation>_<mode>_NN

Purpose

Specifies a name for the export job. The job name is used to identify the export job in subsequent actions, such as when the ATTACH parameter is used to attach to a job. The job name becomes the name of the master table in the current user's schema. The master table is used to control the export job.

Syntax and Description

```
JOB_NAME=jobname_string
```

The *jobname_string* specifies a name of up to 30 bytes for this export job. The bytes must represent printable characters and spaces. If spaces are included, the name must be enclosed in single quotation marks (for example, 'Thursday Export'). The job name is implicitly qualified by the schema of the user performing the export operation.

The default job name is system-generated in the form SYS_<operation>_<mode>_NN, where NN expands to a 2-digit incrementing integer starting at 01. An example of a default name is 'SYS_EXPORT_TABLESPACE_02'.

Example

The following example shows an export operation that is assigned a job name of exp_job:

```
> expdp hr/hr DIRECTORY=dpump_dir1 DUMPFILE=exp_job.dmp JOB_NAME='exp_job'
NOLOGFILE=y
```

LOGFILE

Default: export.log

Purpose

Specifies the name, and optionally, a directory, for the log file of the export job.

Syntax and Description

```
LOGFILE=[directory_object:]file_name
```

You can specify a database *directory_object* previously established by the DBA. This overrides the directory object specified with the DIRECTORY parameter.

The *file_name* specifies a name for the log file. The default behavior is to create a file named export.log in the directory referenced by the directory object specified in the DIRECTORY parameter.

All messages regarding work in progress, work completed, and errors encountered are written to the log file. (For a real-time status of the job, use the STATUS command in interactive mode.)

A log file is always created for an export job unless the NOLOGFILE parameter is specified. As with the dump file set, the log file is relative to the server and not the client.

An existing file matching the filename will be overwritten.

Example

The following example shows how to specify a log file name if you do not want to use the default:

```
> expdp hr/hr DIRECTORY=dpump_dir1 DUMPFILE=hr.dmp LOGFILE=hr_export.log
```

Note: Data Pump Export writes the log file using the database character set. If your client NLS_LANG environment setting sets up a different client character set from the database character set, then it is possible that table names may be different in the log file than they are when displayed on the client output screen.

See Also: STATUS on page 2-41

NETWORK_LINK

Default: none

Purpose

Enables a network export when you specify the name of a valid database link. A network export moves data from a remote database to a dump file set local to the instance running the Data Pump job.

Syntax and Description

NETWORK_LINK=*source_database_link*

The NETWORK_LINK parameter initiates a network export. This means that the expdp client initiates an export request, typically to the local server. That server contacts the remote database referenced by the *source_database_link*, retrieves data from it, and writes the data to a dump file set back on the local system.

The *source_database_link* provided must be the name of a database link to a source system. If the database does not already have a database link, you or your DBA must create one. The following information is required: host machine name, and port number and SID for the database. For more information about the CREATE DATABASE LINK statement, see *Oracle Database SQL Reference*.

If the source database is read-only, then the user on the source database must have a locally managed tablespace assigned as the default temporary tablespace. Otherwise, the job will fail. For further details about this, see the information about creating locally managed temporary tablespaces in the *Oracle Database Administrator's Guide*.

Restrictions

Tables with columns that are object types are not supported in a network export. An ORA-22804 error will be generated and the export will move on to the next table. To work around this restriction, you can manually create the dependent object types within the database from which the export is being run.

Example

The following is an example of using the NETWORK_LINK parameter. The *source_database_link* would be replaced with the name of a valid database link that must already exist.

```
> expdp hr/hr DIRECTORY=dpump_dir1 NETWORK_LINK=source_database_link
  DUMPFILE=network_export.dmp LOGFILE=network_export.log
```

NOLOGFILE

Default: n

Purpose

Specifies whether to suppress creation of a log file.

Syntax and Description

```
NOLOGFILE={y | n}
```

Specify NOLOGFILE=y to suppress the default behavior of creating a log file. Progress and error information is still written to the standard output device of any attached clients, including the client that started the original export operation. If there are no clients attached to a running job and you specify NOLOGFILE=y, you run the risk of losing important progress and error information.

Example

The following is an example of using the NOLOGFILE parameter:

```
> expdp hr/hr DIRECTORY=dpump_dir1 DUMPFILE=hr.dmp NOLOGFILE=y
```

This command results in a schema-mode export in which no log file is written.

PARALLEL

Default: 1

Purpose

Specifies the maximum number of threads of active execution operating on behalf of the export job. This execution set consists of a combination of worker processes and parallel I/O server processes. The master control process and worker processes acting as query coordinators in parallel query operations do not count toward this total.

This parameter enables you to make trade-offs between resource consumption and elapsed time.

Syntax and Description

```
PARALLEL=integer
```

The value you specify for *integer* should match the number of files in the dump file set (or you should specify substitution variables in the dump file specifications). Because each active worker process or I/O server process writes exclusively to one file at a time, an insufficient number of files can have adverse effects. Some of the worker processes will be idle while waiting for files, thereby degrading the overall

performance of the job. More importantly, if any member of a cooperating group of parallel I/O server processes becomes idle, then the export operation will be stopped with an ORA-39095 error. Both situations can be corrected by attaching to the job using the Data Pump Export utility, adding more files using the ADD_FILE command while in interactive mode, and in the case of a stopped job, restarting the job.

To increase or decrease the value of PARALLEL during job execution, use interactive-command mode. Decreasing parallelism does not result in fewer worker processes associated with the job; it merely decreases the number of worker processes that will be executing at any given time. Also, any ongoing work must reach an orderly completion point before the decrease takes effect. Therefore, it may take a while to see any effect from decreasing the value. Idle workers are not deleted until the job exits.

Increasing the parallelism takes effect immediately if there is work that can be performed in parallel.

Restrictions

This parameter is valid only in the Enterprise Edition of Oracle Database 10*g*.

Example

The following is an example of using the PARALLEL parameter:

```
> expdp hr/hr DIRECTORY=dpump_dir1 LOGFILE=parallel_export.log
JOB_NAME=par3_job DUMPFILE=par_exp%u.dmp PARALLEL=3
```

This results in a schema-mode export of the hr schema in which up to three files could be created in the path pointed to by the directory object, dpump_dir1. As a result of the command in this example, the following dump files are created:

- par_exp01.dmp
- par_exp02.dmp
- par_exp03.dmp

See Also:

- DUMPFILE on page 2-12
- Commands Available in Export's Interactive-Command Mode on page 2-37
- Performing a Parallel Full Database Export on page 2-45

PARFILE

Default: none

Purpose

Specifies the name of an export parameter file.

Syntax and Description

PARFILE=[*directory_path*]*file_name*

Unlike dump and log files, which are created and written by the Oracle database, the parameter file is opened and read by the client running the expdp image. Therefore, a directory object name is neither required nor appropriate. The directory path is an operating system-specific directory specification. The default is the user's current directory.

Restrictions

The PARFILE parameter cannot be specified within a parameter file.

Example

The content of an example parameter file, hr.par, might be as follows:

```
SCHEMAS=HR
DUMPFILE=exp.dmp
DIRECTORY=dpump_dir1
LOGFILE=exp.log
```

You could then issue the following Export command to specify the parameter file:

```
> expdp hr/hr parfile=hr.par
```

QUERY

Default: none

Purpose

Enables you to filter the data that is exported by specifying a clause for a SQL SELECT statement, which is applied to all tables in the export job or to a specific table.

Syntax and Description

```
QUERY = [schema.][table_name:] query_clause
```

The *query_clause* is typically a WHERE clause for fine-grained row selection, but could be any SQL clause. For example, an ORDER BY clause could be used to speed up a migration from a heap-organized table to an index-organized table. If a [*schema*.]*table_name* is not supplied, the query is applied to (and must be valid for) all tables in the export job. A table-specific query overrides a query applied to all tables.

When the query is to be applied to a specific table, a colon must separate the table name from the query clause. More than one table-specific query can be specified, but only one can be specified per table. Oracle highly recommends that you place QUERY specifications in a parameter file so that you can avoid having to use operating system-specific escape characters on the command line.

The query must be enclosed in single or double quotation marks.

When the QUERY parameter is used, the external tables method (rather than the direct path method) is used for data access.

To specify a schema other than your own in a table-specific query, you need the EXP_FULL_DATABASE role.

Restrictions

The QUERY parameter cannot be used in conjunction with the following parameters:

- CONTENT=METADATA_ONLY
- ESTIMATE_ONLY
- TRANSPORT_TABLESPACES

Example

The following is an example of using the QUERY parameter:

```
> expdp hr/hr QUERY=employees:'"WHERE department_id > 10 AND salary > 10000"'
NOLOGFILE=y DIRECTORY=dpump_dir1 DUMPFILE=exp1.dmp
```

This example unloads all tables in the hr schema, but only the rows that fit the query expression. In this case, all rows in all tables (except employees) in the hr schema will be unloaded. For the employees table, only rows that meet the query criteria are unloaded.

SCHEMAS

Default: current user's schema

Purpose

Specifies that you want to perform a schema-mode export. This is the default mode for Export.

Syntax and Description

```
SCHEMAS=schema_name [, ...]
```

If you have the EXP_FULL_DATABASE role, then you can specify a single schema other than your own or a list of schema names. The EXP_FULL_DATABASE role also allows you to export additional nonschema object information for each specified schema so that the schemas can be re-created at import time. This additional information includes the user definitions themselves and all associated system and role grants, user password history, and so on.

If no mode is specified, all objects in the current user's schema are exported.

Restrictions

If you do not have the EXP_FULL_DATABASE role, then you can specify only your own schema.

Any tables that are registered in the SYS.KU_NOEXP_TAB dictionary table are excluded in schema mode.

Example

The following is an example of using the SCHEMAS parameter. Note that user hr is allowed to specify more than one schema because the EXP_FULL_DATABASE role was previously assigned to it for the purpose of these examples.

```
> expdp hr/hr DIRECTORY=dpump_dir1 DUMPFILE=expdat.dmp SCHEMAS=hr,sh,oe
```

This results in a schema-mode export in which the schemas, hr, sh, and oe will be written to the expdat.dmp dump file located in the dpump_dir1 directory.

STATUS

Default: 0

Purpose

Displays detailed status of the job, along with a description of the current operation. An estimated completion percentage for the job is also returned

Syntax and Description

```
STATUS=[integer]
```

If you supply a value for *integer*, it specifies how frequently, in seconds, job status should be displayed in logging mode. If no value is entered or if the default value of 0 is used, information is displayed only upon completion of each object type, table, or partition.

This status information is written only to your standard output device, not to the log file (if one is in effect).

Example

The following is an example of using the STATUS parameter.

```
> expdp hr/hr DIRECTORY=dpump_dir1 SCHEMAS=hr,sh STATUS=300
```

This example will export the hr and sh schemas and display the status of the export every 5 minutes (60 seconds x 5 = 300 seconds).

TABLES

Default: none

Purpose

Specifies that you want to perform a table-mode export.

Syntax and Description

```
TABLES=[schema_name.]table_name[:partition_name] [, ...]
```

You can filter the data and metadata that is exported, by specifying a comma-delimited list of tables and partitions or subpartitions. If a partition name is specified, it must be the name of a partition or subpartition in the associated table. Only the specified set of tables, partitions, and their dependent objects are unloaded.

The table name that you specify can be preceded by a qualifying schema name. All table names specified must reside in the same schema. The schema defaults to that

of the current user. To specify a schema other than your own, you must have the EXP_FULL_DATABASE role.

The use of wildcards is supported for one table name per export operation. For example, TABLES=emp% would export all tables having names that start with 'EMP.'

Restrictions

Cross-schema references are not exported. For example, a trigger defined on a table within one of the specified schemas, but that resides in a schema not explicitly specified, is not exported.

Types used by the table are *not* exported in table mode. This means that if you subsequently import the dump file and the TYPE does not already exist in the destination database, the table creation will fail.

The use of synonyms as values for the TABLES parameter is not supported. For example, if the regions table in the hr schema had a synonym of regn, it would not be valid to use TABLES=regn. An error would be returned.

The export of partitions is not supported over network links.

The export of tables that include wildcards in the table name is not supported if the table has partitions.

Examples

The following example shows a simple use of the TABLES parameter to export three tables found in the hr schema: employees, jobs, and departments. Because user hr is exporting tables found in the hr schema, the schema name is not needed before the table names.

```
> expdp hr/hr DIRECTORY=dpump_dir1 DUMPFILE=tables.dmp
TABLES=employees,jobs,departments
```

The following example shows the use of the TABLES parameter to export partitions:

```
> expdp hr/hr DIRECTORY=dpump_dir1 DUMPFILE=tables_part.dmp
TABLES=sh.sales:sales_Q1_2000,sh.sales:sales_Q2_2000
```

This example exports the partitions, sales_Q1_2000 and sales_Q2_2000, from the table sales in the schema sh.

TABLESPACES

Default: none

Purpose

Specifies a list of tablespace names to be exported in tablespace mode.

Syntax and Description

```
TABLESPACES=tablespace_name [, ...]
```

In tablespace mode, only the tables contained in a specified set of tablespaces are unloaded. If a table is unloaded, its dependent objects are also unloaded. If any part of a table resides in the specified set, then that table and all of its dependent objects are exported.

Example

The following is an example of using the TABLESPACES parameter. The example assumes that tablespaces tbs_4, tbs_5, and tbs_6 already exist.

```
> expdp hr/hr DIRECTORY=dpump_dir1 DUMPFILE=tbs.dmp
TABLESPACES=tbs_4, tbs_5, tbs_6
```

This results in a tablespace export in which objects (and their dependent objects) from the specified tablespaces (tbs_4, tbs_5, and tbs_6) will be unloaded.

TRANSPORT_FULL_CHECK

Default: n

Purpose

Specifies whether or not to check for dependencies between those objects inside the transportable set and those outside the transportable set. This parameter is applicable only to a transportable-tablespace mode export.

Syntax and Description

```
TRANSPORT_FULL_CHECK={y | n}
```

If TRANSPORT_FULL_CHECK=y, then Export verifies that there are no dependencies between those objects inside the transportable set and those outside the transportable set. The check addresses two-way dependencies. For example, if a table is inside the transportable set but its index is not, a failure is returned and the export operation is terminated. Similarly, a failure is also returned if an index is in the transportable set but the table is not.

If TRANSPORT_FULL_CHECK=n, then Export verifies only that there are no objects within the transportable set that are dependent on objects outside the transportable set. This check addresses a one-way dependency. For example, a table is not dependent on an index, but an index *is* dependent on a table, because an index without a table has no meaning. Therefore, if the transportable set contains a table, but not its index, then this check succeeds. However, if the transportable set contains an index, but not the table, the export operation is terminated.

In addition to this check, Export always verifies that all storage segments of all tables (and their indexes) defined within the tablespace set specified by TRANSPORT_TABLESPACES are actually contained within the tablespace set.

Example

The following is an example of using the TRANSPORT_FULL_CHECK parameter. It assumes that tablespace tbs_1 exists.

```
> expdp hr/hr DIRECTORY=dpump_dir1 DUMPFILE=tts.dmp
TRANSPORT_TABLESPACES=tbs_1 TRANSPORT_FULL_CHECK=y LOGFILE=tts.log
```

TRANSPORT_TABLESPACES

Default: none

Purpose

Specifies that you want to perform a transportable-tablespace-mode export.

Syntax and Description

```
TRANSPORT_TABLESPACES=tablespace_name [, ...]
```

Use the TRANSPORT_TABLESPACES parameter to specify a list of tablespace names for which object metadata will be exported from the source database into the target database.

Restrictions

Transportable jobs are not restartable.

Transportable jobs are restricted to a degree of parallelism of 1.

Transportable tablespace mode requires that you have the EXP_FULL_DATABASE role.

Example

The following is an example of using the TRANSPORT_TABLESPACES parameter. It assumes that tablespace `tbs_1` exists.

```
> expdp hr/hr DIRECTORY=dpump_dir1 DUMPFILE=tts.dmp
TRANSPORT_TABLESPACES=tbs_1 TRANSPORT_FULL_CHECK=y LOGFILE=tts.log
```

See Also:

- Transportable Tablespace Mode on page 2-5
- *Oracle Database Administrator's Guide* for detailed information about transporting tablespaces between databases

VERSION

Default: COMPATIBLE

Purpose

Specifies the version of database objects to be exported. This can be used to create a dump file set that is compatible with a previous release of Oracle Database.

Syntax and Description

```
VERSION={COMPATIBLE | LATEST | version_string}
```

The legal values for the VERSION parameter are as follows:

- COMPATIBLE - This is the default value. The version of the metadata corresponds to the database compatibility level. Database compatibility must be set to 9.2 or higher.
- LATEST - The version of the metadata corresponds to the database version.
- *version_string* - A specific database version (for example, 10.0.0). In Oracle Database 10*g*, this value cannot be lower than 9.2.

Database objects or attributes that are incompatible with the specified version will not be exported. For example, tables containing new datatypes that are not supported in the specified version will not be exported.

Example

The following example shows an export for which the version of the metadata will correspond to the database version:

```
> expdp hr/hr TABLES=hr.employees VERSION=LATEST DIRECTORY=dpump_dir1
DUMPFILE=emp.dmp NOLOGFILE=y
```

How Data Pump Export Parameters Map to Those of the Original Export Utility

Table 2–1 maps, as closely as possible, Data Pump Export parameters to original Export parameters. In some cases, because of feature redesign, the original Export parameter is no longer needed, so there is no Data Pump parameter to compare it to. Also, as shown in the table, some of the parameter names may be the same, but the functionality is slightly different.

Table 2–1 Original Export Parameters and Their Counterparts in Data Pump Export

Original Export Parameter	Comparable Data Pump Export Parameter
BUFFER	A parameter comparable to BUFFER is not needed.
COMPRESS	A parameter comparable to COMPRESS is not needed.
CONSISTENT	A parameter comparable to CONSISTENT is not needed. Use FLASHBACK_SCN and FLASHBACK_TIME for this functionality.
CONSTRAINTS	EXCLUDE=CONSTRAINT and INCLUDE=CONSTRAINT
DIRECT	A parameter comparable to DIRECT is not needed. Data Pump Export automatically chooses the best method (direct path mode or external tables mode).
FEEDBACK	STATUS
FILE	DUMPFILE
FILESIZE	FILESIZE
FLASHBACK_SCN	FLASHBACK_SCN
FLASHBACK_TIME	FLASHBACK_TIME
FULL	FULL
GRANTS	EXCLUDE=GRANT and INCLUDE=GRANT
HELP	HELP
INDEXES	EXCLUDE=INDEX and INCLUDE=INDEX
LOG	LOGFILE

Table 2–1 (Cont.) Original Export Parameters and Their Counterparts in Data Pump

Original Export Parameter	Comparable Data Pump Export Parameter
OBJECT_CONSISTENT	A parameter comparable to OBJECT_CONSISTENT is not needed.
OWNER	SCHEMAS
PARFILE	PARFILE
QUERY	QUERY
RECORDLENGTH	A parameter comparable to RECORDLENGTH is not needed because sizing is done automatically.
RESUMABLE	A parameter comparable to RESUMABLE is not needed. This functionality is automatically provided.
RESUMABLE_NAME	A parameter comparable to RESUMABLE_NAME is not needed. This functionality is automatically provided.
RESUMABLE_TIMEOUT	A parameter comparable to RESUMABLE_TIMEOUT is not needed. This functionality is automatically provided.
ROWS=N	CONTENT=METADATA_ONLY
ROWS=Y	CONTENT=ALL
STATISTICS	A parameter comparable to STATISTICS is not needed. Statistics are always saved for tables.
TABLES	TABLES
TABLESPACES	TABLESPACES (Same parameter; slightly different behavior)
TRANSPORT_TABLESPACE	TRANSPORT_TABLESPACES (Same parameter; slightly different behavior)
TRIGGERS	EXCLUDE=TRIGGER and INCLUDE=TRIGGER
TTS_FULL_CHECK	TRANSPORT_FULL_CHECK
USERID	A parameter comparable to USERID is not needed. This information is supplied as the *username*/*password* when you invoke Export.
VOLSIZE	A parameter comparable to VOLSIZE is not needed.

This table does not list all Data Pump Export command-line parameters. For information about all Export command-line parameters, see Parameters Available in Export's Command-Line Mode on page 2-8.

See Also: Chapter 20, "Original Export and Import" for details about original Export

Commands Available in Export's Interactive-Command Mode

In interactive-command mode, the current job continues running, but logging to the terminal is suspended and the Export prompt is displayed.

Note: Data Pump Export interactive-command mode is different from the interactive mode for original Export, in which Export prompted you for input. See Interactive Mode on page 20-8 for information about interactive mode in original Export.

To start interactive-command mode, do one of the following:

- From an attached client, press Ctrl+C.
- From a terminal other than the one on which the job is running, specify the ATTACH parameter in an expdp command to attach to the job. This is a useful feature in situations in which you start a job at one location and need to check on it at a later time from a different location.

Table 2–2 lists the activities you can perform for the current job from the Data Pump Export prompt in interactive-command mode.

Table 2–2 Supported Activities in Data Pump Export's Interactive-Command Mode

Activity	Command Used
Add additional dump files.	ADD_FILE on page 2-38
Exit interactive mode and enter logging mode.	CONTINUE_CLIENT on page 2-39
Stop the export client session, but leave the job running.	EXIT_CLIENT on page 2-39
Display a summary of available commands.	HELP on page 2-39
Detach all currently attached client sessions and kill the current job.	KILL_JOB on page 2-40
Increase or decrease the number of active worker processes for the current job. This command is valid only in the Enterprise Edition of Oracle Database 10*g*.	PARALLEL on page 2-40
Restart a stopped job to which you are attached.	START_JOB on page 2-41

Table 2–2 (Cont.) Supported Activities in Data Pump Export's Interactive-Command

Activity	Command Used
Display detailed status for the current job and / or set status interval.	STATUS on page 2-41
Stop the current job for later restart.	STOP_JOB on page 2-42

The following are descriptions of the commands available in the interactive-command mode of Data Pump Export.

ADD_FILE

Purpose

Adds additional files or wildcard file templates to the export dump file set.

Syntax and Description

```
ADD_FILE=[directory_object]file_name [,...]
```

The *file_name* must not contain any directory path information. However, it can include a substitution variable, %U, which indicates that multiple files may be generated using the specified filename as a template. It can also specify another *directory_object*.

The size of the file being added is determined by the setting of the FILESIZE parameter.

See Also:

- DUMPFILE on page 2-12
- FILESIZE on page 2-17
- File Allocation on page 1-12 for information about the effects of using substitution variables

Example

The following example adds two dump files to the dump file set. A directory object is not specified for the dump file named hr2.dmp so the default directory object of dpump_dir1 is assumed. A different directory object, dpump_dir2, is specified for the dump file named hr3.dmp.

```
expdp> ADD_FILE=hr2.dmp, dpump_dir2:hr3.dmp
```

CONTINUE_CLIENT

Purpose

Changes the Export mode from interactive-command mode to logging mode.

Syntax and Description

```
CONTINUE_CLIENT
```

In logging mode, status is continually output to the terminal. If the job is currently stopped, then CONTINUE_CLIENT will also cause the client to attempt to start the job.

Example

```
expdp> CONTINUE_CLIENT
```

EXIT_CLIENT

Purpose

Stops the export client session, exits Export, and discontinues logging to the terminal, but leaves the current job running.

Syntax and Description

```
EXIT_CLIENT
```

Because EXIT_CLIENT stops the client session but leaves the job running, you can attach to the job at a later time. To see the status of the job, you can monitor the log file for the job or you can query the USER_DATAPUMP_JOBS view or the V$SESSION_LONGOPS view.

Example

```
expdp> EXIT_CLIENT
```

HELP

Purpose

Provides information about Data Pump Export commands available in interactive-command mode.

Syntax and Description

```
HELP
```

Displays information about the commands available in interactive-command mode.

Example

```
expdp> HELP
```

KILL_JOB

Purpose

Detaches all currently attached client sessions and then kills the current job. It exits Export and returns to the terminal prompt.

Syntax and Description

```
KILL_JOB
```

A job that is killed using KILL_JOB cannot be restarted. All attached clients, including the one issuing the KILL_JOB command, receive a warning that the job is being killed by the current user and are then detached. After all clients are detached, the job's process structure is immediately run down and the master table and dump files are deleted. Log files are not deleted.

Example

```
expdp> KILL_JOB
```

PARALLEL

Purpose

Enables you to increase or decrease the number of active worker processes for the current job.

Syntax and Description

```
PARALLEL=integer
```

PARALLEL is available as both a command-line parameter and an interactive-command mode parameter. You set it to the desired number of parallel processes. An increase takes effect immediately if there are sufficient files and

resources. A decrease does not take effect until an existing process finishes its current task. If the value is decreased, workers are idled but not deleted until the job exits.

> **See Also:** PARALLEL on page 2-25 for more information about parallelism

Example

```
PARALLEL=10
```

START_JOB

Purpose

Starts the current job to which you are attached.

Syntax and Description

```
START_JOB
```

The START_JOB command restarts the current job to which you are attached (the job cannot be currently executing). The job is restarted with no data loss or corruption after an unexpected failure or after you issued a STOP_JOB command, provided the dump file set and master table remain undisturbed.

Transportable-tablespace-mode exports are not restartable.

Example

```
expdp> START_JOB
```

STATUS

Purpose

Displays cumulative status of the job, along with a description of the current operation. An estimated completion percentage for the job is also returned.

Syntax and Description

```
STATUS[=integer]
```

You have the option of specifying how frequently, in seconds, this status should be displayed in logging mode. If no value is entered or if the default value of 0 is used, the periodic status display is turned off and status is displayed only once.

This status information is written only to your standard output device, not to the log file (even if one is in effect).

Example

The following example will display the status every five minutes (300 seconds):

```
STATUS=300
```

STOP_JOB

Purpose

Stops the current job either immediately or after an orderly shutdown, and exits Export.

Syntax and Description

```
STOP_JOB[=IMMEDIATE]
```

If the master table and dump file set are not disturbed when or after the STOP_JOB command is issued, the job can be attached to and restarted at a later time with the START_JOB command.

To perform an orderly shutdown, use STOP_JOB (without any associated value). A warning requiring confirmation will be issued. An orderly shutdown stops the job after worker processes have finished their current tasks.

To perform an immediate shutdown, specify STOP_JOB=IMMEDIATE. A warning requiring confirmation will be issued. All attached clients, including the one issuing the STOP_JOB command, receive a warning that the job is being stopped by the current user and they will be detached. After all clients are detached, the process structure of the job is immediately run down. That is, the master process will not wait for the worker processes to finish their current tasks. There is no risk of corruption or data loss when you specify STOP_JOB=IMMEDIATE. However, some tasks that were incomplete at the time of shutdown may have to be redone at restart time.

Example

```
expdp> STOP_JOB=IMMEDIATE
```

Examples of Using Data Pump Export

This section provides the following examples of using Data Pump Export:

- Performing a Table-Mode Export
- Data-Only Unload of Selected Tables and Rows
- Estimating Disk Space Needed in a Table-Mode Export
- Performing a Schema-Mode Export
- Performing a Parallel Full Database Export
- Using Interactive Mode to Stop and Reattach to a Job

For information that will help you to successfully use these examples, see Using the Export Parameter Examples on page 2-8.

Performing a Table-Mode Export

Example 2–1 shows a table-mode export, specified using the TABLES parameter. Issue the following Data Pump export command to perform a table export of the tables `employees` and `jobs` from the human resources (`hr`) schema:

Example 2–1 Performing a Table-Mode Export

```
expdp hr/hr TABLES=employees,jobs DUMPFILE=dpump_dir1:table.dmp NOLOGFILE=y
```

Because user `hr` is exporting tables in his own schema, it is not necessary to specify the schema name for the tables. The `NOLOGFILE=y` parameter indicates that an Export log file of the operation will not be generated.

Data-Only Unload of Selected Tables and Rows

Example 2–2 shows the contents of a parameter file (`exp.par`) that you could use to perform a data-only unload of all tables in the human resources (`hr`) schema except for the tables `countries`, `locations`, and `regions`. All rows in the `employees` table are unloaded except those with a `department_id` not equal to 50. The rows are ordered by `employee_id`.

Example 2–2 Data-Only Unload of Selected Tables and Rows

```
DIRECTORY=dpump_dir1
DUMPFILE=dataonly.dmp
CONTENT=DATA_ONLY
```

```
EXCLUDE=TABLE:"IN ('COUNTRIES', 'LOCATIONS', 'REGIONS')"
QUERY=employees:"WHERE department_id !=50 ORDER BY employee_id"
```

You can issue the following command to execute the `exp.par` parameter file:

```
> expdp hr/hr PARFILE=exp.par
```

A schema-mode export (the default mode) is performed, but the `CONTENT` parameter effectively limits the export to an unload of just the tables. The DBA previously created the directory object `dpump_dir1` which points to the directory on the server where user `hr` is authorized to read and write export dump files. The dump file `dataonly.dmp` is created in `dpump_dir1`.

Estimating Disk Space Needed in a Table-Mode Export

Example 2–3 shows the use of the `ESTIMATE_ONLY` parameter to estimate the space that would be consumed in a table-mode export, without actually performing the export operation. Issue the following command to use the `BLOCKS` method to estimate the number of bytes required to export the data in the following three tables located in the human resource (`hr`) schema: `employees`, `departments`, and `locations`.

Example 2–3 Estimating Disk Space Needed in a Schema-Mode Export

```
> expdp hr/hr DIRECTORY=dpump_dir1 ESTIMATE_ONLY=y TABLES=employees,
departments, locations LOGFILE=estimate.log
```

The estimate is printed in the log file and displayed on the client's standard output device. The estimate is for table row data only; it does not include metadata.

Performing a Schema-Mode Export

Example 2–4 shows a schema-mode export of the `hr` schema. In a schema-mode export, only objects belonging to the corresponding schemas are unloaded. Because schema mode is the default mode, it is not necessary to specify the `SCHEMAS` parameter on the command line, unless you are specifying more than one schema or a schema other than your own.

Example 2–4 Performing a Schema Mode Export

```
> expdp hr/hr DUMPFILE=dpump_dir1:expschema.dmp LOGFILE=dpump_dir1:expschema.log
```

Performing a Parallel Full Database Export

Example 2–5 shows a full database Export that will have 3 parallel worker processes.

Example 2–5 Parallel Full Export

```
> expdp hr/hr FULL=y DUMPFILE=dpump_dir1:full1%U.dmp, dpump_dir2:full2%U.dmp
FILESIZE=2G PARALLEL=3 LOGFILE=dpump_dir1:expfull.log JOB_NAME=expfull
```

Because this is a full database export, all data and metadata in the database will be exported. Dump files `full101.dmp`, `full201.dmp`, `full102.dmp`, and so on will be created in a round-robin fashion in the directories pointed to by the `dpump_dir1` and `dpump_dir2` directory objects. For best performance, these should be on separate I/O channels. Each file will be up to 2 gigabytes in size, as necessary. Initially, up to three files will be created. More files will be created, if needed. The job and master table will have a name of `expfull`. The log file will be written to `expfull.log` in the `dpump_dir1` directory.

Using Interactive Mode to Stop and Reattach to a Job

To start this example, reexecute the parallel full export in Example 2–5. While the export is running, press Ctrl+C. This will start the interactive-command interface of Data Pump Export. In the interactive interface, logging to the terminal stops and the Export prompt is displayed.

Issue the following command to stop the job:

```
Export> STOP_JOB=IMMEDIATE
Are you sure you wish to stop this job ([y]/n): y
```

The job is placed in a stopped state and exits the client. Example 2–6 shows how to reattach to the job.

Example 2–6 Attaching to a Stopped Job

Enter the following command to reattach to the job you just stopped:

```
> expdp hr/hr ATTACH=EXPFULL
```

After the job status is displayed, you can issue the `CONTINUE_CLIENT` command to resume logging mode and restart the `expfull` job.

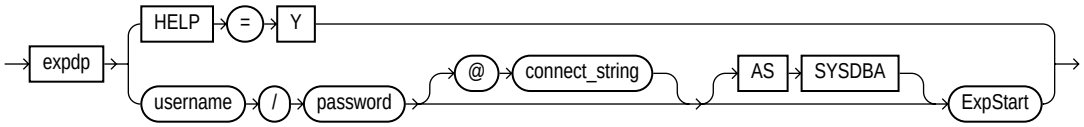
```
Export> CONTINUE_CLIENT
```

A message is displayed that the job has been reopened, and processing status is output to the client.

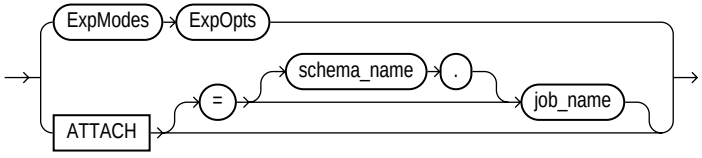
Syntax Diagrams for Data Pump Export

This section provides syntax diagrams for Data Pump Export. These diagrams use standard SQL syntax notation. For more information about SQL syntax notation, see *Oracle Database SQL Reference*.

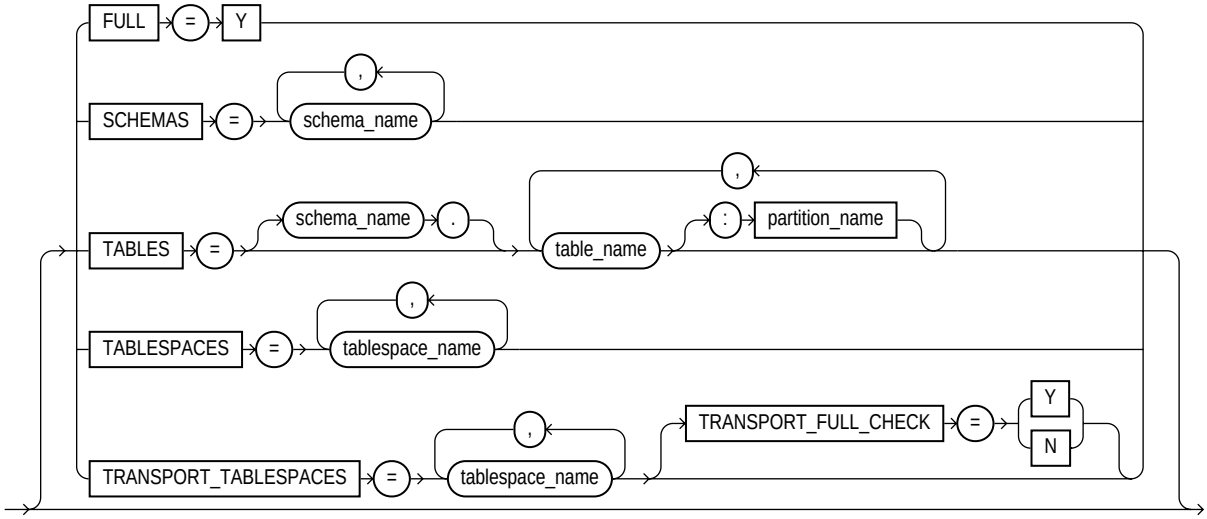
ExpInit



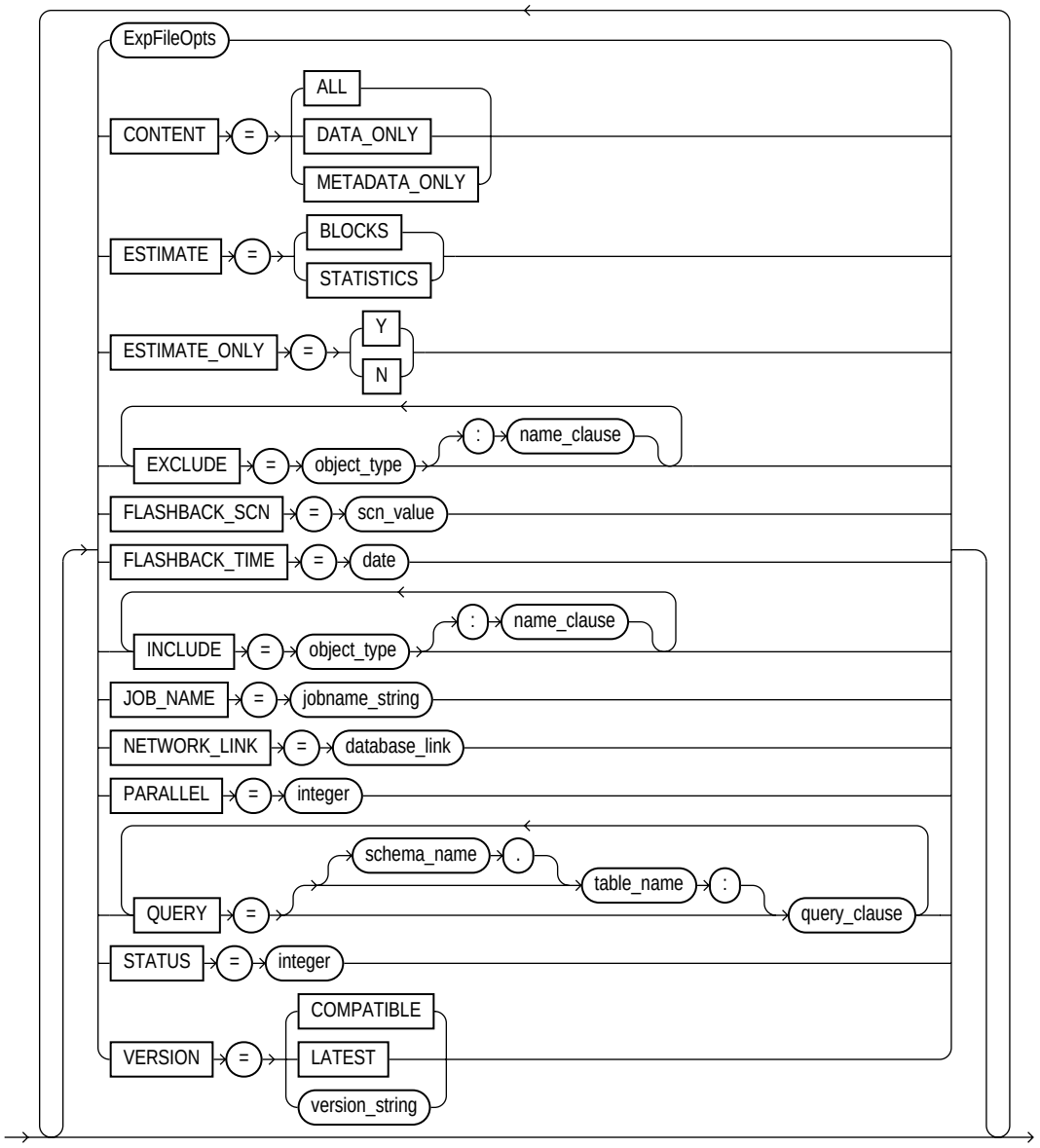
ExpStart



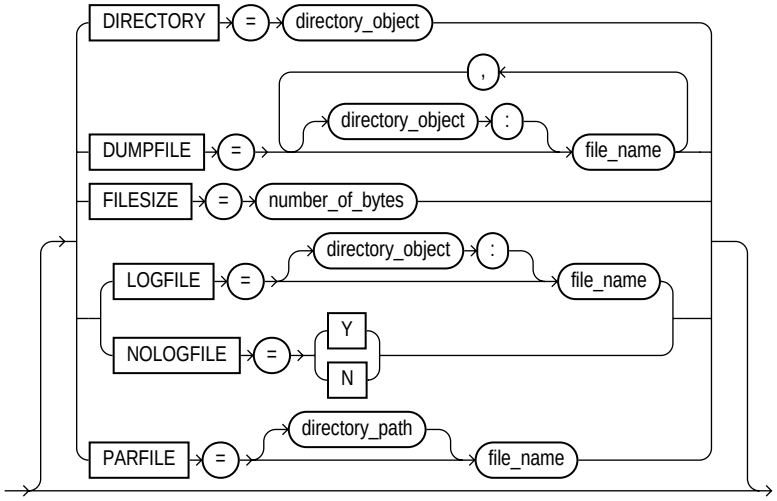
ExpModes


FULL
=
Y
SCHEMAS
=
,
schema_name
TABLES
=
schema_name
.
table_name
:
partition_name
,
TABLESPACES
=
,
tablespace_name
TRANSPORT_TABLESPACES
=
,
tablespace_name
TRANSPORT_FULL_CHECK
=
Y
N

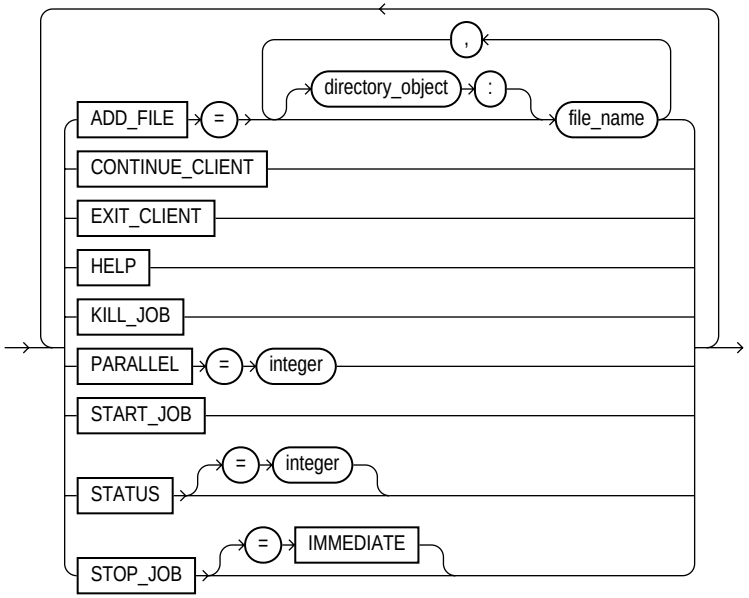
ExpOpts


ExpFileOpts
CONTENT
=
ALL
DATA_ONLY
METADATA_ONLY
ESTIMATE
=
BLOCKS
STATISTICS
ESTIMATE_ONLY
=
Y
N
EXCLUDE
=
object_type
:
name_clause
FLASHBACK_SCN
=
scn_value
FLASHBACK_TIME
=
date
INCLUDE
=
object_type
:
name_clause
JOB_NAME
=
jobname_string
NETWORK_LINK
=
database_link
PARALLEL
=
integer
QUERY
=
schema_name
.
table_name
:
query_clause
STATUS
=
integer
VERSION
=
COMPATIBLE
LATEST
version_string

ExpFileOpts



ExpDynOpts



3

Data Pump Import

This chapter describes the Oracle Data Pump Import utility. The following topics are discussed:

What Is Data Pump Import?

> **Note:** Data Pump Import (invoked with the `impdp` command) is a new utility as of Oracle Database 10*g*. Although its functionality and its parameters are similar to those of the original Import utility (`imp`), they are completely separate utilities and their files are not compatible. See Chapter 20, "Original Export and Import" for a description of the original Import utility.

Data Pump Import (hereinafter referred to as Import for ease of reading) is a utility for loading an export dump file set into a target system. The dump file set is made

up of one or more disk files that contain table data, database object metadata, and control information. The files are written in a proprietary, binary format. During an import operation, the Data Pump Import utility uses these files to locate each database object in the dump file set.

Import can also be used to load a target database directly from a source database with no intervening dump files. This allows export and import operations to run concurrently, minimizing total elapsed time. This is known as a network import.

Data Pump Import enables you to specify whether a job should move a subset of the data and metadata from the dump file set or the source database (in the case of a network import), as determined by the import mode. This is done using data filters and metadata filters, which are implemented through Import commands. See Filtering During Import Operations on page 3-6.

To see some examples of the various ways in which you can use Import, refer to Examples of Using Data Pump Import on page 3-49.

Invoking Data Pump Import

The Data Pump Import utility is invoked using the `impdp` command. The characteristics of the import operation are determined by the import parameters you specify. These parameters can be specified either on the command line or in a parameter file.

Note: Do not invoke Import as `SYSDBA`, except at the request of Oracle technical support. `SYSDBA` is used internally and has specialized functions; its behavior is not the same as for general users.

The following sections contain more information about invoking Import:

- Data Pump Import Interfaces on page 3-2
- Data Pump Import Modes on page 3-3
- Network Considerations on page 3-5

Data Pump Import Interfaces

You can interact with Data Pump Import by using a command line, a parameter file, or an interactive-command mode.

- Command-Line Interface: Enables you to specify the Import parameters directly on the command line. For a complete description of the parameters available in the command-line interface, see Parameters Available in Import's Command-Line Mode on page 3-7.
- Parameter File Interface: Enables you to specify command-line parameters in a parameter file. The only exception is the `PARFILE` parameter because parameter files cannot be nested.
- Interactive-Command Interface: Stops logging to the terminal and displays the Import prompt, from which you can enter various commands, some of which are specific to interactive-command mode. This mode is enabled by pressing Ctrl+C during an import operation started with the command-line interface or the parameter file interface. Interactive-command mode is also enabled when you attach to an executing or stopped job.

  For a complete description of the commands available in interactive-command mode, see Commands Available in Import's Interactive-Command Mode on page 3-44.

Data Pump Import Modes

One of the most significant characteristics of an import operation is its mode, because the mode largely determines what is imported. The specified mode applies to the source of the operation, either a dump file set or another database if the `NETWORK_LINK` parameter is specified.

When the source of the import operation is a dump file set, specifying a mode is optional. If no mode is specified, then Import attempts to load the entire dump file set in the mode in which the export operation was run.

The mode is specified on the command line, using the appropriate parameter. The available modes are as follows:

- Full Import Mode on page 3-4
- Schema Mode on page 3-4
- Table Mode on page 3-4
- Tablespace Mode on page 3-4
- Transportable Tablespace Mode on page 3-5

Full Import Mode

A full import is specified using the FULL parameter. In full import mode, the entire content of the source (dump file set or another database) is loaded into the target database. This is the default for file-based imports. You must have the IMP_FULL_DATABASE role if the source is another database or if the export operation that generated the dump file set required the EXP_FULL_DATABASE role.

See Also: FULL on page 3-17

Schema Mode

A schema import is specified using the SCHEMAS parameter. In a schema import, only objects owned by the current user are loaded. The source can be a full or schema-mode export dump file set or another database. If you have the IMP_FULL_DATABASE role, then a single schema other than your own schema, or a list of schemas can be specified and the schemas themselves (including system privilege grants) are created in the database in addition to the objects contained within those schemas.

Cross-schema references are not imported. For example, a trigger defined on a table within one of the specified schemas, but residing in a schema not explicitly specified, is not imported.

See Also: SCHEMAS on page 3-30

Table Mode

A table-mode import is specified using the TABLES parameter. In table mode, only the specified set of tables, partitions, and their dependent objects are loaded. The source can be a full, schema, tablespace, or table-mode export dump file set or another database. You must have the IMP_FULL_DATABASE role to specify tables that are not in your own schema.

See Also: TABLES on page 3-35

Tablespace Mode

A tablespace-mode import is specified using the TABLESPACES parameter. In tablespace mode, all objects contained within the specified set of tablespaces are loaded. The source can be a full, schema, tablespace, or table-mode export dump file set or another database.

See Also: TABLESPACES on page 3-36

Transportable Tablespace Mode

A transportable tablespace import is specified using the TRANSPORT_TABLESPACES parameter. In transportable tablespace mode, the metadata from a transportable tablespace export dump file set or from another database is loaded. The datafiles specified by the TRANSPORT_DATAFILES parameter must be made available from the source system for use in the target database, typically by copying them over to the target system.

This mode requires the IMP_FULL_DATABASE role.

See Also:

- TRANSPORT_TABLESPACES on page 3-41
- TRANSPORT_FULL_CHECK on page 3-40

Network Considerations

You can use SQL*Net connection strings or connection descriptors when you invoke the Import utility. To do so, the listener must be running (`lsnrctl start`). The following example shows how to invoke Import using a SQL*Net connection:

```
> impdp hr/hr@inst1 DIRECTORY=dpump_dir DUMPFILE=hr.dmp TABLES=employees
```

The `hr/hr@inst1` results in a SQL*Net connection. The *inst1* refers to a service name specified in the `tnsnames.ora` file. This means that the import client is being run remotely from the server to import data from a dump file.

Do not confuse invoking the Import utility using a SQL*Net connection string with performing an import operation using the Import NETWORK_LINK parameter.

The NETWORK_LINK parameter initiates a network import. This means that the `impdp` client initiates the import request, typically to the local database. That server contacts the remote source database referenced by the database link in the NETWORK_LINK parameter, retrieves the data, and writes it directly back to the target database. There are no dump files involved.

See Also:

- NETWORK_LINK on page 3-22
- *Oracle Net Services Administrator's Guide*
- *Oracle Database Heterogeneous Connectivity Administrator's Guide*

Filtering During Import Operations

Data Pump Import provides much greater data and metadata filtering capability than was provided by the original Import utility.

Data Filters

Data filters specify restrictions on the rows that are to be imported. These restrictions can be based on partition names and on the results of subqueries.

Each data filter can only be specified once per table and once per job. If different filters using the same name are applied to both a particular table and to the whole job, the filter parameter supplied for the specific table will take precedence.

Metadata Filters

Data Pump Import provides much greater metadata filtering capability than was provided by the original Import utility. Metadata filtering is implemented through the `EXCLUDE` and `INCLUDE` parameters. The `EXCLUDE` and `INCLUDE` parameters are mutually exclusive.

Metadata filters identify a set of objects to be included or excluded from a Data Pump operation. For example, you could request a full import, but without Package Specifications or Package Bodies.

To use filters correctly and to get the results you expect, remember that *dependent objects of an identified object are processed along with the identified object*. For example, if a filter specifies that a package is to be included in an operation, then grants upon that package will also be included. Likewise, if a table is excluded by a filter, then indexes, constraints, grants, and triggers upon the table will also be excluded by the filter.

If multiple filters are specified for an object type, an implicit `AND` operation is applied to them. That is, objects participating in the job must pass *all* of the filters applied to their object types.

The same filter name can be specified multiple times within a job.

To see which objects can be filtered, you can perform queries on the following views: `DATABASE_EXPORT_OBJECTS`, `SCHEMA_EXPORT_OBJECTS`, and `TABLE_EXPORT_OBJECTS`. For an example of this, see Metadata Filters on page 2-6.

See Also: EXCLUDE on page 3-13 and INCLUDE on page 3-18

Parameters Available in Import's Command-Line Mode

This section provides descriptions of the parameters available in the command-line mode of Data Pump Import. Many of the descriptions include an example of how to use the parameter.

Using the Import Parameter Examples

If you try running the examples that are provided for each parameter, be aware of the following requirements:

- Most of the examples use the sample schemas of the seed database, which is installed by default when you install Oracle Database. In particular, the human resources (`hr`) schema is often used.
- Examples that specify a dump file to import assume that the dump file exists. Wherever possible, the examples use dump files that are generated when you run the Export examples in Chapter 2.
- The examples assume that the directory objects, `dpump_dir1` and `dpump_dir2`, already exist and that `READ` and `WRITE` privileges have been granted to the `hr` schema for these directory objects. See Default Locations for Dump, Log, and SQL Files on page 1-13 for information about creating directory objects and assigning privileges to them.
- Some of the examples require the `EXP_FULL_DATABASE` and `IMP_FULL_DATABASE` roles. The examples assume that the `hr` schema has been granted these roles.

If necessary, ask your DBA for help in creating these directory objects and assigning the necessary privileges and roles.

Syntax diagrams of these parameters are provided in Syntax Diagrams for Data Pump Import on page 3-51.

Unless specifically noted, these parameters can also be specified in a parameter file.

Command-Line Escape Characters in Examples

Some of the examples in this chapter may show certain clauses enclosed in quotation marks and backslashes. This is because those clauses contain a blank, a situation for which most operating systems require that the entire 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, such as the backslash. If the backslashes were not present, the command-line parser that Import uses would not understand the quotation marks

and would remove them. Some examples in this chapter may use escape characters to show you how they would be used. However, in general, Oracle recommends that you place such statements in a parameter file because escape characters are not necessary in parameter files.

See Also:

- Default Locations for Dump, Log, and SQL Files on page 1-13 for information about creating default directory objects
- Examples of Using Data Pump Import on page 3-49
- *Oracle Database Sample Schemas*

Note: If you are accustomed to using the original Import utility, you may be wondering which Data Pump parameters are used to perform the operations you used to perform with original Import. For a comparison, see How Data Pump Import Parameters Map to Those of the Original Import Utility on page 3-42.

ATTACH

Default: current job in user's schema, if there is only one running job

Purpose

Attaches the client session to an existing import job and automatically places you in interactive-command mode.

Syntax and Description

```
ATTACH [=[schema_name.]job_name]
```

Specify a *schema_name* if the schema to which you are attaching is not your own. You must have the IMP_FULL_DATABASE role to do this.

A *job_name* does not have to be specified if only one running job is associated with your schema and the job is active. If the job you are attaching to is stopped, you must supply the job name. To see a list of Data Pump job names, you can query the DBA_DATAPUMP_JOBS view or the USER_DATAPUMP_JOBS view.

When you are attached to the job, Import displays a description of the job and then displays the Import prompt.

Restrictions

When you specify the ATTACH parameter, you cannot specify any other parameters except for the connection string (*user/password*).

You cannot attach to a job in another schema unless it is already running.

Example

The following is an example of using the ATTACH parameter.

```
> impdp hr/hr ATTACH=import_job
```

This example assumes that a job named `import_job` exists in the `hr` schema.

> **See Also:** Commands Available in Import's Interactive-Command Mode on page 3-44

CONTENT

Default: ALL

Purpose

Enables you to filter what is loaded during the import operation.

Syntax and Description

```
CONTENT={ALL | DATA_ONLY | METADATA_ONLY}
```

- ALL loads any data and metadata contained in the source. This is the default.
- DATA_ONLY loads only table row data into existing tables; no database objects are created.
- METADATA_ONLY loads only database object definitions; no table row data is loaded.

Example

The following is an example of using the CONTENT parameter. You can create the `expfull.dmp` dump file used in this example by running the example provided for the Export FULL parameter. See FULL on page 2-19.

```
> impdp hr/hr DIRECTORY=dpump_dir1 DUMPFILE=expfull.dmp CONTENT=METADATA_ONLY
```

This command will execute a full import that will load only the metadata in the expfull.dmp dump file. It executes a full import because that is the default for file-based imports in which no import mode is specified.

DIRECTORY

Default: none for nonprivileged users; DATA_PUMP_DIR for privileged users

Purpose

Specifies the location in which the import job can find the dump file set and where it should create log and SQL files.

Syntax and Description

DIRECTORY=*directory_object*

The *directory_object* is the name of a database directory object (*not the name of an actual directory*) that was previously created by the database administrator (DBA) using the SQL CREATE DIRECTORY statement. You can override this value with individual directory objects specified on the DUMPFILE, LOGFILE, and SQLFILE parameters. You must have read access to the directory used for the dump file set and write access to the directory used to create the log and SQL files.

Example

The following is an example of using the DIRECTORY parameter. You can create the expfull.dmp dump file used in this example by running the example provided for the Export FULL parameter. See FULL on page 2-19.

```
> impdp hr/hr DIRECTORY=dpump_dir1 DUMPFILE=expfull.dmp
LOGFILE=dpump_dir2:expfull.log
```

This command results in the import job looking for the expfull.dmp dump file in the directory pointed to by the dpump_dir1 directory object. The dpump_dir2 directory object specified on the LOGFILE parameter overrides the DIRECTORY parameter so that the log file is written to dpump_dir2.

Note: See DIRECTORY on page 2-10 for an example (Example 2) of using the default DATA_PUMP_DIR directory object available to privileged users

---

See Also:

- Default Locations for Dump, Log, and SQL Files on page 1-13 for more information about default directory objects
- *Oracle Database SQL Reference* for more information about the CREATE DIRECTORY command

DUMPFILE

Default: expdat.dmp

Purpose

Specifies the names and optionally, the directory objects of the dump file set that was created by Export.

Syntax and Description

```
DUMPFILE=[directory_object:]file_name [, ...]
```

The *directory_object* is optional if one has already been established by the DIRECTORY parameter. If you do supply a value here, it must be a directory object that already exists, and it will override a value that was specified by the DIRECTORY parameter.

The *file_name* is the name of the dump file set. The filenames can also be templates that contain the substitution variable, %U. If %U is used, Import examines each file that matches the template (until no match is found) in order to locate all files that are part of the dump file set. The %U expands to a 2-digit incrementing integer starting with 01.

Sufficient information is contained within the files for Import to locate the entire set, provided the file specifications in the DUMPFILE parameter encompass the entire set. The files are not required to have the same names or locations that they had at export time.

Example

The following is an example of using the Import DUMPFILE parameter. You can create the dump files used in this example by running the example provided for the Export DUMPFILE parameter. See DUMPFILE on page 2-12.

```
> impdp hr/hr DIRECTORY=dpump_dir1 DUMPFILE=dpump_dir2:exp1.dmp, exp2%U.dmp
```

Because a directory object (dpump_dir2) is specified for the exp1.dmp dump file, the import job will look there for the file. It will also look in dpump_dir1 for any dump files of the form exp2<nn>.dmp. The log file will be written to dpump_dir1.

See Also:

- File Allocation on page 1-12
- Performing a Data-Only Table-Mode Import on page 3-50

ESTIMATE

Default: BLOCKS

Purpose

Instructs the source system in a network import operation to estimate how much data will be generated.

Syntax and Description

```
ESTIMATE={BLOCKS | STATISTICS}
```

The valid choices for the ESTIMATE parameter are as follows:

- BLOCKS - The estimate is calculated by multiplying the number of database blocks used by the target objects times the appropriate block sizes.
- STATISTICS - The estimate is calculated using statistics for each table. For this method to be as accurate as possible, all tables should have been analyzed recently.

The estimate that is generated can be used to determine a percentage complete throughout the execution of the import job. When the import source is a dump file set, the amount of data to be loaded is already known, so the percentage complete is automatically calculated.

Restrictions

The Import ESTIMATE parameter is valid only if the NETWORK_LINK parameter is also specified.

Example

In the following example, *source_database_link* would be replaced with the name of a valid link to the source database.

```
> impdp hr/hr TABLES=job_history NETWORK_LINK=source_database_link
  DIRECTORY=dpump_dir1 ESTIMATE=statistics
```

The job_history table in the hr schema is imported from the source database. A log file is created by default and written to the directory pointed to by the dpump_dir1 directory object. When the job begins, an estimate for the job is calculated based on table statistics.

EXCLUDE

Default: none

Purpose

Enables you to filter the metadata that is imported by specifying objects and object types that you want to exclude from the import job.

Syntax and Description

```
EXCLUDE=object_type[:name_clause] [, ...]
```

For the given mode of import, all object types contained within the source (and their dependents) are included, except those specified in an EXCLUDE statement. If an object is excluded, all of its dependent objects are also excluded. For example, excluding a table will also exclude all indexes and triggers on the table.

The *name_clause* is optional. It allows fine-grained selection of specific objects within an object type. It is a SQL expression used as a filter on the object names of the type. It consists of a SQL operator and the values against which the object names of the specified type are to be compared. The name clause applies only to object types whose instances have names (for example, it is applicable to TABLE and VIEW, but not to GRANT). The optional name clause must be separated from the object type with a colon and enclosed in double quotation marks, because single-quotation marks are required to delimit the name strings. For example, you could set EXCLUDE=INDEX:"LIKE 'DEPT%'" to exclude all indexes whose names start with dept.

More than one EXCLUDE statement can be specified. Oracle recommends that you place EXCLUDE statements in a parameter file to avoid having to use operating system-specific escape characters on the command line.

As explained in the following sections, you should be aware of the effects of specifying certain objects for exclusion, in particular, CONSTRAINT, GRANT, and USER.

Excluding Constraints

The following constraints cannot be excluded:

- NOT NULL constraints.
- Constraints needed for the table to be created and loaded successfully (for example, primary key constraints for index-organized tables or REF SCOPE and WITH ROWID constraints for tables with REF columns).

This means that the following EXCLUDE statements will be interpreted as follows:

- EXCLUDE=CONSTRAINT will exclude all nonreferential constraints, except for NOT NULL constraints and any constraints needed for successful table creation and loading.
- EXCLUDE=REF_CONSTRAINT will exclude referential integrity (foreign key) constraints.

Excluding Grants and Users

Specifying EXCLUDE=GRANT excludes object grants on all object types and system privilege grants.

Specifying EXCLUDE=USER excludes only the definitions of users, not the objects contained within users' schemas.

To exclude a specific user and all objects of that user, specify a filter such as the following (where hr is the schema name of the user you want to exclude):

```
EXCLUDE=SCHEMA:"= 'HR'"
```

If you try to exclude a user by using a statement such as EXCLUDE=USER:"= 'HR'", only CREATE USER hr DDL statements will be excluded, and you may not get the results you expect.

Restrictions

The EXCLUDE and INCLUDE parameters are mutually exclusive.

Neither EXCLUDE nor INCLUDE can be used if the CONTENT=DATA_ONLY parameter is specified, because that implies import of row data only.

Example

Assume the following is in a parameter file, exclude.par, being used by a DBA or some other user with the IMP_FULL_DATABASE role. (If you want to try the example, you will need to create this file.)

```
EXCLUDE=FUNCTION
EXCLUDE=PROCEDURE
EXCLUDE=PACKAGE
EXCLUDE=INDEX:"LIKE 'EMP%' "
```

You could then issue the following command. You can create the `expfull.dmp` dump file used in this command by running the example provided for the Export FULL parameter. See FULL on page 2-19.

```
> impdp hr/hr DIRECTORY=dpump_dir1 DUMPFILE=expfull.dmp PARFILE=exclude.par
```

All data from the `expfull.dmp` dump file will be loaded except for functions, procedures, packages, and indexes whose names start with `emp`.

> **See Also:** Filtering During Import Operations on page 3-6 for more information about the effects of using the EXCLUDE parameter

FLASHBACK_SCN

Default: none

Purpose

Specifies the system change number (SCN) that Import will use to enable the Flashback utility.

Syntax and Description

```
FLASHBACK_SCN=scn_number
```

The import operation is performed with data that is consistent as of the specified *scn_number*.

Note: If you are on a logical standby system, the FLASHBACK_SCN parameter is ignored because SCNs are selected by logical standby. See *Oracle Data Guard Concepts and Administration* for information about logical standby databases.

Restrictions

The FLASHBACK_SCN parameter is valid only when the NETWORK_LINK parameter is also specified. This is because the value is passed to the source system to provide an SCN-consistent data extraction.

FLASHBACK_SCN and FLASHBACK_TIME are mutually exclusive.

Example

The following is an example of using the FLASHBACK_SCN parameter.

```
> impdp hr/hr DIRECTORY=dpump_dir1 FLASHBACK_SCN=scn_number
NETWORK_LINK=source_database_link
```

The *source_database_link* in this example would be replaced with the name of a source database from which you were importing data. Also, an actual integer value would be supplied for the *scn_number*.

FLASHBACK_TIME

Default: none

Purpose

Specifies the time of a particular SCN.

Syntax and Description

```
FLASHBACK_TIME="TO_TIMESTAMP()"
```

The SCN that most closely matches the specified time is found, and this SCN is used to enable the Flashback utility. The import operation is performed with data that is consistent as of this SCN.

Note: If you are on a logical standby system, the FLASHBACK_TIME parameter is ignored because SCNs are selected by logical standby. See *Oracle Data Guard Concepts and Administration* for information about logical standby databases.

Restrictions

This parameter is valid only when the NETWORK_LINK parameter is also specified. This is because the value is passed to the source system to provide a time-consistent import.

FLASHBACK_TIME and FLASHBACK_SCN are mutually exclusive.

Example

You can specify the time in any format that the DBMS_FLASHBACK.ENABLE_AT_TIME procedure accepts, for example:

```
> impdp hr/hr DIRECTORY=dpump_dir1 NETWORK_LINK=source_database_link
FLASHBACK_TIME="TO_TIMESTAMP('25-08-2003 14:35:00', 'DD-MM-YYYY HH24:MI:SS')"
```

This example may require the use of escape characters, depending on your operating system. See Command-Line Escape Characters in Examples on page 3-7.

> **See Also:** *Oracle Database Application Developer's Guide - Fundamentals* for information about using flashback

FULL

Default: Y

Purpose

Specifies that you want to perform a full database import.

Syntax and Description

```
FULL=y
```

If you specify FULL=y, then everything from the source (either a dump file set or another database) is imported.

The IMP_FULL_DATABASE role is required if the NETWORK_LINK parameter is also used or if the export operation that generated the dump file set required the EXP_FULL_DATABASE role.

Example

The following is an example of using the FULL parameter. You can create the expfull.dmp dump file used in this example by running the example provided for the Export FULL parameter. See FULL on page 2-19.

```
> impdp hr/hr DUMPFILE=dpump_dir1:expfull.dmp FULL=y
LOGFILE=dpump_dir2:full_imp.log
```

This example imports everything from the expfull.dmp dump file. In this example, a DIRECTORY parameter is not provided. Therefore, a directory object

must be provided on both the DUMPFILE parameter and the LOGFILE parameter. The directory objects can be different, as shown in this example.

HELP

Default: n

Purpose

Displays online help for the Import utility.

Syntax and Description

```
HELP=y
```

If HELP=y is specified, Import displays a summary of all Import command-line parameters and interactive commands.

Example

```
> impdp HELP = Y
```

This example will display a brief description of all Import parameters and commands.

INCLUDE

Default: none

Purpose

Enables you to filter the metadata that is imported by specifying objects and object types for the current import mode.

Syntax and Description

```
INCLUDE = object_type[:name_clause] [, ...]
```

Only object types in the source that are explicitly specified in the INCLUDE statement are imported.

The *name_clause* is optional. It allows fine-grained selection of specific objects within an object type. It is a SQL expression used as a filter on the object names of the type. It consists of a SQL operator and the values against which the object names of the specified type are to be compared. The name clause applies only to

object types whose instances have names (for example, it is applicable to TABLE, but not to GRANT). The optional name clause must be separated from the object type with a colon and enclosed in double quotation marks, because single-quotation marks are required to delimit the name strings.

More than one INCLUDE statement can be specified. Oracle recommends that you place INCLUDE statements in a parameter file to avoid having to use operating system-specific escape characters on the command line.

To see a list of valid paths for use with the INCLUDE parameter, you can query the following views: DATABASE_EXPORT_OBJECTS, SCHEMA_EXPORT_OBJECTS, and TABLE_EXPORT_OBJECTS.

Restrictions

The INCLUDE and EXCLUDE parameters are mutually exclusive.

Neither INCLUDE nor EXCLUDE can be used if the CONTENT=DATA_ONLY parameter is specified, because that implies import of table row data only.

Example

Assume the following is in a parameter file, imp_include.par, being used by a DBA or some other user with the IMP_FULL_DATABASE role:

```
INCLUDE=FUNCTION
INCLUDE=PROCEDURE
INCLUDE=PACKAGE
INCLUDE=INDEX:"LIKE 'EMP%' "
```

You can then issue the following command:

```
> impdp hr/hr SCHEMAS=hr DIRECTORY=dpump_dir1 DUMPFILE=expfull.dmp
PARFILE=imp_include.par
```

You can create the expfull.dmp dump file used in this example by running the example provided for the Export FULL parameter. See FULL on page 2-19.

The Import operation will load only functions, procedures, and packages from the hr schema and indexes whose names start with EMP. Although this is a privileged-mode import (the user must have the IMP_FULL_DATABASE role), the schema definition is not imported, because the USER object type was not specified in an INCLUDE statement.

JOB_NAME

Default: system-generated name of the form SYS_<operation>_<mode>_NN

Purpose

Specifies a name for the import job. The job name is used to identify the import job in subsequent actions, such as ATTACH. The job name becomes the name of the master table in the current user's schema. The master table controls the import job.

Syntax and Description

```
JOB_NAME=jobname_string
```

The *jobname_string* specifies a name of up to 30 bytes for this import job. The bytes must represent printable characters and spaces. If spaces are included, the name must be enclosed in single quotation marks (for example, 'Thursday Import'). The job name is implicitly qualified by the schema of the user performing the import operation.

The default job name is system-generated in the form SYS_<operation>_<mode>_NN, where NN expands to a 2-digit incrementing integer starting at 01. An example of a default name is 'SYS_IMPORT_TABLESPACE_02'.

Example

The following is an example of using the JOB_NAME parameter. You can create the expfull.dmp dump file used in this example by running the example provided for the Export FULL parameter. See FULL on page 2-19.

```
> impdp hr/hr DIRECTORY=dpump_dir1 DUMPFILE=expfull.dmp JOB_NAME=impjob01
```

LOGFILE

Default: import.log

Purpose

Specifies the name, and optionally, a directory object, for the log file of the import job.

Syntax and Description

```
LOGFILE=[directory_object:]file_name
```

If you specify a *directory_object*, it must be one that was previously established by the DBA. This overrides the directory object specified with the DIRECTORY parameter. The default behavior is to create import.log in the directory referenced by the directory object specified in the DIRECTORY parameter.

If the *file_name* you specify already exists, it will be overwritten.

All messages regarding work in progress, work completed, and errors encountered are written to the log file. (For a real-time status of the job, use the STATUS command in interactive mode.)

A log file is always created unless the NOLOGFILE parameter is specified. As with the dump file set, the log file is relative to the server and not the client.

Note: Data Pump Import writes the log file using the database character set. If your client NLS_LANG environment sets up a different client character set from the database character set, then it is possible that table names may be different in the log file than they are when displayed on the client output screen.

Example

The following is an example of using the LOGFILE parameter. You can create the expfull.dmp dump file used in this example by running the example provided for the Export FULL parameter. See FULL on page 2-19.

```
> impdp hr/hr SCHEMAS=HR DIRECTORY=dpump_dir2 LOGFILE=imp.log
 DUMPFILE=dpump_dir1:expfull.dmp
```

The DUMPFILE parameter specifies a directory object (dpump_dir1) and a dump file name (expfull.dmp). The directory object specified on the DUMPFILE parameter overrides the directory object specified on the DIRECTORY parameter. Because no directory object is specified on the LOGFILE parameter, the log file is written to the directory object specified on the DIRECTORY parameter.

See Also:

- STATUS on page 3-48
- Using Directory Objects When Automatic Storage Management Is Enabled on page 1-14 for information about the effect of Automatic Storage Management on the location of log files

NETWORK_LINK

Default: none

Purpose

Enables a network import when you specify the name of a valid database link to a source system.

Syntax and Description

```
NETWORK_LINK=source_database_link
```

The NETWORK_LINK parameter initiates a network import. This means that the impdp client initiates the import request, typically to the local database. That server contacts the remote source database referenced by *source_database_link*, retrieves the data, and writes it directly back to the target database. There are no dump files involved.

The *source_database_link* provided must be the name of a valid link to a source database. If the database does not already have a database link, you or your DBA must create one. For more information about the CREATE DATABASE LINK statement, see *Oracle Database SQL Reference*.

If the source database is read-only, then the user on the source database must have a locally-managed tablespace assigned as a default temporary tablespace. Otherwise, the job will fail. For further details about this, see the information about creating locally managed temporary tablespaces in the *Oracle Database Administrator's Guide*.

This parameter is required when any of the following parameters are specified: FLASHBACK_SCN, FLASHBACK_TIME, ESTIMATE, or TRANSPORT_TABLESPACES.

Restrictions

Network imports do not support the use of evolved types.

Example

In the following example, the *source_database_link* would be replaced with the name of a valid database link that must already exist.

```
> impdp hr/hr TABLES=employees DIRECTORY=dpump_dir1
NETWORK_LINK=source_database_link EXCLUDE=CONSTRAINT
```

This example results in an import of the employees table (excluding constraints) from the source database. The log file is written to dpump_dir1, specified on the DIRECTORY parameter.

NOLOGFILE

Default: n

Purpose

Specifies whether or not to suppress the default behavior of creating a log file.

Syntax and Description

```
NOLOGFILE={y | n}
```

If you specify NOLOGFILE=Y to suppress creation of a log file, progress and error information is still written to the standard output device of any attached clients, including the client that started the original export operation. If there are no clients attached to a running job and you specify NOLOGFILE=Y, you run the risk of losing important progress and error information.

Example

The following is an example of using the NOLOGFILE parameter.

```
> impdp hr/hr DIRECTORY=dpump_dir1 DUMPFILE=expfull.dmp NOLOGFILE=Y
```

This command results in a full mode import (the default for file-based imports) of the expfull.dmp dump file. No log file is written because NOLOGFILE is set to y.

PARALLEL

Default: 1

Purpose

Specifies the maximum number of threads of active execution operating on behalf of the import job.

Syntax and Description

```
PARALLEL=integer
```

The value you specify for *integer* specifies the maximum number of threads of active execution operating on behalf of the import job. This execution set consists of a combination of worker processes and parallel I/O server processes. The master control process, idle workers, and worker processes acting as parallel execution coordinators in parallel I/O operations do not count toward this total. This parameter enables you to make trade-offs between resource consumption and elapsed time.

If the source of the import is a dump file set consisting of files, multiple processes can read from the same file, but performance may be limited by I/O contention.

To increase or decrease the value of PARALLEL during job execution, use interactive-command mode.

Parallelism is used to best advantage for loading user data and package bodies, and for building indexes.

Restrictions

This parameter is valid only in the Enterprise Edition of Oracle Database 10*g*.

Example

The following is an example of using the PARALLEL parameter.

```
> impdp hr/hr DIRECTORY=dpump_dir1 LOGFILE=parallel_import.log
JOB_NAME=imp_par3 DUMPFILE=par_exp%U.dmp PARALLEL=3
```

This command imports the dump file set that is created when you run the example for the Export PARALLEL parameter. (See PARALLEL on page 2-25.) The names of the dump files are `par_exp01.dmp`, `par_exp02.dmp`, and `par_exp03.dmp`.

The import job looks for the dump files in the location indicated by the `dpump_dir1` directory object specified on the DIRECTORY parameter. This is also the location to which the log file is written.

PARFILE

Default: none

Purpose

Specifies the name of an import parameter file.

Syntax and Description

PARFILE=[*directory_path*]*file_name*

Unlike dump files, log files, and SQL files which are created and written by the server, the parameter file is opened and read by the client running the impdp image. Therefore, a directory object name is neither required nor appropriate. The default is the user's current directory.

Restrictions

The PARFILE parameter cannot be specified within a parameter file.

Example

The content of an example parameter file, hr_imp.par, might be as follows:

```
TABLES= countries, locations, regions
DUMPFILE=dpump_dir2:exp1.dmp,exp2%U.dmp
DIRECTORY=dpump_dir1
PARALLEL=3
```

You could then issue the following command to execute the parameter file:

```
> impdp hr/hr PARFILE=hr_imp.par
```

The tables named countries, locations, and regions will be imported from the dump file set that is created when you run the example for the Export DUMPFILE parameter. (See DUMPFILE on page 2-12.) The import job looks for the exp1.dmp file in the location pointed to by dpump_dir2. It looks for any dump files of the form exp2<nn>.dmp in the location pointed to by dpump_dir1. The log file for the job will also be written to dpump_dir1.

QUERY

Default: none

Purpose

Enables you to filter the data that is imported by specifying a SQL SELECT statement, which is applied to all tables in the import job or to a specific table.

Syntax and Description

QUERY=[[*schema_name*.]*table_name*:]*query_clause*

The *query_clause* is typically a WHERE clause for fine-grained row selection, but could be any SQL clause.

If a schema and table are not supplied, the query is applied to (and must be valid for) all tables in the source dump file set or database.

When the query is to be applied to a specific table, a colon must separate the table name from the query clause. More than one table-specific query can be specified, but only one can be specified per table.

The query must be enclosed in single or double quotation marks. Double quotation marks are recommended, because strings within the clause must be enclosed in single quotation marks.

Oracle recommends that you place QUERY specifications in a parameter file to avoid having to use operating system-specific escape characters on the command line.

When the QUERY parameter is used, the external tables method (rather than the direct path method) is used for data access.

To specify a schema other than your own in a table-specific query, you need the IMP_FULL_DATABASE role.

Restrictions

The QUERY parameter cannot be used in conjunction with the following parameters:

- CONTENT=METADATA_ONLY
- SQLFILE
- TRANSPORT_TABLESPACES

Example

The following is an example of using the QUERY parameter. You can create the expfull.dmp dump file used in this example by running the example provided for the Export FULL parameter. See FULL on page 2-19.

```
> impdp hr/hr DIRECTORY=dpump_dir1 DUMPFILE=expfull.dmp
 QUERY=departments:'"WHERE department_id < 120"' NOLOGFILE=Y
```

Only data in expfull.dmp that meets the criteria specified in the QUERY parameter is imported.

This example may require the use of escape characters, depending on your operating system. See Command-Line Escape Characters in Examples on page 3-7.

REMAP_DATAFILE

Default: none

Purpose

Changes the name of the source datafile to the target datafile name in all SQL statements where the source datafile is referenced: CREATE TABLESPACE, CREATE LIBRARY, and CREATE DIRECTORY.

Syntax and Description

REMAP_DATAFILE=*source_datafile*:*target_datafile*

Remapping datafiles is useful when you move databases between platforms that have different file naming conventions. The *source_datafile* and *target_datafile* names should be exactly as you want them to appear in the SQL statements where they are referenced. Oracle recommends that you enclose datafile names in quotation marks to eliminate ambiguity on platforms for which a colon is a valid file specification character.

You must have the IMP_FULL_DATABASE role to specify this parameter.

Example

The following is an example of using the REMAP_DATAFILE parameter.

```
> impdp hr/hr FULL=y DIRECTORY=dpump_dir1 DUMPFILE=db_full.dmp
REMAP_DATAFILE='DB1$:[HRDATA.PAYROLL]tbs6.f':'/db1/hrdata/payroll/tbs6.f'
```

This example remaps a VMS file specification (DR1$:[HRDATA.PAYROLL]tbs6.f) to a UNIX file specification, (/db1/hrdata/payroll/tbs6.f) for all SQL DDL statements during the import. The dump file, db_full.dmp, is located in the directory object, dpump_dir1.

REMAP_SCHEMA

Default: none

Purpose

Loads all objects from the source schema into a target schema.

Syntax and Description

REMAP_SCHEMA=*source_schema*:*target_schema*

Multiple REMAP_SCHEMA lines can be specified, but the source schema must be different for each one. However, different source schemas can map to the same target schema. The mapping may not be 100 percent complete, because there are certain schema references that Import is not capable of finding. For example, Import will not find schema references embedded within the body of definitions of types, views, procedures, and packages.

This parameter requires the IMP_FULL_DATABASE role.

If the schema you are remapping to does not already exist, the import operation creates it, provided the dump file set contains the necessary CREATE USER metadata and you are importing with enough privileges. For example, the following Export commands would create the dump file sets with the necessary metadata to create a schema, because the user SYSTEM has the necessary privileges:

```
> expdp SYSTEM/password SCHEMAS=hr
> expdp SYSTEM/password FULL=y
```

If you do not have enough privileges to perform an import that creates dump files containing the metadata necessary to create a schema, then you must create the target schema before performing the import operation. This is because the dump files do not contain the necessary information for the import to create the schema automatically.

If the import operation does create the schema, then after the import is complete, you must assign it a valid password in order to connect to it. The SQL statement to do this, which requires privileges, is:

```
SQL> ALTER USER [schema_name] IDENTIFIED BY [new_pswd]
```

When you remap a schema, it is possible to constrain the rows loaded into the destination table by specifying a query. However, Data Pump Import performs remapping before applying any queries. This means that by the time the query is applied, the schema has already been remapped. Therefore, to constrain the rows that are loaded, the query must be applied to the target schema table rather than to the source schema table.

Example

Suppose that you execute the following Export and Import commands to remap the hr schema into the blake schema:

```
> expdp SYSTEM/password SCHEMAS=hr DIRECTORY=dpump_dir1 DUMPFILE=hr.dmp
```

```
> impdp SYSTEM/password DIRECTORY=dpump_dir1 DUMPFILE=hr.dmp
REMAP_SCHEMA=hr:scott
```

In this example, if user scott already exists before the import, then the Import REMAP_SCHEMA command will add objects from the hr schema into the existing scott schema. You can connect to the scott schema after the import by using the existing password (without resetting it).

If user scott does not exist before you execute the import operation, Import automatically creates it with an unusable password. This is possible because the dump file, hr.dmp, was created by SYSTEM, which has the privileges necessary to create a dump file that contains the metadata needed to create a schema. However, you cannot connect to scott on completion of the import, unless you reset the password for scott on the target database after the import completes.

REMAP_TABLESPACE

Default: none

Purpose

Remaps all objects selected for import with persistent data in the source tablespace to be created in the target tablespace.

Syntax and Description

REMAP_TABLESPACE=*source_tablespace*:*target_tablespace*

Multiple REMAP_TABLESPACE parameters can be specified, but no two can have the same source tablespace. The target schema must have sufficient quota in the target tablespace.

Note that use of the REMAP_TABLESPACE parameter is the *only* way to remap a tablespace in Data Pump Import. This is a simpler and cleaner method than the one provided in the original Import utility. In original Import, if you wanted to change the default tablespace for a user, you had to perform several steps, including exporting and dropping the user, creating the user in the new tablespace, and importing the user from the dump file. That method was subject to many restrictions (including the number of tablespace subclauses) which sometimes resulted in the failure of some DDL commands.

By contrast, the Data Pump Import method of using the REMAP_TABLESPACE parameter works for all objects, including the user, and it works regardless of how many tablespace subclauses are in the DDL statement.

Example

The following is an example of using the REMAP_TABLESPACE parameter.

```
> impdp hr/hr REMAP_TABLESPACE='tbs_1':'tbs_6' DIRECTORY=dpump_dir1 PARALLEL=2
JOB_NAME=cf1n02 DUMPFILE=employees.dmp NOLOGFILE=Y
```

REUSE_DATAFILES

Default: n

Purpose

Specifies whether or not the import job should reuse existing datafiles for tablespace creation.

Syntax and Description

```
REUSE_DATAFILES={y | n}
```

If the default (n) is used and the datafiles specified in CREATE TABLESPACE statements already exist, an error message from the failing CREATE TABLESPACE statement is issued, but the import job continues.

If this parameter is specified as y, a warning is issued and the existing datafiles are reinitialized. Be aware that specifying Y can result in a loss of data.

Example

The following is an example of using the REUSE_DATAFILES parameter. You can create the expfull.dmp dump file used in this example by running the example provided for the Export FULL parameter. See FULL on page 2-19.

```
> impdp hr/hr DIRECTORY=dpump_dir1 DUMPFILE=expfull.dmp LOGFILE=reuse.log
REUSE_DATAFILES=Y
```

This example reinitializes datafiles referenced by CREATE TABLESPACE statements in the expfull.dmp file.

SCHEMAS

Default: none

Purpose

Specifies that a schema-mode import is to be performed.

Syntax and Description

```
SCHEMAS=schema_name [,...]
```

If you have the IMP_FULL_DATABASE role, you can use this parameter to perform a schema-mode import by specifying a single schema other than your own or a list of schemas to import. First, the schemas themselves are created (if they do not already exist), including system and role grants, password history, and so on. Then all objects contained within the schemas are imported. Nonprivileged users can specify only their own schemas. In that case, no information about the schema definition is imported, only the objects contained within it.

Example

The following is an example of using the SCHEMAS parameter. You can create the expdat.dmp file used in this example by running the example provided for the Export SCHEMAS parameter. See SCHEMAS on page 2-29.

```
> impdp hr/hr SCHEMAS=hr,oe DIRECTORY=dpump_dir1 LOGFILE=schemas.log
DUMPFILE=expdat.dmp
```

The hr and oe schemas are imported from the expdat.dmp file. The log file, schemas.log, is written to dpump_dir1.

SKIP_UNUSABLE_INDEXES

Default: n

Purpose

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

Syntax and Description

```
SKIP_UNUSABLE_INDEXES={y | n}
```

If SKIP_UNUSABLE_INDEXES is set to y, then the import job does not load tables that have indexes that were previously set to an Index Unusable state. Other tables, not previously set Unusable, continue to be updated as rows are inserted.

This parameter enables 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.

If SKIP_UNUSABLE_INDEXES is set to n (the default) and an index in the Unusable state is encountered, the load of that table or partition will fail.

Example

The following is an example of using the SKIP_UNUSABLE_INDEXES parameter. You can create the expfull.dmp dump file used in this example by running the example provided for the Export FULL parameter. See FULL on page 2-19.

```
> impdp hr/hr DIRECTORY=dpump_dir1 DUMPFILE=expfull.dmp LOGFILE=skip.log
SKIP_UNUSABLE_INDEXES=y
```

SQLFILE

Default: none

Purpose

Specifies a file into which all of the SQL DDL that Import would have executed, based on other parameters, is written.

Syntax and Description

```
SQLFILE=[directory_object:]file_name
```

The *file_name* specifies where the import job will write the DDL that would be executed during the job. The SQL is not actually executed, and the target system remains unchanged. The file is written to the directory object specified in the DIRECTORY parameter, unless another *directory_object* is explicitly specified here. Any existing file that has a name matching the one specified with this parameter is overwritten.

Note that passwords are commented out in the SQL file. For example, if a CONNECT statement is part of the DDL that was executed, it will be commented out and the schema name will be shown but the password will not. In the following example, the dashes indicate that a comment follows, and the hr schema name is shown, but not the password.

```
-- CONNECT hr
```

Therefore, before you can execute the SQL file, you must edit it by removing the dashes indicating a comment and adding the password for the hr schema (in this case, the password is also hr), as follows:

```
CONNECT hr/hr
```

Example

The following is an example of using the SQLFILE parameter. You can create the expfull.dmp dump file used in this example by running the example provided for the Export FULL parameter. See FULL on page 2-19.

```
> impdp hr/hr DIRECTORY=dpump_dir1 DUMPFILE=expfull.dmp
SQLFILE=dpump_dir2:expfull.sql
```

A SQL file named expfull.sql is written to dpump_dir2.

STATUS

Default: 0

Purpose

Displays detailed status of the job, along with a description of the current operation. An estimated completion percentage for the job is also returned.

Syntax and Description

```
STATUS[=integer]
```

If you supply a value for *integer*, it specifies how frequently, in seconds, job status should be displayed in logging mode. If no value is entered or if the default value of 0 is used, information is displayed only upon completion of each object type, table, or partition

This status information is written only to your standard output device, not to the log file (if one is in effect).

Example

The following is an example of using the STATUS parameter. You can create the expfull.dmp dump file used in this example by running the example provided for the Export FULL parameter. See FULL on page 2-19.

```
> impdp hr/hr NOLOGFILE=y STATUS=120 DIRECTORY=dpump_dir1 DUMPFILE=expfull.dmp
```

In this example, the status is shown every two minutes (120 seconds).

STREAMS_CONFIGURATION

Default: y

Purpose

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

Syntax and Description

```
STREAMS_CONFIGURATION={y | n}
```

Example

The following is an example of using the STREAMS_CONFIGURATION parameter. You can create the expfull.dmp dump file used in this example by running the example provided for the Export FULL parameter. See FULL on page 2-19.

```
> impdp hr/hr DIRECTORY=dpump_dir1 DUMPFILE=expfull.dmp STREAMS_CONFIGURATION=n
```

See Also: *Oracle Streams Replication Administrator's Guide*

TABLE_EXISTS_ACTION

Default: SKIP (Note that if CONTENT=DATA_ONLY is specified, the default is APPEND, not SKIP.)

Purpose

Tells Import what to do if the table it is trying to create already exists.

Syntax and Description

```
TABLE_EXISTS_ACTION={SKIP | APPEND | TRUNCATE | REPLACE}
```

The possible values have the following effects:

- SKIP leaves the table as is and moves on to the next object. This is not a valid option if the CONTENT parameter is set to DATA_ONLY.
- APPEND loads rows from the source and leaves existing rows unchanged.
- TRUNCATE deletes existing rows and then loads rows from the source.
- REPLACE drops the existing table and then creates and loads it from the source. This is not a valid option if the CONTENT parameter is set to DATA_ONLY.

The following considerations apply when you are using these options:

- When you use TRUNCATE or REPLACE, make sure that rows in the affected tables are not targets of any referential constraints.

- When you use SKIP, APPEND, or TRUNCATE, existing table-dependent objects in the source, such as indexes, grants, triggers, and constraints, are ignored. For REPLACE, the dependent objects are dropped and re-created from the source, if they were not explicitly or implicitly excluded (using EXCLUDE) and they exist in the source dump file or system.
- When you use APPEND or TRUNCATE, checks are made to ensure that rows from the source are compatible with the existing table prior to performing any action.

  The existing table is loaded using the external tables access method because the external tables feature honors active constraints and triggers. However, be aware that if any row violates an active constraint, the load fails and no data is loaded.

  If you have data that must be loaded, but may cause constraint violations, consider disabling the constraints, loading the data, and then deleting the problem rows before reenabling the constraints.
- When you use APPEND, the data is always loaded into new space; existing space, even if available, is not reused. For this reason, you may wish to compress your data after the load.

Restrictions

TRUNCATE cannot be used on clustered tables or over network links.

Example

The following is an example of using the TABLE_EXISTS_ACTION parameter. You can create the expfull.dmp dump file used in this example by running the example provided for the Export FULL parameter. See FULL on page 2-19.

```
> impdp hr/hr TABLES=employees DIRECTORY=dpump_dir1 DUMPFILE=expfull.dmp
TABLE_EXISTS_ACTION=REPLACE
```

TABLES

Default: none

Purpose

Specifies that you want to perform a table-mode import.

Syntax and Description

TABLES=[*schema_name*.]*table_name*[:*partition_name*]

In a table-mode import, you can filter the data that is imported from the source by specifying a comma-delimited list of tables and partitions or subpartitions.

If you do not supply a *schema_name*, it defaults to that of the current user. To specify a schema other than your own, you must have the IMP_FULL_DATABASE role.

If a *partition_name* is specified, it must be the name of a partition or subpartition in the associated table.

The use of wildcards with table names is also supported. For example, TABLES=emp% would import all tables having names that start with 'EMP'.

Restrictions

The use of synonyms as values for the TABLES parameter is not supported. For example, if the regions table in the hr schema had a synonym of regn, it would not be valid to use TABLES=regn. An error would be returned.

If you specify more than one *table_name*, they must all reside in the same schema.

Example

The following example shows a simple use of the TABLES parameter to import only the employees and jobs tables from the expfull.dmp file. You can create the expfull.dmp dump file used in this example by running the example provided for the Export FULL parameter. See FULL on page 2-19.

```
> impdp hr/hr DIRECTORY=dpump_dir1 DUMPFILE=expfull.dmp TABLES=employees,jobs
```

The following example shows the use of the TABLES parameter to import partitions:

```
> impdp hr/hr DIRECTORY=dpump_dir1 DUMPFILE=expdat.dmp
TABLES=sh.sales:sales_Q1_2000,sh.sales:sales_Q2_2000
```

This example imports the partitions sales_Q1_2000 and sales_Q2_2000 for the table sales in the schema sh.

TABLESPACES

Default: none

Purpose

Specifies that you want to perform a tablespace-mode import.

Syntax and Description

```
TABLESPACES=tablespace_name [, ...]
```

Use TABLESPACES to specify a list of tablespace names whose tables and dependent objects are to be imported from the source (full, schema, tablespace, or table-mode export dump file set or another database).

Example

The following is an example of using the TABLESPACES parameter. It assumes that the tablespaces already exist. You can create the expfull.dmp dump file used in this example by running the example provided for the Export FULL parameter. See FULL on page 2-19.

```
> impdp hr/hr DIRECTORY=dpump_dir1 DUMPFILE=expfull.dmp TABLESPACES=tbs_1,tbs_2,
tbs_3,tbs_4
```

TRANSFORM

Default: none

Purpose

Enables you to alter object creation DDL for specific objects, as well as for all applicable objects being loaded.

Syntax and Description

```
TRANSFORM = transform_name:value[:object_type]
```

The *transform_name* specifies the name of the transform. There are two possible options:

- SEGMENT_ATTRIBUTES - if the value is specified as y, then segment attributes (physical attributes, storage attributes, tablespaces, and logging) are included, with appropriate DDL. The default is y.
- STORAGE - If the value is specified as y, the storage clauses are included, with appropriate DDL. The default is y. This parameter is ignored if SEGMENT_ATTRIBUTES=n.

The *value* specifies whether to include (y) or omit (n) segment or storage attributes (as specified by the *transform_name*) on applicable object DDL.

The *object_type* is optional. If supplied, it designates the object type to which the transform will be applied. If no object type is specified then the transform applies to all object types. The object type must be either TABLE or INDEX.

Example

For the following example, assume that you have exported the employees table in the hr schema. The SQL CREATE TABLE statement that results when you then import the table is similar to the following:

```
CREATE TABLE "HR"."EMPLOYEES"
   ( "EMPLOYEE_ID" NUMBER(6,0),
     "FIRST_NAME" VARCHAR2(20),
     "LAST_NAME" VARCHAR2(25) CONSTRAINT "EMP_LAST_NAME_NN" NOT NULL ENABLE,
     "EMAIL" VARCHAR2(25) CONSTRAINT "EMP_EMAIL_NN" NOT NULL ENABLE,
     "PHONE_NUMBER" VARCHAR2(20),
     "HIRE_DATE" DATE CONSTRAINT "EMP_HIRE_DATE_NN" NOT NULL ENABLE,
     "JOB_ID" VARCHAR2(10) CONSTRAINT "EMP_JOB_NN" NOT NULL ENABLE,
     "SALARY" NUMBER(8,2),
     "COMMISSION_PCT" NUMBER(2,2),
     "MANAGER_ID" NUMBER(6,0),
     "DEPARTMENT_ID" NUMBER(4,0)
   ) PCTFREE 10 PCTUSED 40 INITRANS 1 MAXTRANS 255 NOCOMPRESS LOGGING
  STORAGE(INITIAL 10240 NEXT 16384 MINEXTENTS 1 MAXEXTENTS 121
  PCTINCREASE 50 FREELISTS 1 FREELIST GROUPS 1 BUFFER_POOL DEFAULT)
  TABLESPACE "SYSTEM" ;
```

If you do not want to retain the STORAGE clause or TABLESPACE clause, you can remove them from the CREATE STATEMENT by using the Import TRANSFORM parameter. Specify the value of SEGMENT_ATTRIBUTES as n. This results in the exclusion of segment attributes (both storage and tablespace) from the table.

```
> impdp hr/hr TABLES=hr.employees \
  DIRECTORY=dpump_dir1 DUMPFILE=hr_emp.dmp \
  TRANSFORM=SEGMENT_ATTRIBUTES:n:table
```

The resulting CREATE TABLE statement for the employees table would then look similar to the following. It does not contain a STORAGE or TABLESPACE clause; the attributes for the default tablespace for the HR schema will be used instead.

```
CREATE TABLE "HR"."EMPLOYEES"
   ( "EMPLOYEE_ID" NUMBER(6,0),
     "FIRST_NAME" VARCHAR2(20),
```

```
  "LAST_NAME" VARCHAR2(25) CONSTRAINT "EMP_LAST_NAME_NN" NOT NULL ENABLE,
  "EMAIL" VARCHAR2(25) CONSTRAINT "EMP_EMAIL_NN" NOT NULL ENABLE,
  "PHONE_NUMBER" VARCHAR2(20),
  "HIRE_DATE" DATE CONSTRAINT "EMP_HIRE_DATE_NN" NOT NULL ENABLE,
  "JOB_ID" VARCHAR2(10) CONSTRAINT "EMP_JOB_NN" NOT NULL ENABLE,
  "SALARY" NUMBER(8,2),
  "COMMISSION_PCT" NUMBER(2,2),
  "MANAGER_ID" NUMBER(6,0),
  "DEPARTMENT_ID" NUMBER(4,0)
 );
```

As shown in the previous example, the SEGMENT_ATTRIBUTES transform applies to both storage and tablespace attributes. To omit only the STORAGE clause and retain the TABLESPACE clause, you can use the STORAGE transform, as follows:

```
> impdp hr/hr TABLES=hr.employees \
  DIRECTORY=dpump_dir1 DUMPFILE=hr_emp.dmp \
  TRANSFORM=STORAGE:n:table
```

The SEGMENT_ATTRIBUTES and STORAGE transforms can be applied to all applicable table and index objects by not specifying the object type on the TRANSFORM parameter, as shown in the following command:

```
> impdp hr/hr DIRECTORY=dpump_dir1 DUMPFILE=hr.dmp \
 SCHEMAS=hr TRANSFORM=SEGMENT_ATTRIBUTES:n
```

TRANSPORT_DATAFILES

Default: none

Purpose

Specifies a list of datafiles to be imported into the target database by a transportable-mode import. The files must already have been copied from the source database system.

Syntax and Description

```
TRANSPORT_DATAFILES=datafile_name
```

The *datafile_name* must include an absolute directory path specification (*not* a directory object name) that is valid on the system where the target database resides.

Example

The following is an example of using the TRANSPORT_DATAFILES parameter.

```
> impdp hr/hr DIRECTORY=dpump_dir1 DUMPFILE=tts.dmp
TRANSPORT_DATAFILES='/user01/data/tbs1.f'
```

TRANSPORT_FULL_CHECK

Default: n

Purpose

Specifies whether or not to verify that the specified transportable tablespace set has no dependencies.

Syntax and Description

```
TRANSPORT_FULL_CHECK={y | n}
```

If TRANSPORT_FULL_CHECK=y, then Import verifies that there are no dependencies between those objects inside the transportable set and those outside the transportable set. The check addresses two-way dependencies. For example, if a table is inside the transportable set but its index is not, a failure is returned and the import operation is terminated. Similarly, a failure is also returned if an index is in the transportable set but the table is not.

If TRANSPORT_FULL_CHECK=n, then Import verifies only that there are no objects within the transportable set that are dependent on objects outside the transportable set. This check addresses a one-way dependency. For example, a table is not dependent on an index, but an index *is* dependent on a table, because an index without a table has no meaning. Therefore, if the transportable set contains a table, but not its index, then this check succeeds. However, if the transportable set contains an index, but not the table, the import operation is terminated.

In addition to this check, Import always verifies that all storage segments of all tables (and their indexes) defined within the tablespace set specified by TRANSPORT_TABLESPACES are actually contained within the tablespace set.

Restrictions

This parameter is valid for transportable mode only when the NETWORK_LINK parameter is specified.

Example

In the following example, *source_database_link* would be replaced with the name of a valid database link. The example also assumes that a datafile named tbs6.f already exists.

```
> impdp hr/hr DIRECTORY=dpump_dir1 TRANSPORT_TABLESPACES=tbs_6
NETWORK_LINK=source_database_link TRANSPORT_FULL_CHECK=y
TRANSPORT_DATAFILES='/wkdir/data/tbs6.f'
```

TRANSPORT_TABLESPACES

Default: none

Purpose

Specifies that you want to perform a transportable-tablespace-mode import.

Syntax and Description

```
TRANSPORT_TABLESPACES=tablespace_name [, ...]
```

Use the TRANSPORT_TABLESPACES parameter to specify a list of tablespace names for which object metadata will be imported from the source database into the target database.

Example

In the following example, the *source_database_link* would be replaced with the name of a valid database link. The example also assumes that a datafile named tbs6.f already exists.

```
> impdp hr/hr DIRECTORY=dpump_dir1 NETWORK_LINK=source_database_link
TRANSPORT_TABLESPACES=tbs_6 TRANSPORT_FULL_CHECK=n
TRANSPORT_DATAFILES='user01/data/tbs6.f'
```

VERSION

Default: COMPATIBLE

Purpose

Specifies the version of database objects to be imported.

Syntax and Description

```
VERSION={COMPATIBLE | LATEST | version_string}
```

This parameter can be used to load a target system whose Oracle database is at an earlier version than that of the source system. Database objects or attributes on the source system that are incompatible with the specified version will not be moved to the target. For example, tables containing new datatypes that are not supported in the specified version will not be imported. Legal values for this parameter are as follows:

- COMPATIBLE - This is the default value. The version of the metadata corresponds to the database compatibility level. Database compatibility must be set to 9.2.0 or higher.
- LATEST - The version of the metadata corresponds to the database version.
- *version_string* - A specific database version (for example, 10.0.0). In Oracle Database 10*g*, this value cannot be lower than 10.0.0.

Example

The following is an example of using the VERSION parameter. You can create the expfull.dmp dump file used in this example by running the example provided for the Export FULL parameter. See FULL on page 2-19.

```
> impdp hr/hr DIRECTORY=dpump_dir1 DUMPFILE=expfull.dmp TABLES=employees
VERSION=LATEST
```

How Data Pump Import Parameters Map to Those of the Original Import Utility

Table 3–1 maps, as closely as possible, Data Pump Import parameters to original Import parameters. In some cases, because of feature redesign, the original Import parameter is no longer needed so there is no Data Pump command to compare it to. Also, as shown in the table, some of the parameter names may be the same, but the functionality is slightly different.

Table 3–1 Original Import Parameters and Their Counterparts in Data Pump Import

Original Import Parameter	Comparable Data Pump Import Parameter
BUFFER	A parameter comparable to BUFFER is not needed.
CHARSET	A parameter comparable to CHARSET is not needed.
COMMIT	A parameter comparable to COMMIT is not supported.
COMPILE	A parameter comparable to COMPILE is not supported.

Table 3–1 (Cont.) Original Import Parameters and Their Counterparts in Data Pump

Original Import Parameter	Comparable Data Pump Import Parameter
CONSTRAINTS	EXCLUDE=CONSTRAINT and INCLUDE=CONSTRAINT
DATAFILES	TRANSPORT_DATAFILES
DESTROY	REUSE_DATAFILES
FEEDBACK	STATUS
FILE	DUMPFILE
FILESIZE	Not necessary. It is included in the dump file set.
FROMUSER	SCHEMAS
FULL	FULL
GRANTS	EXCLUDE=GRANT and INCLUDE=GRANT
HELP	HELP
IGNORE	TABLE_EXISTS_ACTION
INDEXES	EXCLUDE=INDEX and INCLUDE=INDEX
INDEXFILE	SQLFILE
LOG	LOGFILE
PARFILE	PARFILE
RECORDLENGTH	A parameter comparable to RECORDLENGTH is not needed.
RESUMABLE	A parameter comparable to RESUMABLE is not needed. It is automatically defaulted.
RESUMABLE_NAME	A parameter comparable to RESUMABLE_NAME is not needed. It is automatically defaulted.
RESUMABLE_TIMEOUT	A parameter comparable to RESUMABLE_TIMEOUT is not needed. It is automatically defaulted.
ROWS=N	CONTENT=METADATA_ONLY
ROWS=Y	CONTENT=ALL
SHOW	SQLFILE
SKIP_UNUSABLE_INDEXES	SKIP_UNUSABLE_INDEXES
STATISTICS	A parameter comparable to STATISTICS is not needed. If the source table has statistics, they are imported.

Table 3–1 (Cont.) Original Import Parameters and Their Counterparts in Data Pump

Original Import Parameter	Comparable Data Pump Import Parameter
STREAMS_CONFIGURATION	STREAMS_CONFIGURATION
STREAMS_INSTANTIATION	A parameter comparable to STREAMS_INSTANTIATION is not needed.
TABLES	TABLES
TABLESPACES	This parameter still exists, but some of its functionality is now performed using the TRANSPORT_TABLESPACES parameter.
TOID_NOVALIDATE	A command comparable to TOID_NOVALIDATE is not needed. OIDs are no longer used for type validation.
TOUSER	REMAP_SCHEMA
TRANSPORT_TABLESPACE	TRANSPORT_TABLESPACES (see command description)
TTS_OWNERS	A parameter comparable to TTS_OWNERS is not needed because the information is stored in the dump file set.
USERID	A parameter comparable to USERID is not needed. This information is supplied as the *username*/*password* when you invoke Import.
VOLSIZE	A parameter comparable to VOLSIZE is not needed because tapes are not supported.

This table does not list all Data Pump Import command-line parameters. For information about all Import command-line parameters, see Commands Available in Import's Interactive-Command Mode on page 3-44.

Commands Available in Import's Interactive-Command Mode

In interactive-command mode, the current job continues running, but logging to the terminal is stopped and the Import prompt is displayed.

> **Note:** Data Pump Import interactive-command mode is different from the interactive mode for original Import, in which Import prompted you for input. See Interactive Mode on page 20-8 for information about interactive mode in original Import.

To start interactive-command mode, do one of the following:

- From an attached client, press Ctrl+C.
- From a terminal other than the one on which the job is running, use the ATTACH parameter to attach to the job. This is a useful feature in situations in which you start a job at one location and need to check on it at a later time from a different location.

Table 3–2 lists the activities you can perform for the current job from the Data Pump Import prompt in interactive-command mode.

Table 3–2 Supported Activities in Data Pump Import's Interactive-Command Mode

Activity	Command Used
Exit interactive-command mode	CONTINUE_CLIENT on page 3-45
Stop the import client session, but leave the current job running	EXIT_CLIENT on page 3-46
Display a summary of available commands	HELP on page 3-46
Detach all currently attached client sessions and kill the current job	KILL_JOB on page 3-47
Increase or decrease the number of active worker processes for the current job. This command is valid only in the Enterprise Edition of Oracle Database 10*g*.	PARALLEL on page 3-47
Restart a stopped job to which you are attached	START_JOB on page 3-48
Display detailed status for the current job	STATUS on page 3-48
Stop the current job	STOP_JOB on page 3-49

The following are descriptions of the commands available in the interactive-command mode of Data Pump Import.

CONTINUE_CLIENT

Purpose

Changes the mode from interactive-command mode to logging mode.

Syntax and Description

```
CONTINUE_CLIENT
```

In logging mode, the job status is continually output to the terminal. If the job is currently stopped, then CONTINUE_CLIENT will also cause the client to attempt to start the job.

Example

```
impdp> CONTINUE_CLIENT
```

EXIT_CLIENT

Purpose

Stops the import client session, exits Import, and discontinues logging to the terminal, but leaves the current job running.

Syntax and Description

```
EXIT_CLIENT
```

Because EXIT_CLIENT stops the client session but leaves the job running, you can attach to the job at a later time if it is still executing or in a stopped state. To see the status of the job, you can monitor the log file for the job or you can query the USER_DATAPUMP_JOBS view or the V$SESSION_LONGOPS view.

Example

```
impdp> EXIT_CLIENT
```

HELP

Purpose

Provides information about Data Pump Import commands available in interactive-command mode.

Syntax and Description

```
HELP
```

Displays information about the commands available in interactive-command mode.

Example

```
impdp> HELP
```

KILL_JOB

Purpose

Detaches all currently attached client sessions and then kills the current job. It exits Import and returns to the terminal prompt.

Syntax and Description

```
KILL_JOB
```

A job that is killed using KILL_JOB cannot be restarted. All attached clients, including the one issuing the KILL_JOB command, receive a warning that the job is being killed by the current user and are then detached. After all clients are detached, the job's process structure is immediately run down and the master table and dump files are deleted. Log files are not deleted.

Example

```
impdp> KILL_JOB
```

PARALLEL

Purpose

Enables you to increase or decrease the number of active worker processes for the current job.

Syntax and Description

```
PARALLEL=integer
```

PARALLEL is available as both a command-line parameter and an interactive-mode parameter. You set it to the desired number of parallel processes. An increase takes effect immediately if there are sufficient files and resources. A decrease does not take effect until an existing process finishes its current task. If the integer value is decreased, workers are idled but not deleted until the job exits.

> **See Also:** PARALLEL on page 3-23 for more information about parallelism

Example

```
PARALLEL=10
```

START_JOB

Purpose

Starts the current job to which you are attached.

Syntax and Description

```
START_JOB[=SKIP_CURRENT]
```

The START_JOB command restarts the job to which you are currently attached (the job cannot be currently executing). The job is restarted with no data loss or corruption after an unexpected failure or after you issue a STOP_JOB command, provided the dump file set and master table remain undisturbed.

The SKIP_CURRENT option allows you to restart a job that previously failed to restart because execution of some DDL statement failed. The failing statement is skipped and the job is restarted from the next work item.

Transportable-tablespace-mode imports are not restartable.

Example

```
impdp> START_JOB
```

STATUS

Purpose

Displays the cumulative status of the job, along with a description of the current operation. A completion percentage for the job is also returned.

Syntax and Description

```
STATUS[=integer]
```

You have the option of specifying how frequently, in seconds, this status should be displayed in logging mode. If no value is entered or if the default value of 0 is used, the periodic status display is turned off and status is displayed only once.

This status information is written only to your standard output device, not to the log file (even if one is in effect).

Example

The following example will display the status every two minutes (120 seconds).

```
STATUS=120
```

STOP_JOB

Purpose

Stops the current job either immediately or after an orderly shutdown, and exits Import.

Syntax and Description

```
STOP_JOB[=IMMEDIATE]
```

If the master table and dump file set are not disturbed when or after the STOP_JOB command is issued, the job can be attached to and restarted at a later time with the START_JOB command.

To perform an orderly shutdown, use STOP_JOB (without any associated value). A warning requiring confirmation will be issued. An orderly shutdown stops the job after worker processes have finished their current tasks.

To perform an immediate shutdown, specify STOP_JOB=IMMEDIATE. A warning requiring confirmation will be issued. All attached clients, including the one issuing the STOP_JOB command, receive a warning that the job is being stopped by the current user and they will be detached. After all clients are detached, the process structure of the job is immediately run down. That is, the master process will not wait for the worker processes to finish their current tasks. There is no risk of corruption or data loss when you specify STOP_JOB=IMMEDIATE. However, some tasks that were incomplete at the time of shutdown may have to be redone at restart time.

Example

```
impdp> STOP_JOB=IMMEDIATE
```

Examples of Using Data Pump Import

This section provides examples of the following ways in which you might use Data Pump Import:

- Performing a Data-Only Table-Mode Import
- Performing a Schema-Mode Import
- Performing a Network-Mode Import

For information that will help you to successfully use these examples, see Using the Import Parameter Examples on page 3-7.

Performing a Data-Only Table-Mode Import

Example 3–1 shows how to perform a data-only table-mode import of the table named `employees`. It uses the dump file created in Example 2–1.

Example 3–1 Performing a Data-Only Table-Mode Import

```
> impdp hr/hr TABLES=employees CONTENT=DATA_ONLY DUMPFILE=dpump_dir1:table.dmp
NOLOGFILE=y
```

The `CONTENT=DATA_ONLY` parameter filters out any database object definitions (metadata). Only table row data is loaded.

Performing a Schema-Mode Import

Example 3–2 shows a schema-mode import of the dump file set created in Example 2–4.

Example 3–2 Performing a Schema-Mode Import

```
> impdp hr/hr SCHEMAS=hr DIRECTORY=dpump_dir1 DUMPFILE=expschema.dmp
 EXCLUDE=CONSTRAINT, REF_CONSTRAINT, INDEX TABLE_EXISTS_ACTION=REPLACE
```

The `EXCLUDE` parameter filters the metadata that is imported. For the given mode of import, all the objects contained within the source, and all their dependent objects, are included except those specified in an `EXCLUDE` statement. If an object is excluded, all of its dependent objects are also excluded.

The `TABLE_EXISTS_ACTION=REPLACE` parameter tells Import to drop the table if it already exists and to then re-create and load it using the dump file contents.

Performing a Network-Mode Import

Example 3–3 performs a network-mode import where the source is the database defined by the `NETWORK_LINK` parameter.

Example 3–3 Network-Mode Import of Schemas

```
> impdp hr/hr TABLES=employees REMAP_SCHEMA=hr:scott DIRECTORY=dpump_dir1
NETWORK_LINK=dblink
```

This example imports the `employees` table from the `hr` schema into the `scott` schema. `The dblink` references a source database that is different than the target database.

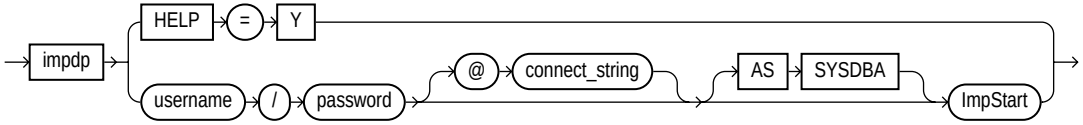
`REMAP_SCHEMA` loads all the objects from the source schema into the target schema.

> **See Also:** NETWORK_LINK on page 3-22 for more information about database links

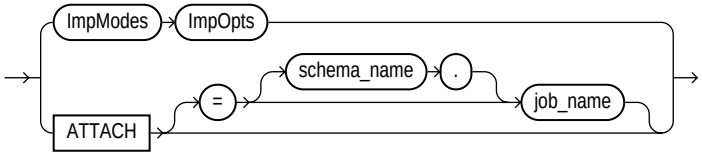
Syntax Diagrams for Data Pump Import

This section provides syntax diagrams for Data Pump Import. These diagrams use standard SQL syntax notation. For more information about SQL syntax notation, see *Oracle Database SQL Reference*.

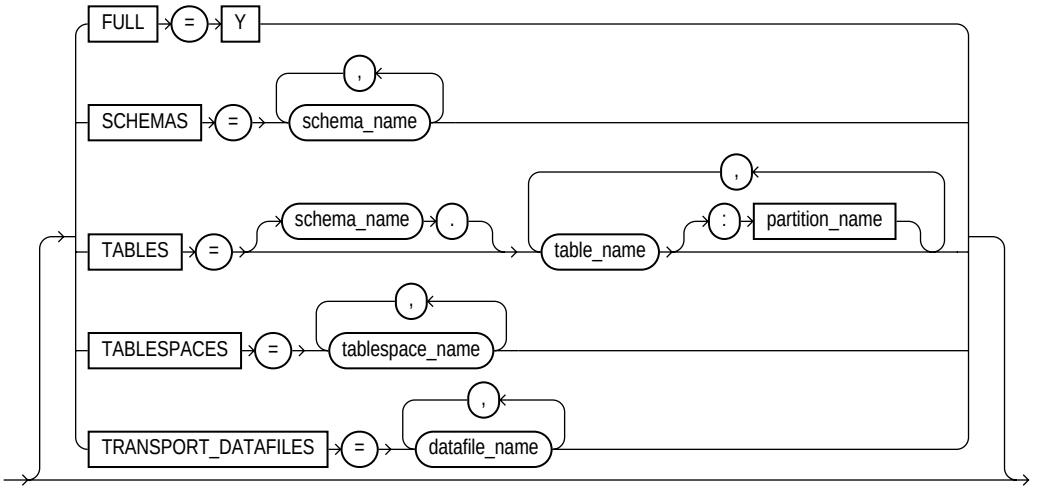
ImpInit



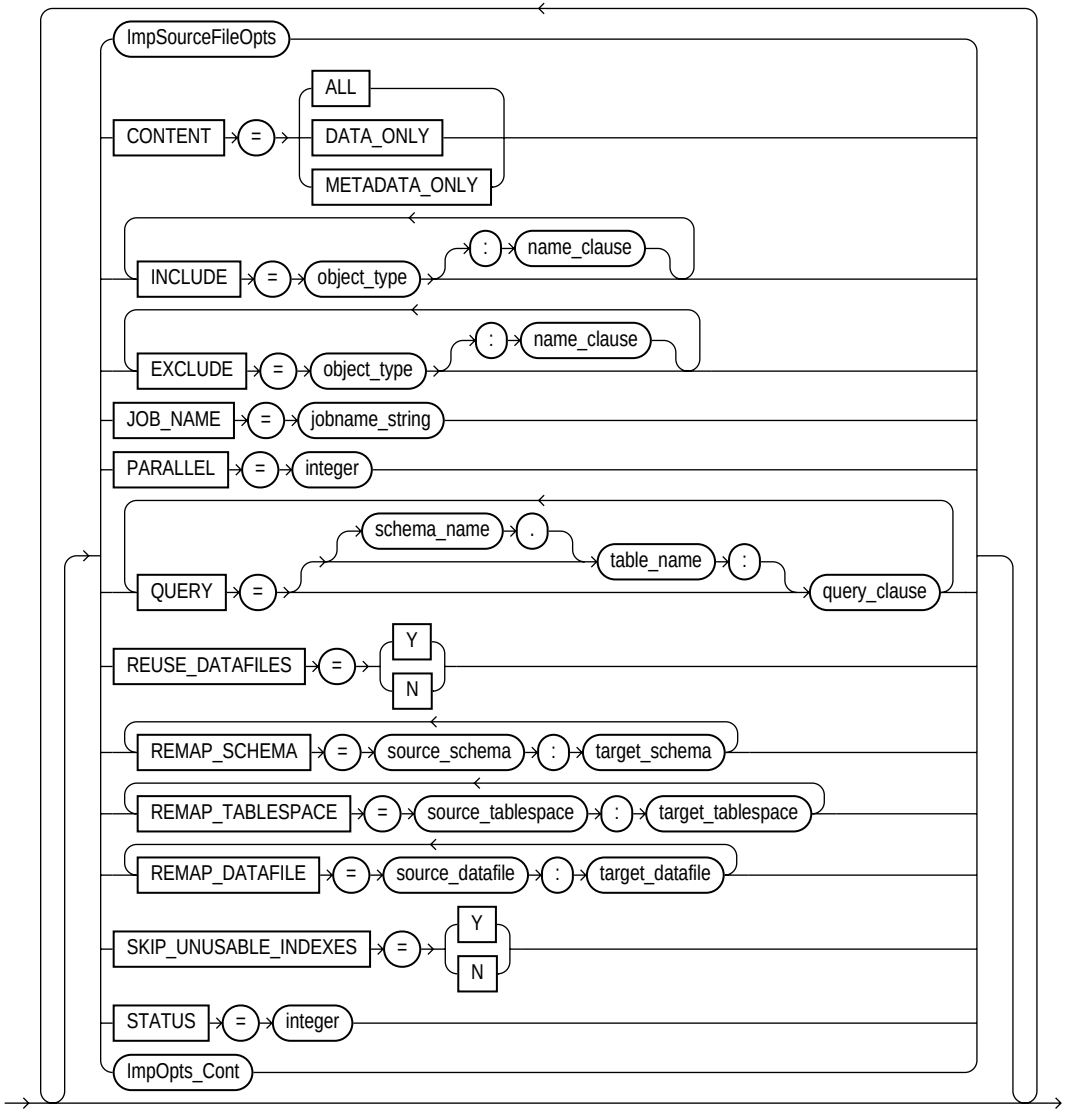
ImpStart



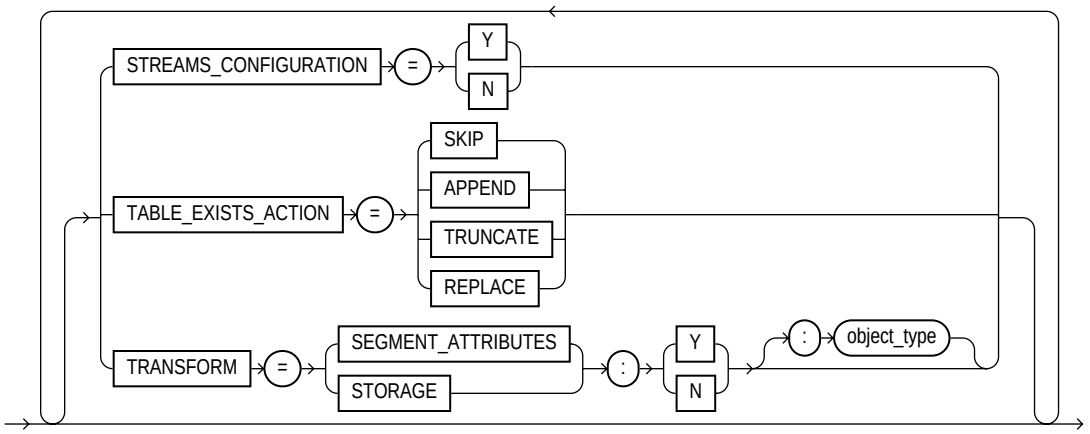
ImpModes


FULL
=
Y
SCHEMAS
=
,
schema_name
TABLES
=
schema_name
.
,
table_name
:
partition_name
TABLESPACES
=
,
tablespace_name
TRANSPORT_DATAFILES
=
,
datafile_name

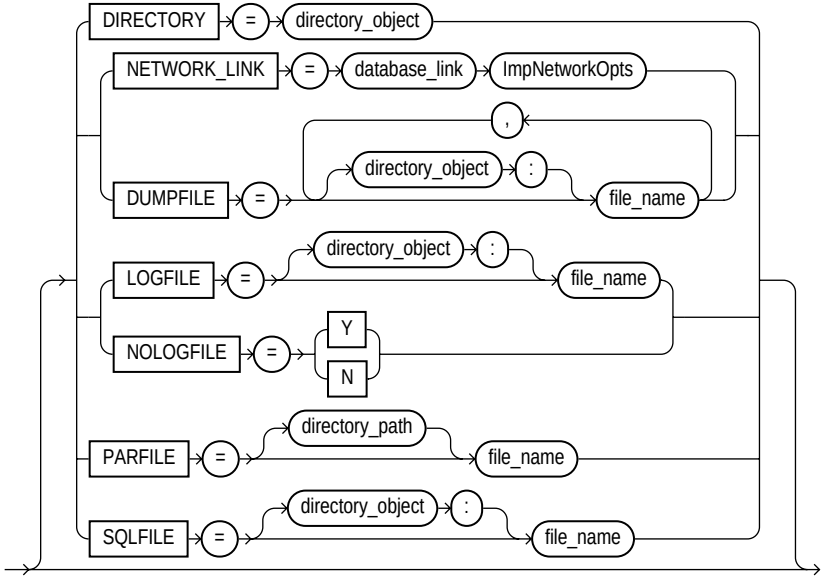
ImpOpts


ImpSourceFileOpts
CONTENT
=
ALL
DATA_ONLY
METADATA_ONLY
INCLUDE
=
object_type
:
name_clause
EXCLUDE
=
object_type
:
name_clause
JOB_NAME
=
jobname_string
PARALLEL
=
integer
QUERY
=
schema_name
.
table_name
:
query_clause
REUSE_DATAFILES
=
Y
N
REMAP_SCHEMA
=
source_schema
:
target_schema
REMAP_TABLESPACE
=
source_tablespace
:
target_tablespace
REMAP_DATAFILE
=
source_datafile
:
target_datafile
SKIP_UNUSABLE_INDEXES
=
Y
N
STATUS
=
integer
ImpOpts_Cont

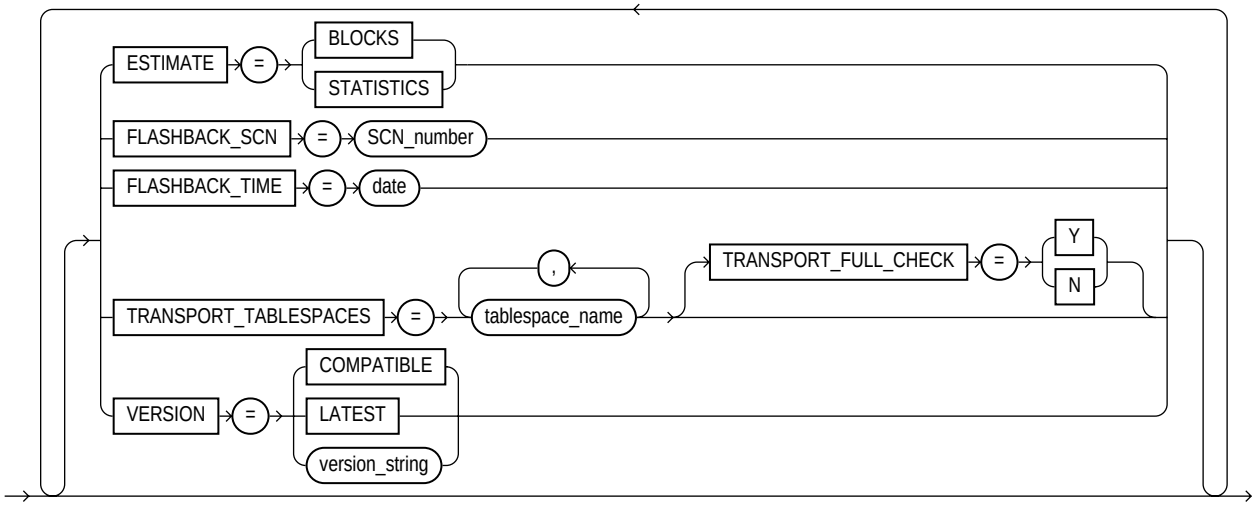
ImpOpts_Cont



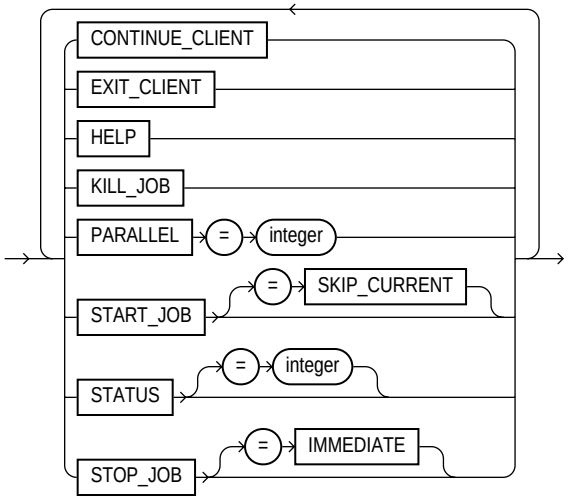
ImpSourceFileOpts



ImpNetworkOpts



ImpDynOpts



4

Data Pump Performance

The Data Pump utilities are designed especially for very large databases. If your site has very large quantities of data versus metadata, you should experience a dramatic increase in performance compared to the original Export and Import utilities. This chapter briefly discusses why the performance is better and also suggests specific steps you can take to enhance performance of export and import operations.

This chapter contains the following sections:

Performance of metadata extraction and database object creation in Data Pump Export and Import remains essentially equivalent to that of the original Export and Import utilities.

Data Performance Improvements for Data Pump Export and Import

The improved performance of the Data Pump Export and Import utilities is attributable to several factors, including the following:

- Multiple worker processes can perform intertable and interpartition parallelism to load and unload tables in multiple, parallel, direct-path streams.
- For very large tables and partitions, single worker processes can choose intrapartition parallelism through multiple parallel queries and parallel DML I/O server processes when the external tables method is used to access data.
- Data Pump uses parallelism to build indexes and load package bodies.

- Dump files are read and written directly by the server and, therefore, do not require any data movement to the client.
- The dump file storage format is the internal stream format of the direct path API. This format is very similar to the format stored in Oracle database datafiles inside of tablespaces. Therefore, no client-side conversion to `INSERT` statement bind variables is performed.
- The supported data access methods, direct path and external tables, are faster than conventional SQL. The direct path API provides the fastest single-stream performance. The external tables feature makes efficient use of the parallel queries and parallel DML capabilities of the Oracle database.
- Metadata and data extraction can be overlapped during export.

Tuning Performance

Data Pump technology fully uses all available resources to maximize throughput and minimize elapsed job time. For this to happen, a system must be well-balanced across CPU, memory, and I/O. In addition, standard performance tuning principles apply. For example, for maximum performance you should ensure that the files that are members of a dump file set reside on separate disks, because the dump files will be written and read in parallel. Also, the disks should not be the same ones on which the source or target tablespaces reside.

Any performance tuning activity involves making trade-offs between performance and resource consumption.

Controlling Resource Consumption

The Data Pump Export and Import utilities enable you to dynamically increase and decrease resource consumption for each job. This is done using the PARALLEL parameter to specify a degree of parallelism for the job. (The PARALLEL parameter is the only tuning parameter that is specific to Data Pump.) For maximum throughput, do not set PARALLEL to much more than 2x the CPU count.

See Also:

- PARALLEL on page 2-25 for more information about the Export PARALLEL parameter
- PARALLEL on page 3-23 for more information about the Import PARALLEL parameter

As you increase the degree of parallelism, CPU usage, memory consumption, and I/O bandwidth usage also increase. You must ensure that adequate amounts of these resources are available. If necessary, you can distribute files across different disk devices or channels to get the needed I/O bandwidth.

The PARALLEL parameter is valid only in the Enterprise Edition of Oracle Database 10*g*.

Initialization Parameters That Affect Data Pump Performance

The settings for certain initialization parameters can affect the performance of Data Pump Export and Import. In particular, you can try using the following settings to improve performance, although the effect may not be the same on all platforms.

- DISK_ASYNCH_IO=TRUE
- DB_BLOCK_CHECKING=FALSE
- DB_BLOCK_CHECKSUM=FALSE

Additionally, the following initialization parameters must have values set high enough to allow for maximum parallelism:

- PROCESSES
- SESSIONS
- PARALLEL_MAX_SERVERS

5

The Data Pump API

The Data Pump API, DBMS_DATAPUMP, provides a high-speed mechanism to move all or part of the data and metadata for a site from one database to another. The Data Pump Export and Data Pump Import utilities are based on the Data Pump API.

You should read this chapter if you want more details about how the Data Pump API works. The following topics are covered:

- How Does the Client Interface to the Data Pump API Work?
- What Are the Basic Steps in Using the Data Pump API?
- Examples of Using the Data Pump API

See Also:

- *PL/SQL Packages and Types Reference* for a detailed description of the procedures available in the DBMS_DATAPUMP package
- Chapter 1, "Overview of Oracle Data Pump" for additional explanation of Data Pump concepts

How Does the Client Interface to the Data Pump API Work?

The main structure used in the client interface is a job handle, which appears to the caller as an integer. Handles are created using the DBMS_DATAPUMP.OPEN or DBMS_DATAPUMP.ATTACH function. Other sessions can attach to a job to monitor and control its progress. This allows a DBA to start up a job before departing from work and then watch the progress of the job from home. Handles are session specific. The same job can create different handles in different sessions.

Job States

There is a state associated with each phase of a job, as follows:

- Undefined - before a handle is created
- Defining - when the handle is first created
- Executing - when the DBMS_DATAPUMP.START_JOB procedure is executed
- Completing - when the job is completing
- Completed - when the job is completed
- Stop Pending - when an orderly job shutdown has been requested
- Stopping - when the job is stopping
- Stopped - when DBMS_DATAPUMP.STOP_JOB is performed against an executing job
- Idling - for a restarted job, the period before a START_JOB is executed
- Not Running - when a master table exists for a job that is not running (has no Data Pump processes associated with it)

Performing DBMS_DATAPUMP.START_JOB on a job in an Idling state will return it to an Executing state.

If all users execute DBMS_DATAPUMP.DETACH to detach from a job in the Defining state, the job will be totally removed from the database.

When a job abnormally terminates or when an instance running the job is shut down, the job is placed in the Not Running state if it was previously executing or idling. It can then be restarted by the user.

The master control process is active in the Defining, Idling, Executing, Stopping, Stop Pending, and Completing states. It is also active briefly in the Stopped and Completed states. The master table for the job exists in all states except the Undefined state. Worker processes are only active in the Executing and Stop Pending states.

Detaching while a job is in the Executing state will not halt the job. You can be explicitly or implicitly detached from a job. An explicit detachment occurs when you execute the DBMS_DATAPUMP.DETACH procedure. An implicit detachment occurs when a session is run down, an instance is started, or the STOP_JOB procedure is called.

The Not Running state indicates that a master table exists outside the context of an executing job. This will occur if a master table has been explicitly retained upon job

completion, if a job has been stopped (probably to be restarted later), or if a job has abnormally terminated. This state can also be seen momentarily during job state transitions at the beginning of a job, and at the end of a job before the master table is dropped. Note that the Not Running state is shown only in the DBA_DATAPUMP_JOBS view and the USER_DATAPUMP_JOBS view. It is never shown in the master table or returned by the GET_STATUS procedure.

Table 5–1 shows the valid job states in which DBMS_DATAPUMP procedures can be executed. The states listed are valid for both export and import jobs, unless otherwise noted.

Table 5–1 Valid Job States in Which DBMS_DATAPUMP Procedures Can Be Executed

Procedure Name	Valid States	Description
ADD_FILE	Defining (valid for both export and import jobs)	Specifies a file for the dump file set, the log file, or the SQL_FILE output.
	Executing and Idling (valid only for specifying dump files for export jobs)	
ATTACH	Defining, Executing, Idling, Stopped, Completed	Allows a user session to monitor a job or to continue a stopped job.
DATA_FILTER	Defining	Restricts data processed by a job.
DETACH	All	Disconnects a user session from a job.
GET_STATUS	All	Obtains the status of a job.
LOG_ENTRY	Defining, Executing, Idling, Completing	Adds an entry to the log file.
METADATA_FILTER	Defining	Restricts metadata processed by a job.
METADATA_REMAP	Defining	Remaps metadata processed by a job.
METADATA_TRANSFORM	Defining	Alters metadata processed by a job.
OPEN	Undefined	Creates a new job.
SET_PARALLEL	Defining, Executing, Idling	Specifies parallelism for a job.
SET_PARAMETER	Defining	Alters default processing by a job.
START_JOB	Defining, Idling	Begins or resumes execution of a job.
STOP_JOB	Defining, Stopping, Executing, Idling	Initiates shutdown of a job.

What Are the Basic Steps in Using the Data Pump API?

To use the Data Pump API, you use the procedures provided in the DBMS_DATAPUMP package. The following steps list the basic activities involved in using the Data Pump API. The steps are presented in the order in which the activities would generally be performed:

1. Execute the DBMS_DATAPUMP.OPEN procedure to create a Data Pump job and its infrastructure.
2. Define any parameters for the job.
3. Start the job.
4. Optionally, monitor the job until it completes.
5. Optionally, detach from the job and reattach at a later time.
6. Optionally, stop the job.
7. Optionally, restart the job, if desired.

These concepts are illustrated in the examples provided in the next section.

> **See Also:** *PL/SQL Packages and Types Reference* for a complete description of the DBMS_DATAPUMP package

Examples of Using the Data Pump API

This section provides the following examples to help you get started using the Data Pump API:

- Example 5–1, "Performing a Simple Schema Export"
- Example 5–2, "Importing a Dump File and Remapping All Schema Objects"
- Example 5–3, "Using Exception Handling During a Simple Schema Export"

The examples are in the form of PL/SQL scripts. If you choose to copy these scripts and run them, you must first do the following, using SQL*Plus:

- Create a directory object and grant READ and WRITE access to it. For example, to create a directory object named dmpdir to which you have access, do the following. Replace *user* with your username.

```
SQL> CREATE DIRECTORY dmpdir AS '/rdbms/work';
SQL> GRANT READ, WRITE ON DIRECTORY dmpdir TO user
```

- Ensure that you have the EXP_FULL_DATABASE and IMP_FULL_DATABASE roles. To see a list of all roles assigned to you within your security domain, do the following:

  ```
  SQL> SELECT * FROM SESSION_ROLES;
  ```

  If you do not have the necessary roles assigned to you, contact your system administrator for help.

- Turn on server output if it is not already on. This is done as follows:

  ```
  SQL> SET SERVEROUTPUT ON
  ```

  If you do not do this, you will not see any output to your screen. You must do this in the same session in which you invoke the example. Also, if you exit SQL*Plus, this setting is lost and must be reset when you begin a new session.

Example 5–1 Performing a Simple Schema Export

The PL/SQL script provided in this example shows how to use the Data Pump API to perform a simple schema export of the HR schema. It shows how to create a job, start it, and monitor it. Additional information about the example is contained in the comments within the script. To keep the example simple, exceptions from any of the API calls will not be trapped. However, in a production environment, Oracle recommends that you define exception handlers and call GET_STATUS to retrieve more detailed error information when a failure occurs.

```
DECLARE
  ind NUMBER;              -- Loop index
  h1 NUMBER;               -- Data Pump job handle
  percent_done NUMBER;     -- Percentage of job complete
  job_state VARCHAR2(30);  -- To keep track of job state
  le ku$_LogEntry;         -- For WIP and error messages
  js ku$_JobStatus;        -- The job status from get_status
  jd ku$_JobDesc;          -- The job description from get_status
  sts ku$_Status;          -- The status object returned by get_status
BEGIN

-- Create a (user-named) Data Pump job to do a schema export.

  h1 := DBMS_DATAPUMP.OPEN('EXPORT','SCHEMA',NULL,'EXAMPLE1','LATEST');

-- Specify a single dump file for the job (using the handle just returned)
-- and a directory object, which must already be defined and accessible
```

```
-- to the user running this procedure.

  DBMS_DATAPUMP.ADD_FILE(h1,'example1.dmp','DMPDIR');

-- A metadata filter is used to specify the schema that will be exported.

  DBMS_DATAPUMP.METADATA_FILTER(h1,'SCHEMA_EXPR','IN (''HR'')');

-- Start the job. An exception will be generated if something is not set up
-- properly.

  DBMS_DATAPUMP.START_JOB(h1);

-- The export job should now be running. In the following loop, the job
-- is monitored until it completes. In the meantime, progress information is
-- displayed.

  percent_done := 0;
  job_state := 'UNDEFINED';
  while (job_state != 'COMPLETED') and (job_state != 'STOPPED') loop
    dbms_datapump.get_status(h1,
           dbms_datapump.ku$_status_job_error +
           dbms_datapump.ku$_status_job_status +
           dbms_datapump.ku$_status_wip,-1,job_state,sts);
    js := sts.job_status;

-- If the percentage done changed, display the new value.

    if js.percent_done != percent_done
    then
      dbms_output.put_line('*** Job percent done = ' ||
                           to_char(js.percent_done));
      percent_done := js.percent_done;
    end if;

-- If any work-in-progress (WIP) or error messages were received for the job,
-- display them.

   if (bitand(sts.mask,dbms_datapump.ku$_status_wip) != 0)
    then
      le := sts.wip;
    else
      if (bitand(sts.mask,dbms_datapump.ku$_status_job_error) != 0)
      then
        le := sts.error;
```

```
      else
        le := null;
      end if;
    end if;
    if le is not null
    then
      ind := le.FIRST;
      while ind is not null loop
        dbms_output.put_line(le(ind).LogText);
        ind := le.NEXT(ind);
      end loop;
    end if;
  end loop;

-- Indicate that the job finished and detach from it.

  dbms_output.put_line('Job has completed');
  dbms_output.put_line('Final job state = ' || job_state);
  dbms_datapump.detach(h1);
END;
/
```

Example 5–2 Importing a Dump File and Remapping All Schema Objects

This example imports the dump file created in Example 5–1 (an export of the `hr` schema). All schema objects are remapped from the `hr` schema to the `blake` schema. To keep the example simple, exceptions from any of the API calls will not be trapped. However, in a production environment, Oracle recommends that you define exception handlers and call `GET_STATUS` to retrieve more detailed error information when a failure occurs.

```
DECLARE
  ind NUMBER;              -- Loop index
  h1 NUMBER;               -- Data Pump job handle
  percent_done NUMBER;     -- Percentage of job complete
  job_state VARCHAR2(30);  -- To keep track of job state
  le ku$_LogEntry;         -- For WIP and error messages
  js ku$_JobStatus;        -- The job status from get_status
  jd ku$_JobDesc;          -- The job description from get_status
  sts ku$_Status;          -- The status object returned by get_status
BEGIN

-- Create a (user-named) Data Pump job to do a "full" import (everything
-- in the dump file without filtering).
```

```
  h1 := DBMS_DATAPUMP.OPEN('IMPORT','FULL',NULL,'EXAMPLE2');

-- Specify the single dump file for the job (using the handle just returned)
-- and directory object, which must already be defined and accessible
-- to the user running this procedure. This is the dump file created by
-- the export operation in the first example.

  DBMS_DATAPUMP.ADD_FILE(h1,'example1.dmp','DMPDIR');

-- A metadata remap will map all schema objects from hr to blake.

  DBMS_DATAPUMP.METADATA_REMAP(h1,'REMAP_SCHEMA','hr','blake');

-- If a table already exists in the destination schema, skip it (leave
-- the preexisting table alone). This is the default, but it does not hurt
-- to specify it explicitly.

  DBMS_DATAPUMP.SET_PARAMETER(h1,'TABLE_EXISTS_ACTION','SKIP');

-- Start the job. An exception is returned if something is not set up properly.

  DBMS_DATAPUMP.START_JOB(h1);

-- The import job should now be running. In the following loop, the job is
-- monitored until it completes. In the meantime, progress information is
-- displayed. Note: this is identical to the export example.

 percent_done := 0;
  job_state := 'UNDEFINED';
  while (job_state != 'COMPLETED') and (job_state != 'STOPPED') loop
    dbms_datapump.get_status(h1,
           dbms_datapump.ku$_status_job_error +
           dbms_datapump.ku$_status_job_status +
           dbms_datapump.ku$_status_wip,-1,job_state,sts);
    js := sts.job_status;

-- If the percentage done changed, display the new value.

     if js.percent_done != percent_done
    then
      dbms_output.put_line('*** Job percent done = ' ||
                           to_char(js.percent_done));
      percent_done := js.percent_done;
    end if;
```

```
-- If any work-in-progress (WIP) or Error messages were received for the job,
-- display them.

       if (bitand(sts.mask,dbms_datapump.ku$_status_wip) != 0)
    then
      le := sts.wip;
    else
      if (bitand(sts.mask,dbms_datapump.ku$_status_job_error) != 0)
      then
        le := sts.error;
      else
        le := null;
      end if;
    end if;
    if le is not null
    then
      ind := le.FIRST;
      while ind is not null loop
        dbms_output.put_line(le(ind).LogText);
        ind := le.NEXT(ind);
      end loop;
    end if;
  end loop;

-- Indicate that the job finished and gracefully detach from it.

  dbms_output.put_line('Job has completed');
  dbms_output.put_line('Final job state = ' || job_state);
  dbms_datapump.detach(h1);
END;
/
```

Example 5–3 Using Exception Handling During a Simple Schema Export

This example shows a simple schema export using the Data Pump API. It extends Example 5–1 to show how to use exception handling to catch the SUCCESS_WITH_INFO case, and how to use the GET_STATUS procedure to retrieve additional information about errors. If you want to get status up to the current point, but a handle has not yet been obtained, you can use NULL for DBMS_DATAPUMP.GET_STATUS.

```
DECLARE
  ind NUMBER;              -- Loop index
  spos NUMBER;             -- String starting position
  slen NUMBER;             -- String length for output
```

```
  h1 NUMBER;               -- Data Pump job handle
  percent_done NUMBER;     -- Percentage of job complete
  job_state VARCHAR2(30);  -- To keep track of job state
  le ku$_LogEntry;         -- For WIP and error messages
  js ku$_JobStatus;        -- The job status from get_status
  jd ku$_JobDesc;          -- The job description from get_status
  sts ku$_Status;          -- The status object returned by get_status
BEGIN

-- Create a (user-named) Data Pump job to do a schema export.

  h1 := dbms_datapump.open('EXPORT','SCHEMA',NULL,'EXAMPLE3','LATEST');

-- Specify a single dump file for the job (using the handle just returned)
-- and a directory object, which must already be defined and accessible
-- to the user running this procedure.

  dbms_datapump.add_file(h1,'example3.dmp','DMPDIR');

-- A metadata filter is used to specify the schema that will be exported.

  dbms_datapump.metadata_filter(h1,'SCHEMA_EXPR','IN (''HR'')');

-- Start the job. An exception will be returned if something is not set up
-- properly.One possible exception that will be handled differently is the
-- success_with_info exception. success_with_info means the job started
-- successfully, but more information is available through get_status about
-- conditions around the start_job that the user might want to be aware of.

    begin
    dbms_datapump.start_job(h1);
    dbms_output.put_line('Data Pump job started successfully');
    exception
      when others then
        if sqlcode = dbms_datapump.success_with_info_num
        then
          dbms_output.put_line('Data Pump job started with info available:');
          dbms_datapump.get_status(h1,
                                   dbms_datapump.ku$_status_job_error,0,
                                   job_state,sts);
          if (bitand(sts.mask,dbms_datapump.ku$_status_job_error) != 0)
          then
            le := sts.error;
            if le is not null
            then
```

```
              ind := le.FIRST;
              while ind is not null loop
                dbms_output.put_line(le(ind).LogText);
                ind := le.NEXT(ind);
              end loop;
            end if;
          end if;
        else
          raise;
        end if;
  end;

-- The export job should now be running. In the following loop, we will monitor
-- the job until it completes. In the meantime, progress information is
-- displayed.

 percent_done := 0;
  job_state := 'UNDEFINED';
  while (job_state != 'COMPLETED') and (job_state != 'STOPPED') loop
    dbms_datapump.get_status(h1,
           dbms_datapump.ku$_status_job_error +
           dbms_datapump.ku$_status_job_status +
           dbms_datapump.ku$_status_wip,-1,job_state,sts);
    js := sts.job_status;

-- If the percentage done changed, display the new value.

     if js.percent_done != percent_done
    then
      dbms_output.put_line('*** Job percent done = ' ||
                           to_char(js.percent_done));
      percent_done := js.percent_done;
    end if;

-- Display any work-in-progress (WIP) or error messages that were received for
-- the job.

      if (bitand(sts.mask,dbms_datapump.ku$_status_wip) != 0)
    then
      le := sts.wip;
    else
      if (bitand(sts.mask,dbms_datapump.ku$_status_job_error) != 0)
      then
        le := sts.error;
      else
```

```
        le := null;
      end if;
    end if;
    if le is not null
    then
      ind := le.FIRST;
      while ind is not null loop
        dbms_output.put_line(le(ind).LogText);
        ind := le.NEXT(ind);
      end loop;
    end if;
  end loop;

-- Indicate that the job finished and detach from it.

  dbms_output.put_line('Job has completed');
  dbms_output.put_line('Final job state = ' || job_state);
  dbms_datapump.detach(h1);

-- Any exceptions that propagated to this point will be captured. The
-- details will be retrieved from get_status and displayed.

  exception
    when others then
      dbms_output.put_line('Exception in Data Pump job');
      dbms_datapump.get_status(h1,dbms_datapump.ku$_status_job_error,0,
                               job_state,sts);
      if (bitand(sts.mask,dbms_datapump.ku$_status_job_error) != 0)
      then
        le := sts.error;
        if le is not null
        then
          ind := le.FIRST;
          while ind is not null loop
            spos := 1;
            slen := length(le(ind).LogText);
            if slen > 255
            then
              slen := 255;
            end if;
            while slen > 0 loop
              dbms_output.put_line(substr(le(ind).LogText,spos,slen));
              spos := spos + 255;
              slen := length(le(ind).LogText) + 1 - spos;
            end loop;
```

```
            ind := le.NEXT(ind);
          end loop;
        end if;
      end if;
END;
/
```

Part II

SQL*Loader

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

Chapter 6, "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 7, "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 8, "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 9, "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 10, "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 11, "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 12, "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.

6

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

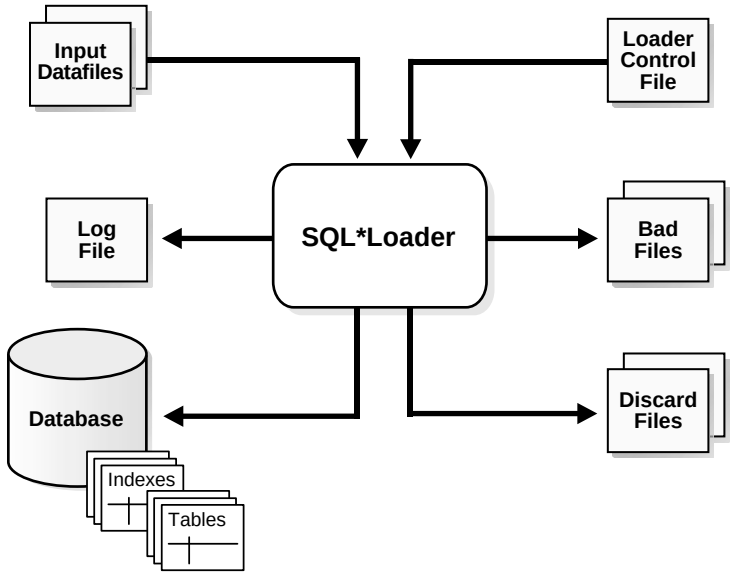
SQL*Loader loads data from external files into tables of an Oracle database. It has a powerful data parsing engine that puts little limitation on the format of the data in the datafile. You can use SQL*Loader to do the following:

- Load data across a network. This means that you can run the SQL*Loader client on a different system from the one that is running the SQL*Loader server.

- 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 aid 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 11.

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 6–1.

*Figure 6–1 SQL*Loader Overview*



SQL*Loader Parameters

SQL*Loader is invoked when you specify the `sqlldr` command and, optionally, parameters that establish session characteristics.

In situations where you always use the same parameters for which the values seldom change, it can be more efficient to specify parameters using the following methods, rather than on the command line:

- Parameters can be grouped together in a parameter file. You could then specify the name of the parameter file on the command line using the `PARFILE` parameter.
- Certain parameters can also be specified within the SQL*Loader control file by using the `OPTIONS` clause.

Parameters specified on the command line override any parameter values specified in a parameter file or `OPTIONS` clause.

See Also:

- Chapter 7 for descriptions of the SQL*Loader parameters
- PARFILE (parameter file) on page 7-10
- OPTIONS Clause on page 8-4

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 sessionwide 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 keywords `CONSTANT` and `ZONE` have special meaning to SQL*Loader and are therefore reserved. To avoid potential conflicts, Oracle recommends that you do not use either `CONSTANT` or `ZONE` as a name for any tables or columns.

See Also: Chapter 8 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 6–1 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 8-23.

Example 6–1 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 6–2 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 8-23.

Example 6–2 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 6–3 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 6–3 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 8-27
- Case Study 4: Loading Combined Physical Records on page 12-14 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:

- Specifying the Position of a Data Field on page 9-3
- Specifying Delimiters on page 9-25

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 10-22
- Secondary Datafiles (SDFs) on page 10-32

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 accepts the data and executes the `INSERT` statement to store the data in the database.

The Oracle database 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. 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 Database Rejects

After a datafile record is accepted for processing by SQL*Loader, it is sent to the Oracle database for insertion into a table as a row. If the Oracle database 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 8-12
- Case Study 4: Loading Combined Physical Records on page 12-14 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 12-14
- Specifying the Discard File on page 8-14

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 12, "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 11-1
- Bind Arrays and Conventional Path Loads on page 8-45

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 much of the data processing that normally takes place. Direct path load is much faster than conventional path load, but entails several restrictions.

See Also: Direct Path Load on page 11-5

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 11-31

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 13, "External Tables Concepts"
- Chapter 14, "The ORACLE_LOADER Access Driver"

Choosing External Tables Versus SQL*Loader

The record parsing of external tables and SQL*Loader is very similar, so normally there is not a major performance difference for the same record format. However, due to the different architecture of external tables and SQL*Loader, there are situations in which one method is more appropriate than the other.

In the following situations, use external tables for the best load performance:

- You want to transform the data as it is being loaded into the database.
- You want to use transparent parallel processing without having to split the external data first.

However, in the following situations, use SQL*Loader for the best load performance:

- You want to load data remotely.
- Transformations are not required on the data, and the data does not need to be loaded in parallel.

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 *Oracle Database Concepts* and in the *Oracle 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 10-1
- Loading Object Tables on page 10-12

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 10-29 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 10-18 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 provides a direct path load API for application developers. See the *Oracle Call Interface Programmer's Guide* for more information.

7

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=sample.ctl, LOG=sample.log, BAD=baz.bad, DATA=etc.dat
   USERID=scott/tiger, ERRORS=999, LOAD=2000, DISCARD=toss.dsc,
   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
.
.
.
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)
commit_discontinued -- commit loaded rows when load is discontinued  (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.
```

See Also: Command-Line Parameters on page 7-3 for descriptions of all the command-line parameters

Alternative Ways to Specify Parameters

If the length of the command line exceeds the size of the maximum command line on your system, you can put certain command-line parameters in the control file by using the `OPTIONS` clause.

You can also group parameters together in a parameter file. You specify the name of this file on the command line using the `PARFILE` parameter when you invoke SQL*Loader.

These alternative ways of specifying parameters are useful when you often use the same parameters with the same values.

Parameter values specified on the command line override parameter values specified in either a parameter file or in the `OPTIONS` clause.

See Also:

- OPTIONS Clause on page 8-4
- PARFILE (parameter file) on page 7-10

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 8-12 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 7-1.

`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 8-45
- READSIZE (read buffer size) on page 7-10

COLUMNARRAYROWS

Default: To see the default value for this parameter, invoke SQL*Loader without any parameters, as described in Invoking SQL*Loader on page 7-1.

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 8-28
- Specifying the Number of Column Array Rows and Size of Stream Buffers on page 11-21

CONTROL (control file)

Default: none

`CONTROL` specifies the name of the SQL*Loader control file that describes how to load the 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 8 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 11-22

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 11, "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 6-10 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 7-1.

`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 rejected rows are 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.
- 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.

  If you use EXTERNAL_TABLE=EXECUTE and also use the SEQUENCE parameter in your SQL*Loader control file, then SQL*Loader creates a database sequence, loads the table using that sequence, and then delete the sequence. The results of doing the load this way will be different than if the load were done with conventional or direct path. (For more information about creating sequences, see CREATE SEQUENCE in *Oracle Database SQL Reference.)*

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 12-18), 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 13, "External Tables Concepts"
- Chapter 14, "The ORACLE_LOADER Access Driver"

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 11-31

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 11-23

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 11-31

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 7-1.

The READSIZE parameter is used *only* when reading data from datafiles. When reading records from a control file, a value of 64 kilobytes (KB) 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 20 megabytes (MB) 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 operation 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 kilobytes (KB).

See BINDSIZE (maximum size) on page 7-4.

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:

- *Oracle Database Concepts*
- *Oracle 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 terminated, without finishing.

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 7-1.

Keep in mind that if you specify a low value for `ROWS` and then attempt to compress data using table compression, your compression ratio will probably be degraded. Oracle recommends that you either specify a high value or accept the default value when compressing data.

Conventional path loads only: `ROWS` specifies the number of rows in the bind array. See Bind Arrays and Conventional Path Loads on page 8-45.

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 11-13. The actual number of rows loaded into a table on a save is approximately the value of `ROWS` minus the number of discarded and rejected records since the last save.

Note: Do not attempt to use the ROWS parameter when loading data into an Index Organized Table (IOT). Such use is not supported and will likely result in an error.

SILENT (feedback mode)

When SQL*Loader begins, information about the SQL*Loader version being used appears on the screen and is placed in the log file. 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 was loaded into each table. It is not used for multiple-table direct loads when a different number of records was loaded into each table.

If a WHEN clause is also present and the load involves secondary data, the secondary data is skipped only if the WHEN clause succeeds for the record in the primary data file.

See Also: Interrupted Loads on page 8-24

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 to be marked Index Unusable instead, 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: The value of the Oracle database configuration parameter, SKIP_UNUSABLE_INDEXES, as specified in the initialization parameter file. The default database setting is TRUE.

Both SQL*Loader and the Oracle database provide a SKIP_UNUSABLE_INDEXES parameter. The SQL*Loader SKIP_UNUSABLE_INDEXES parameter is specified at the SQL*Loader command line. The Oracle database SKIP_UNUSABLE_INDEXES parameter is specified as a configuration parameter in the initialization parameter file. It is important to understand how they affect each other.

If you specify a value for SKIP_UNUSABLE_INDEXES at the SQL*Loader command line, it overrides the value of the SKIP_UNUSABLE_INDEXES configuration parameter in the initialization parameter file.

If you do not specify a value for SKIP_UNUSABLE_INDEXES at the SQL*Loader command line, then SQL*Loader uses the database setting for the SKIP_UNUSABLE_INDEXES configuration parameter, as specified in the initialization parameter file. If the initialization parameter file does not specify a database setting for SKIP_UNUSABLE_INDEXES, then the default database setting is TRUE.

A value of TRUE for SKIP_UNUSABLE_INDEXES means that if an index in an Index Unusable state is encountered, it is skipped and the load operation continues. This allows SQL*Loader to load a table with indexes that are in an Unusable state prior to the beginning of the load. Indexes that are not in an Unusable state at load time will be maintained by SQL*Loader. Indexes that are in an Unusable state at load time will not be maintained but will remain in an Unusable state at load completion.

Note: Indexes that are unique and marked Unusable 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.

The SKIP_UNUSABLE_INDEXES parameter applies to both conventional and direct path loads.

STREAMSIZE

Default: To see the default value for this parameter, invoke SQL*Loader without any parameters, as described in Invoking SQL*Loader on page 7-1.

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 11-21

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\' sample.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 7–1 shows the exit codes for various results.

Table 7–1 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
```

8

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:

- Sessionwide information
- Table and field-list information
- Input data (optional section)

Example 8–1 shows a sample control file.

Example 8–1 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 8-4.
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 8-8.
4. The `BADFILE` clause specifies the name of a file into which rejected records are placed. See Specifying the Bad File on page 8-12.
5. The `DISCARDFILE` clause specifies the name of a file into which discarded records are placed. See Specifying the Discard File on page 8-14.
6. The `APPEND` clause 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 8-34.

   To load data into a table that is empty, you would use the `INSERT` clause. See Loading Data into Empty Tables on page 8-34.
7. The `INTO TABLE` clause enables 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 8-32.
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 8-36.
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 8-38.

10. The remainder of the control file contains the field list, which provides information about column formats in the table being loaded. See Chapter 9 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 12-11.

Specifying Command-Line Parameters in the Control File

You can specify command-line parameters in the SQL*Loader control file using the OPTIONS clause. This can be 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 following command-line parameters can be specified using the OPTIONS clause. These parameters are described in greater detail in Chapter 7.

```
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 = {HEADER | FEEDBACK | ERRORS | DISCARDS | PARTITIONS | ALL}
SKIP = n
SKIP_INDEX_MAINTENANCE = {TRUE | FALSE}
SKIP_UNUSABLE_INDEXES = {TRUE | FALSE}
STREAMSIZE = n
```

The following is an example use of the OPTIONS clause that you could use in a SQL*Loader control file:

```
OPTIONS (BINDSIZE=100000, SILENT=(ERRORS, FEEDBACK) )
```

Note: Parameter values specified on the command line override parameter values specified in the control file OPTIONS clause.

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: *Oracle Database 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 9-52

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.

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.

Using the Backslash as an Escape Character

If your operating system uses the backslash character to separate directories in a path name, *and* if the version of the Oracle database 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.

Escape Character Is Sometimes Disallowed

The version of the Oracle database 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.

Identifying XML Type Tables

As of Oracle Database 10*g*, the XMLTYPE clause is available for use in a SQL*Loader control file. This clause is of the format `XMLTYPE(field name)`. It is used to identify XML type tables so that the correct SQL statement can be constructed. Example 8–2 shows how the clause can be used in a SQL*Loader control file.

Example 8–2 Identifying XML Type Tables in the SQL*Loader Control File

```
LOAD DATA
INFILE *
INTO TABLE po_tab
APPEND
XMLTYPE (xmldata)
FIELDS
(xmldata CHAR(2000))
BEGINDATA

<?xml version="1.0"?>
<purchaseOrder xmlns="http://www.oracle.com/PO"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.oracle.com/PO
http://www.oracle.com/scha0/po1.xsd"
```

```
orderDate="1999-10-20">

<shipTo country="US">
<name>Alice Smith</name>
<street>123 Maple Street</street>
<city>Mill Valley</city>
<state>CA</state>
<zip>90952</zip>
</shipTo>

<billTo country="US">
<name>Robert Smith</name>
<street>8 Oak Avenue</street>
<city>Old Town</city>
<state>PA</state>
<zip>95819</zip>
</billTo>

<comment>Hurry, my lawn is going wild!</comment>

<items>
<item partNum="872-AA">
<productName>Lawnmower</productName>
<quantity>1</quantity>
<USPrice>148.95</USPrice>
<comment>Confirm this is electric</comment>
</item>
<item partNum="926-AA">
<productName>Baby Monitor</productName>
<quantity>1</quantity>
<USPrice>39.98</USPrice>
<shipDate>1999-05-21</shipDate>
</item>
</items>
</purchaseOrder>
```

Specifying Datafiles

To specify a datafile that contains the data to be loaded, use the `INFILE` keyword, followed by the filename and optional file processing options string. You can specify multiple files by using multiple `INFILE` keywords.

> **Note:** You can also specify the datafile from the command line, using the DATA parameter described in Command-Line Parameters on page 7-3. 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 8-11.

> **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 10-22.
>
> For information about SDFs, see Secondary Datafiles (SDFs) on page 10-32.

The syntax for INFILE is as follows:

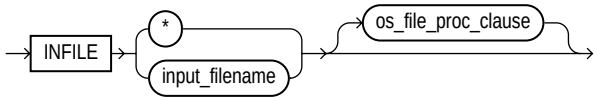


Table 8–1 describes the parameters for the INFILE keyword.

Table 8–1 Parameters for the INFILE Keyword

Parameter	Description
INFILE	Specifies that a datafile specification follows.
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 8-5.

Table 8–1 (Cont.) Parameters for the INFILE Keyword

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 8-12.

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 `sample` with a default extension of `.dat`:

  ```
  INFILE  sample
  ```

- 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.

Specifying Multiple Datafiles

To load data from multiple datafiles in one SQL*Loader run, use an `INFILE` clause 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` statement before the first data record. The syntax is:

```
BEGINDATA
data
```

Keep the following points in mind when using the `BEGINDATA` statement:

- If you omit the `BEGINDATA` statement 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` statement.
- Do not use spaces or other characters on the same line as the `BEGINDATA` statement, 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.

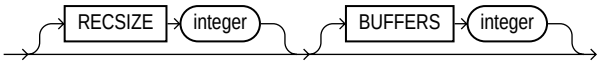
See Also:

- Specifying Datafiles on page 8-8 for an explanation of using `INFILE`
- Case Study 1: Loading Variable-Length Data on page 12-5

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"
```

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.

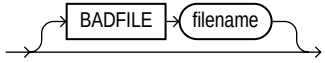
> **Note:** On some systems, a new version of the file may be created if a file with the same name already exists.

To specify the name of the bad file, use the `BADFILE` clause, followed by a 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 7-3.

A filename specified on the command line is associated with the first `INFILE` 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 you can reload the data after you correct it. 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` clause specifies that a filename for the bad file follows.

The *`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 `sample` and default file extension or file type of `.bad`, enter:

```
BADFILE sample
```

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 12-14 demonstrates rejected records.

Specifying the Discard File

During execution, SQL*Loader 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*, DISCARDS, or DISCARDMAX.

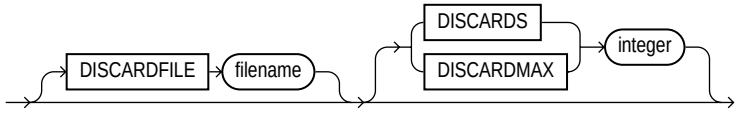
To create a discard file from the command line, specify either DISCARD or DISCARDMAX.

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 clause, followed by the filename.



The DISCARDFILE clause specifies that a discard filename follows.

The *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 7-6 for information about 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 12-28 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`* for either the `DISCARDS` or `DISCARDMAX` keyword.

When the discard limit is reached, processing of the datafile terminates and continues with the next datafile, if one exists.

You can specify a different number of discards for each datafile. 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: *Oracle 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 12-47
- *Oracle Database Globalization Support Guide* for more information about Unicode encoding

Database Character Sets

The Oracle database 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 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 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 also by 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:

- *Oracle Database Concepts* for more information about character-length semantics in the database
- Character-Length Semantics on page 8-23
- *Oracle Database Globalization Support Guide*

Considerations When Loading Data into VARRAYs or Primary-Key-Based REFs

If you use SQL*Loader conventional path or the Oracle Call Interface (OCI) to load data into VARRAYs or into primary-key-based REFs, and the data being loaded is in a different character set than the database character set, problems such as the following might occur:

- Rows might be rejected for the reason that a field is too large for the database column, but in reality the field is not too large.
- A load might be abnormally terminated without any rows being loaded, when only the field that really was too large should have been rejected.
- Rows might be reported as loaded correctly, but the primary-key-based REF columns are returned as blank when they are selected with SQL*Plus.

To avoid these problems, set the client character set (using the NLS_LANG environment variable) to the database character set before you load the data.

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 9-39
- *Oracle Database Globalization Support Guide* for more information about the names of the supported character sets
- Control File Character Set on page 8-22
- Case Study 11: Loading Data in the Unicode Character Set on page 12-47 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'"
```

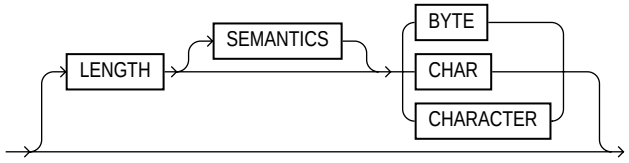
Any data included after the BEGINDATA statement is also assumed to be in the character set specified for your session by the NLS_LANG parameter.

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 Ctrl+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 7-14

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 reports the value for the SKIP parameter only 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 reports the value for the SKIP parameter only if it is the same for all tables.

Load Discontinued Because a Ctrl+C Was Issued

If SQL*Loader is in the middle of saving data when a Ctrl+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 11-13 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 7-14

Assembling Logical Records from Physical Records

As of Oracle9*i*, user-defined record sizes larger than 64 KB are supported (see READSIZE (read buffer size) on page 7-10). This reduces the need to break up logical records into multiple physical records. 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 7-4
- Specifying the Number of Column Array Rows and Size of Stream Buffers on page 11-21

Using CONTINUEIF to Assemble Logical Records

Use CONTINUEIF if the number of physical records to be combined varies. The CONTINUEIF clause 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:

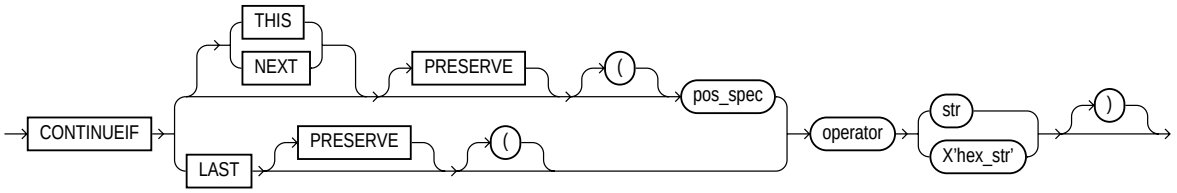


Table 8–2 describes the parameters for the CONTINUEIF clause.

Table 8–2 Parameters for the CONTINUEIF Clause

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 (!= or <>). 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. LAST allows only a single character-continuation field (as opposed to THIS and NEXT, which allow multiple character-continuation fields).
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.

Table 8–2 (Cont.) Parameters for the CONTINUEIF Clause

Parameter	Description
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.

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.

For CONTINUEIF THIS and CONTINUEIF LAST, if the PRESERVE parameter is not specified, 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. For example, if CONTINUEIF THIS(3:5)='***' is specified, then positions 3 through 5 are removed from all records. This means that the continuation characters are removed if they are in positions 3 through 5 of the record. It also means that the characters in positions 3 through 5 are removed from the record even if the continuation characters are not in positions 3 through 5.

For CONTINUEIF THIS and CONTINUEIF LAST, if the PRESERVE parameter is used, the continuation field is kept in all physical records when the logical record is assembled.

CONTINUEIF LAST differs from CONTINUEIF THIS and CONTINUEIF NEXT. For CONTINUEIF LAST, where the positions of the continuation field vary from record to record, the continuation field is never removed, even if PRESERVE is not specified.

Example 8–3 through Example 8–6 show the use of CONTINUEIF THIS and CONTINUEIF NEXT, with and without the PRESERVE parameter.

Example 8–3 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....
%%ddddddddd..
```

```
%%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....
ddddddddddd..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 8–4 CONTINUEIF THIS with the PRESERVE Parameter

Assume that you have the same physical records as in Example 8–3.

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....
%%ddddddddddd..%%eeeeeeeeee....ffffffffff..
```

Note that columns 1 and 2 are not removed from the physical records when the logical records are assembled.

Example 8–5 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....
..ddddddddddd..
%%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 8–3).

```
aaaaaaaa....bbbbbbbb....cccccccc....
ddddddddddd..eeeeeeeeee..ffffffffff..
```

Example 8–6 CONTINUEIF NEXT with the PRESERVE Parameter

Assume that you have the same physical records as in Example 8–5.

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....
..ddddddddddd..%%eeeeeeeeee..%%ffffffffff..
```

See Also: Case Study 4: Loading Combined Physical Records on page 12-14 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

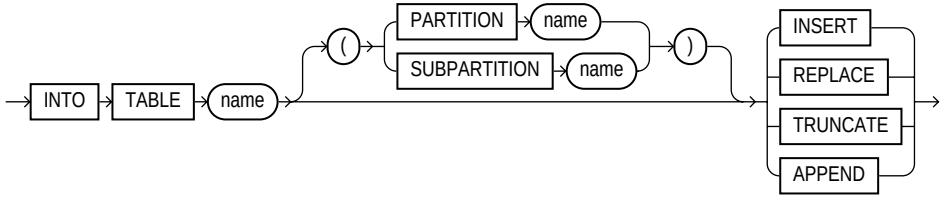
The `INTO TABLE` clause of the `LOAD DATA` statement enables you to identify tables, fields, and datatypes. It defines the relationship between records in the datafile and tables in the database. The specification of fields and datatypes is described in later sections.

INTO TABLE Clause

Among its many functions, the INTO TABLE clause enables 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 refers to the table emp in the scott schema).

Note: SQL*Loader considers the default schema to be whatever schema is current after your connect to the database finishes executing. This means that the default schema will not necessarily be the one you specified in the connect string, if there are logon triggers present that get executed during connection to a database.

If you have a logon trigger that changes your current schema to a different one when you connect to a certain database, then SQL*Loader uses that new schema as the default.

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 12-5 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 statement 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.

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 12-11 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 12-14 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 about cascaded deletes, see the information about data integrity in *Oracle 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 *Oracle Database 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.

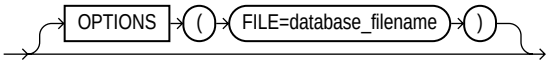
The table must be in your schema, or you must have the DROP ANY TABLE privilege.

> **See Also:** *Oracle 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 valid only for a parallel load.)

The syntax for the OPTIONS parameter is as follows:

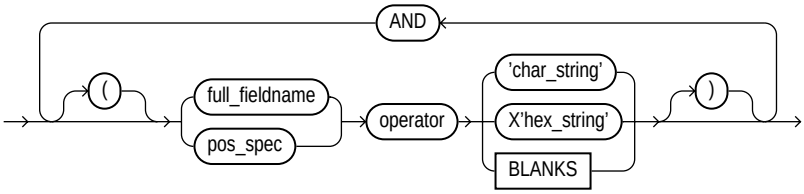


> **See Also:** Parameters for Parallel Direct Path Loads on page 11-34

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 9-33 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 12-18 provides an example of the WHEN clause

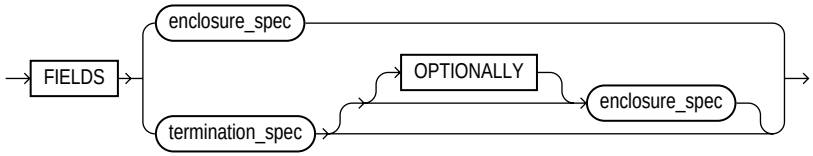
Using the WHEN Clause with LOBFILEs and SDFs

If a record with a LOBFILE or SDF is discarded, SQL*Loader skips the corresponding data in that LOBFILE or SDF.

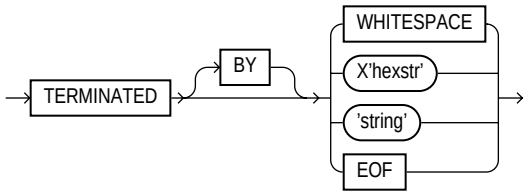
Specifying Default Data Delimiters

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

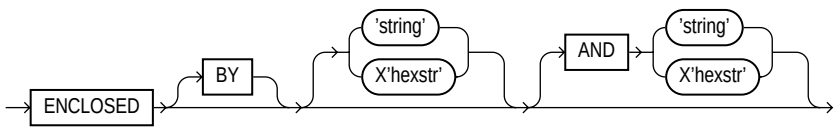


termination_spec



Note: Terminator strings can contain one or more characters. Also, `TERMINATED BY EOF` applies only to loading LOBs from a 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 12-11 provides an example.

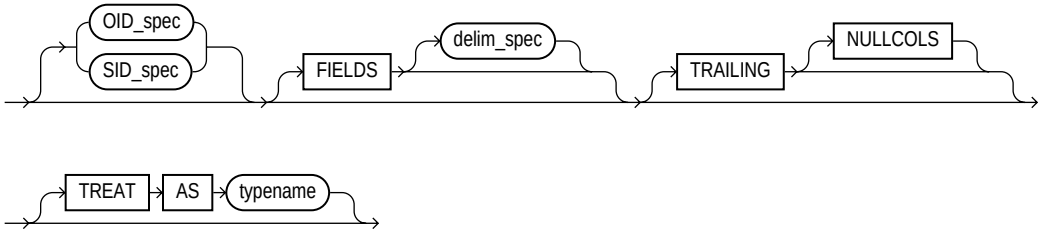
See Also:

- Specifying Delimiters on page 9-25 for a complete description of the syntax
- Loading LOB Data from LOBFILEs on page 10-22

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 12-28 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 11-18

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 operation, 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 enable 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 of 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 8-41 and in Loading Data into Multiple Tables on page 8-44.

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 their row objects 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.

> **Note:** Multiple subtypes cannot be loaded with the same `INTO TABLE` statement. Instead, you must use multiple `INTO TABLE` statements and have each one load a different subtype.

```
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 10-1 for more information about loading object types

Loading Data into Multiple Tables

By using the `POSITION` parameter 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 12-18.

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 statement 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 7-3.

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 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 7-4) or the `OPTIONS` clause in the control file (see OPTIONS Clause on page 8-4), 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 7-12) or the `OPTIONS` clause in the control file (see OPTIONS Clause on page 8-4).

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 9-8. 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 9-8. 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 8–3 through Table 8–6 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 about these values, see SQL*Loader Datatypes on page 9-8.

Table 8–3 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 8–4 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 8–5 Graphic Fields

Datatype	Default Size	Length Specified with POSITION	Length Specified with DATATYPE
GRAPHIC	None	L	2*L

Table 8–5 (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 8–6 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.

9

Field List Reference

This chapter describes the field-list portion of the SQL*Loader control file. The following topics are included:

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 9–1 shows the field list section of the sample control file that was introduced in Chapter 8.

Example 9–1 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 9-60.
2. POSITION specifies the position of a data field. See Specifying the Position of a Data Field on page 9-3.

   INTEGER EXTERNAL is the datatype for the field. See Specifying the Datatype of a Data Field on page 9-7 and Numeric EXTERNAL on page 9-21.

   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 9-33.

   In this sample, the field is being compared to blanks, using the BLANKS parameter. See Comparing Fields to BLANKS on page 9-32.
3. The TERMINATED BY WHITESPACE clause is one of the delimiters it is possible to specify for a field. See TERMINATED Fields on page 9-26.
4. The ENCLOSED BY clause is another possible field delimiter. See ENCLOSED Fields on page 9-26.

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:

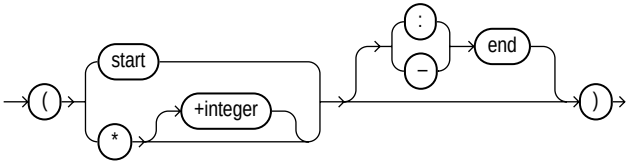


Table 9–1 describes the parameters for the position specification clause.

Table 9–1 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 9-25

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 8-40.

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 9-58.

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, and BFILE).

  To avoid ambiguity, if a Filler field is referenced in a directive, such as BFILE, and that field is declared in the control file inside of a column object, then the field name must be qualified with the name of the column object. This is illustrated in the following example:

```
LOAD DATA
INFILE *
INTO TABLE BFILE1O_TBL REPLACE
FIELDS TERMINATED BY ','
(
   emp_number char,
   emp_info_b column object
   (
   bfile_name FILLER char(12),
   emp_b BFILE(constant "SQLOP_DIR", emp_info_b.bfile_name) NULLIF
  emp_info_b.bfile_name = 'NULL'
   )
)
BEGINDATA
00001,bfile1.dat,
00002,bfile2.dat,
00003,bfile3.dat,
```

- 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 9-8 describes how SQL*Loader datatypes are converted into Oracle datatypes and gives detailed information about 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-10.

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.

INTEGERs 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 9-38

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 9-39.

> **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.

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 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 9-39.

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 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 9-39.

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.)

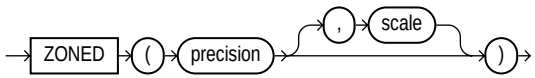
An example of the syntax for this datatype is:

```
(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:

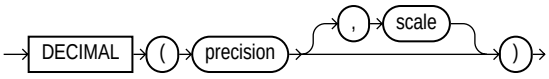
```
sal   POSITION(32)   ZONED(8),
```

The Oracle database 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 8-20 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.)

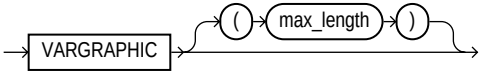
VARGRAPHIC

The data is a varying-length, double-byte character set (DBCS). It consists of a length subfield followed by a string of double-byte characters. The Oracle database does not support double-byte character sets; however, SQL*Loader reads them as single bytes and loads them 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 9-10 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 9-39.

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.

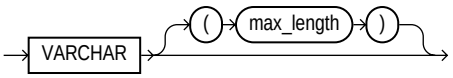
VARCHAR

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 8-23.

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 9-39.

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 9-10 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 8-23.

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 9-39.

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 9-39.

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-10.

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.

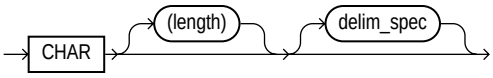
CHAR

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 9-25

Datetime and Interval Datatypes

Both datetimes and intervals are made up of fields. The values of these fields determine the value of the datatype.

The datetime datatypes are:

- DATE
- TIME
- TIME WITH TIME ZONE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE
- TIMESTAMP WITH LOCAL TIME ZONE

Values of datetime datatypes are sometimes called datetimes. In the following descriptions of the datetime datatypes you will see that, except for DATE, you are allowed to optionally specify a value for fractional_second_precision. The fractional_second_precision specifies the number of digits stored in the fractional part of the SECOND datetime field. When you create a column of this datatype, the value can be a number in the range 0 to 9. The default is 6.

The interval datatypes are:

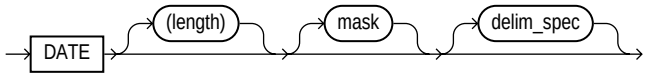
- INTERVAL YEAR TO MONTH
- INTERVAL DAY TO SECOND

Values of interval datatypes are sometimes called intervals. The INTERVAL YEAR TO MONTH datatype allows you to optionally specify a value for year_precision. The year_precision value is the number of digits in the YEAR datetime field. The default value is 2.

The INTERVAL DAY TO SECOND datatype allows you to optionally specify values for day_precision and fractional_second_precision. The day_precision is the number of digits in the DAY datetime field. Accepted values are 0 to 9. The default is 2. The fractional_second_precision specifies the number of digits stored in the fractional part of the SECOND datetime field. When you create a column of this datatype, the value can be a number in the range 0 to 9. The default is 6.

> **See Also:** *Oracle Database SQL Reference* for more detailed information about specifying datetime and interval datatypes, including the use of fractional_second_precision, year_precision, and day_precision

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-2002
1-Apr-2002 28-Feb-2002
```

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 8-23.

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 12-11 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 9-25.

TIME The TIME datatype stores hour, minute, and second values. It is specified as follows:

```
TIME [(fractional_second_precision)]
```

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). It is specified as follows:

```
TIME [(fractional_second_precision)] WITH [LOCAL] TIME ZONE
```

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 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. It is specified as follows:

```
TIMESTAMP [(fractional_second_precision)]
```

If you specify a date value without a time component, the default time is 12:00:00 a.m. (midnight).

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). It is specified as follows:

```
TIMESTAMP [(fractional_second_precision)] WITH TIME ZONE
```

TIMESTAMP WITH LOCAL TIME ZONE The TIMESTAMP WITH LOCAL TIME ZONE datatype is another variant of TIMESTAMP that includes a time zone offset in its value. 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. It is specified as follows:

```
TIMESTAMP [(fractional_second_precision)] WITH LOCAL TIME ZONE
```

INTERVAL YEAR TO MONTH The INTERVAL YEAR TO MONTH datatype stores a period of time using the YEAR and MONTH datetime fields. It is specified as follows:

```
INTERVAL YEAR [(year_precision)] TO MONTH
```

INTERVAL DAY TO SECOND The INTERVAL DAY TO SECOND datatype stores a period of time using the DAY and SECOND datetime fields. It is specified as follows:

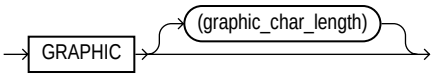
```
INTERVAL DAY [(day_precision)] TO SECOND [(fractional_second_precision)]
```

GRAPHIC

The data is in the form of a double-byte character set (DBCS). The Oracle database does not support double-byte character sets; however, SQL*Loader reads them as

single bytes. Like RAW data, GRAPHIC fields are stored without modification in whichever column you specify.

The syntax for the GRAPHIC datatype is:



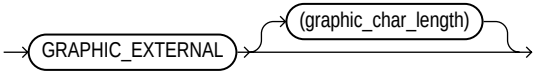
For GRAPHIC and GRAPHIC EXTERNAL, specifying POSITION*(start:end)* gives the exact location of the field in the logical record.

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 9-19).

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 8-23.

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 9-15 for a complete description of these rules.

The syntax for the numeric EXTERNAL datatypes is shown as part of datatype_spec on page A-10.

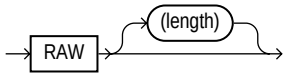
Note: The data is a number in character form, not binary representation. Therefore, these datatypes are identical to CHAR and are treated identically, *except for the use of DEFAULTIF*. If you want the default to be null, use CHAR; if you want it to be zero, use EXTERNAL. See Using the WHEN, NULLIF, and DEFAULTIF Clauses on page 9-33.

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. If it is loaded into a CHAR column, the Oracle database 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 8-23.

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 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 is capable of converting may be loaded into these or other database columns.

Datatype Conversions for Datetime and Interval Datatypes

Table 9–2 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 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.

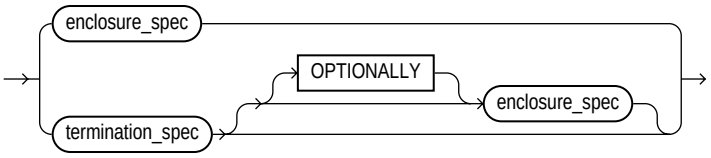
Table 9–2 Datatype Conversions for Datetime and Interval Datatypes

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. To see the syntax for `enclosure_spec` and `termination_spec`, see Syntax for Termination and Enclosure Specification on page 9-26.

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.

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.

Syntax for Termination and Enclosure Specification

The following diagram shows the syntax for `termination_spec` and `enclosure_spec`.

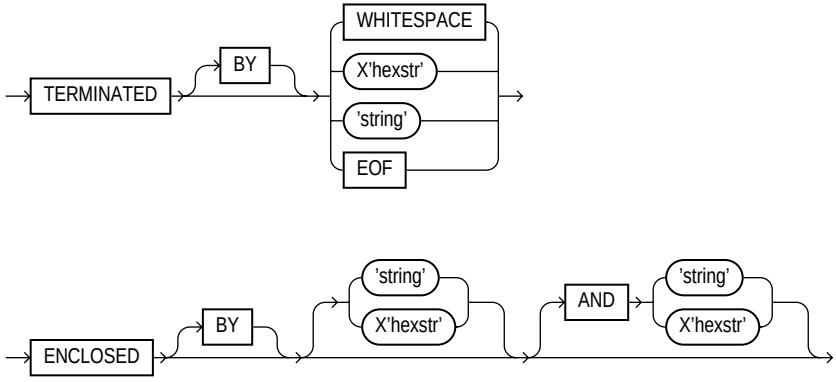


Table 9–3 describes the syntax for the termination and enclosure specifications used to specify delimiters.

Table 9–3 Parameters Used for Specifying Delimiters

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.

Table 9–3 (Cont.) Parameters Used for Specifying Delimiters

Parameter	Description
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 9-44
- How the PRESERVE BLANKS Option Affects Whitespace Trimming on page 9-51

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 an error.

Date Field Masks

The length of a date field depends on the mask, if a mask is specified. The mask provides a format pattern, telling SQL*Loader how to interpret the data in the record. For example, 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 9-16 for more information about the `DATE` field

Specifying Field Conditions

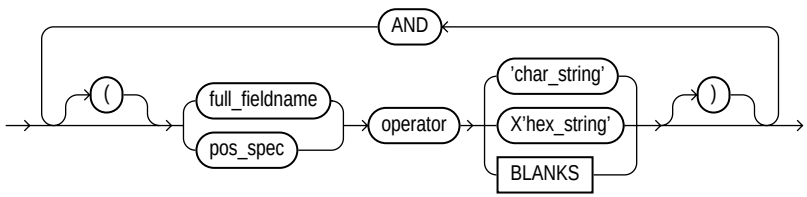
A field condition is a statement about a field in a logical record that evaluates as true or false. It is used in the `WHEN`, `NULLIF`, and `DEFAULTIF` clauses.

> **Note:** If a field used in a clause evaluation has a NULL value, then that clause will always evaluate to FALSE. This feature is illustrated in Example 9–5.

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:

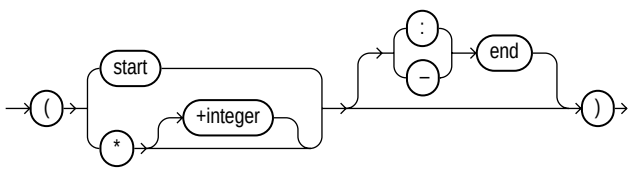


Table 9–4 describes the parameters used for the field condition clause. For a full description of the position specification parameters, see Table 9–1.

Table 9–4 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.
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	Enables 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 about multibyte character sets, see Multibyte (Asian) Character Sets on page 8-17.

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 9-44 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.

Note: As demonstrated in these steps, the presence of NULLIF and DEFAULTIF clauses results in extra processing that SQL*Loader must perform. This can affect performance. Note that during Step 1, SQL*Loader will set a field to NULL if its evaluated length is zero. To improve performance, consider whether it might be possible for you to change your data to take advantage of this. The detection of NULLs as part of Step 1 occurs much more quickly than the processing of a NULLIF or DEFAULTIF clause.

For example, a CHAR(5) will have zero length if it falls off the end of the logical record or if it contains all blanks and blank trimming is in effect. A delimited field will have zero length if there are no characters between the start of the field and the terminator.

Also, for character fields, NULLIF is usually faster to process than DEFAULTIF (the default for character fields is NULL).

Examples of Using the WHEN, NULLIF, and DEFAULTIF Clauses

Example 9–2 through Example 9–5 clarify the results for different situations in which the WHEN, NULLIF, and DEFAULTIF clauses might be used. 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 9–2 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 9–2, 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 9–3 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 9–3, `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 9–4 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 9–4, `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 9–5 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 9–5, 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 9-39. 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.

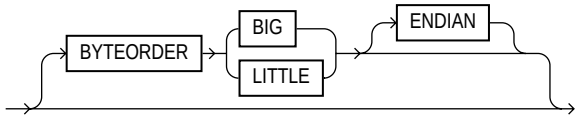
Byte order in a datafile is only an issue if the datafile that contains the byte-order-dependent data is created on a system that has a different byte order from the system on which SQL*Loader is running. If SQL*Loader knows the byte order of the data, it swaps the bytes as necessary to ensure that the 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 12-47 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.

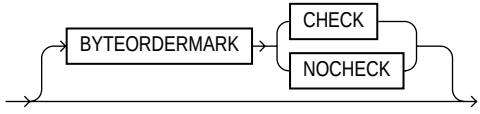
In summary, the precedence of the byte-order indicators for LOBFILEs and SDFs is as follows:

- BYTEORDER parameter value specified with the LOBFILE or SDF
- The byte-order setting established for the first primary datafile

> **Note:** If the character set of your datafile is a unicode character set and there is a byte-order mark in the first few bytes of the file, do not use the SKIP parameter. If you do, the byte-order mark will not be read and interpreted as a byte-order mark.

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 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 or numeric 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 12-24 for an example of how to load all-blank fields as NULL with the NULLIF clause
- Trimming Whitespace on page 9-44
- How the PRESERVE BLANKS Option Affects Whitespace Trimming on page 9-51

Trimming Whitespace

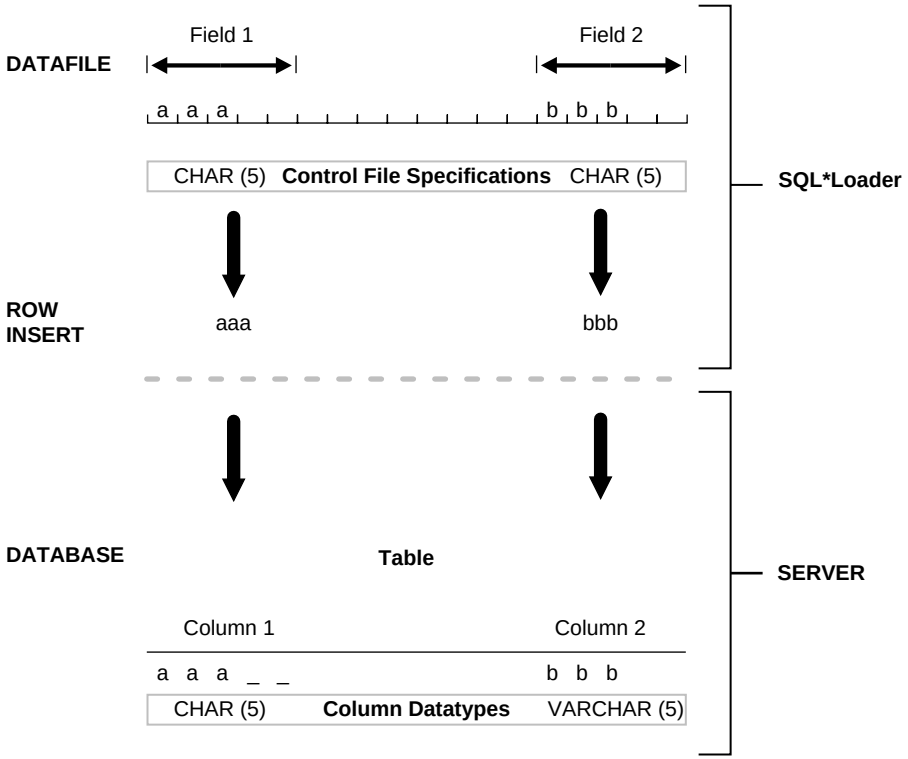
Blanks, tabs, and other nonprinting characters (such as carriage returns and line feeds) constitute whitespace. Leading whitespace occurs at the beginning of a field. Trailing whitespace occurs at the end of a field. Depending on how the field is specified, whitespace may or may not be included when the field is inserted into the database. This is illustrated in Figure 9–1, 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 handling any necessary conversions.

By default, SQL*Loader removes trailing spaces from CHAR data before passing it to the database. So, in Figure 9–1, 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 9–1 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 9–5 summarizes when and how whitespace is removed from input data fields when PRESERVE BLANKS is not specified. See How the PRESERVE BLANKS Option Affects Whitespace Trimming on page 9-51 for details on how to prevent whitespace trimming.

Table 9–5 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
- Specifying Field Length for Datatypes for Which Whitespace Can Be Trimmed
- Relative Positioning of Fields
- Leading Whitespace
- Trimming Trailing Whitespace

- Trimming 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.

Specifying Field Length 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

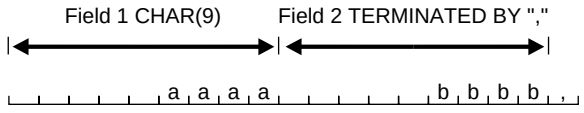
This section describes how SQL*Loader determines the starting position of a field in the following situations:

- 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 9–2 illustrates this situation when the previous field (Field 1) has a predetermined size.

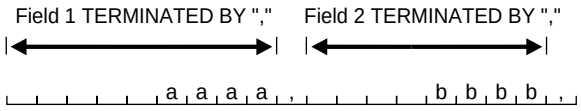
Figure 9–2 Relative Positioning After a Fixed Field



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 9–3.

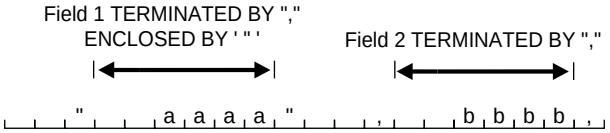
Figure 9–3 Relative Positioning After a Delimited Field



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 9–4. If a nonwhitespace character is found after the enclosure delimiter, but before the terminator, then SQL*Loader generates an error.

Figure 9–4 Relative Positioning After Enclosure Delimiters



Leading Whitespace

In Figure 9–4, both fields are stored with leading whitespace. Fields do *not* include leading whitespace in the following cases:

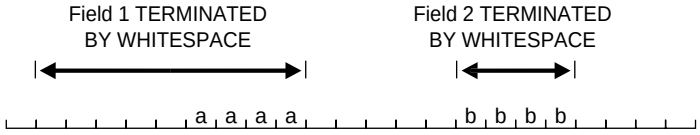
- When the previous field is terminated by whitespace, and no starting position is specified for the current field
- When optional enclosure delimiters are specified for the field, and the enclosure delimiters are *not* present

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 9–5 illustrates this case.

Figure 9–5 Fields Terminated by Whitespace



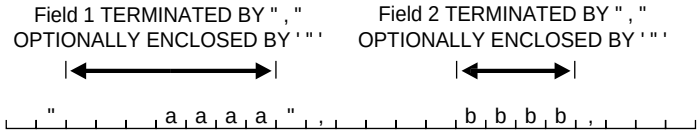
This situation occurs when the previous field is explicitly specified with the TERMINATED BY WHITESPACE clause, as shown in the example. It also occurs when you use the global FIELDS TERMINATED BY WHITESPACE clause.

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 9–6. (In Field 1 the whitespace is included because SQL*Loader found enclosure delimiters for the field.)

Figure 9–6 Fields Terminated by Optional Enclosure Delimiters



Unlike the case when the previous field is TERMINATED BY WHITESPACE, this specification removes leading whitespace even when a starting position is specified for the current field.

Note: If enclosure delimiters are present, leading whitespace after the initial enclosure delimiter is kept, but whitespace before this delimiter is discarded. See the first quotation mark in Field 1, Figure 9–6.

Trimming 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.

Trimming Enclosed Fields

If a field is enclosed, or terminated and enclosed, like the first field shown in Figure 9–6, 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.

How the PRESERVE BLANKS Option Affects Whitespace Trimming

To prevent whitespace trimming in *all* CHAR, DATE, and numeric EXTERNAL fields, you specify PRESERVE BLANKS as part of the LOAD statement in the control file. However, there may be times when you do not want to preserve blanks for *all* CHAR, DATE, and numeric EXTERNAL fields. Therefore, SQL*Loader also enables you to specify PRESERVE BLANKS as part of the datatype specification for individual fields, rather than specifying it globally as part of the LOAD statement.

In the following example, assume that PRESERVE BLANKS has not been specified as part of the LOAD statement, but you want the c1 field to default to zero when blanks are present. You can achieve this by specifying PRESERVE BLANKS on the individual field. Only that field is affected; blanks will still be removed on other fields.

```
c1 INTEGER EXTERNAL(10) PRESERVE BLANKS DEFAULTIF c1=BLANKS
```

In this example, if PRESERVE BLANKS were not specified for the field, it would result in the field being improperly loaded as NULL (instead of as 0).

There may be times when you want to specify PRESERVE BLANKS as an option to the LOAD statement and have it apply to most CHAR, DATE, and numeric EXTERNAL

fields. You can override it for an individual field by specifying NO PRESERVE BLANKS as part of the datatype specification for that field, as follows:

```
c1 INTEGER EXTERNAL(10) NO PRESERVE BLANKS
```

How [NO] PRESERVE BLANKS Works with Delimiter Clauses

The PRESERVE BLANKS option is affected by the presence of the delimiter clauses, as follows:

- Leading whitespace is left intact when optional enclosure delimiters are not present
- Trailing whitespace is left intact when fields are specified with a predetermined size

For example, consider the following field, where underscores represent blanks:

```
__aa__,
```

Suppose this field is loaded with the following delimiter clause:

```
TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
```

In such a case, if PRESERVE BLANKS is specified, then both the leading whitespace and the trailing whitespace are retained. If PRESERVE BLANKS is not specified, then the leading whitespace is trimmed.

Now suppose the field is loaded with the following clause:

```
TERMINATED BY WHITESPACE
```

In such a case, if PRESERVE BLANKS is specified, it does not retain 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.

See Also: Trimming Whitespace on page 9-44

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 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 *Oracle Database 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 Grant
Jason Taylor
```

The following requirements and restrictions apply when you are using SQL strings:

- If your control file specifies character input that has an associated SQL string, SQL*Loader makes no attempt to modify the data. This is because SQL*Loader assumes that character input data that is modified using a SQL operator will yield results that are correct for database insertion.
- 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 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, BFILEs, 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 about the use of quotation marks inside quoted strings, see Specifying Filenames and Object Names on page 8-5.
- 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 12-28 to load data from a formatted report.

Using SQL Strings to Load the ANYDATA Database Type

The ANYDATA database type can contain data of different types. To load the ANYDATA type using SQL*loader, it must be explicitly constructed by using a function call. The function is invoked using support for SQL strings as has been described in this section.

For example, suppose you have a table with a column named `miscellaneous` which is of type ANYDATA. You could load the column by doing the following, which would create an ANYDATA type containing a number.

```
LOAD DATA
INFILE *
APPEND INTO TABLE  ORDERS
(
miscellaneous CHAR "SYS.ANYDATA.CONVERTNUMBER(:miscellaneous)"
)
BEGINDATA
4
```

There can also be more complex situations in which you create an ANYDATA type that contains a different type depending upon the values in the record. To do this, you could write your own PL/SQL function that would determine what type should be in the ANYDATA type, based on the value in the record, and then call the appropriate ANYDATA.`Convert*()` function to create it.

See Also:

- *Oracle Database SQL Reference* for more information about the ANYDATA database type
- *PL/SQL Packages and Types Reference* for more information about using ANYDATA with PL/SQL

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 *Oracle Database 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.

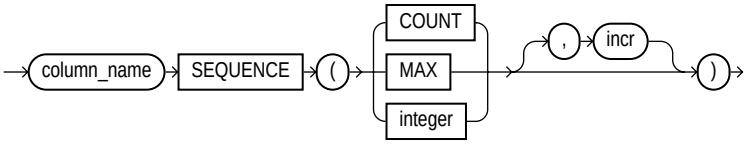


Table 9–6 describes the parameters used for column specification.

Table 9–6 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.
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 12-11 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.

10

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 10–1 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 10–1 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
```

Notes

1. This type of column object specification can be applied recursively to describe nested column objects.

Loading Column Objects in Variable Record Format

Example 10–2 shows a case in which the data is in delimited fields.

Example 10–2 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 6 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 10–3 shows a control file describing nested column objects (one column object nested in another column object).

Example 10–3 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 10–4 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 10–4 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 10–5 demonstrates this.

Example 10–5 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 10–5, 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 10–6.

Example 10–6 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 9-6

Loading Column Objects with User-Defined Constructors

The Oracle database 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 10–7 illustrates this difference.

Example 10–7 Loading a Column Object with Constructors That Match

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
--User-defined constructor makes sure that the name attribute is 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 10–8.

Example 10–8 Loading a Column Object with Constructors That Do Not Match

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 user-defined constructor 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 10–7 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 10–9 shows how this can be done.

Example 10–9 Using SQL to Load Column Objects When Constructors Do Not Match

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),
1      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 10–9 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 10–10 demonstrates loading an object table with primary-key-based object identifiers (OIDs).

Example 10–10 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, an object table with primary-key-based OIDs, or a relational table.

You may want to load data that already contains system-generated 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 system-generated 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-based OIDs.

Example 10–11 demonstrates loading system-generated OIDs with the row objects.

Example 10–11 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 10–12.

Example 10–12 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 system-generated OID REF columns, primary-key-based REF columns, and unscoped REF columns that allow primary keys.

System-Generated OID REF Columns

SQL*Loader assumes, when loading system-generated 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 10–13 demonstrates loading system-generated OID REF columns.

Example 10–13 Loading System-Generated 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-delimited 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 10–14 demonstrates loading primary key REF columns.

Example 10–14 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 system-generated OID REF column or into a primary-key-based REF column. See Example 10–13, "Loading System-Generated REF Columns" and Example 10–14, "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 System-Generated OID REF Columns on page 10-15 also apply here.
- If you are loading primary key REFs into this REF column, any limitations described in Primary Key REF Columns on page 10-15 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 actual values, they can be null, or they can be empty. SQL*Loader creates an empty LOB when there is a 0-length field to store in the LOB. (Note that this is different than other datatypes where SQL*Loader sets the column to NULL for any 0-length string.) This means that *the only way to load NULL values into a LOB column is to use the NULLIF clause.*

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 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 10–15.

> **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 10–15 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. Because the DEFAULTIF clause is used, if the data field containing the resume is empty, the result is an empty LOB rather than a null LOB. However, if a

NULLIF clause had been used instead of DEFAULTIF, the empty data field would be null.

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.

> **Note:** SQL*Loader defaults to 255 bytes when moving CLOB data, but a value of up to 2 gigabytes can be specified. For a delimited field, if a length is specified, that length is used as a maximum. If no maximum is specified, it defaults to 255 bytes. For a CHAR field that is delimited and is also greater than 255 bytes, you must specify a maximum length. See CHAR on page 9-15 for more information about the CHAR datatype.

Example 10–16 shows an example of loading LOB data in delimited fields.

Example 10–16 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 8-23.
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 10–17 demonstrates loading LOB data in length-value pair fields.

Example 10–17 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 8-23.
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 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 10-23 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 10–18, each LOBFILE is the source of a single LOB. To load LOB data that is organized in this way, the column or field name is followed by the LOBFILE datatype specifications.

Example 10–18 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 10–19, 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 10–19 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 8-23.

Delimited LOBs In Example 10–20, 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 10–20 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 10–21 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 10–21 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 8-23.
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 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 *Oracle Database Application Developer's Guide - Large Objects*.

Loading BFILE Columns

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 *Oracle Database Application Developer's Guide - Large Objects*.

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 *Oracle Database SQL Reference* for more information.

In the next two examples of loading BFILEs, Example 10–22 has only the filename specified dynamically, while Example 10–23 demonstrates specifying both the BFILE and the directory object dynamically.

Example 10–22 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 in quotation marks; therefore, the string is used as is and is not capitalized.

Example 10–23 Loading Data Using BFILEs: Filename and Directory 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 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 10-32 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 10-1. 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.

Also, be aware that if you are loading into a table containing nested tables, SQL*Loader will not automatically split the load into multiple loads and generate a set ID.

Example 10–24 demonstrates loading a VARRAY and a nested table.

Example 10–24 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 8-17).
- 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 7-10
- OPTIONS Clause on page 8-4
- sdf_spec on page A-12

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 10–25 illustrates how to load a parent table with user-provided SIDs.

Example 10–25 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 10–26 illustrates how to load a child table (the nested table storage table) with user-provided SIDs.

Example 10–26 Loading a Child 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.

11

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 12-24. 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.

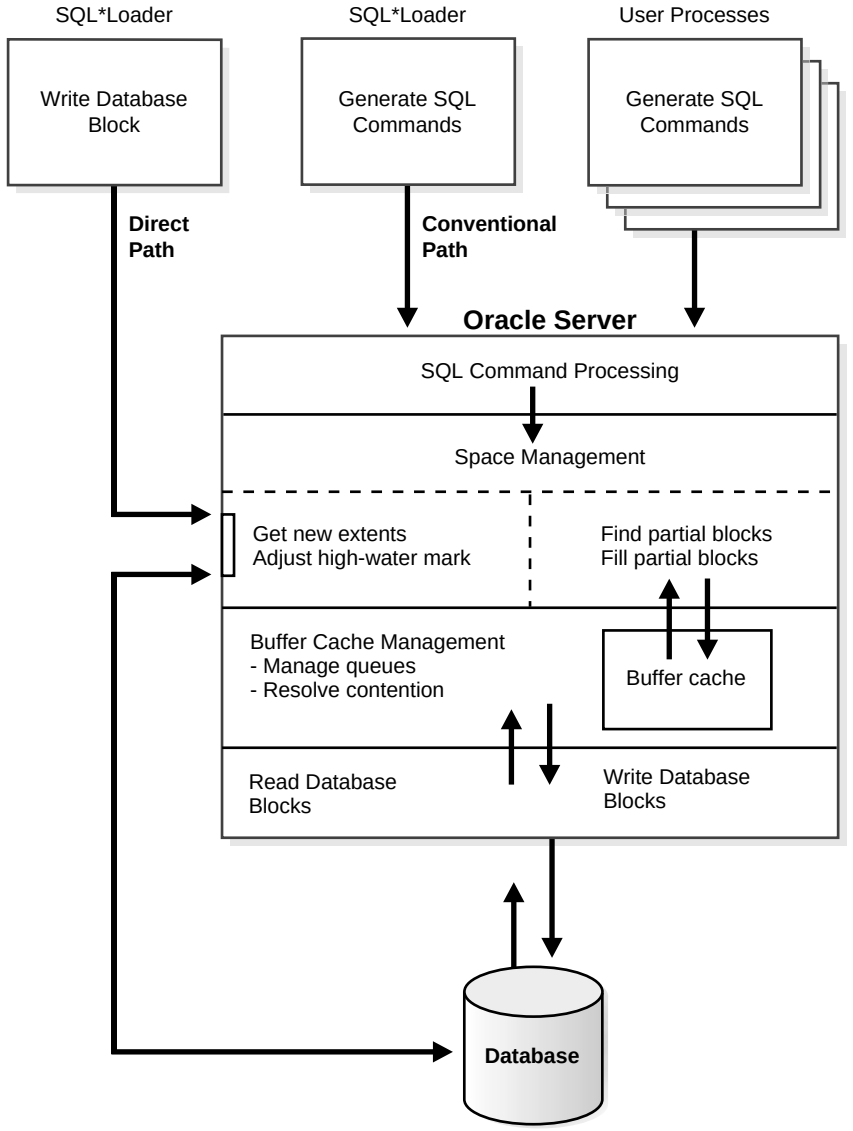
The tables to be loaded must already exist in the database. SQL*Loader never creates tables. It loads existing tables that either already contain data or are empty.

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 11–1 shows how conventional and direct path loads perform database writes.

Figure 11–1 Database Writes on SQL*Loader Direct Path and Conventional Path



Loading ROWID Columns

In both conventional path and direct path, you can specify a text value for a `ROWID` column. (This is the same text you get when you perform a `SELECT ROWID FROM table_name` operation.) The character string interpretation of the `ROWID` is converted into the `ROWID` type for a column in a table.

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 statements are generated, passed to Oracle, and executed.

The Oracle database 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 8-25

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 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 8-25

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 that 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 11–1 or set an NLS_DATE_FORMAT environment variable as shown in Example 11–2.

Example 11–1 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 11–2 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 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 11-13 for information about 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 enables 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:
  - The Oracle database has the SQL `NOARCHIVELOG` parameter enabled
  - The SQL*Loader `UNRECOVERABLE` clause is enabled
  - The object being loaded has the SQL `NOLOGGING` parameter set

  See Instance Recovery and Direct Path Loads on page 11-15.

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 database prior to Oracle9*i*, you can perform a SQL*Loader direct path load only 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.

  Beginning with Oracle9*i*, you can perform a SQL*Loader direct path load when the client and server are different versions. However, both versions must be at least release 9.0.1 and the client version must be higher than the server version. For example, you can perform a direct path load from a release 9.0.1 database into a release 9.2 database. However, you cannot use direct path load to load data from a release 10.0.0 database into a release 9.2 database.

The following features are not available with direct path load:

- 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 11-31.

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 11-25.

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 12-24
- Optimizing Performance of Direct Path Loads on page 11-17 for information about parameters you can use to optimize performance of direct path loads
- Optimizing Direct Path Loads on Multiple-CPU Systems on page 11-23 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 8-39.

> **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 11-17 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:** *Oracle Database Administrator's Guide* for information about 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 11-18 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 are 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 11-15 for more information about 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.

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 and data block space utilization.

Data Save Versus Commit

In a conventional load, ROWS is the number of rows to read before a commit operation. 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 from the loss of a database file. You must be operating in ARCHIVELOG mode to recover after you lose a database file.
- Instance - 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 can always recover from instance failures, even when redo logs are not archived.

> **See Also:** *Oracle 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:** *Oracle Database Backup and Recovery Advanced User's Guide* for more information about the RMAN 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 11-13 for information about 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 10-18
- Specifying the Number of Column Array Rows and Size of Stream Buffers on page 11-21

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 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 three 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 *Oracle 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 enables 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 about estimating storage requirements, see Temporary Segment Storage Requirements on page 11-12.

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 12-24 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. 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. A small ROWS value can also result in wasted data block space because the last data block is not written to after a save, even if the data block is not full.

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 SQL NOLOGGING parameter for 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 SQL ARCHIVELOG and NOARCHIVELOG parameters to set the archiving mode. See the *Oracle Database Administrator's Guide* for more information about archiving.

Specifying the SQL*Loader UNRECOVERABLE Clause

To save time and space in the redo log file, use the SQL*Loader UNRECOVERABLE clause in the control file when you load data. An unrecoverable load does not record loaded data in the redo log file; instead, it generates invalidation redo.

The UNRECOVERABLE clause 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 clause, 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.

Example: Specifying the UNRECOVERABLE Clause in the Control File

```
UNRECOVERABLE
LOAD DATA
INFILE 'sample.dat'
INTO TABLE emp
(ename VARCHAR2(10), empno NUMBER(4));
```

Setting the SQL NOLOGGING Parameter

If a data or index segment has the SQL NOLOGGING parameter set, then full image redo logging is disabled for that segment (invalidation redo is generated). Use of the NOLOGGING parameter 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. Note that when VARRAYs are loaded using direct path, the COLUMNARRAYROWS parameter defaults to 100 to avoid client object cache thrashing.

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 7-1.

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 11-23 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 enables 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 7-5

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 11-21 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 8-39.

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 about maintaining data integrity in the *Oracle Database 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 11-13.

Disabled Constraints

During a direct path load, the following constraints are automatically disabled by default:

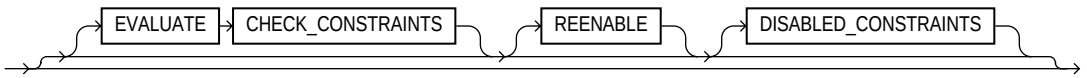
- CHECK constraints
- Referential constraints (FOREIGN KEY)

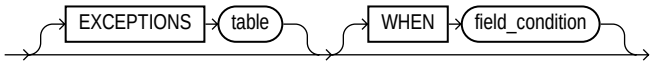
You can override the automatic disabling of CHECK constraints by specifying the EVALUATE CHECK_CONSTRAINTS clause. SQL*Loader will then evaluate CHECK constraints during a direct path load. Any row that violates the CHECK constraint is rejected. The following example shows the use of the EVALUATE CHECK_CONSTRAINTS clause in a SQL*Loader control file:

```
LOAD DATA
INFILE *
APPEND
INTO TABLE emp
EVALUATE CHECK_CONSTRAINTS
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
(c1 CHAR(10) ,c2)
BEGINDATA
Jones,10
Smith,20
Brown,30
Taylor,40
```

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 *Oracle Database 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 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 the triggers that these application insert 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 Packages and Types 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 terminated 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 *Oracle Database 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 11-31. 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

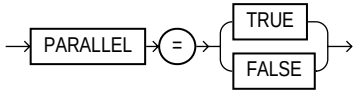
The following restrictions are enforced on parallel direct path loads:

- 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 enables 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. These options are specified with the FILE and STORAGE parameters. These parameters are valid only for parallel loads.

Using the FILE Parameter to Specify Temporary Segments

To allow for maximum I/O throughput, Oracle recommends that each concurrent direct path load session use files located on different disks. In the SQL*Loader control file, 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 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.

For example, the following OPTIONS clause could be used to specify STORAGE parameters:

```
OPTIONS (STORAGE=(INITIAL 100M NEXT 100M PCTINCREASE 0))
```

You can use the STORAGE parameter only in the SQL*Loader 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.

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 enables you to manually create the required indexes in parallel to save time before enabling the constraint.

See Also: *Oracle Database Performance Tuning Guide*

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 using `CONTINUEIF` and `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. SQL*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 character set
    - Database 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 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 8-28.

Additionally, the performance tips provided in Performance Hints When Using External Tables on page 13-10 also apply to SQL*Loader.

12

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:

The Case Studies

This chapter contains the following case studies:

- Case Study 1: Loading Variable-Length Data on page 12-5: 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 12-8: Loads data from a separate datafile.
- Case Study 3: Loading a Delimited, Free-Format File on page 12-11: 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 12-14: Combines multiple physical records into one logical record corresponding to one database row.
- Case Study 5: Loading Data into Multiple Tables on page 12-18: Loads data into multiple tables in one run.
- Case Study 6: Loading Data Using the Direct Path Load Method on page 12-24: Loads data using the direct path load method.
- Case Study 7: Extracting Data from a Formatted Report on page 12-28: Extracts data from a formatted report.
- Case Study 8: Loading Partitioned Tables on page 12-34: Loads partitioned tables.
- Case Study 9: Loading LOBFILEs (CLOBs) on page 12-38: 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 12-43: 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 12-47: 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 12–1 lists the files associated with each case.

Table 12–1 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

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 8-11.
- Loading data in stream format, with both types of delimited fields: terminated and enclosed. See Specifying Field Length for Datatypes for Which Whitespace Can Be Trimmed on page 9-47.

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
.
.
.
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 8-8.
- Data conversions. See Datatype Conversions on page 9-24.

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
```

```
.
.
.
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 9-30.
- Loading dates using the datatype DATE. See Datetime and Interval Datatypes on page 9-16.
- Using SEQUENCE numbers to generate unique keys for loaded data. See Setting a Column to a Unique Sequence Number on page 9-60.
- Using APPEND to indicate that the table need not be empty before inserting new records. See Table-Specific Loading Method on page 8-34.
- Using Comments in the control file set off by two hyphens. See Comments in the Control File on page 8-4.

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
.
.
.
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 8-28.
- 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 8-34.

- Specifying a discard file in the control file using DISCARDFILE; see Specifying the Discard File on page 8-14.
- Specifying a maximum number of discards using DISCARDMAX; see Specifying the Discard File on page 8-14.
- Rejecting records due to duplicate values in a unique index or due to invalid data values; see Criteria for Rejected Records on page 8-14.

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
.
.
.
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 8-44.
- 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 8-40.
- Using a WHEN clause. See Loading Records Based on a Condition on page 8-36.
- 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
.
.
.
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 11.
- How to specify the indexes for which the data is presorted. See Presorting Data for Faster Indexing on page 11-18.
- The `NULLIF` clause. See Using the WHEN, NULLIF, and DEFAULTIF Clauses on page 9-33.

- Loading all-blank numeric fields as NULL. See Loading All-Blank Fields on page 9-44.

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 9-33.

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
.
.
.
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 *Oracle Database Application Developer's Guide - Fundamentals* for more information about database triggers.
- Use of the SQL string to manipulate data; see Applying SQL Operators to Fields on page 9-52.
- Different initial and trailing delimiters. See Specifying Delimiters on page 9-25.
- Use of SYSDATE; see Setting a Column to the Current Date on page 9-60.
- Use of the TRAILING NULLCOLS clause; see TRAILING NULLCOLS Clause on page 8-38.
- Ambiguous field length warnings; see Conflicting Native Datatype Field Lengths on page 9-23 and Conflicting Field Lengths for Character Datatypes on page 9-29.
- Use of a discard file. See Specifying the Discard File in the Control File on page 8-15.

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 12-31.

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 12-32. 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
.
.
.
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 *Oracle Database Concepts* for more information about partitioned data concepts.
- Explicitly defined field positions and datatypes.
- Loading using the fixed record length option. See Input Data and Datafiles on page 6-5.

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
.
.
.
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 9-6 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 10-22 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
.
.
.
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 about setting the COMPATIBILITY parameter, see *Oracle Database Upgrade Guide*.

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 9-6 for more information.
2. This field is created as a REF field. See Loading REF Columns on page 10-14 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 10-1 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
.
.
.
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 8-17
   - Byte Ordering on page 9-39

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
.
.
.
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 part describe the use of external tables.

Chapter 13, "External Tables Concepts"

This chapter describes basic concepts about external tables.

Chapter 14, "The ORACLE_LOADER Access Driver"

This chapter describes the `ORACLE_LOADER` access driver.

Chapter 15, "The ORACLE_DATAPUMP Access Driver"

This chapter describes the `ORACLE_DATAPUMP` access driver, including its parameters, and information about loading and unloading supported datatypes.

13

External Tables Concepts

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

Prior to Oracle Database 10*g*, external tables were read-only. However, as of Oracle Database 10*g*, external tables can also be written to. Note that 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 13-12 for more information about 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 if the `ORACLE_LOADER` access driver is used and the datafiles are in text format. 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:

How Are External Tables Created?

External tables are created using the SQL CREATE TABLE...ORGANIZATION EXTERNAL statement. When you create an external table, you specify the following attributes:

- TYPE - specifies the type of external table. The two available types are the ORACLE_LOADER type and the ORACLE_DATAPUMP type. Each type of external table is supported by its own access driver.
  - The ORACLE_LOADER access driver is the default. It can perform only data loads, and the data must come from text datafiles. Loads from external tables to internal tables are done by reading from the external tables' text-only datafiles.
  - The ORACLE_DATAPUMP access driver can perform both loads and unloads. The data must come from binary dump files. Loads to internal tables from external tables are done by fetching from the binary dump files. Unloads from internal tables to external tables are done by populating the external tables' binary dump files.
- DEFAULT DIRECTORY - specifies the default location of files that are read or written by external tables. The location is specified with a directory object, not a directory path. See Location of Datafiles and Output Files on page 13-3 for more information.
- ACCESS PARAMETERS - describe the external data source and implements the type of external table that was specified. Each type of external table has its own access driver that provides access parameters unique to that type of external table. See Access Parameters on page 13-3.
- LOCATION - specifies the location of the external data. The location is specified as a list of directory objects and filenames. If the directory object is not specified, then the default directory object is used as the file location.

The following example shows the use of each of these attributes:

```
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 CHAR(2),
                                                                  employee_dob CHAR(20),
                                                                  employee_last_name CHAR(18),
                                                                  employee_first_name CHAR(11),
                                                                  employee_middle_name CHAR(11)))
                                      LOCATION ('info.dat'));
```

The information you provide through the access driver ensures that data from the data source is processed so that it matches the definition of the external table. The fields listed after `CREATE TABLE emp_load` are actually defining the metadata for the data in the `info.dat` source file. The access parameters are optional.

Access Parameters

When you create an external table of a particular type, you can specify access parameters to modify the default behavior of the access driver. Each access driver has its own syntax for access parameters.

See Also:

- Chapter 14, "The ORACLE_LOADER Access Driver"
- Chapter 15, "The ORACLE_DATAPUMP Access Driver"

Location of Datafiles and Output Files

The access driver runs inside 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 output files created by the access driver: the log file, bad file, and discard file, as well as any dump files created by the `ORACLE_DATAPUMP` access driver.

The access driver does not allow you to specify a complete specification for files. 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 might overwrite a file that you normally would not 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 `ext_tab_dir` that is mapped to a directory located at `/usr/apps/datafiles`.

```
CREATE DIRECTORY ext_tab_dir 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 and WRITE privileges on the directory to other users. These privileges must be explicitly granted, rather than assigned through the use of roles. For example, to allow the server to read files on behalf of user scott in the directory named by ext_tab_dir, the user who created the directory object must execute the following command:

```
GRANT READ ON DIRECTORY ext_tab_dir 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 clause 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 enables 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 means only that the Oracle database will read or write that file on your behalf. You are not given direct access to those files outside of the Oracle database unless you have the appropriate operating system privileges. Similarly, the Oracle database requires permission from the operating system to read and write files in the directories.

Example: Creating and Loading an External Table Using ORACLE_LOADER

The steps in this section show an example of using the ORACLE_LOADER access driver to create and load an external table. A traditional table named emp is defined along with an external table named emp_load. The external data is then loaded into an internal table.

1. Assume your .dat file looks as follows:

```
56november, 15, 1980  baker            mary       alice
87december, 20, 1970  roper            lisa       marie
```

2. Execute the following SQL statements to set up a default directory (which contains the data source) and to grant access to it:

```
CREATE DIRECTORY ext_tab_dir AS '/usr/apps/datafiles';
GRANT READ ON DIRECTORY ext_tab_dir TO SCOTT;
```

3. Create a traditional table named emp:

```
CREATE TABLE emp (emp_no CHAR(6), last_name CHAR(25), first_name CHAR(20),
middle_initial CHAR(1));
```

4. Create an external table named emp_load:

```
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 DELIMITED BY NEWLINE FIELDS
                       (employee_number CHAR(2),
                       employee_dob CHAR(20),
                       employee_last_name CHAR(18),
                       employee_first_name CHAR(11),
                       employee_middle_name CHAR(11)))
                       LOCATION ('info.dat'));
```

5. Load the data from the external table emp_load into the table emp:

```
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);
```

6. Perform the following select operation to verify that the information in the .dat file was loaded into the emp table:

```
SQL> SELECT * FROM emp;

EMP_NO LAST_NAME                 FIRST_NAME           M
------ ------------------------- -------------------- -
56     baker                     mary                 a
87     roper                     lisa                 m
```

Notes about this 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.

Using External Tables to Load and Unload Data

In the context of external tables, loading data refers to the act of reading data from an external table and loading it into a table in the database. Unloading data refers to the act of reading data from a table in the database and inserting it into an external table.

Note: Data can only be unloaded using the ORACLE_DATAPUMP access driver.

Loading Data

When data is loaded, the data stream is read from the files specified by the LOCATION and DEFAULT DIRECTORY clauses. The INSERT statement generates a flow of data from the external data source to the Oracle SQL engine, where data is processed. As data from the external source is parsed by the access driver and provided to the external table interface, it is converted from its external representation to its Oracle internal datatype.

Unloading Data Using the ORACLE_DATAPUMP Access Driver

To unload data, you use the ORACLE_DATAPUMP access driver. The data stream that is unloaded is in a proprietary format and contains all the column data for every row being unloaded.

An unload operation also creates a metadata stream that describes the contents of the data stream. The information in the metadata stream is required for loading the data stream. Therefore, the metadata stream is written to the datafile and placed before the data stream.

Dealing with Column Objects

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 enables 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 ('info.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);
```

Datatype Conversion During External Table Use

When data is moved into or out of an external table, it is possible that the same column will have a different datatype in each of the following three places:

- The database: This is the source when data is unloaded *into* an external table and it is the destination when data is loaded *from* an external table.
- The external table: When data is unloaded into an external table, the data from the database is converted, if necessary, to match the datatype of the column in

the external table. Also, you can apply SQL operators to the source data to change its datatype before the data gets moved to the external table. Similarly, when loading from the external table into a database, the data from the external table is automatically converted to match the datatype of the column in the database. Again, you can perform other conversions by using SQL operators in the SQL statement that is selecting from the external table. For better performance, the datatypes in the external table should match those in the database.

- The datafile: When you unload data into an external table, the datatypes for fields in the datafile exactly match the datatypes of fields in the external table. However, when you load data from the external table, the datatypes in the datafile may not match the datatypes in the external table. In this case, the data from the datafile is converted to match the datatypes of the external table. If there is an error converting a column, then the record containing that column is not loaded. For better performance, the datatypes in the datafile should match the datatypes in the external table.

Any conversion errors that occur between the datafile and the external table cause the row with the error to be ignored. Any errors between the external table and the column in the database (including conversion errors and constraint violations) cause the entire operation to terminate unsuccessfully.

When data is unloaded into an external table, data conversion occurs if the datatype of a column in the source table does not match the datatype of the column in the external table. If a conversion error occurs, then the datafile may not contain all the rows that were processed up to that point and the datafile will not be readable. To avoid problems with conversion errors causing the operation to fail, the datatype of the column in the external table should match the datatype of the column in the database. This is not always possible, because external tables do not support all datatypes. In these cases, the unsupported datatypes in the source table must be converted into a datatype that the external table can support. For example, if a source table has a LONG column, the corresponding column in the external table must be a CLOB and the SELECT subquery that is used to populate the external table must use the TO_LOB operator to load the column. For example:

```
CREATE TABLE LONG_TAB_XT (LONG_COL CLOB) ORGANIZATION EXTERNAL...SELECT TO_
LOB(LONG_COL) FROM LONG_TAB;
```

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. Each access driver supports parallel access slightly differently.

Parallel Access with ORACLE_LOADER

The ORACLE_LOADER 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

Specifying a PARALLEL clause is of value only when large amounts of data are involved.

Parallel Access with ORACLE_DATAPUMP

When you use the ORACLE_DATAPUMP access driver to unload data, the data is unloaded in parallel when the PARALLEL clause or parallel hint has been specified and when multiple locations have been specified for the external table.

Each parallel process writes to its own file. Therefore, the LOCATION clause should specify as many files as there are degrees of parallelism. If there are fewer files than the degree of parallelism specified, then the degree of parallelism will be limited to the number of files specified. If there are more files than the degree of parallelism specified, then the extra files will not be used.

In addition to unloading data, the ORACLE_DATAPUMP access driver can also load data. Parallel processes can read multiple dump files or even chunks of the same dump file concurrently. Thus, data can be loaded in parallel even if there is only one dump file, as long as that file is large enough to contain multiple file offsets. This is because when the ORACLE_DATAPUMP access driver unloads data, it periodically remembers the offset into the dump file of the start of a new data chunk and writes

that information into the file when the unload completes. For nonparallel loads, file offsets are ignored because only one process at a time can access a file. For parallel loads, file offsets are distributed among parallel processes for multiple concurrent processing on a file or within a set of files.

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 enables 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.

An additional consideration is that the access drivers use large I/O buffers for better performance. On databases with shared servers, all memory used by the access drivers comes out of the system global area (SGA). For this reason, you should be careful when using external tables on shared servers. Note that for the `ORACLE_LOADER` access driver, you can use the `READSIZE` clause in the access parameters to specify the size of the buffers.

Performance Hints Specific to the ORACLE_LOADER Access Driver

The information about performance provided in this section is specific to the `ORACLE_LOADER` access driver.

Performance can 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 DATE_CACHE on page 14-11 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.
- 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.

See Also: Choosing External Tables Versus SQL*Loader on page 6-14

External Table Restrictions

This section lists what the external tables feature 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.
- Column processing: By default, the external tables feature fetches all columns defined for an external table. This guarantees a consistent result set for all queries. However, for performance reasons you can decide to process only the referenced columns of an external table, thus minimizing the amount of data conversion and data handling required to execute a query. In this case, 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. You can change this column-processing behavior with the ALTER TABLE command.

- An external table cannot load data into a LONG column.
- When identifiers (for example, column or table names) are specified in the external table access parameters, certain values are considered to be reserved words by the access parameter parser. If a reserved word is used as an identifier, it must be enclosed in double quotation marks.

  See Also:

  - Reserved Words for the ORACLE_LOADER Access Driver on page 14-36
  - Reserved Words for the ORACLE_DATAPUMP Access Driver on page 15-18

Restrictions Specific to the ORACLE_DATAPUMP Access Driver

In addition to the restrictions just described, the ORACLE_DATAPUMP access driver has the following restrictions:

- 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 necessary only 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.
- The external tables feature does not support the use of the backslash (\) escape character within strings. See Use of the Backslash Escape Character on page 13-13.

Behavior Differences Between SQL*Loader and External Tables

This section describes important differences between loading data with external tables, using the ORACLE_LOADER access driver, as opposed to loading data with SQL*Loader conventional and direct path loads. This information does not apply to the ORACLE_DATAPUMP access driver.

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: `LONG`s, nested tables, `VARRAY`s, `REF`s, primary key `REF`s, and `SID`s

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, the settings of NLS environment variables on the *client* determine the default character set and date masks. For fields in external tables, the setting of NLS environment variables on the *server* determine the default character set and date masks.

Use of the Backslash Escape Character

In SQL*Loader, you can use the backslash (\) escape character to mark a single quotation mark as a single quotation mark, as follows:

```
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '\''
```

In external tables, the use of the backslash escape character within a string will raise an error. The workaround is to use double quotation marks to mark the separation string, as follows:

```
TERMINATED BY ',' ENCLOSED BY "'"
```

14

The ORACLE_LOADER Access Driver

This chapter describes the access parameters for the default external tables access driver, ORACLE_LOADER. You specify these access parameters when you create the external table.

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.

The following topics are discussed in this chapter:

- access_parameters Clause
- record_format_info Clause
- field_definitions Clause
- column_transforms Clause
- Reserved Words for the ORACLE_LOADER Access Driver

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 7-7

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.
- When identifiers (for example, column or table names) are specified in the external table access parameters, certain values are considered to be reserved words by the access parameter parser. If a reserved word is used as an identifier, it must be enclosed in double quotation marks. See Reserved Words for the ORACLE_LOADER Access Driver on page 14-36.

access_parameters Clause

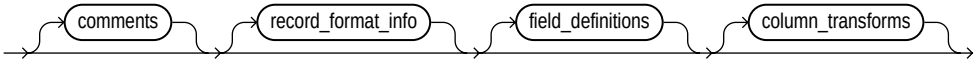
The access parameters clause contains comments, record formatting, and field formatting information.

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

As stated earlier, the access driver ensures that data from the data source is processed so that it matches the definition of the external table.

The syntax for the `access_parameters` clause is as follows:



comments

Comments are lines that begin with two hyphens 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 is an optional clause that 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. For a full description of the syntax, see record_format_info Clause on page 14-3.

field_definitions

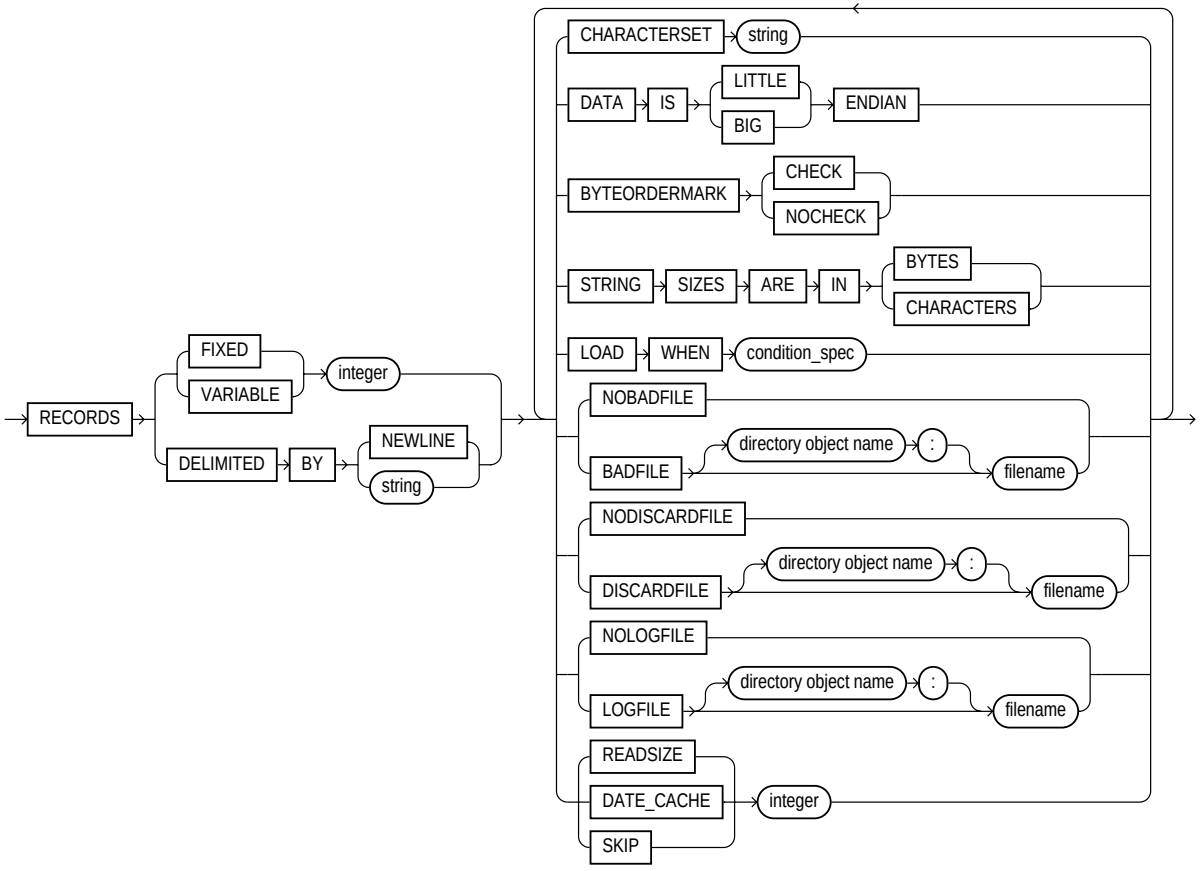
The `field_definitions` clause is used to describe the fields in the datafile. If a datafile field has the same name as a column in the external table, then the data from the field is used for that column. For a full description of the syntax, see field_definitions Clause on page 14-15.

column_transforms

The `column_transforms` clause is an optional clause used to describe how to load columns in the external table that do not map directly to columns in the datafile. This is done using the following transforms: `NULL`, `CONSTANT`, `CONCAT`, and `LOBFILE`. For a full description of the syntax, see column_transforms Clause on page 14-33.

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 ('info.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 ('info.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 14-11.

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 ('info.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 14-11.

Note: The settings of NLS environment variables on the client have no effect on the character set used for the database.

See Also: *Oracle 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
- BINARY_FLOAT
- DOUBLE
- BINARY_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

A common platform that generates little-endian data is 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. UTF-16 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.

BYTEORDERMARK (CHECK | NOCHECK)

The BYTEORDERMARK clause is used to specify whether or not the datafile should be checked for the presence of a byte-order mark (BOM). This clause is meaningful only when the character set is Unicode.

BYTEORDERMARK 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.

BYTEORDERMARK 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
  BYTEORDERMARK 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 BYTEORDERMARK 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 9-39

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 this clause is 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 references only 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 14-12.

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 to 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, and it will have an extension of .bad.

See [directory object name:] filename on page 14-13.

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 and it will have an extension of .dsc.

See [directory object name:] filename on page 14-13.

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 and it will have an extension of .log.

See [directory object name:] filename on page 14-13.

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 512 KB (524288 bytes). You must specify a larger value if any of the records in the datafile are larger than 512 KB. 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.

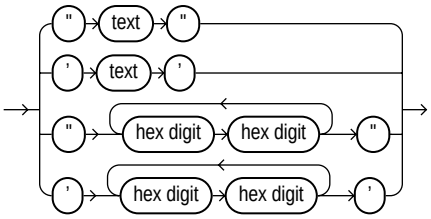
The date cache feature is enabled by default. The default date cache size is 1000 elements. If the default size is used and the number of unique input values loaded exceeds 1000, then the date cache feature is automatically disabled for that table. However, if you override the default and specify a nonzero 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 11-22

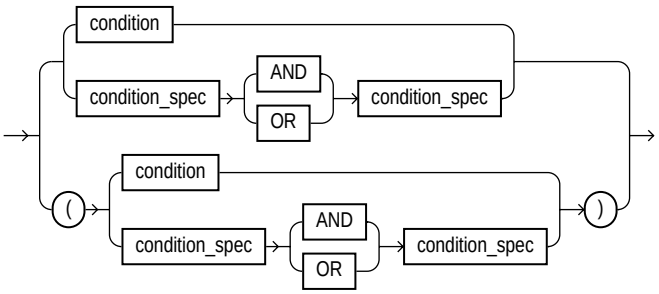
string

A string is a quoted series of characters or hexadecimal digits. If it is a series of characters, then those characters will be converted into the character set of the data file. If it is a series of hexadecimal digits, then there must be an even number of hexadecimal digits. The hexadecimal digits are converted into their binary translation, and the translation is treated as a character string in the character set of the data file. This means that once the hexadecimal digits have been converted into their binary translation, there is no other character set translation that occurs. 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 14-13

[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...ORGANIZATION 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.

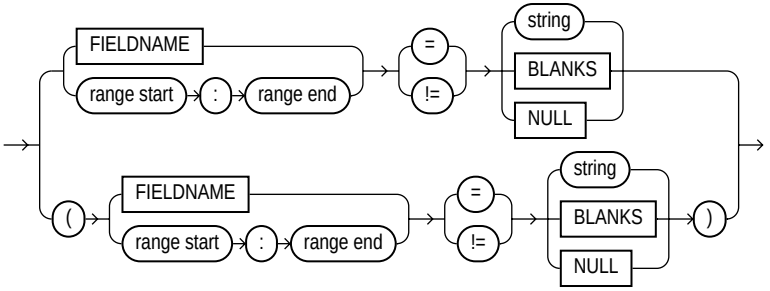
If %p or %a is not used to create unique filenames for output files and an external table is being accessed in parallel, then output files may be corrupted or agents may be unable to write to the files.

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:

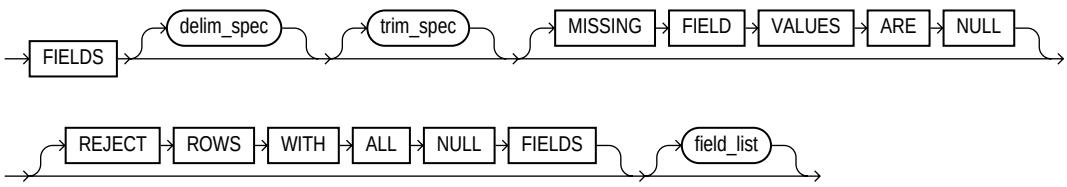
- The fields are assumed to be delimited by ','
- The fields are assumed to be character type
- The maximum length of the field is assumed to be 255
- The order of the fields in the datafile is the order in which the fields were defined in the external table
- No blanks are trimmed from the field

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 ('info.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 14-16.

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 14-19.

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 14-20.

REJECT ROWS WITH ALL NULL FIELDS

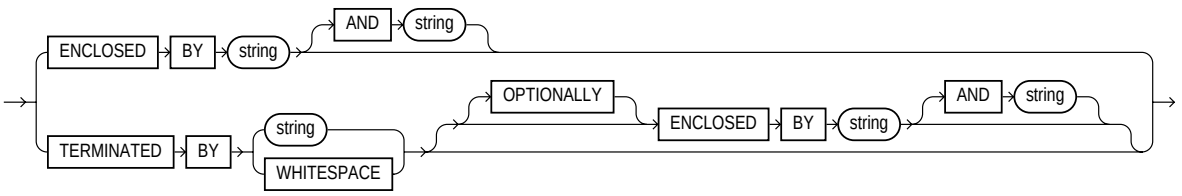
`REJECT ROWS WITH ALL NULL FIELDS` indicates that a row will not be loaded into the external table if all referenced fields in the row are null. If this parameter is not specified, the default value is to accept rows with all null fields. The setting of this parameter is written to the log file either 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 14-21.

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, leading blanks are trimmed.

> **See Also:** Table 9–5 for a description of the access driver's default trimming behavior. You can override this behavior with LTRIM and RTRIM.

After the delimiters have been found, the current position in the record is set to the spot 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.

Note: The use of the backslash character (\) within strings is not supported in external tables.

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 ('info.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 ('info.dat'));
```

```
(Alvin) ,   (Tolliver),(1976)
(Kenneth),  (Baer) ,(1963)
(Mary),(Dube) ,   (1973)
```

Example: External Table with Optional Enclosure Delimiters

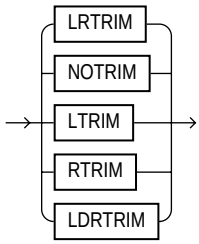
The following is an example of an external table that uses optional enclosure delimiters. Note that LRTRIM is used to trim leading and trailing blanks from fields. The example is followed by a sample of the datafile that can be used to load it.

```
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 ('info.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 ('info.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 ('info.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:

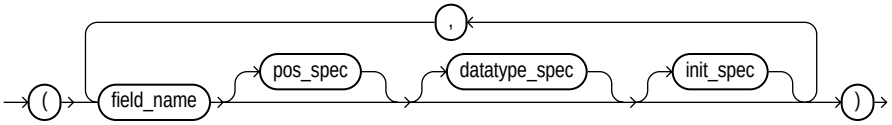
- If no datatype is specified for a field, it is assumed to be `CHAR(1)` for a nondelimited field, and `CHAR(255)`for a delimited field.
- If no field list is specified, then the fields in the datafile are assumed to be in the same order as the fields in the external table. The datatype for all fields is `CHAR(255)` unless the column in the database is `CHAR` or `VARCHAR`. If the column in the database is `CHAR` or `VARCHAR`, then the datatype for the field is still `CHAR` but the length is either 255 or the length of the column, whichever is greater.
- 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 ('info.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 14-22.

datatype_spec

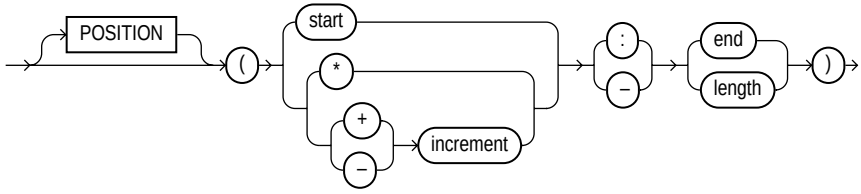
The `datatype_spec` clause indicates the datatype of the field. If `datatype_spec` is omitted, the access driver assumes the datatype is `CHAR(255)`. For a full description of the syntax, see datatype_spec Clause on page 14-24.

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 14-32.

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 *.

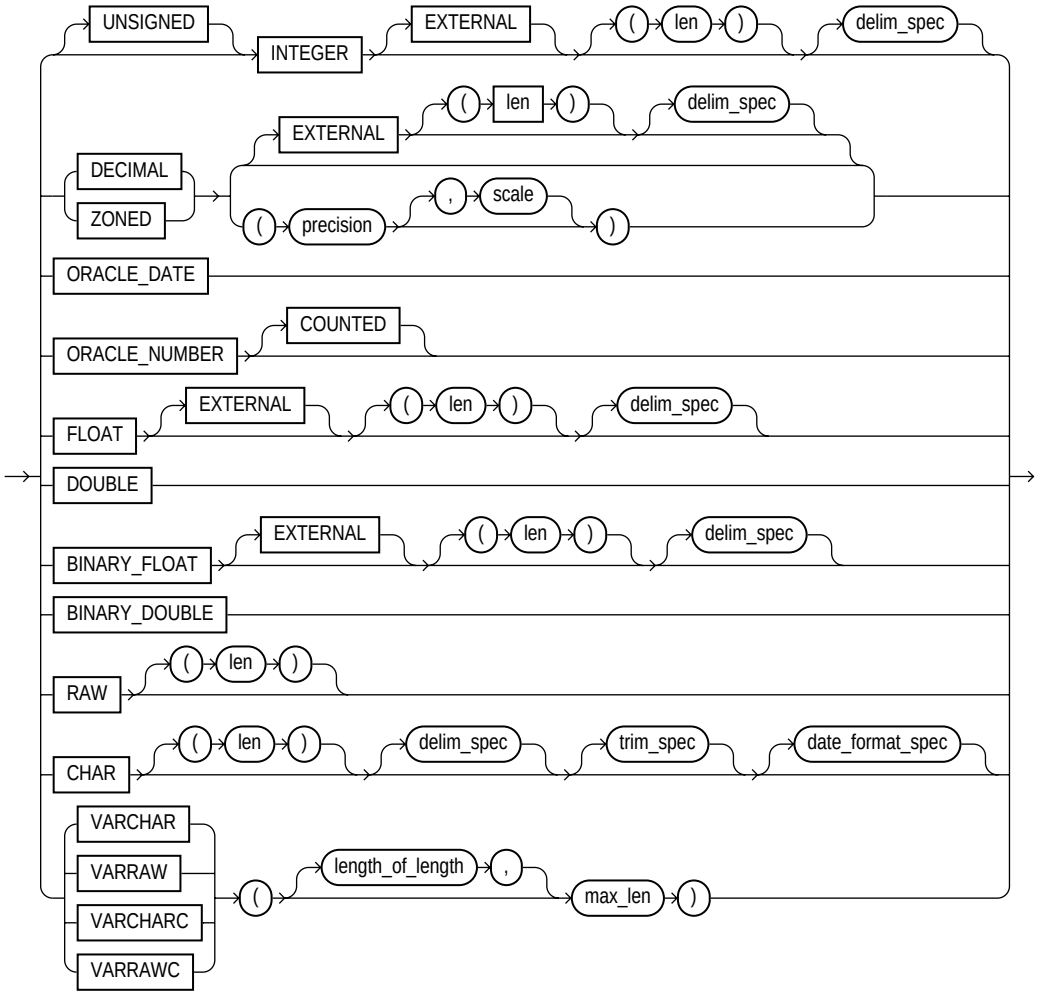
The following example shows various ways of using `pos_spec`. It is followed by a sample of the datafile that can be used to load it.

```
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 ('info.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 14-32 for more information about NULLIF and DEFAULTIF
- *Oracle Database SQL Reference* for more information about datatypes

[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 Oracle Call Interface (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.

Floating-Point Numbers

The following four datatypes, DOUBLE, FLOAT, BINARY_DOUBLE, and BINARY_FLOAT are floating-point numbers.

DOUBLE and FLOAT are the floating-point formats used natively on the platform in use. They are the same datatypes used by default for the DOUBLE and FLOAT datatypes in a C program on that platform. BINARY_FLOAT and BINARY_DOUBLE are floating-point numbers that conform substantially with the Institute for Electrical and Electronics Engineers (IEEE) Standard for Binary Floating-Point Arithmetic, IEEE Standard 754-1985. Because most platforms use the IEEE standard as their native floating-point format, FLOAT and BINARY_FLOAT are the same on those platforms and DOUBLE and BINARY_DOUBLE are also the same.

Note: See *Oracle Database SQL Reference* for more information about floating-point numbers

---

DOUBLE

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.

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. See

BINARY_DOUBLE

BINARY_DOUBLE is a 64-bit, double-precision, floating-point number datatype. Each BINARY_DOUBLE value requires 9 bytes, including a length byte. See the information in the note provided for the FLOAT datatype for more details about floating-point numbers.

BINARY_FLOAT

BINARY_FLOAT is a 32-bit, single-precision, floating-point number datatype. Each BINARY_FLOAT value requires 5 bytes, including a length byte. See the information in the note provided for the FLOAT datatype for more details about floating-point numbers.

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.

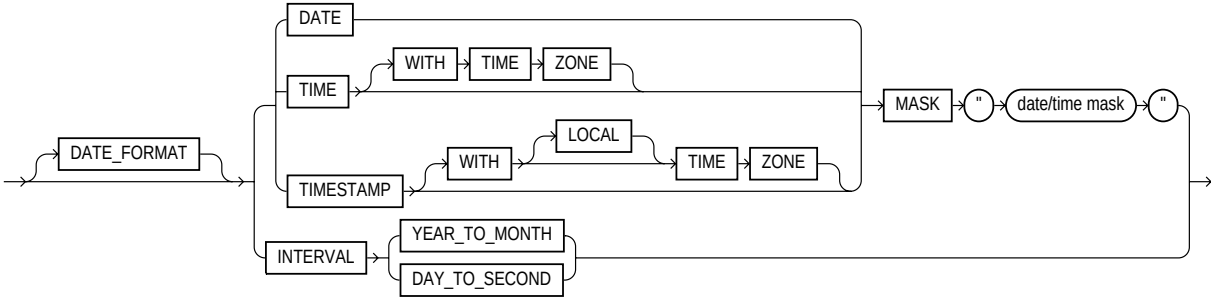
The following example shows the use of the CHAR clause. It is followed by a sample of the datafile that can be used to load it.

```
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 ('info.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 settings of NLS parameters

for the session (*not* the client settings) for the appropriate globalization parameter for the datatype are 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 14-8 for information about 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 ('info.dat'));
```

*0005*Alvin*0008*Tolliver*0000001D*Alvin 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 14-8 for information about 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.

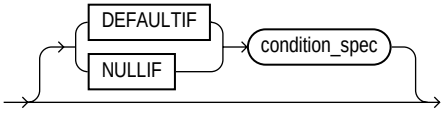
The following example shows various uses of VARCHARC and VARRAWC. The length of the picture field is 0, which means the field is set to NULL.

```
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 ('info.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:



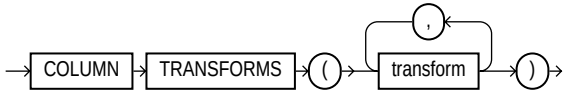
Only one NULLIF clause and only one DEFAULTIF clause can be specified for any field. These clauses behave as follows:

- 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.

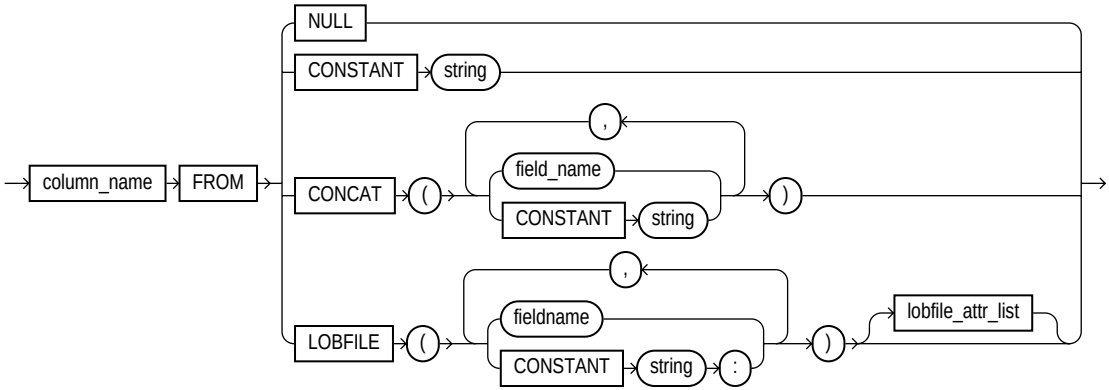
column_transforms Clause

The optional column_transforms clause provides transforms that you can use to describe how to load columns in the external table that do not map directly to columns in the datafile. The syntax for the column_transforms clause is as follows:



transform

Each transform specified in the transform clause identifies a column in the external table and then a specifies how to calculate the value of the column. The syntax is as follows:



The NULL transform is used to set the external table column to NULL in every row. The CONSTANT transform is used to set the external table column to the same value in every row. The CONCAT transform is used to set the external table column to the concatenation of constant strings and / or fields in the current record from the datafile. The LOBFILE transform is used to load data into a field for a record from

another datafile. Each of these transforms is explained further in the following sections.

column_name

The `column_name` uniquely identifies a column in the external table to be loaded. Note that if the name of a column is mentioned in the `transform` clause, then that name cannot be specified in the `FIELDS` clause as a field in the datafile.

NULL

When the `NULL` transform is specified, every value of the field is set to `NULL` for every record.

CONSTANT

The `CONSTANT` transform uses the value of the string specified as the value of the column in the record. If the column in the external table is not a character string type, then the constant string will be converted to the datatype of the column. This conversion will be done for every row.

The character set of the string used for datatype conversions is the character set of the database.

CONCAT

The `CONCAT` transform concatenates constant strings and fields in the datafile together to form one string. Only fields that are character datatypes and that are listed in the `fields` clause can be used as part of the concatenation. Other column transforms cannot be specified as part of the concatenation.

LOBFILE

The `LOBFILE` transform is used to identify a file whose contents are to be used as the value for a column in the external table. All LOBFILEs are identified by an optional directory object and a filename in the form <`directory object`>:<`filename`>. The following rules apply to use of the `LOBFILE` transform:

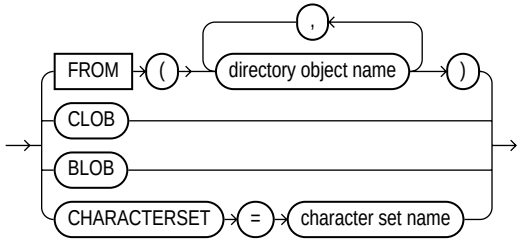
- Both the directory object and the filename can be either a constant string or the name of a field in the field clause.
- If a constant string is specified, then that string is used to find the LOBFILE for every row in the table.

- If a field name is specified, then the value of that field in the datafile is used to find the LOBFILE.
- If a field name is specified for either the directory object or the filename and if the value of that field is NULL, then the column being loaded by the LOBFILE is also set to NULL.
- If the directory object is not specified, then the default directory specified for the external table is used.
- If a field name is specified for the directory object, the FROM clause also needs to be specified.

Note that the entire file is used as the value of the LOB column. If the same file is referenced in multiple rows, then that file is reopened and reread in order to populate each column.

lobfile_attr_list

The lobfile_attr_list lists additional attributes of the LOBFILE. The syntax is as follows:



The FROM clause lists the names of all directory objects that will be used for LOBFILEs. It is used only when a field name is specified for the directory object of the name of the LOBFILE. The purpose of the FROM clause is to determine the type of access allowed to the named directory objects during initialization. If directory object in the value of field is not a directory object in this list, then the row will be rejected.

The CLOB attribute indicates that the data in the LOBFILE is character data (as opposed to RAW data). Character data may need to be translated into the character set used to store the LOB in the database.

The CHARACTERSET attribute contains the name of the character set for the data in the LOBFILEs.

The BLOB attribute indicates that the data in the LOBFILE is raw data.

If neither CLOB nor BLOB is specified, then CLOB is assumed. If no character set is specified for character LOBFILEs, then the character set of the datafile is assumed.

Reserved Words for the ORACLE_LOADER Access Driver

When identifiers (for example, column or table names) are specified in the external table access parameters, certain values are considered to be reserved words by the access parameter parser. If a reserved word is used as an identifier it must be enclosed in double quotation marks. The following are the reserved words for the ORACLE_LOADER access driver:

- ALL
- AND
- ARE
- ASTERISK
- AT
- ATSIGN
- BADFILE
- BADFILENAME
- BACKSLASH
- BENDIAN
- BIG
- BLANKS
- BY
- BYTES
- BYTESTR
- CHAR
- CHARACTERS
- CHARACTERSET
- CHARSET

- CHARSTR
- CHECK
- CLOB
- COLLENGTH
- COLON
- COLUMN
- COMMA
- CONCAT
- CONSTANT
- COUNTED
- DATA
- DATE
- DATE_CACHE
- DATE_FORMAT
- DATEMASK
- DAY
- DEBUG
- DECIMAL
- DEFAULTIF
- DELIMITBY
- DELIMITED
- DISCARDFILE
- DOT
- DOUBLE
- DOUBLETYPE
- DQSTRING
- DQUOTE
- DSCFILENAME

- ENCLOSED
- ENDIAN
- ENDPOS
- EOF
- EQUAL
- EXIT
- EXTENDED_IO_PARAMETERS
- EXTERNAL
- EXTERNALKW
- EXTPARM
- FIELD
- FIELDS
- FILE
- FILEDIR
- FILENAME
- FIXED
- FLOAT
- FLOATTYPE
- FOR
- FROM
- HASH
- HEXPREFIX
- IN
- INTEGER
- INTERVAL
- LANGUAGE
- IS
- LEFTCB

- LEFTTXTDELIM
- LEFTP
- LENDIAN
- LDRTRIM
- LITTLE
- LOAD
- LOBFILE
- LOBPC
- LOBPCCONST
- LOCAL
- LOCALTZONE
- LOGFILE
- LOGFILENAME
- LRTRIM
- LTRIM
- MAKE_REF
- MASK
- MINUSSIGN
- MISSING
- MISSINGFLD
- MONTH
- NEWLINE
- NO
- NOCHECK
- NOT
- NOBADFILE
- NODISCARDFILE
- NOLOGFILE

- NOTEQUAL
- NOTERMBY
- NOTRIM
- NULL
- NULLIF
- OID
- OPTENCLOSE
- OPTIONALLY
- OPTIONS
- OR
- ORACLE_DATE
- ORACLE_NUMBER
- PLUSSIGN
- POSITION
- PROCESSING
- QUOTE
- RAW
- READSIZE
- RECNUM
- RECORDS
- REJECT
- RIGHTCB
- RIGHTTXTDELIM
- RIGHTP
- ROW
- ROWS
- RTRIM
- SCALE

- SECOND
- SEMI
- SETID
- SIGN
- SIZES
- SKIP
- STRING
- TERMBY
- TERMEOF
- TERMINATED
- TERMWS
- TERRITORY
- TIME
- TIMESTAMP
- TIMEZONE
- TO
- TRANSFORMS
- UNDERSCORE
- UINTEGER
- UNSIGNED
- VALUES
- VARCHAR
- VARCHARC
- VARIABLE
- VARRAW
- VARRAWC
- VLENELN
- VMAXLEN

- WHEN
- WHITESPACE
- WITH
- YEAR
- ZONED

15

The ORACLE_DATAPUMP Access Driver

This chapter describes the ORACLE_DATAPUMP access driver. The following topics are discussed:

- access_parameters Clause
- Unloading and Loading Data with the ORACLE_DATAPUMP Access Driver
- Supported Datatypes
- Unsupported Datatypes
- Reserved Words for the ORACLE_DATAPUMP Access Driver

To use the information in this chapter, you must know enough about SQL to be able to create an external table and perform queries against it.

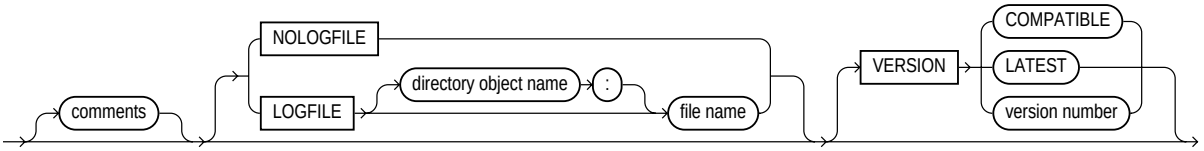
Notes:

- 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.
- When identifiers (for example, column or table names) are specified in the external table access parameters, certain values are considered to be reserved words by the access parameter parser. If a reserved word is used as an identifier, it must be enclosed in double quotation marks. See Reserved Words for the ORACLE_DATAPUMP Access Driver on page 15-18.

---

access_parameters Clause

When you create the external table, you can specify certain parameters in an `access_parameters` clause. This clause is optional, as are its individual parameters. For example, you could specify `LOGFILE`, but not `VERSION`, or vice versa. The syntax for the `access_parameters` clause is as follows:



comments

Comments are lines that begin with two hyphens followed by text. Comments must be placed *before* any access parameters, for example:

```
--This is a comment.
--This is another comment.
NOLOG
```

All text to the right of the double hyphen is ignored, until the end of the line.

LOGFILE | NOLOGFILE

`LOGFILE` specifies the name of the log file that contains any messages generated while the dump file was being accessed. `NOLOGFILE` prevents the creation of a log file. If a directory object is not specified as part of the log file name, then the directory object specified by the `DEFAULT DIRECTORY` attribute is used. If a directory object is not specified and no default directory was specified, an error is returned.

If `LOGFILE` is not specified, a log file is created in the default directory and the name of the log file is generated from the table name and the process ID with an extension of `.log`. If a log file already exists by the same name, the access driver reopens that log file and appends the new log information to the end. See Filenames for LOGFILE on page 15-3 for information about using wildcards to create unique filenames during parallel loads or unloads.

See Example of LOGFILE Usage for ORACLE_DATAPUMP on page 15-3.

Filenames for LOGFILE

The access driver does some symbol substitution to help make filenames unique in the case of parallel loads. The symbol substitutions supported are as follows:

- %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 `exttab_%a.log` was specified as the filename, then the agent would create a file named `exttab_003.log`.
- %% is replaced by '%'. If there is a need to have a percent sign in the filename, then this symbol substitution must be used.

If the '%' character is followed by anything other than one of the characters in the preceding list, 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, output files may be corrupted or agents may be unable to write to the files.

If no extension is supplied for the file, a default extension of `.log` will be used. If the name generated is not a valid filename, an error is returned and no data is loaded or unloaded.

Example of LOGFILE Usage for ORACLE_DATAPUMP

In the following example, the dump file, `dept_dmp`, is in the directory identified by the directory object, `load_dir`, but the log file, `deptxt.log`, is in the directory identified by the directory object, `log_dir`.

```
CREATE TABLE dept_xt (dept_no INT, dept_name CHAR(20), location CHAR(20))
ORGANIZATION EXTERNAL (TYPE ORACLE_DATAPUMP DEFAULT DIRECTORY load_dir
ACCESS PARAMETERS (LOGFILE log_dir:deptxt) LOCATION ('dept_dmp'));
```

VERSION Clause

The `VERSION` clause is used when data is unloaded from an Oracle database version that is later than the database version where the data will be loaded. In this case, the `VERSION` clause should be specified for the external table used to unload

the data and the value specified should be the version that will be used to read the data.

Unloading and Loading Data with the ORACLE_DATAPUMP Access Driver

The ORACLE_DATAPUMP access driver can be used to populate a file with data. The data in the file is written in a binary format that can only be read by the ORACLE_DATAPUMP access driver. Once the file has been populated with data, that file can be used as the dump file for another external table in the same database or in a different database.

The following steps use the sample schema, oe, to show an extended example of how you can use the ORACLE_DATAPUMP access driver to unload and load data. (The example assumes that the directory object def_dir1 already exists, and that user oe has read and write access to it.)

1. An external table will populate a file with data only as part of creating the external table with the AS SELECT clause. The following example creates an external table named inventories_xt and populates the dump file for the external table with the data from table inventories in the oe schema.

```
SQL> CREATE TABLE inventories_xt
  2  ORGANIZATION EXTERNAL
  3  (
  4    TYPE ORACLE_DATAPUMP
  5    DEFAULT DIRECTORY def_dir1
  6    LOCATION ('inv_xt.dmp')
  7  )
  8  AS SELECT * FROM inventories;

Table created.
```

2. Describe both inventories and the new external table, as follows. They should both match.

```
SQL> DESCRIBE inventories
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------
 PRODUCT_ID                                NOT NULL NUMBER(6)
 WAREHOUSE_ID                              NOT NULL NUMBER(3)
 QUANTITY_ON_HAND                          NOT NULL NUMBER(8)

SQL> DESCRIBE inventories_xt
```

```
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------
 PRODUCT_ID                                NOT NULL NUMBER(6)
 WAREHOUSE_ID                              NOT NULL NUMBER(3)
 QUANTITY_ON_HAND                          NOT NULL NUMBER(8)
```

3. Now that the external table is created, it can be queried just like any other table. For example, select the count of records in the external table, as follows:

```
SQL> SELECT COUNT(*) FROM inventories_xt;

  COUNT(*)
----------
      1112
```

4. Compare the data in the external table against the data in `inventories`. There should be no differences.

```
SQL> SELECT * FROM inventories MINUS SELECT * FROM inventories_xt;

no rows selected
```

5. After an external table has been created and the dump file populated by the CREATE TABLE AS SELECT statement, no rows may be added, updated, or deleted from the external table. Any attempt to modify the data in the external table will fail with an error.

   The following example shows an attempt to use data manipulation language (DML) on an existing external table. This will return an error, as shown.

```
SQL> DELETE FROM inventories_xt WHERE warehouse_id = 5;
DELETE FROM inventories_xt WHERE warehouse_id = 5
            *
ERROR at line 1:
ORA-30657: operation not supported on external organized table
```

6. The dump file created for the external table can now be moved and used as the dump file for another external table in the same database or different database. Note that when you create an external table that uses an existing file, there is no AS SELECT clause for the CREATE TABLE statement.

```
SQL> CREATE TABLE inventories_xt2
  2  (
  3    product_id          NUMBER(6),
  4    warehouse_id        NUMBER(3),
  5    quantity_on_hand    NUMBER(8)
```

```
 6  )
 7  ORGANIZATION EXTERNAL
 8  (
 9    TYPE ORACLE_DATAPUMP
10    DEFAULT DIRECTORY def_dir1
11    LOCATION ('inv_xt.dmp')
12  );

Table created.
```

7. Compare the data for the new external table against the data in the `inventories` table. The `product_id` field will be converted to a compatible datatype before the comparison is done. There should be no differences.

```
SQL> SELECT * FROM inventories MINUS SELECT * FROM inventories_xt2;

no rows selected
```

8. Create an external table with three dump files and with a degree of parallelism of three.

```
SQL> CREATE TABLE inventories_xt3
  2  ORGANIZATION EXTERNAL
  3  (
  4    TYPE ORACLE_DATAPUMP
  5    DEFAULT DIRECTORY def_dir1
  6    LOCATION ('inv_xt1.dmp', 'inv_xt2.dmp', 'inv_xt3.dmp')
  7  )
  8  PARALLEL 3
  9  AS SELECT * FROM inventories;

Table created.
```

9. Compare the data unload against `inventories`. There should be no differences.

```
SQL> SELECT * FROM inventories MINUS SELECT * FROM inventories_xt3;

no rows selected
```

10. Create an external table containing some rows from table `inventories`.

```
SQL> CREATE TABLE inv_part_xt
  2  ORGANIZATION EXTERNAL
  3  (
  4  TYPE ORACLE_DATAPUMP
```

```
  5  DEFAULT DIRECTORY def_dir1
  6  LOCATION ('inv_p1_xt.dmp')
  7  )
  8  AS SELECT * FROM inventories WHERE warehouse_id < 5;

Table created.
```

11. Create another external table containing the rest of the rows from `inventories`.

```
SQL> drop table inv_part_xt;

Table dropped.

SQL>
SQL> CREATE TABLE inv_part_xt
  2  ORGANIZATION EXTERNAL
  3  (
  4  TYPE ORACLE_DATAPUMP
  5  DEFAULT DIRECTORY def_dir1
  6  LOCATION ('inv_p2_xt.dmp')
  7  )
  8  AS SELECT * FROM inventories WHERE warehouse_id >= 5;

Table created.
```

12. Create an external table that uses the two dump files created in Steps 10 and 11.

```
SQL> CREATE TABLE inv_part_all_xt
  2  (
  3  product_id NUMBER(6),
  4  warehouse_id NUMBER(3),
  5  quantity_on_hand NUMBER(8)
  6  )
  7  ORGANIZATION EXTERNAL
  8  (
  9  TYPE ORACLE_DATAPUMP
 10  DEFAULT DIRECTORY def_dir1
 11  LOCATION ('inv_p1_xt.dmp','inv_p2_xt.dmp')
 12  );

Table created.
```

13. Compare the new external table to the `inventories` table. There should be no differences. This is because the two dump files used to create the external table

have the same metadata (for example, the same table name inv_part_xt and the same column information)

```
SQL> SELECT * FROM inventories MINUS SELECT * FROM inv_part_all_xt;

no rows selected
```

Parallel Loading and Unloading

The dump file must be on a disk big enough to hold all the data being written. If there is insufficient space for all of the data, then an error will be returned for the CREATE TABLE AS SELECT statement. One way to alleviate the problem is to create multiple files in multiple directory objects (assuming those directories are on different disks) when executing the CREATE TABLE AS SELECT statement. Multiple files can be created by specifying multiple locations in the form directory:file in the LOCATION clause and by specifying the PARALLEL clause. Each parallel I/O server process that is created to populate the external table writes to its own file. The number of files in the LOCATION clause should match the degree of parallelization because each I/O server process requires its own files. Any extra files that are specified will be ignored. If there are not enough files for the degree of parallelization specified, then the degree of parallelization will be lowered to match the number of files in the LOCATION clause.

Here is an example of unloading the inventories table into three files.

```
SQL> CREATE TABLE inventories_XT_3
  2  ORGANIZATION EXTERNAL
  3  (
  4    TYPE ORACLE_DATAPUMP
  5    DEFAULT DIRECTORY def_dir1
  6    LOCATION ('inv_xt1.dmp', 'inv_xt2.dmp', 'inv_xt3.dmp')
  7  )
  8  PARALLEL 3
  9  AS SELECT * FROM oe.inventories;

Table created.
```

The degree of parallelization is not tied to the number of files in the LOCATION clause when reading from ORACLE_DATAPUMP external tables. There is information in the dump files so that multiple parallel I/O server processes can read different portions of the same file. So, even if there is only one dump file, the degree of parallelization can be increased to speed the time required to read the file.

Combining Dump Files

Dump files populated by different external tables can all be specified in the LOCATION clause of another external table. For example, data from different production databases can be unloaded into separate files, and then those files can all be included in an external table defined in a data warehouse. This provides an easy way of aggregating data from multiple sources. The only restriction is that the metadata for all of the external tables be exactly the same. This means that the character set, time zone, schema name, table name, and column names must all match. Also, the columns must be defined in the same order, and their datatypes must be exactly alike. This means that after you create the first external table you must drop it so that you can use the same table name for the second external table. This ensures that the metadata listed in the two dump files is the same and they can be used together to create the same external table.

```
SQL> CREATE TABLE inv_part_1_xt
  2  ORGANIZATION EXTERNAL
  3  (
  4    TYPE ORACLE_DATAPUMP
  5    DEFAULT DIRECTORY def_dir1
  6    LOCATION ('inv_p1_xt.dmp')
  7  )
  8  AS SELECT * FROM oe.inventories WHERE warehouse_id < 5;

Table created.

SQL> DROP TABLE inv_part_1_xt;

SQL> CREATE TABLE inv_part_1_xt
  2  ORGANIZATION EXTERNAL
  3  (
  4    TYPE ORACLE_DATAPUMP
  5    DEFAULT directory def_dir1
  6    LOCATION ('inv_p2_xt.dmp')
  7  )
  8  AS SELECT * FROM oe.inventories WHERE warehouse_id >= 5;

Table created.

SQL> CREATE TABLE inv_part_all_xt
  2  (
  3    PRODUCT_ID          NUMBER(6),
  4    WAREHOUSE_ID        NUMBER(3),
  5    QUANTITY_ON_HAND    NUMBER(8)
```

```
  6  )
  7  ORGANIZATION EXTERNAL
  8  (
  9    TYPE ORACLE_DATAPUMP
 10    DEFAULT DIRECTORY def_dir1
 11    LOCATION ('inv_p1_xt.dmp','inv_p2_xt.dmp')
 12  );

Table created.

SQL> SELECT * FROM inv_part_all_xt MINUS SELECT * FROM oe.inventories;

no rows selected
```

Supported Datatypes

You may encounter the following situations when you use external tables to move data between databases:

- The database character set and the database national character set may be different between the two platforms.
- The endianness of the platforms for the two databases may be different.

The ORACLE_DATAPUMP access driver automatically resolves some of these situations.

The following datatypes are automatically converted during loads and unloads:

- Character (CHAR, NCHAR, VARCHAR2, NVARCHAR2)
- RAW
- NUMBER
- Date/Time
- BLOB
- CLOB and NCLOB
- ROWID and UROWID

If you attempt to use a datatype that is not supported for external tables, you will receive an error. This is demonstrated in the following example, in which the unsupported datatype, LONG, is used:

```
SQL> CREATE TABLE bad_datatype_xt
  2  (
```

```
  3    product_id             NUMBER(6),
  4    language_id            VARCHAR2(3),
  5    translated_name        NVARCHAR2(50),
  6    translated_description LONG
  7  )
  8  ORGANIZATION EXTERNAL
  9  (
 10    TYPE ORACLE_DATAPUMP
 11    DEFAULT DIRECTORY def_dir1
 12    LOCATION ('proddesc.dmp')
 13  );
  translated_description LONG
  *
ERROR at line 6:
ORA-30656: column type not supported on external organized table
```

See Also: Unsupported Datatypes on page 15-11

Unsupported Datatypes

An external table supports a subset of all possible datatypes for columns. In particular, it supports character datatypes (except LONG), the RAW datatype, all numeric datatypes, and all date, timestamp, and interval datatypes.

This section describes how you can use the ORACLE_DATAPUMP access driver to unload and reload data for some of the unsupported datatypes, specifically:

- BFILE
- LONG and LONG RAW
- Final object types
- Tables of final object types

Unloading and Loading BFILE Datatypes

The BFILE datatype has two pieces of information stored in it: the directory object for the file and the name of the file within that directory object.

You can unload BFILE columns using the ORACLE_DATAPUMP access driver by storing the directory object name and the filename in two columns in the external table. The procedure DBMS_LOB.FILEGETNAME will return both parts of the name. However, because this is a procedure, it cannot be used in a SELECT statement.

Instead, two functions are needed. The first will return the name of the directory object, and the second will return the name of the file.

The steps in the following extended example demonstrate the unloading and loading of BFILE datatypes.

1. Create a function to extract the directory object for a BFILE column. Note that if the column is NULL, then NULL is returned.

```
SQL> CREATE FUNCTION get_dir_name (bf BFILE) RETURN VARCHAR2 IS
  2  DIR_ALIAS VARCHAR2(255);
  3  FILE_NAME VARCHAR2(255);
  4  BEGIN
  5    IF bf is NULL
  6    THEN
  7      RETURN NULL;
  8    ELSE
  9      DBMS_LOB.FILEGETNAME (bf, dir_alias, file_name);
 10      RETURN dir_alias;
 11    END IF;
 12  END;
 13  /

Function created.
```

2. Create a function to extract the filename for a BFILE column.

```
SQL> CREATE FUNCTION get_file_name (bf BFILE) RETURN VARCHAR2 is
  2  dir_alias VARCHAR2(255);
  3  file_name VARCHAR2(255);
  4  BEGIN
  5    IF bf is NULL
  6    THEN
  7      RETURN NULL;
  8    ELSE
  9      DBMS_LOB.FILEGETNAME (bf, dir_alias, file_name);
 10      RETURN file_name;
 11    END IF;
 12  END;
 13  /

Function created.
```

3. You can then add a row with a NULL value for the BFILE column, as follows:

```
SQL> INSERT INTO PRINT_MEDIA (product_id, ad_id, ad_graphic)
  2  VALUES (3515, 12001, NULL);
```

```
1 row created.
```

You can use the newly created functions to populate an external table. Note that the functions should set columns ad_graphic_dir and ad_graphic_file to NULL if the BFILE column is NULL.

4. Create an external table to contain the data from the print_media table. Use the get_dir_name and get_file_name functions to get the components of the BFILE column.

```
SQL> CREATE TABLE print_media_xt
  2  ORGANIZATION EXTERNAL
  3  (
  4    TYPE oracle_datapump
  5    DEFAULT DIRECTORY def_dir1
  6    LOCATION ('pm_xt.dmp')
  7  ) AS
  8  SELECT product_id, ad_id,
  9         get_dir_name (ad_graphic) ad_graphic_dir,
 10         get_file_name(ad_graphic) ad_graphic_file
 11  FROM print_media;

Table created.
```

5. Create a function to load a BFILE column from the data that is in the external table. This function will return NULL if the ad_graphic_dir column in the external table is NULL.

```
SQL> CREATE FUNCTION get_bfile (dir VARCHAR2, file VARCHAR2) RETURN
BFILE is
  2  bf BFILE;
  3  BEGIN
  4    IF dir IS NULL
  5    THEN
  6      RETURN NULL;
  7    ELSE
  8      RETURN BFILENAME(dir,file);
  9    END IF;
 10  END;
 11  /

Function created.
```

6. The `get_bfile` function can be used to populate a new table containing a `BFILE` column.

```
SQL> CREATE TABLE print_media_int AS
  2  SELECT product_id, ad_id,
  3         get_bfile (ad_graphic_dir, ad_graphic_file) ad_graphic
  4  FROM print_media_xt;

Table created.
```

7. The data in the columns of the newly loaded table should match the data in the columns of the `print_media` table.

```
SQL> SELECT product_id, ad_id,
  2         get_dir_name(ad_graphic),
  3         get_file_name(ad_graphic)
  4  FROM print_media_int
  5  MINUS
  6  SELECT product_id, ad_id,
  7         get_dir_name(ad_graphic),
  8         get_file_name(ad_graphic)
  9  FROM print_media;

no rows selected
```

Unloading LONG and LONG RAW Datatypes

The ORACLE_DATAPUMP access driver can be used to unload LONG and LONG RAW columns, but that data can only be loaded back into LOB fields. The steps in the following extended example demonstrate the unloading of LONG and LONG RAW datatypes.

1. If a table to be unloaded contains a LONG or LONG RAW column, then define the corresponding columns in the external table as CLOB for LONG columns or BLOB for LONG RAW columns.

```
SQL> CREATE TABLE long_tab
  2  (
  3    key                   SMALLINT,
  4    description           LONG
  5  );

Table created.

SQL> INSERT INTO long_tab VALUES (1, 'Description Text');
```

```
1 row created.
```

2. Now, an external table can be created that contains a CLOB column to contain the data from the LONG column. Note that when loading the external table, the TO_LOB operator is used to convert the LONG column into a CLOB.

```
SQL> CREATE TABLE long_tab_xt
  2  ORGANIZATION EXTERNAL
  3  (
  4    TYPE ORACLE_DATAPUMP
  5    DEFAULT DIRECTORY def_dir1
  6    LOCATION ('long_tab_xt.dmp')
  7  )
  8  AS SELECT key, TO_LOB(description) description FROM long_tab;

Table created.
```

3. The data in the external table can be used to create another table exactly like the one that was unloaded except the new table will contain a LOB column instead of a LONG column.

```
SQL> CREATE TABLE lob_tab
  2  AS SELECT * from long_tab_xt;

Table created.
```

4. Verify that the table was created correctly.

```
SQL> SELECT * FROM lob_tab;

       KEY  DESCRIPTION
---------- --------------------------------------------------------------------------------
         1  Description Text
```

Unloading and Loading Columns Containing Final Object Types

Final column objects are populated into an external table by moving each attribute in the object type into a column in the external table. In addition, the external table needs a new column to track whether the column object is atomically null. The following steps demonstrate the unloading and loading of columns containing final object types.

1. In the following example, the warehouse column in the external table is used to track whether the warehouse column in the source table is atomically NULL.

```
SQL> CREATE TABLE inventories_obj_xt
```

```
 2  ORGANIZATION EXTERNAL
 3  (
 4    TYPE ORACLE_DATAPUMP
 5    DEFAULT DIRECTORY def_dir1
 6    LOCATION ('inv_obj_xt.dmp')
 7  )
 8  AS
 9  SELECT oi.product_id,
10         DECODE (oi.warehouse, NULL, 0, 1) warehouse,
11         oi.warehouse.location_id location_id,
12         oi.warehouse.warehouse_id warehouse_id,
13         oi.warehouse.warehouse_name warehouse_name,
14         oi.quantity_on_hand
15  FROM oc_inventories oi;

Table created.
```

The columns in the external table containing the attributes of the object type can now be used as arguments to the type constructor function when loading a column of that type. Note that the `warehouse` column in the external table is used to determine whether to call the constructor function for the object or set the column to `NULL`.

2. Load a new internal table that looks exactly like the `oc_inventories` view. (The use of the `WHERE 1=0` clause creates a new table that looks exactly like the old table but does not copy any data from the old table into the new table.)

```
SQL> CREATE TABLE oc_inventories_2 AS SELECT * FROM oc_inventories
WHERE 1 = 0;

Table created.

SQL> INSERT INTO oc_inventories_2
  2  SELECT product_id,
  3         DECODE (warehouse, 0, NULL,
  4                 warehouse_typ(warehouse_id, warehouse_name,
  5                 location_id)), quantity_on_hand
  6  FROM inventories_obj_xt;

1112 rows created.
```

Tables of Final Object Types

Object tables have an object identifier that uniquely identifies every row in the table. The following situations can occur:

- If there is no need to unload and reload the object identifier, then the external table only needs to contain fields for the attributes of the type for the object table.
- If the object identifier (OID) needs to be unloaded and reloaded and the OID for the table is one or more fields in the table, (also known as primary-key-based OIDs), then the external table has one column for every attribute of the type for the table.
- If the OID needs to be unloaded and the OID for the table is system-generated, then the procedure is more complicated. In addition to the attributes of the type, another column needs to be created to hold the system-generated OID.

The steps in the following example demonstrate this last situation.

1. Create a table of a type with system-generated OIDs:

```
SQL> CREATE TYPE person AS OBJECT (name varchar2(20)) NOT FINAL
  2  /

Type created.

SQL> CREATE TABLE people OF person;

Table created.

SQL> INSERT INTO people VALUES ('Euclid');

1 row created.
```

2. Create an external table in which the column `OID` is used to hold the column containing the system-generated OID.

```
SQL> CREATE TABLE people_xt
  2  ORGANIZATION EXTERNAL
  3  (
  4    TYPE ORACLE_DATAPUMP
  5    DEFAULT DIRECTORY def_dir1
  6    LOCATION ('people.dmp')
  7  )
  8  AS SELECT SYS_NC_OID$ oid, name FROM people;

Table created.
```

3. Create another table of the same type with system-generated OIDs. Then, execute an INSERT statement to load the new table with data unloaded from the old table.

```
SQL> CREATE TABLE people2 OF person;

Table created.

SQL>
SQL> INSERT INTO people2 (SYS_NC_OID$, SYS_NC_ROWINFO$)
  2  SELECT oid, person(name) FROM people_xt;

1 row created.

SQL>
SQL> SELECT SYS_NC_OID$, name FROM people
  2  MINUS
  3  SELECT SYS_NC_OID$, name FROM people2;

no rows selected
```

Reserved Words for the ORACLE_DATAPUMP Access Driver

When identifiers (for example, column or table names) are specified in the external table access parameters, certain values are considered to be reserved words by the access parameter parser. If a reserved word is used as an identifier, it must be enclosed in double quotation marks. The following are the reserved words for the ORACLE_DATAPUMP access driver:

- BADFILE
- COMPATIBLE
- DATAPUMP
- DEBUG
- INTERNAL
- JOB
- LATEST
- LOGFILE
- NOBADFILE
- NOLOGFILE

- PARALLEL
- TABLE
- VERSION
- WORKERID

Part IV

Other Utilities

This part contains the following chapters:

Chapter 16, "DBVERIFY: Offline Database Verification Utility"

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

Chapter 17, "DBNEWID Utility"

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

Chapter 18, "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.

Chapter 19, "Using LogMiner to Analyze Redo Log Files"

This chapter describes the Oracle LogMiner utility, which enables you to query redo logs through a SQL interface.

Chapter 20, "Original Export and Import"

This chapter describes the original Export and Import utilities.

16

DBVERIFY: Offline Database Verification Utility

DBVERIFY is an external command-line utility that performs a physical data structure integrity check. It can be used on offline or online databases, as well on backup 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. Both interfaces are started with the `dbv` command. The following sections provide descriptions of these interfaces:

- Using DBVERIFY to Validate Disk Blocks of a Single Datafile
- Using DBVERIFY to Validate a Segment

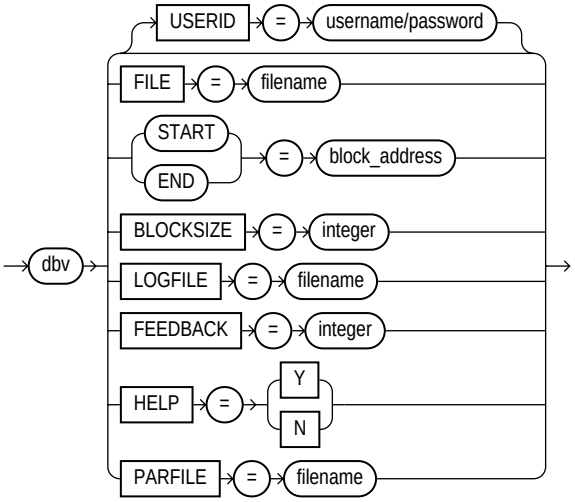
Using DBVERIFY to Validate Disk Blocks of a Single Datafile

In this mode, DBVERIFY scans one or more disk blocks of a single datafile and performs page checks.

Note: If the file you are verifying is an Automatic Storage Management (ASM) file, you must supply a USERID. This is because DBVERIFY needs to connect to an Oracle instance to access ASM files.

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
USERID	Specifies your username and password. This parameter is only necessary when the files being verified are ASM files.
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.

Parameter	Description
END	The ending block address to verify. If you do not specify END, DBVERIFY defaults to the last block in the file.
BLOCKSIZE	BLOCKSIZE is required only if the file to be verified 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 enables 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 is a sample verification of the file t_db1.dbf. The feedback parameter has been given the value 100 to display one period (.) for every 100 pages processed. A portion of the resulting output is also shown.

```
% dbv FILE=t_db1.dbf FEEDBACK=100
.
.
.
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

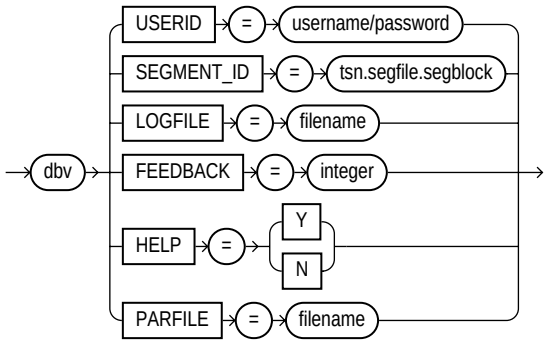
In this mode, DBVERIFY enables you to specify a table segment or index segment for verification. It checks to make sure that a row chain pointer is within the segment being verified.

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 enables 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
```

17

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?

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. This is similar to creating a database except that the data is already in the datafiles. After you change the DBID, backups and archive logs that were created prior to the change can no longer be used because they still have the original DBID, which does not match the current 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 *Oracle 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.

Note: Do not change the DBID or DBNAME of a database if you are using a capture process to capture changes to the database. See *Oracle Streams Concepts and Administration* for more information about capture processes.

Considerations for Global Database Names

If you are dealing with a database in a distributed database system, then each database should have a unique global database name. *The DBNEWID utility does not change global database names.* This can only be done with the SQL ALTER DATABASE statement, for which the syntax is as follows:

```
ALTER DATABASE RENAME GLOBAL_NAME TO <newname>.<domain>;
```

The global database name is made up of a database name and a domain, which are determined by the DB_NAME and DB_DOMAIN initialization parameters when the database is first created.

The following example changes the database name to sales in the domain us.oracle.com:

```
ALTER DATABASE RENAME GLOBAL_NAME TO sales.us.oracle.com
```

You would do this after you finished using DBNEWID to change the database name.

> **See Also:** *Oracle Database Administrator's Guide* for more information about global database names

Changing the DBID and DBNAME of a Database

This section contains these topics:

- Changing the DBID and Database Name
- Changing Only the Database ID
- Changing Only the Database Name
- Troubleshooting DBNEWID

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_db`:

   ```
   % nid TARGET=SYS/oracle@test DBNAME=test_db
   ```

   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 (and the DBNAME, if

specified, as in this example) for each datafile, including offline normal and read-only datafiles, shuts down the database, and then exits. The following is an example of what the output for this would look like:

```
.
.
.
Connected to database PROD (DBID=86997811)

Connected to server version 10.1.0

Control Files in database:
    /oracle/TEST_DB/data/cf1.f
    /oracle/TEST_DB/data/cf2.f

The following datafiles are offline clean:
    /oracle/TEST_DB/data/tbs_61.f (23)
    /oracle/TEST_DB/data/tbs_62.f (24)
    /oracle/TEST_DB/data/temp3.f (3)
These files must be writable by this utility.

The following datafiles are read-only:
    /oracle/TEST_DB/data/tbs_51.f (15)
    /oracle/TEST_DB/data/tbs_52.f (16)
    /oracle/TEST_DB/data/tbs_53.f (22)
These files must be writable by this utility.

Changing database ID from 86997811 to 1250654267
Changing database name from PROD to TEST_DB
    Control File /oracle/TEST_DB/data/cf1.f - modified
    Control File /oracle/TEST_DB/data/cf2.f - modified
    Datafile /oracle/TEST_DB/data/tbs_01.f - dbid changed, wrote new name
    Datafile /oracle/TEST_DB/data/tbs_ax1.f - dbid changed, wrote new name
    Datafile /oracle/TEST_DB/data/tbs_02.f - dbid changed, wrote new name
    Datafile /oracle/TEST_DB/data/tbs_11.f - dbid changed, wrote new name
    Datafile /oracle/TEST_DB/data/tbs_12.f - dbid changed, wrote new name
    Datafile /oracle/TEST_DB/data/temp1.f - dbid changed, wrote new name
    Control File /oracle/TEST_DB/data/cf1.f - dbid changed, wrote new name
    Control File /oracle/TEST_DB/data/cf2.f - dbid changed, wrote new name
    Instance shut down

Database name changed to TEST_DB.
Modify parameter file and generate a new password file before restarting.
Database ID for database TEST_DB changed to 1250654267.
All previous backups and archived redo logs for this database are unusable.
```

```
Database has been shutdown, open database with RESETLOGS option.
Successfully changed database name and ID.
DBNEWID - Completed successfully.
```

If validation is not successful, then DBNEWID terminates and leaves the target database intact, as shown in the following sample output. You can open the database, fix the error, and then either resume the DBNEWID operation or continue using the database without changing its DBID.

```
.
.
.
     Connected to database PROD (DBID=86997811)

     Connected to server version 10.1.0

     Control Files in database:
         /oracle/TEST_DB/data/cf1.f
         /oracle/TEST_DB/data/cf2.f

     The following datafiles are offline clean:
         /oracle/TEST_DB/data/tbs_61.f (23)
         /oracle/TEST_DB/data/tbs_62.f (24)
         /oracle/TEST_DB/data/temp3.f (3)
     These files must be writable by this utility.

     The following datafiles are read-only:
         /oracle/TEST_DB/data/tbs_51.f (15)
         /oracle/TEST_DB/data/tbs_52.f (16)
         /oracle/TEST_DB/data/tbs_53.f (22)
     These files must be writable by this utility.

     The following datafiles are offline immediate:
         /oracle/TEST_DB/data/tbs_71.f (25)
         /oracle/TEST_DB/data/tbs_72.f (26)

NID-00122: Database should have no offline immediate datafiles

Change of database name failed during validation - database is intact.
DBNEWID - Completed with validation errors.
```

4. Mount the database. For example:

```
STARTUP MOUNT
```

5. 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 ID

To change the database ID without changing the database name, follow the steps in Changing the DBID and Database Name on page 17-3, but in Step 3 do not specify the optional database name (DBNAME). The following is an example of the type of output that is generated when only the database ID is changed.

```
.
.
.
     Connected to database PROD (DBID=86997811)

     Connected to server version 10.1.0

     Control Files in database:
         /oracle/TEST_DB/data/cf1.f
         /oracle/TEST_DB/data/cf2.f

     The following datafiles are offline clean:
         /oracle/TEST_DB/data/tbs_61.f (23)
         /oracle/TEST_DB/data/tbs_62.f (24)
         /oracle/TEST_DB/data/temp3.f (3)
     These files must be writable by this utility.

     The following datafiles are read-only:
         /oracle/TEST_DB/data/tbs_51.f (15)
         /oracle/TEST_DB/data/tbs_52.f (16)
         /oracle/TEST_DB/data/tbs_53.f (22)
     These files must be writable by this utility.

     Changing database ID from 86997811 to 4004383693
         Control File /oracle/TEST_DB/data/cf1.f - modified
         Control File /oracle/TEST_DB/data/cf2.f - modified
         Datafile /oracle/TEST_DB/data/tbs_01.f - dbid changed
         Datafile /oracle/TEST_DB/data/tbs_ax1.f - dbid changed
         Datafile /oracle/TEST_DB/data/tbs_02.f - dbid changed
         Datafile /oracle/TEST_DB/data/tbs_11.f - dbid changed
```

```
    Datafile /oracle/TEST_DB/data/tbs_12.f - dbid changed
    Datafile /oracle/TEST_DB/data/temp1.f - dbid changed
    Control File /oracle/TEST_DB/data/cf1.f - dbid changed
    Control File /oracle/TEST_DB/data/cf2.f - dbid changed
    Instance shut down

Database ID for database TEST_DB changed to 4004383693.
All previous backups and archived redo logs for this database are unusable.
Database has been shutdown, open database with RESETLOGS option.
Succesfully changed database ID.
DBNEWID - Completed succesfully.
```

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_db`:

   ```
   % nid TARGET=SYS/oracle@test_db DBNAME=test_db 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, shuts down the database and exits. The following is an example of what the output for this would look like:

   ```
   .
   .
   .
   Connected to server version 10.1.0

   Control Files in database:
       /oracle/TEST_DB/data/cf1.f
       /oracle/TEST_DB/data/cf2.f
   ```

```
The following datafiles are offline clean:
    /oracle/TEST_DB/data/tbs_61.f (23)
    /oracle/TEST_DB/data/tbs_62.f (24)
    /oracle/TEST_DB/data/temp3.f (3)
These files must be writable by this utility.

The following datafiles are read-only:
    /oracle/TEST_DB/data/tbs_51.f (15)
    /oracle/TEST_DB/data/tbs_52.f (16)
    /oracle/TEST_DB/data/tbs_53.f (22)
These files must be writable by this utility.

Changing database name from PROD to TEST_DB
    Control File /oracle/TEST_DB/data/cf1.f - modified
    Control File /oracle/TEST_DB/data/cf2.f - modified
    Datafile /oracle/TEST_DB/data/tbs_01.f - wrote new name
    Datafile /oracle/TEST_DB/data/tbs_ax1.f - wrote new name
    Datafile /oracle/TEST_DB/data/tbs_02.f - wrote new name
    Datafile /oracle/TEST_DB/data/tbs_11.f - wrote new name
    Datafile /oracle/TEST_DB/data/tbs_12.f - wrote new name
    Datafile /oracle/TEST_DB/data/temp1.f - wrote new name
    Control File /oracle/TEST_DB/data/cf1.f - wrote new name
    Control File /oracle/TEST_DB/data/cf2.f - wrote new name
    Instance shut down

Database name changed to TEST_DB.
Modify parameter file and generate a new password file before restarting.
Successfully changed database name.
DBNEWID - Completed successfully.
```

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. (For an example of what the output looks like for an unsuccessful validation, see Step 3 in Changing the DBID and Database Name on page 17-3.)

4. Set the DB_NAME initialization parameter in the initialization parameter file (PFILE) to the new database name.

Note: The DBNEWID utility does not change the server parameter file (SPFILE). Therefore, if you use SPFILE to start your Oracle database, you must re-create the initialization parameter file from the server parameter file, remove the server parameter file, change the DB_NAME in the initialization parameter file, and then re-create the server parameter file.

5. Create a new password file.
6. Start up the database and resume normal use. For example:

   ```
   STARTUP
   ```

   Because you have changed only the database name, and not the database ID, it is not necessary to use the RESETLOGS option when you open the database. This means that all previous backups are still usable.

Troubleshooting DBNEWID

If the DBNEWID utility succeeds in its validation stage but detects an error while performing the requested change, 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 with the change, then reexecute your original command. The DBNEWID utility resumes and attempts to continue the change until all datafiles and control files have the new value or values. At this point, the database is shut down. You should mount it 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.

If DBNEWID is run against a release 10.1 or later Oracle database, a summary of the operation is written to the alert file. For example, for a change of database name and database ID, you might see something similar to the following:

```
*** DBNEWID utility started ***
DBID will be changed from 86997811 to new DBID of 1250452230 for
database PROD
DBNAME will be changed from PROD to new DBNAME of TEST_DB
Starting datafile conversion
Setting recovery target incarnation to 1
Datafile conversion complete
Database name changed to TEST_DB.
Modify parameter file and generate a new password file before restarting.
Database ID for database TEST_DB changed to 1250452230.
All previous backups and archived redo logs for this database are unusable.
Database has been shutdown, open with RESETLOGS option.
Successfully changed database name and ID.
*** DBNEWID utility finished successfully ***
```

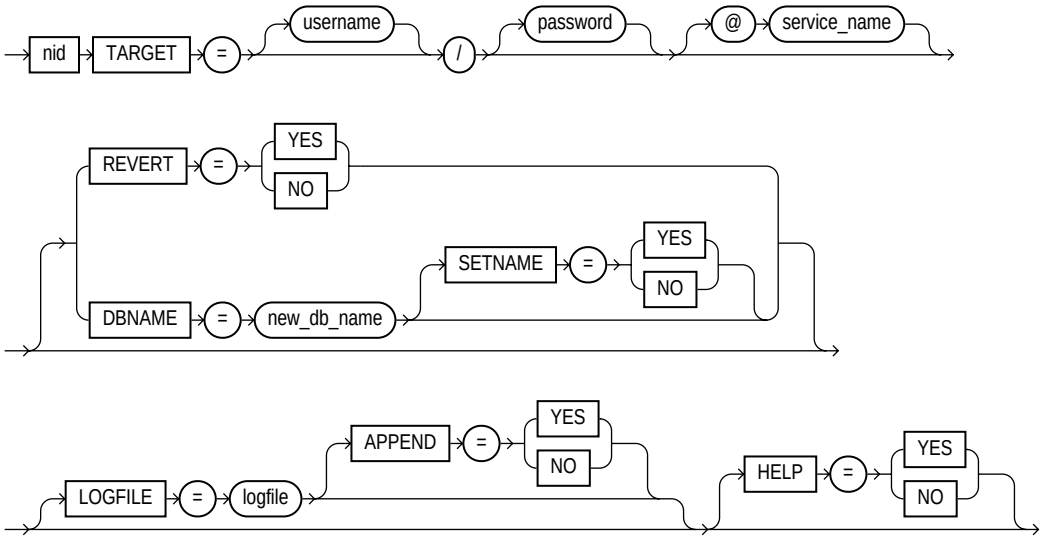
For a change of just the database name, the alert file might show something similar to the following:

```
*** DBNEWID utility started ***
DBNAME will be changed from PROD to new DBNAME of TEST_DB
Starting datafile conversion
Datafile conversion complete
Database name changed to TEST_DB.
Modify parameter file and generate a new password file before restarting.
Successfully changed database name.
*** DBNEWID utility finished successfully ***

In case of failure during DBNEWID the alert will also log the failure:
*** DBNEWID utility started ***
DBID will be changed from 86997811 to new DBID of 86966847 for database
AV3
Change of database ID failed.
Must finish change or REVERT changes before attempting any database
operation.
*** DBNEWID utility finished with errors ***
```

DBNEWID Syntax

The following diagrams show the syntax for the DBNEWID utility.



Parameters

Table 17–1 describes the parameters in the DBNEWID syntax.

Table 17–1 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 valid only 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 writes only to the target database control files.

Table 17–1 (Cont.) Parameters for the DBNEWID Utility

Parameter	Description
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:

- 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. However, 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 terminates unsuccessfully.
- 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 *Oracle Database Administrator's Guide*).
- The DBNEWID utility does not change global database names. See Considerations for Global Database Names on page 17-2.

Additional Restrictions for Releases Prior to Oracle Database 10*g*

The following additional restrictions apply if the DBNEWID utility is run against an Oracle Database release prior to 10.1:

- 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.

18

Using the Metadata API

This chapter describes the Metadata application programming interface (API), which provides a means for you to do the following:

- Retrieve an object's metadata as XML
- Transform the XML in a variety of ways, including transforming it into SQL DDL
- Submit the XML to re-create the object extracted by the retrieval

The following topics are discussed in this chapter:

Why Use the Metadata API?

Over time, as you have used the Oracle database, you may have developed your own code for extracting metadata from the dictionary, manipulating the metadata (adding columns, changing column datatypes, and so on) and then converting the metadata to DDL so that you could re-create the object on the same or another

database. Keeping that code updated to support new dictionary features has probably proven to be challenging.

The Metadata API eliminates the need for you to write and maintain your own code for metadata extraction. It provides a centralized facility for the extraction, manipulation, and resubmission of dictionary metadata. And it supports all dictionary objects at their most current level.

Although the Metadata API can dramatically decrease the amount of custom code you are writing and maintaining, it does not involve any changes to your normal database procedures. The Metadata API is installed in the same way as data dictionary views, by running `catproc.sql` to invoke a SQL script at database installation time. Once it is installed, it is available whenever the instance is operational, even in restricted mode.

The Metadata API does not require you to make any source code changes when you change database versions because it is upwardly compatible across different Oracle versions. XML documents retrieved by one version can be processed by the submit interface on the same or later version. For example, XML documents retrieved by an Oracle9*i* database can be submitted to Oracle Database 10*g*.

Overview of the Metadata API

For the purposes of the Metadata API, every entity in the database is modeled as an object that belongs to an object type. For example, the table `scott.emp` is an object and its object type is `TABLE`. When you fetch an object's metadata you must specify the object type.

In order to fetch a particular object or set of objects within an object type, you specify a filter. Different filters are defined for each object type. For example, two of the filters defined for the `TABLE` object type are `SCHEMA` and `NAME`. They allow you to say, for example, that you want the table whose schema is `scott` and whose name is `emp`.

The Metadata API makes use of XML (Extensible Markup Language) and XSLT (Extensible Stylesheet Language Transformation). The Metadata API represents object metadata as XML because it is a universal format that can be easily parsed and transformed. The Metadata API uses XSLT to transform XML documents into either other XML documents or into SQL DDL.

You can use the Metadata API to specify one or more transforms (XSLT scripts) to be applied to the XML when the metadata is fetched (or when it is resubmitted). The API provides some predefined transforms, including one named DDL that transforms the XML document into SQL creation DDL.

You can then specify conditions on the transform by using transform parameters. You can also specify optional parse items to access specific attributes of an object's metadata. For more details about all of these options, as well as examples of their implementation, see the following sections:

- Using the Metadata API to Retrieve an Object's Metadata
- Using the Metadata API to Re-Create a Retrieved Object
- Retrieving Collections of Different Object Types

Using the Metadata API to Retrieve an Object's Metadata

The Metadata API's retrieval interface lets you specify the kind of object to be retrieved. This can be either a particular object type (such as a table, index, or procedure) or a heterogeneous collection of object types that form a logical unit (such as a database export or schema export). By default, metadata that you fetch is returned in an XML document.

Note: To access objects that are not in your own schema you must have the SELECT_CATALOG_ROLE. However, roles are disabled within many PL/SQL objects (stored procedures, functions, definer's rights packages). Therefore, if you are writing a PL/SQL program that will access objects in another schema (or, in general, any objects for which you need SELECT_CATALOG_ROLE), you must put the code in an invoker's rights package.

You can use the retrieval interface for casual browsing, or you can use it to develop applications. You would use the browsing interface if you simply wanted to make ad hoc queries of the system metadata. You would use the programmatic interface when you want to extract dictionary metadata as part of an application. In such cases, the procedures provided by the Metadata API can be used in place of SQL scripts and customized code that you may be currently using to do the same thing.

Typical Steps Used for Basic Metadata Retrieval

When you retrieve metadata, you use the DBMS_METADATA PL/SQL package, which contains procedures for the Metadata API. The following examples illustrate the programmatic and browsing interfaces.

See Also:

- Table 18–1 for descriptions of DBMS_METADATA procedures used in the programmatic interface
- Table 18–2 for descriptions of DBMS_METADATA procedures used in the browsing interface

Example 18–1 provides a basic demonstration how you might use the Metadata API programmatic interface to retrieve metadata for one table. It creates a Metadata API program that creates a function named `get_table_md`. This function returns metadata for one table.

Example 18–1 Using the DBMS_METADATA Programmatic Interface to Retrieve Data

1. Create a Metadata API program that creates a function named `get_table_md`, which will return the metadata for one table. The content of such a program looks as follows. (For this example, we will name the program `metadata_program.sql`.)

```
CREATE OR REPLACE FUNCTION get_table_md RETURN CLOB IS
-- Define local variables.
h NUMBER; --handle returned by OPEN
th NUMBER; -- handle returned by ADD_TRANSFORM
doc CLOB;
BEGIN

-- Specify the object type.
h := DBMS_METADATA.OPEN('TABLE');

-- Use filters to specify the particular object desired.
DBMS_METADATA.SET_FILTER(h,'SCHEMA','HR');
DBMS_METADATA.SET_FILTER(h,'NAME','TIMECARDS');

 -- Request that the metadata be transformed into creation DDL.
th := DBMS_METADATA.ADD_TRANSFORM(h,'DDL');

 -- Fetch the object.
doc := DBMS_METADATA.FETCH_CLOB(h);

 -- Release resources.
DBMS_METADATA.CLOSE(h);
RETURN doc;
END;
/
```

2. Connect as `hr`/`hr`.
3. Run the program to create the `get_table_md` function:

   ```
   SQL> @metadata_program
   ```

4. Use the newly created `get_table_md` function in a select operation. 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 1000000
   SQL> SELECT get_table_md FROM dual;
   ```

5. The output, which shows the metadata for the `timecards` table in the `hr` schema, looks similar to the following:

   ```
   CREATE TABLE "HR"."TIMECARDS"
    (    "EMPLOYEE_ID" NUMBER(6,0),
         "WEEK" NUMBER(2,0),
         "JOB_ID" VARCHAR2(10),
         "HOURS_WORKED" NUMBER(4,2),
          FOREIGN KEY ("EMPLOYEE_ID")
           REFERENCES "HR"."EMPLOYEES" ("EMPLOYEE_ID") ENABLE
    ) PCTFREE 10 PCTUSED 40 INITRANS 1 MAXTRANS 255 NOCOMPRESS LOGGING
   STORAGE(INITIAL 65536 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS 2147483645
   PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1 BUFFER_POOL DEFAULT)
   TABLESPACE "EXAMPLE"
   ```

You can use the browsing interface and get the same results, as shown in Example 18–2.

Example 18–2 Using the DBMS_METADATA Browsing Interface to Retrieve Data

```
SQL> SET PAGESIZE 0
SQL> SET LONG 1000000
SQL> SELECT DBMS_METADATA.GET_DDL('TABLE','TIMECARDS','HR') FROM dual;
```

The results will be the same as shown for Example 18–1.

Retrieving Multiple Objects

In Example 18–1, the `FETCH_CLOB` procedure was called only once, because it was known that there was only one object. However, you can also retrieve multiple

objects, for example, all the tables in schema `scott`. To do this, you need to use the following construct:

```
LOOP
  doc := DBMS_METADATA.FETCH_CLOB(h);
  --
  -- When there are no more objects to be retrieved, FETCH_CLOB returns NULL.
  --
  EXIT WHEN doc IS NULL;
END LOOP;
```

Example 18–3 demonstrates use of this construct and retrieving multiple objects.

Example 18–3 Retrieving Multiple Objects

```
-- Because not all objects can be returned, they are stored in a table and
queried at the end.

CONNECT scott/tiger
DROP TABLE my_metadata;
CREATE TABLE my_metadata (md clob);
CREATE OR REPLACE PROCEDURE get_tables_md IS
-- Define local variables
h       NUMBER;         -- handle returned by 'OPEN'
th      NUMBER;         -- handle returned by 'ADD_TRANSFORM'
doc     CLOB;           -- metadata is returned in a CLOB
BEGIN
 -- Specify the object type.
 h := DBMS_METADATA.OPEN('TABLE');

 -- Use filters to specify the schema.
 DBMS_METADATA.SET_FILTER(h,'SCHEMA','SCOTT');

 -- Request that the metadata be transformed into creation DDL.
 th := DBMS_METADATA.ADD_TRANSFORM(h,'DDL');

 -- Fetch the objects.
 LOOP
   doc := DBMS_METADATA.FETCH_CLOB(h);

  -- When there are no more objects to be retrieved, FETCH_CLOB returns NULL.
   EXIT WHEN doc IS NULL;

   -- Store the metadata in a table.
   INSERT INTO my_metadata(md) VALUES (doc);
```

```
   COMMIT;
 END LOOP;

 -- Release resources.
 DBMS_METADATA.CLOSE(h);
END;
/
-- Execute the procedure.

EXECUTE get_tables_md;

-- See what was retrieved.

SET LONG 9000000
SET PAGES 0
SELECT * FROM my_metadata;
```

Placing Conditions on Transforms

You can use transform parameters to specify conditions on the transforms you add. To do this, you use the SET_TRANSFORM_PARAM procedure. For example, if you have added the DDL transform for a TABLE object, you can specify the SEGMENT_ATTRIBUTES transform parameter to indicate that you do not want segment attributes (physical, storage, logging, and so on) to appear in the DDL.

Example 18–4 shows use of the SET_TRANSFORM_PARAM procedure.

Example 18–4 Placing Conditions on Transforms

```
CREATE OR REPLACE FUNCTION get_table_md RETURN CLOB IS
 -- Define local variables.
 h    NUMBER;   -- handle returned by 'OPEN'
 th   NUMBER;   -- handle returned by 'ADD_TRANSFORM'
 doc  CLOB;
BEGIN
 -- Specify the object type.
 h := DBMS_METADATA.OPEN('TABLE');

 -- Use filters to specify the particular object desired.
 DBMS_METADATA.SET_FILTER(h,'SCHEMA','HR');
 DBMS_METADATA.SET_FILTER(h,'NAME','TIMECARDS');

 -- Request that the metadata be transformed into creation DDL.
 th := dbms_metadata.add_transform(h,'DDL');
```

```
 -- Specify that segment attributes are not to be returned.
DBMS_METADATA.SET_TRANSFORM_PARAM(th,'SEGMENT_ATTRIBUTES',false);

 -- Fetch the object.
 doc := DBMS_METADATA.FETCH_CLOB(h);

 -- Release resources.
 DBMS_METADATA.CLOSE(h);

 RETURN doc;
END;
/
```

When you run the program, the output looks similar to the following:

```
  CREATE TABLE "HR"."TIMECARDS"
   (    "EMPLOYEE_ID" NUMBER(6,0),
        "WEEK" NUMBER(2,0),
        "JOB_ID" VARCHAR2(10),
        "HOURS_WORKED" NUMBER(4,2),
         FOREIGN KEY ("EMPLOYEE_ID")
          REFERENCES "HR"."EMPLOYEES" ("EMPLOYEE_ID") ENABLE
   )
```

The examples shown up to this point have used a single transform, the DDL transform. The Metadata API also enables you to specify multiple transforms, with the output of the first being the input to the next and so on.

Oracle supplies a transform called MODIFY that modifies an XML document. You can do things like change schema names or tablespace names. To do this, you use remap parameters and the SET_REMAP_PARAM procedure.

Example 18–5 shows a sample use of the SET_REMAP_PARAM procedure. It first adds the MODIFY transform and specifies remap parameters to change the schema name from hr to scott. It then adds the DDL transform. The output of the MODIFY transform is an XML document that becomes the input to the DDL transform. The end result is the creation DDL for the timecards table with all instances of schema hr changed to scott.

Example 18–5 Modifying an XML Document

```
CREATE OR REPLACE FUNCTION remap_schema RETURN CLOB IS
-- Define local variables.
h NUMBER; --handle returned by OPEN
th NUMBER; -- handle returned by ADD_TRANSFORM
```

```
doc CLOB;
BEGIN

-- Specify the object type.
h := DBMS_METADATA.OPEN('TABLE');

-- Use filters to specify the particular object desired.
DBMS_METADATA.SET_FILTER(h,'SCHEMA','HR');
DBMS_METADATA.SET_FILTER(h,'NAME','TIMECARDS');

-- Request that the schema name be modified.
th := DBMS_METADATA.ADD_TRANSFORM(h,'MODIFY');
DBMS_METADATA.SET_REMAP_PARAM(th,'REMAP_SCHEMA','HR','SCOTT');

-- Request that the metadata be transformed into creation DDL.
th := DBMS_METADATA.ADD_TRANSFORM(h,'DDL');

-- Specify that segment attributes are not to be returned.

DBMS_METADATA.SET_TRANSFORM_PARAM(th,'SEGMENT_ATTRIBUTES',false);

-- Fetch the object.
doc := DBMS_METADATA.FETCH_CLOB(h);

-- Release resources.
DBMS_METADATA.CLOSE(h);
RETURN doc;
END;
/
```

When you run the program (`SELECT remap_schema from DUAL;`), the output looks similar to the following:

```
  CREATE TABLE "SCOTT"."TIMECARDS"
   (    "EMPLOYEE_ID" NUMBER(6,0),
        "WEEK" NUMBER(2,0),
        "JOB_ID" VARCHAR2(10),
        "HOURS_WORKED" NUMBER(4,2),
         FOREIGN KEY ("EMPLOYEE_ID")
          REFERENCES "SCOTT"."EMPLOYEES" ("EMPLOYEE_ID") ENABLE
   )
```

If you are familiar with XSLT, you can add your own user-written transforms to process the XML.

Accessing Specific Metadata Attributes

It is often desirable to access specific attributes of an object's metadata, for example, its name or schema. You could get this information by parsing the returned metadata, but the Metadata API provides another mechanism; you can specify parse items, specific attributes that will be parsed out of the metadata and returned in a separate data structure. To do this, you use the SET_PARSE_ITEM procedure.

Example 18–6 fetches all tables in a schema. For each table, a parse item is used to get its name. The name is then used to get all indexes on the table. The example illustrates the use of the FETCH_DDL function, which returns metadata in a sys.ku$_ddls object.

This example assumes you are connected to a schema that contains some tables and indexes. It also creates a table named my_metadata.

Example 18–6 Using Parse Items to Access Specific Metadata Attributes

```
DROP TABLE my_metadata;
CREATE TABLE my_metadata (
  object_type   VARCHAR2(30),
  name          VARCHAR2(30),
  md            CLOB);
CREATE OR REPLACE PROCEDURE get_tables_and_indexes IS
-- Define local variables.
h1      NUMBER;         -- handle returned by OPEN for tables
h2      NUMBER;         -- handle returned by OPEN for indexes
th1     NUMBER;         -- handle returned by ADD_TRANSFORM for tables
th2     NUMBER;         -- handle returned by ADD_TRANSFORM for indexes
doc     sys.ku$_ddls;   -- metadata is returned in sys.ku$_ddls,
                        --  a nested table of sys.ku$_ddl objects
ddl     CLOB;           -- creation DDL for an object
pi      sys.ku$_parsed_items;   -- parse items are returned in this object
                                -- which is contained in sys.ku$_ddl
objname VARCHAR2(30);   -- the parsed object name
BEGIN
 -- This procedure has an outer loop that fetches tables,
 -- and an inner loop that fetches indexes.

 -- Specify the object type: TABLE.
 h1 := DBMS_METADATA.OPEN('TABLE');

 -- Request that the table name be returned as a parse item.
 DBMS_METADATA.SET_PARSE_ITEM(h1,'NAME');
```

```
 -- Request that the metadata be transformed into creation DDL.
 th1 := DBMS_METADATA.ADD_TRANSFORM(h1,'DDL');

 -- Specify that segment attributes are not to be returned.
 DBMS_METADATA.SET_TRANSFORM_PARAM(th1,'SEGMENT_ATTRIBUTES',false);

 -- Set up the outer loop: fetch the TABLE objects.
 LOOP
   doc := dbms_metadata.fetch_ddl(h1);

-- When there are no more objects to be retrieved, FETCH_DDL returns NULL.
   EXIT WHEN doc IS NULL;

-- Loop through the rows of the ku$_ddls nested table.
   FOR i IN doc.FIRST..doc.LAST LOOP
     ddl := doc(i).ddlText;
     pi := doc(i).parsedItems;
     -- Loop through the returned parse items.
     IF pi IS NOT NULL AND pi.COUNT > 0 THEN
       FOR j IN pi.FIRST..pi.LAST LOOP
         IF pi(j).item='NAME' THEN
           objname := pi(j).value;
         END IF;
       END LOOP;
     END IF;
     -- Insert information about this object into our table.
     INSERT INTO my_metadata(object_type, name, md)
       VALUES ('TABLE',objname,ddl);
     COMMIT;
   END LOOP;

   -- Now fetch indexes using the parsed table name as
   --  a BASE_OBJECT_NAME filter.

   -- Specify the object type.
   h2 := DBMS_METADATA.OPEN('INDEX');

   -- The base object is the table retrieved in the outer loop.
   DBMS_METADATA.SET_FILTER(h2,'BASE_OBJECT_NAME',objname);

   -- Exclude system-generated indexes.
   DBMS_METADATA.SET_FILTER(h2,'SYSTEM_GENERATED',false);

   -- Request that the metadata be transformed into creation DDL.
   th2 := DBMS_METADATA.ADD_TRANSFORM(h2,'DDL');
```

```
   -- Specify that segment attributes are not to be returned.
   DBMS_METADATA.SET_TRANSFORM_PARAM(th2,'SEGMENT_ATTRIBUTES',false);

   -- Set up the inner loop: fetch the INDEX objects.
   LOOP
     DDL := DBMS_METADATA.FETCH_CLOB(h2);

     -- When there are no more objects to be retrieved, FETCH_CLOB returns NULL.
     EXIT WHEN ddl IS NULL;

     -- Store the metadata in our table.
     INSERT INTO my_metadata(object_type, name, md)
       VALUES ('INDEX',NULL,ddl);
     COMMIT;
   END LOOP;
   DBMS_METADATA.CLOSE(h2);
 END LOOP;
 DBMS_METADATA.CLOSE(h1);
END;
/

-- Execute the procedure.

EXECUTE get_tables_and_indexes;

-- Perform a query to check what was retrieved.

SET LONG 9000000
SET PAGES 0
SELECT * FROM my_metadata;
```

Using the Metadata API to Re-Create a Retrieved Object

When you fetch metadata for an object, you may want to use it to re-create the object in a different database or schema.

You may not be ready to make remapping decisions when you fetch the metadata. You may want to defer these decisions until later. To accomplish this, you fetch the metadata as XML and store it in a file or table. Later you can use the submit interface to re-create the object.

The submit interface is similar in form to the retrieval interface. It has an OPENW procedure in which you specify the object type of the object to be created. You can

specify transforms, transform parameters, and parse items. You can call the CONVERT function to convert the XML to DDL, or you can call the PUT function to both convert XML to DDL and submit the DDL to create the object.

> **See Also:** Table 18–3 on page 18-27 for descriptions of DBMS_METADATA procedures and functions used in the submit interface

Example 18–7 fetches the XML for a table in one schema, and then uses the submit interface to re-create the table in another schema.

Example 18–7 Using the Submit Interface to Re-Create a Retrieved Object

```
-- Connect as a privileged user.

CONNECT system/manager

-- Create an invoker's rights package to hold the procedure
-- because access to objects in another schema depends on the
-- SELECT_CATALOG_ROLE.  In a definer's rights PL/SQL object
-- (such as a procedure or function), roles are disabled.

CREATE OR REPLACE PACKAGE example_pkg AUTHID current_user IS
  PROCEDURE move_table(
        table_name  in VARCHAR2,
        from_schema in VARCHAR2,
        to_schema   in VARCHAR2 );
END example_pkg;
/
CREATE OR REPLACE PACKAGE BODY example_pkg IS
PROCEDURE move_table(
        table_name  in VARCHAR2,
        from_schema in VARCHAR2,
        to_schema   in VARCHAR2 ) IS

-- Define local variables.
h1      NUMBER;         -- handle returned by OPEN
h2      NUMBER;         -- handle returned by OPENW
th1     NUMBER;         -- handle returned by ADD_TRANSFORM for MODIFY
th2     NUMBER;         -- handle returned by ADD_TRANSFORM for DDL
xml     CLOB;           -- XML document
errs    sys.ku$_SubmitResults := sys.ku$_SubmitResults();
err     sys.ku$_SubmitResult;
result  BOOLEAN;
BEGIN
```

```
-- Specify the object type.
h1 := DBMS_METADATA.OPEN('TABLE');

-- Use filters to specify the name and schema of the table.
DBMS_METADATA.SET_FILTER(h1,'NAME',table_name);
DBMS_METADATA.SET_FILTER(h1,'SCHEMA',from_schema);

-- Fetch the XML.
xml := DBMS_METADATA.FETCH_CLOB(h1);
IF xml IS NULL THEN
    DBMS_OUTPUT.PUT_LINE('Table ' || from_schema || '.' || table_name
|| ' not found');
    RETURN;
  END IF;

  -- Release resources.
  DBMS_METADATA.CLOSE(h1);

  -- Use the submit interface to re-create the object in another schema.

  -- Specify the object type using OPENW (instead of OPEN).
  h2 := DBMS_METADATA.OPENW('TABLE');

  -- First, add the MODIFY transform.
  th1 := DBMS_METADATA.ADD_TRANSFORM(h2,'MODIFY');

-- Specify the desired modification: remap the schema name.
DBMS_METADATA.SET_REMAP_PARAM(th1,'REMAP_SCHEMA',from_schema,to_schema);

-- Now add the DDL transform so that the modified XML can be
--  transformed into creation DDL.
th2 := DBMS_METADATA.ADD_TRANSFORM(h2,'DDL');

-- Call PUT to re-create the object.
result := DBMS_METADATA.PUT(h2,xml,0,errs);

DBMS_METADATA.CLOSE(h2);
  IF NOT result THEN
    -- Process the error information.
    FOR i IN errs.FIRST..errs.LAST LOOP
      err := errs(i);
      FOR j IN err.errorLines.FIRST..err.errorLines.LAST LOOP
        dbms_output.put_line(err.errorLines(j).errorText);
      END LOOP;
```

```
    END LOOP;
  END IF;
END;
END example_pkg;
/

-- Now try it: create a table in SCOTT...

CONNECT scott/tiger
DROP TABLE my_example;
CREATE TABLE my_example (a NUMBER, b VARCHAR2(30));

CONNECT system/manager

SET LONG 9000000
SET PAGESIZE 0
SET SERVEROUTPUT ON SIZE 100000

-- ...and copy it to SYSTEM.

DROP TABLE my_example;
EXECUTE example_pkg.move_table('MY_EXAMPLE','SCOTT','SYSTEM');

-- Verify that it worked.

SELECT DBMS_METADATA.GET_DDL('TABLE','MY_EXAMPLE') FROM dual;
```

Retrieving Collections of Different Object Types

There may be times when you need to retrieve collections of objects in which the objects are of different types, but comprise a logical unit. For example, you might need to retrieve all the objects in a database or a schema, or a table and all its dependent indexes, constraints, grants, audits, and so on. To make such a retrieval possible, the Metadata API provides a number of heterogeneous object types. A heterogeneous object type is an ordered set of object types.

Oracle supplies a number of heterogeneous object types:

- TABLE_EXPORT - a table and its dependent objects
- SCHEMA_EXPORT - a schema and its contents
- DATABASE_EXPORT - the objects in the database

These object types were developed for use by the Data Pump Export utility, but you can use them in your own applications.

You can use only the programmatic retrieval interface (OPEN, FETCH, CLOSE) with these types, not the browsing interface or the submit interface.

You can specify filters for heterogeneous object types, just as you do for the homogeneous types. For example, you can specify the SCHEMA and NAME filters for TABLE_EXPORT, or the SCHEMA filter for SCHEMA_EXPORT.

Example 18–8 shows how to retrieve collections of different object types.

Example 18–8 Retrieving Heterogeneous Object Types

```
-- Create a table to store the retrieved objects.
CONNECT scott/tiger
DROP TABLE my_metadata;
CREATE TABLE my_metadata (md CLOB);
CREATE OR REPLACE PROCEDURE get_schema_md IS

-- Define local variables.
h       NUMBER;         -- handle returned by OPEN
th      NUMBER;         -- handle returned by ADD_TRANSFORM
doc     CLOB;           -- metadata is returned in a CLOB
BEGIN

-- Specify the object type.
 h := DBMS_METADATA.OPEN('SCHEMA_EXPORT');

 -- Use filters to specify the schema.
 DBMS_METADATA.SET_FILTER(h,'SCHEMA','SCOTT');

 -- Request that the metadata be transformed into creation DDL.
 th := DBMS_METADATA.ADD_TRANSFORM(h,'DDL');

 -- Fetch the objects.
 LOOP
   doc := DBMS_METADATA.FETCH_CLOB(h);

   -- When there are no more objects to be retrieved, FETCH_CLOB returns NULL.
   EXIT WHEN doc IS NULL;

   -- Store the metadata in the table.
   INSERT INTO my_metadata(md) VALUES (doc);
   COMMIT;
 END LOOP;
```

```
 -- Release resources.
 DBMS_METADATA.CLOSE(h);
END;
/
-- Execute the procedure.

EXECUTE get_schema_md;

-- See what was retrieved.

SET LONG 9000000
SET PAGESIZE 0
SELECT * FROM my_metadata;
```

Note the following about this example:

The objects are returned ordered by object type; for example, all tables are returned, then all grants on tables, then all indexes on tables, and so on. The order is, generally speaking, a valid creation order. Thus, if you take the objects in the order in which they were returned and use the submit interface to re-create them in the same order in another schema or database, there will usually be no errors. (The exceptions usually involve circular references; for example, if package A contains a call to package B, and B contains a call to A, then one of the packages will need to be recompiled.)

Filtering the Return of Heterogeneous Object Types

If you want finer control of the objects returned, you can use the SET_FILTER procedure and specify that the filter apply only to a specific member type. You do this by specifying the path name of the member type as the fourth parameter to SET_FILTER. In addition, you can use the EXCLUDE_PATH_EXPR filter to exclude all objects of an object type. For a list of valid path names, see the TABLE_EXPORT_OBJECTS catalog view.

Example 18–9 shows how you can use SET_FILTER to specify finer control on the objects returned.

Example 18–9 Filtering the Return of Heterogeneous Object Types

```
-- Create a table to store the retrieved objects.
CONNECT scott/tiger
DROP TABLE my_metadata;
CREATE TABLE my_metadata (md CLOB);
```

```
CREATE OR REPLACE PROCEDURE get_schema_md2 IS

-- Define local variables.
h       NUMBER;         -- handle returned by 'OPEN'
th      NUMBER;         -- handle returned by 'ADD_TRANSFORM'
doc     CLOB;           -- metadata is returned in a CLOB
BEGIN

 -- Specify the object type.
 h := DBMS_METADATA.OPEN('SCHEMA_EXPORT');

 -- Use filters to specify the schema.
 DBMS_METADATA.SET_FILTER(h,'SCHEMA','SCOTT');

 -- Use the fourth parameter to SET_FILTER to specify a filter
 -- that applies to a specific member object type.
 DBMS_METADATA.SET_FILTER(h,'NAME_EXPR','!=''MY_METADATA''','TABLE');

 -- Use the EXCLUDE_PATH_EXPR filter to exclude procedures.
 DBMS_METADATA.SET_FILTER(h,'EXCLUDE_PATH_EXPR','=''PROCEDURE''');

 -- Request that the metadata be transformed into creation DDL.
 th := DBMS_METADATA.ADD_TRANSFORM(h,'DDL');

 -- Use the fourth parameter to SET_TRANSFORM_PARAM to specify a parameter
 --  that applies to a specific member object type.
DBMS_METADATA.SET_TRANSFORM_PARAM(th,'SEGMENT_ATTRIBUTES',false,'TABLE');

 -- Fetch the objects.
 LOOP
   doc := dbms_metadata.fetch_clob(h);

   -- When there are no more objects to be retrieved, FETCH_CLOB returns NULL.
   EXIT WHEN doc IS NULL;

   -- Store the metadata in the table.
   INSERT INTO my_metadata(md) VALUES (doc);
   COMMIT;
 END LOOP;

 -- Release resources.
 DBMS_METADATA.CLOSE(h);
END;
/
-- Execute the procedure.
```

```
EXECUTE get_schema_md2;

-- See what was retrieved.

SET LONG 9000000
SET PAGESIZE 0
SELECT * FROM my_metadata;
```

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 tables, then all indexes, then all 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. Example Usage of the Metadata API on page 18-19 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. 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: *Oracle Database Application Developer's Guide - Large Objects*

Example Usage of the Metadata API

This section provides an example of how the Metadata API could be used. A script is provided that automatically runs the demo for you by performing the following actions:

- Establishes a schema (MDDEMO) and some payroll users.

- Creates three payroll-like tables within the schema, as well as any associated indexes, triggers, and grants.
- Creates a package, PAYROLL_DEMO, that uses the Metadata API. The PAYROLL_DEMO package contains a procedure, GET_PAYROLL_TABLES, that retrieves the DDL for the two tables in the MDDEMO schema that start with PAYROLL. For each table, it retrieves the DDL for the table's associated dependent objects; indexes, grants, and triggers. All the DDL is written to a table named MDDEMO.DDL.

To execute the example, do the following:

1. Start SQL*Plus as `system/manager`:

   ```
   sqlplus system/manager
   ```

2. Install the demo, which is located in `rdbms/demo`:

   ```
   SQL> @mddemo
   ```

   For an explanation of what happens during this step, see What Does the Metadata API Example Do? on page 18-21.

3. Connect as user `mddemo/mddemo`:

   ```
   SQL> CONNECT mddemo/mddemo
   ```

4. Set the following parameters so that query output will be complete and readable:

   ```
   SQL> SET PAGESIZE 0
   SQL> SET LONG 1000000
   ```

5. Execute the GET_PAYROLL_TABLES procedure, as follows:

   ```
   SQL> CALL payroll_demo.get_payroll_tables();
   ```

6. Execute the following SQL query:

   ```
   SQL> SELECT ddl FROM DDL ORDER BY SEQNO;
   ```

   The output generated is the result of the execution of the GET_PAYROLL_TABLES procedure. It shows all the DDL that was performed in Step 2 when the demo was installed. See Output Generated from the GET_PAYROLL_TABLES Procedure on page 18-23 for a listing of the actual output.

What Does the Metadata API Example Do?

When the mddemo script is run, the following steps take place. You can adapt these steps to your own situation.

1. Drops users as follows, if they exist. This will ensure that you are starting out with fresh data. If the users do not exist, a message to that effect is displayed, no harm is done, and the demo continues to execute.

```
CONNECT system/manager
SQL> DROP USER mddemo CASCADE;
SQL> DROP USER mddemo_clerk CASCADE;
SQL> DROP USER mddemo_mgr CASCADE;
```

2. Creates user mddemo, identified by mddemo:

```
SQL> CREATE USER mddemo IDENTIFIED BY mddemo;
SQL> GRANT resource, connect, create session,
  1     create table,
  2     create procedure,
  3     create sequence,
  4     create trigger,
  5     create view,
  6     create synonym,
  7     alter session,
  8  TO mddemo;
```

3. Creates user mddemo_clerk, identified by clerk:

```
CREATE USER mddemo_clerk IDENTIFIED BY clerk;
```

4. Creates user mddemo_mgr, identified by mgr:

```
CREATE USER mddemo_mgr IDENTIFIED BY mgr;
```

5. Connects to SQL*Plus as mddemo:

```
CONNECT mddemo/mddemo
```

6. Creates some payroll-type tables:

```
SQL> CREATE TABLE payroll_emps
  2  ( lastname VARCHAR2(60) NOT NULL,
  3  firstname VARCHAR2(20) NOT NULL,
  4  mi VARCHAR2(2),
  5  suffix VARCHAR2(10),
  6  dob DATE NOT NULL,
```

```
  7  badge_no NUMBER(6) PRIMARY KEY,
  8  exempt VARCHAR(1) NOT NULL,
  9  salary NUMBER (9,2),
  10 hourly_rate NUMBER (7,2) )
  11 /

SQL> CREATE TABLE payroll_timecards
  2  (badge_no NUMBER(6) REFERENCES payroll_emps (badge_no),
  3  week NUMBER(2),
  4  job_id NUMBER(5),
  5  hours_worked NUMBER(4,2) )
  6 /
```

7. Creates a dummy table, `audit_trail`. This table is used to show that tables that do not start with "PAYROLL" are not retrieved by the GET_PAYROLL_TABLES procedure.

```
SQL> CREATE TABLE audit_trail
  2  (action_time DATE,
  3  lastname VARCHAR2(60),
  4  action LONG )
  5  /
```

8. Creates some grants on the tables just created:

```
SQL> GRANT UPDATE (salary,hourly_rate) ON payroll_emps TO mddemo_clerk;
SQL> GRANT ALL ON payroll_emps TO mddemo_mgr WITH GRANT OPTION;

SQL> GRANT INSERT,UPDATE ON payroll_timecards TO mddemo_clerk;
SQL> GRANT ALL ON payroll_timecards TO mddemo_mgr WITH GRANT OPTION;
```

9. Creates some indexes on the tables just created:

```
SQL> CREATE INDEX i_payroll_emps_name ON payroll_emps(lastname);
SQL> CREATE INDEX i_payroll_emps_dob ON payroll_emps(dob);
SQL> CREATE INDEX i_payroll_timecards_badge ON payroll_timecards(badge_no);
```

10. Creates some triggers on the tables just created:

```
SQL> CREATE OR REPLACE PROCEDURE check_sal( salary in number) AS BEGIN
  2  RETURN;
  3  END;
  4  /
```

Note that the security is kept fairly loose to keep the example simple.

```
SQL> 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)
/

SQL> 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;
/
```

11. Sets up a table to hold the generated DDL:

```
CREATE TABLE ddl (ddl CLOB, seqno NUMBER);
```

12. Creates the PAYROLL_DEMO package, which provides examples of how DBMS_METADATA procedures can be used.

```
SQL> CREATE OR REPLACE PACKAGE payroll_demo AS PROCEDURE get_payroll_tables;
END;
/
```

> **Note:** To see the entire script for this example, including the contents of the PAYROLL_DEMO package, see the file `rdbms/demo/mddemo.sql` located in your $ORACLE_HOME directory.

Output Generated from the GET_PAYROLL_TABLES Procedure

After you execute the `mddemo.payroll_demo.get_payroll_tables` procedure, you can execute the following query:

```
SQL> SELECT ddl FROM ddl ORDER BY seqno;
```

The results are as follows, which reflect all the DDL executed by the script as described in the previous section.

```
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" ;
```

Summary of DBMS_METADATA Procedures

This section provides brief descriptions of the procedures provided by the Metadata API. For detailed descriptions of these procedures, see *PL/SQL Packages and Types Reference*.

Table 18–1 provides a brief description of the procedures provided by the DBMS_METADATA programmatic interface for retrieving multiple objects.

Table 18–1 DBMS_METADATA Procedures Used for Retrieving Multiple Objects

PL/SQL Procedure Name	Description
DBMS_METADATA.OPEN()	Specifies the type of object to be retrieved, the version of its metadata, and the object model.
DBMS_METADATA.SET_FILTER()	Specifies restrictions on the objects to be retrieved, for example, the object name or schema.
DBMS_METADATA.SET_COUNT()	Specifies the maximum number of objects to be retrieved in a single FETCH_XXX call.
DBMS_METADATA.GET_QUERY()	Returns the text of the queries that are used by FETCH_XXX. You can use this as a debugging aid.
DBMS_METADATA.SET_PARSE_ITEM()	Enables output parsing by specifying an object attribute to be parsed and returned.
DBMS_METADATA.ADD_TRANSFORM()	Specifies a transform that FETCH_XXX applies to the XML representation of the retrieved objects.
DBMS_METADATA.SET_TRANSFORM_PARAM()	Specifies parameters to the XSLT stylesheet identified by transform_handle.
DBMS_METADATA.SET_REMAP_PARAM()	Specifies parameters to the XSLT stylesheet identified by transform_handle.
DBMS_METADATA.FETCH_xxx()	Returns metadata for objects meeting the criteria established by OPEN, SET_FILTER, SET_COUNT, ADD_TRANSFORM, and so on.
DBMS_METADATA.CLOSE()	Invalidates the handle returned by OPEN and cleans up the associated state.

Table 18–2 lists the procedures provided by the DBMS_METADATA browsing interface and provides a brief description of each one. These functions return metadata for one or more dependent or granted objects. These procedures do not support heterogeneous object types.

Table 18–2 DBMS_METADATA Procedures Used for the 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.

Table 18–3 provides a brief description of the DBMS_METADATA procedures and functions used for XML submission.

Table 18–3 DBMS_METADATA Procedures and Functions for Submitting XML Data

PL/SQL Name	Description
DBMS_METADATA.OPENW()	Opens a write context.
DBMS_METADATA.ADD_TRANSFORM()	Specifies a transform for the XML documents
DBMS_METADATA.SET_TRANSFORM_PARAM() and DBMS_METADATA.SET_REMAP_PARAM()	SET_TRANSFORM_PARAM specifies a parameter to a transform. SET_REMAP_PARAM specifies a remapping for a transform.
DBMS_METADATA.SET_PARSE_ITEM()	Specifies an object attribute to be parsed.
DBMS_METADATA.CONVERT()	Converts an XML document to DDL.
DBMS_METADATA.PUT()	Submits an XML document to the database.
DBMS_METADATA.CLOSE()	Closes the context opened with OPENW.

19

Using LogMiner to Analyze Redo Log Files

Oracle LogMiner, which is part of Oracle Database, enables you to query online and archived redo log files through a SQL interface. Redo log files contain information about the history of activity on a database.

This chapter contains the following sections:

This chapter describes LogMiner as it is used from the command line. You can also access LogMiner through the Oracle LogMiner Viewer graphical user interface. Oracle LogMiner Viewer is a part of Oracle Enterprise Manager. See the Oracle

Enterprise Manager online Help for more information about Oracle LogMiner Viewer.

LogMiner Benefits

All changes made to user data or to the database dictionary are recorded in the Oracle redo log files so that database recovery operations can be performed.

Because LogMiner provides a well-defined, easy-to-use, and comprehensive relational interface to redo log files, it can be used as a powerful data audit tool, as well as a tool for sophisticated data analysis. The following list describes some key capabilities of LogMiner:

- Pinpointing when a logical corruption to a database, such as errors made at the application level, may have begun. These might include errors such as those where the wrong rows were deleted because of incorrect values in a `WHERE` clause, rows were updated with incorrect values, the wrong index was dropped, and so forth. For example, a user application could mistakenly update a database to give all employees 100 percent salary increases rather than 10 percent increases, or a database administrator (DBA) could accidently delete a critical system table. It is important to know exactly when an error was made so that you know when to initiate time-based or change-based recovery. This enables you to restore the database to the state it was in just before corruption. See Querying V$LOGMNR_CONTENTS Based on Column Values on page 19-17 for details about how you can use LogMiner to accomplish this.
- Determining what actions you would have to take to perform fine-grained recovery at the transaction level. If you fully understand and take into account existing dependencies, it may be possible to perform a table-specific undo operation to return the table to its original state. This is achieved by applying table-specific reconstructed SQL statements that LogMiner provides in the reverse order from which they were originally issued. See Scenario 1: Using LogMiner to Track Changes Made by a Specific User on page 19-82 for an example.

  Normally you would have to restore the table to its previous state, and then apply an archived redo log file to roll it forward.
- Performance tuning and capacity planning through trend analysis. You can determine which tables get the most updates and inserts. That information provides a historical perspective on disk access statistics, which can be used for tuning purposes. See Scenario 2: Using LogMiner to Calculate Table Access Statistics on page 19-84 for an example.

- Performing postauditing. LogMiner can be used to track any data manipulation language (DML) and data definition language (DDL) statements executed on the database, the order in which they were executed, and who executed them. (However, to use LogMiner for such a purpose, you need to have an idea when the event occurred so that you can specify the appropriate logs for analysis; otherwise you might have to mine a large number of redo log files, which can take a long time. Consider using LogMiner as a complementary activity to auditing database use. See the *Oracle Database Administrator's Guide* for information about database auditing.)

Introduction to LogMiner

The following sections provide a brief introduction to LogMiner, including the following topics:

- LogMiner Configuration
- Directing LogMiner Operations and Retrieving Data of Interest

The remaining sections in this chapter describe these concepts and related topics in more detail.

LogMiner Configuration

There are four basic objects in a LogMiner configuration that you should be familiar with: the source database, the mining database, the LogMiner dictionary, and the redo log files containing the data of interest:

- The **source database** is the database that produces all the redo log files that you want LogMiner to analyze.
- The **mining database** is the database that LogMiner uses when it performs the analysis.
- The **LogMiner dictionary** allows LogMiner to provide table and column names, instead of internal object IDs, when it presents the redo log data that you request.

  LogMiner uses the dictionary to translate internal object identifiers and datatypes to object names and external data formats. Without a dictionary, LogMiner returns internal object IDs and presents data as binary data.

  For example, consider the following the SQL statement:

```
INSERT INTO HR.JOBS(JOB_ID, JOB_TITLE, MIN_SALARY, MAX_SALARY)
```

```
  VALUES('IT_WT','Technical Writer', 4000, 11000);
```

Without the dictionary, LogMiner will display:

```
insert into "UNKNOWN"."OBJ# 45522"("COL 1","COL 2","COL 3","COL 4") values
(HEXTORAW('45465f4748'),HEXTORAW('546563686e6963616c20577269746572'),
HEXTORAW('c229'),HEXTORAW('c3020b'));
```

- The **redo log files** contain the changes made to the database or database dictionary.

Sample Configuration

Figure 19–1 shows a sample LogMiner configuration. In this figure, the source database in Boston generates redo log files that are archived and shipped to a database in San Francisco. A LogMiner dictionary has been extracted to these redo log files. The mining database, where LogMiner will actually analyze the redo log files, is in San Francisco. The Boston database is running Oracle9*i*, and the San Francisco database is running Oracle Database 10*g*.

Figure 19–1 Sample LogMiner Database Configuration

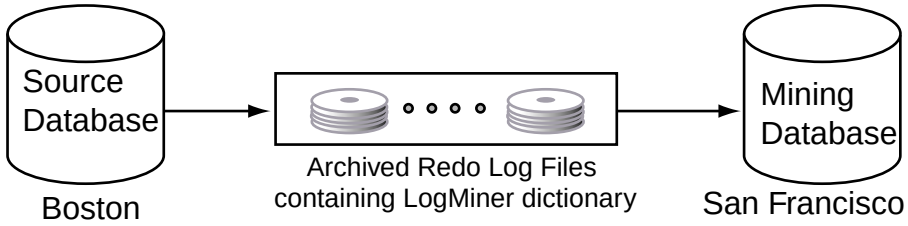


Figure 19–1 shows just one valid LogMiner configuration. Other valid configurations are those that use the same database for both the source and mining database, or use another method for providing the data dictionary. These other data dictionary options are described in the section about LogMiner Dictionary Options on page 19-7.

Requirements

The following are requirements for the source and mining database, the data dictionary, and the redo log files that LogMiner will mine:

- Source and mining database

  - Both the source database and the mining database must be running on the same hardware platform.
  - The mining database can be the same as, or completely separate from, the source database.
  - The mining database must run the same version or a later version of the Oracle Database software as the source database.
  - The mining database must use the same character set (or a superset of the character set) used by the source database.
- LogMiner dictionary
  - The dictionary must be produced by the same source database that generates the redo log files that LogMiner will analyze.
- All redo log files:
  - Must be produced by the same source database.
  - Must be associated with the same database `RESETLOGS SCN`.
  - Must be from a release 8.0 or later Oracle Database. However, several of the LogMiner features introduced as of release 9.0.1 work only with redo log files produced on an Oracle9*i* or later database. See Supported Databases and Redo Log File Versions on page 19-86.

LogMiner does not allow you to mix redo log files from different databases or to use a dictionary from a different database than the one that generated the redo log files to be analyzed.

Note: Oracle strongly recommends that you enable supplemental logging prior to using LogMiner; ideally all the log files that LogMiner will process contain supplemental log data.

When you enable supplemental logging, additional information is recorded in the redo stream that is needed to make the information in the redo log files useful to you. At least, you should enable minimal supplemental logging, as follows:

```
SQL> ALTER DATABASE ADD SUPPLEMENTAL LOG DATA;
```

To determine whether supplemental logging is enabled, query the V$DATABASE view, as follows:

```
SQL> SELECT SUPPLEMENTAL_LOG_DATA_MIN FROM V$DATABASE;
```

If the query returns a value of YES or IMPLICIT, minimal supplemental logging is enabled. See Supplemental Logging on page 19-28 for complete information about supplemental logging.

Directing LogMiner Operations and Retrieving Data of Interest

You direct LogMiner operations using the DBMS_LOGMNR and DBMS_LOGMNR_D PL/SQL packages, and retrieve data of interest using the V$LOGMNR_CONTENTS view, as follows:

1. Specify a LogMiner dictionary.

   Use the DBMS_LOGMNR_D.BUILD procedure or specify the dictionary when you start LogMiner (in Step 3), or both, depending on the type of dictionary you plan to use.

2. Specify a list of redo log files for analysis.

   Use the DBMS_LOGMNR.ADD_LOGFILE procedure, or direct LogMiner to create a list of log files for analysis automatically when you start LogMiner (in Step 3).

3. Start LogMiner.

   Use the DBMS_LOGMNR.START_LOGMNR procedure.

4. Request the redo data of interest.

   Query the V$LOGMNR_CONTENTS view.

5. End the LogMiner session.

Use the DBMS_LOGMNR.END_LOGMNR procedure.

You must have been granted the EXECUTE_CATALOG_ROLE role to use the LogMiner PL/SQL packages and to query the V$LOGMNR_CONTENTS view.

> **See Also:** Steps in a Typical LogMiner Session on page 19-43 for an example of using LogMiner

LogMiner Dictionary Files and Redo Log Files

Before you begin using LogMiner, it is important to understand how LogMiner works with the LogMiner dictionary file (or files) and redo log files. This will help you to get accurate results and to plan the use of your system resources.

The following concepts are discussed in this section:

- LogMiner Dictionary Options
- Redo Log File Options

LogMiner Dictionary Options

LogMiner requires a dictionary to translate object IDs into object names when it returns redo data to you. LogMiner gives you three options for supplying the dictionary:

- Using the Online Catalog

  Oracle recommends that you use this option when you will have access to the source database from which the redo log files were created and when no changes to the column definitions in the tables of interest are anticipated. This is the most efficient and easy-to-use option.

- Extracting a LogMiner Dictionary to the Redo Log Files

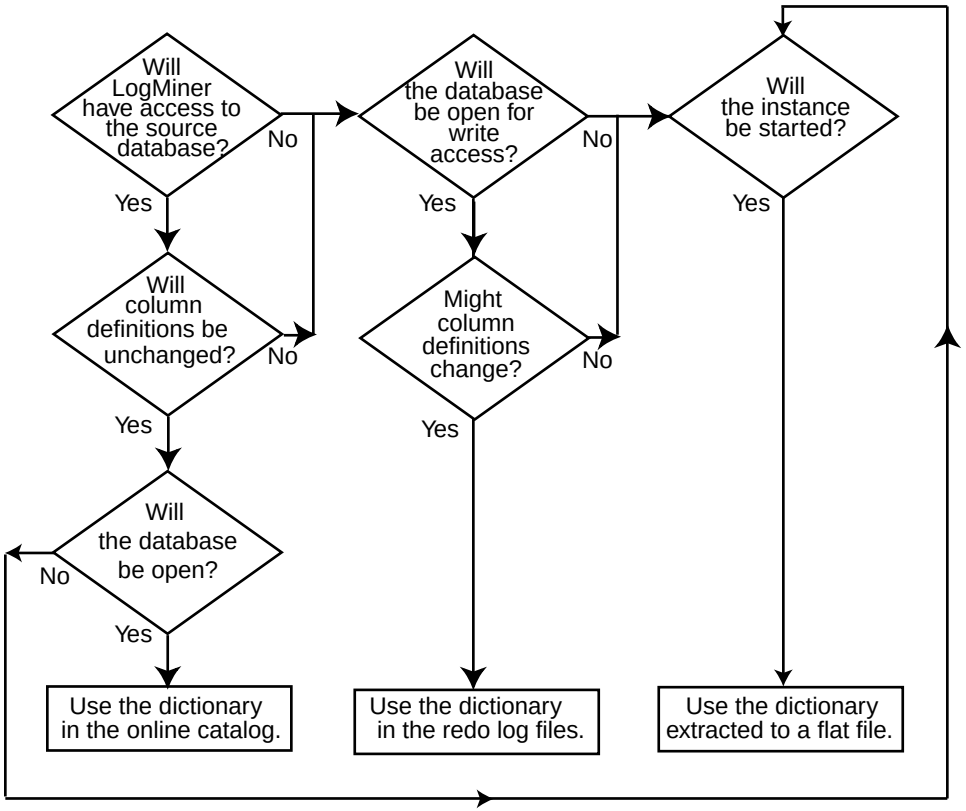
  Oracle recommends that you use this option when you do not expect to have access to the source database from which the redo log files were created, or if you anticipate that changes will be made to the column definitions in the tables of interest.

- Extracting the LogMiner Dictionary to a Flat File

  This option is maintained for backward compatibility with previous releases. This option does not guarantee transactional consistency. Oracle recommends that you use either the online catalog or extract the dictionary from redo log files instead.

Figure 19–2 shows a decision tree to help you select a LogMiner dictionary, depending on your situation.

Figure 19–2 Decision Tree for Choosing a LogMiner Dictionary



The following sections provide instructions on how to specify each of the available dictionary options.

Using the Online Catalog

To direct LogMiner to use the dictionary currently in use for the database, specify the online catalog as your dictionary source when you start LogMiner, as follows:

```
SQL> EXECUTE DBMS_LOGMNR.START_LOGMNR(-
       OPTIONS => DBMS_LOGMNR.DICT_FROM_ONLINE_CATALOG);
```

In addition to using the online catalog to analyze online redo log files, you can use it to analyze archived redo log files, if you are on the same system that generated the archived redo log files.

The online catalog contains the latest information about the database and may be the fastest way to start your analysis. Because DDL operations that change important tables are somewhat rare, the online catalog generally contains the information you need for your analysis.

Remember, however, that the online catalog can only reconstruct SQL statements that are executed on the latest version of a table. As soon as a table is altered, the online catalog no longer reflects the previous version of the table. This means that LogMiner will not be able to reconstruct any SQL statements that were executed on the previous version of the table. Instead, LogMiner generates nonexecutable SQL (including hexadecimal-to-raw formatting of binary values) in the SQL_REDO column of the V$LOGMNR_CONTENTS view similar to the following example:

```
insert into HR.EMPLOYEES(col#1, col#2) values (hextoraw('4a6f686e20446f65'),
hextoraw('c306'));"
```

The online catalog option requires that the database be open.

The online catalog option is not valid with the DDL_DICT_TRACKING option of DBMS_LOGMNR.START_LOGMNR.

Extracting a LogMiner Dictionary to the Redo Log Files

To extract a LogMiner dictionary to the redo log files, the database must be open and in ARCHIVELOG mode and archiving must be enabled. While the dictionary is being extracted to the redo log stream, no DDL statements can be executed. Therefore, the dictionary extracted to the redo log files is guaranteed to be consistent (whereas the dictionary extracted to a flat file is not).

To extract dictionary information to the redo log files, use the DBMS_LOGMNR_D.BUILD procedure with the STORE_IN_REDO_LOGS option. Do not specify a filename or location.

```
SQL> EXECUTE DBMS_LOGMNR_D.BUILD( -
       OPTIONS=> DBMS_LOGMNR_D.STORE_IN_REDO_LOGS);
```

> **See Also:** *Oracle Database Backup and Recovery Basics* for more information about ARCHIVELOG mode and the *PL/SQL Packages and Types Reference* for a complete description of the DBMS_LOGMNR_D.BUILD procedure

The process of extracting the dictionary to the redo log files does consume database resources, but if you limit the extraction to off-peak hours, this should not be a problem, and it is faster than extracting to a flat file. Depending on the size of the dictionary, it may be contained in multiple redo log files. If the relevant redo log files have been archived, you can find out which redo log files contain the start and end of an extracted dictionary. To do so, query the V$ARCHIVED_LOG view, as follows:

```
SQL> SELECT NAME FROM V$ARCHIVED_LOG WHERE DICTIONARY_BEGIN='YES';
SQL> SELECT NAME FROM V$ARCHIVED_LOG WHERE DICTIONARY_END='YES';
```

Specify the names of the start and end redo log files, and possibly other logs in between them, with the ADD_LOGFILE procedure when you are preparing to begin a LogMiner session.

Oracle recommends that you periodically back up the redo log files so that the information is saved and available at a later date. Ideally, this will not involve any extra steps because if your database is being properly managed, there should already be a process in place for backing up and restoring archived redo log files. Again, because of the time required, it is good practice to do this during off-peak hours.

Extracting the LogMiner Dictionary to a Flat File

When the LogMiner dictionary is in a flat file, fewer system resources are used than when it is contained in the redo log files. Oracle recommends that you regularly back up the dictionary extract to ensure correct analysis of older redo log files.

To extract database dictionary information to a flat file, use the DBMS_LOGMNR_D.BUILD procedure with the STORE_IN_FLAT_FILE option.

Be sure that no DDL operations occur while the dictionary is being built.

The following steps describe how to extract a dictionary to a flat file. Steps 1 and 2 are preparation steps. You only need to do them once, and then you can extract a dictionary to a flat file as many times as you wish.

1. The DBMS_LOGMNR_D.BUILD procedure requires access to a directory where it can place the dictionary file. Because PL/SQL procedures do not normally access user directories, you must specify a directory for use by the DBMS_LOGMNR_D.BUILD procedure or the procedure will fail. To specify a directory, set the initialization parameter, UTL_FILE_DIR, in the initialization parameter file.

> **See Also:** *Oracle Database Reference* for more information about the initialization parameter file (`init.ora`) and the *PL/SQL Packages and Types Reference* for a complete description of the `DBMS_LOGMNR_D.BUILD` procedure

For example, to set `UTL_FILE_DIR` to use `/oracle/database` as the directory where the dictionary file is placed, enter the following in the initialization parameter file:

```
UTL_FILE_DIR = /oracle/database
```

Remember that for the changes to the initialization parameter file to take effect, you must stop and restart the database.

2. If the database is closed, use SQL*Plus to mount and then open the database whose redo log files you want to analyze. For example, entering the `STARTUP` command mounts and opens the database:

```
SQL> STARTUP
```

3. Execute the PL/SQL procedure `DBMS_LOGMNR_D.BUILD`. Specify a filename for the dictionary and a directory path name for the file. This procedure creates the dictionary file. For example, enter the following to create the file `dictionary.ora` in `/oracle/database`:

```
SQL> EXECUTE DBMS_LOGMNR_D.BUILD('dictionary.ora', -
       '/oracle/database/', -
       DBMS_LOGMNR_D.STORE_IN_FLAT_FILE);
```

You could also specify a filename and location without specifying the `STORE_IN_FLAT_FILE` option. The result would be the same.

Redo Log File Options

To mine data in the redo log files, LogMiner needs information about which redo log files to mine. Changes made to the database that are found in these redo log files are delivered to you through the `V$LOGMNR_CONTENTS` view.

You can direct LogMiner to automatically and dynamically create a list of redo log files to analyze, or you can explicitly specify a list of redo log files for LogMiner to analyze, as follows:

- Automatically

If LogMiner is being used on the source database, then you can direct LogMiner to find and create a list of redo log files for analysis automatically. Use the CONTINUOUS_MINE option when you start LogMiner with the DBMS_LOGMNR.START_LOGMNR procedure, and specify a time or SCN range. Although this example specifies the dictionary from the online catalog, any LogMiner dictionary can be used.

LogMiner will use the database control file to find and add redo log files that satisfy your specified time or SCN range to the LogMiner redo log file list. For example:

```
SQL> ALTER SESSION SET NLS_DATE_FORMAT = 'DD-MON-YYYY HH24:MI:SS';
SQL> EXECUTE DBMS_LOGMNR.START_LOGMNR( -
   STARTTIME => '01-Jan-2003 08:30:00', -
   ENDTIME => '01-Jan-2003 08:45:00', -
   OPTIONS => DBMS_LOGMNR.DICT_FROM_ONLINE_CATALOG + -
   DBMS_LOGMNR.CONTINUOUS_MINE);
```

(To avoid the need to specify the date format in the call to the DBMS_LOGMNR.START_LOGMNR procedure, this example uses the ALTER SESSION SET NLS_DATE_FORMAT statement first.)

You can also direct LogMiner to automatically build a list of redo log files to analyze by specifying just one redo log file using DBMS_LOGMNR.ADD_LOGFILE, and then specifying the CONTINUOUS_MINE option when you start LogMiner. The previously described method is more typical, however.

- Manually

  Use the DBMS_LOGMNR.ADD_LOGFILE procedure to manually create a list of redo log files before you start LogMiner. After the first redo log file has been added to the list, each subsequently added redo log file must be from the same database and associated with the same database RESETLOGS SCN. When using this method, LogMiner need not be connected to the source database.

  For example, to start a new list of redo log files, specify the NEW option of the DBMS_LOGMNR.ADD_LOGFILE procedure to signal that this is the beginning of a new list. For example, enter the following to specify /oracle/logs/log1.f:

```
SQL> EXECUTE DBMS_LOGMNR.ADD_LOGFILE( -
       LOGFILENAME => '/oracle/logs/log1.f', -
       OPTIONS => DBMS_LOGMNR.NEW);
```

If desired, add more redo log files by specifying the ADDFILE option of the DBMS_LOGMNR.ADD_LOGFILE procedure. For example, enter the following to add /oracle/logs/log2.f:

```
SQL> EXECUTE DBMS_LOGMNR.ADD_LOGFILE( -
       LOGFILENAME => '/oracle/logs/log2.f', -
       OPTIONS => DBMS_LOGMNR.ADDFILE);
```

To determine which redo log files are being analyzed in the current LogMiner session, you can query the V$LOGMNR_LOGS view, which contains one row for each redo log file.

Starting LogMiner

You call the DBMS_LOGMNR.START_LOGMNR procedure to start LogMiner. Because the options available with the DBMS_LOGMNR.START_LOGMNR procedure allow you to control output to the V$LOGMNR_CONTENTS view, you must call DBMS_LOGMNR.START_LOGMNR before querying the V$LOGMNR_CONTENTS view.

When you start LogMiner, you can:

- Specify how LogMiner should filter data it returns (for example, by starting and ending time or SCN value)
- Specify options for formatting the data returned by LogMiner
- Specify the LogMiner dictionary to use

The following list is a summary of LogMiner settings that you can specify with the OPTIONS parameter to DBMS_LOGMNR.START_LOGMNR and where to find more information about them.

- DICT_FROM_ONLINE_CATALOG — See Using the Online Catalog on page 19-8
- DICT_FROM_REDO_LOGS — See Start LogMiner on page 19-46
- CONTINUOUS_MINE — See Redo Log File Options on page 19-11
- COMMITTED_DATA_ONLY — See Showing Only Committed Transactions on page 19-19
- SKIP_CORRUPTION — See Skipping Redo Corruptions on page 19-22
- NO_SQL_DELIMITER — See Formatting Reconstructed SQL Statements for Reexecution on page 19-24
- PRINT_PRETTY_SQL — See Formatting the Appearance of Returned Data for Readability on page 19-25

- NO_ROWID_IN_STMT — See Formatting Reconstructed SQL Statements for Reexecution on page 19-24
- DDL_DICT_TRACKING — See Tracking DDL Statements in the LogMiner Dictionary on page 19-35

When you execute the DBMS_LOGMNR.START_LOGMNR procedure, LogMiner checks to ensure that the combination of options and parameters that you have specified is valid and that the dictionary and redo log files that you have specified are available. However, the V$LOGMNR_CONTENTS view is not populated until you query the view, as described in How the V$LOGMNR_CONTENTS View Is Populated on page 19-16.

Note that parameters and options are not persistent across calls to DBMS_LOGMNR.START_LOGMNR. You must specify all desired parameters and options (including SCN and time ranges) each time you call DBMS_LOGMNR.START_LOGMNR.

Querying V$LOGMNR_CONTENTS for Redo Data of Interest

You access the redo data of interest by querying the V$LOGMNR_CONTENTS view. This view provides historical information about changes made to the database, including (but not limited to) the following:

- The type of change made to the database: INSERT, UPDATE, DELETE, or DDL (OPERATION column).
- The SCN at which a change was made (SCN column).
- The SCN at which a change was committed (COMMIT_SCN column).
- The transaction to which a change belongs (XIDUSN, XIDSLT, and XIDSQN columns).
- The table and schema name of the modified object (SEG_NAME and SEG_OWNER columns).
- The name of the user who issued the DDL or DML statement to make the change (USERNAME column).
- If the change was due to a SQL DML statement, the reconstructed SQL statements showing SQL DML that is equivalent (but not necessarily identical) to the SQL DML used to generate the redo records (SQL_REDO column).

- If a password is part of the statement in a SQL_REDO column, the password is encrypted. SQL_REDO column values that correspond to DDL statements are always identical to the SQL DDL used to generate the redo records.
- If the change was due to a SQL DML change, the reconstructed SQL statements showing the SQL DML statements needed to undo the change (SQL_UNDO column).

  SQL_UNDO columns that correspond to DDL statements are always NULL. The SQL_UNDO column may be NULL also for some datatypes and for rolled back operations.

For example, suppose you wanted to find out about any delete operations that a user named Ron had performed on the oe.orders table. You could issue a query similar to the following:

```
SQL> SELECT OPERATION, SQL_REDO, SQL_UNDO
       FROM V$LOGMNR_CONTENTS
       WHERE SEG_OWNER = 'OE' AND SEG_NAME = 'ORDERS' AND
       OPERATION = 'DELETE' AND USERNAME = 'RON';
```

The following output would be produced. The formatting may be different on your display than that shown here.

```
OPERATION   SQL_REDO                          SQL_UNDO

DELETE      delete from "OE"."ORDERS"        insert into "OE"."ORDERS"
            where "ORDER_ID" = '2413'        ("ORDER_ID","ORDER_MODE",
            and "ORDER_MODE" = 'direct'       "CUSTOMER_ID","ORDER_STATUS",
            and "CUSTOMER_ID" = '101'         "ORDER_TOTAL","SALES_REP_ID",
            and "ORDER_STATUS" = '5'          "PROMOTION_ID")
            and "ORDER_TOTAL" = '48552'       values ('2413','direct','101',
            and "SALES_REP_ID" = '161'        '5','48552','161',NULL);
            and "PROMOTION_ID" IS NULL
            and ROWID = 'AAAHTCAABAAAZAPAAN';

DELETE      delete from "OE"."ORDERS"         insert into "OE"."ORDERS"
            where "ORDER_ID" = '2430'         ("ORDER_ID","ORDER_MODE",
            and "ORDER_MODE" = 'direct'        "CUSTOMER_ID","ORDER_STATUS",
            and "CUSTOMER_ID" = '101'          "ORDER_TOTAL","SALES_REP_ID",
            and "ORDER_STATUS" = '8'           "PROMOTION_ID")
            and "ORDER_TOTAL" = '29669.9'      values('2430','direct','101',
            and "SALES_REP_ID" = '159'         '8','29669.9','159',NULL);
            and "PROMOTION_ID" IS NULL
            and ROWID = 'AAAHTCAABAAAZAPAAe';
```

This output shows that user Ron deleted two rows from the `oe.orders` table. The reconstructed SQL statements are equivalent, but not necessarily identical, to the actual statement that Ron issued. The reason for this is that the original `WHERE` clause is not logged in the redo log files, so LogMiner can only show deleted (or updated or inserted) rows individually.

Therefore, even though a single `DELETE` statement may have been responsible for the deletion of both rows, the output in `V$LOGMNR_CONTENTS` does not reflect that. Thus, the actual `DELETE` statement may have been `DELETE FROM OE.ORDERS WHERE CUSTOMER_ID ='101'` or it might have been `DELETE FROM OE.ORDERS WHERE PROMOTION_ID = NULL.`

How the V$LOGMNR_CONTENTS View Is Populated

The `V$LOGMNR_CONTENTS` fixed view is unlike other views in that it is not a selective presentation of data stored in a table. Instead, it is a relational presentation of the data that you request from the redo log files. LogMiner populates the view only in response to a query against it. You must successfully start LogMiner before you can query `V$LOGMNR_CONTENTS`.

When a SQL select operation is executed against the `V$LOGMNR_CONTENTS` view, the redo log files are read sequentially. Translated information from the redo log files is returned as rows in the `V$LOGMNR_CONTENTS` view. This continues until either the filter criteria specified at startup are met or the end of the redo log file is reached.

LogMiner returns all the rows in SCN order unless you have used the `COMMITTED_DATA_ONLY` option to specify that only committed transactions should be retrieved. SCN order is the order normally applied in media recovery.

> **See Also:** Showing Only Committed Transactions on page 19-19 for more information about the `COMMITTED_DATA_ONLY` option to `DBMS_LOGMNR.START_LOGMNR`

Note: Because LogMiner populates the V$LOGMNR_CONTENTS view only in response to a query and does not store the requested data in the database, the following is true:

- Every time you query V$LOGMNR_CONTENTS, LogMiner analyzes the redo log files for the data you request.
- The amount of memory consumed by the query is not dependent on the number of rows that must be returned to satisfy a query.
- The time it takes to return the requested data is dependent on the amount and type of redo log data that must be mined to find that data.

For the reasons stated in the previous note, Oracle recommends that you create a table to temporarily hold the results from a query of V$LOGMNR_CONTENTS if you need to maintain the data for further analysis, particularly if the amount of data returned by a query is small in comparison to the amount of redo data that LogMiner must analyze to provide that data.

Querying V$LOGMNR_CONTENTS Based on Column Values

LogMiner lets you make queries based on column values. For instance, you can perform a query to show all updates to the hr.employees table that increase salary more than a certain amount. Data such as this can be used to analyze system behavior and to perform auditing tasks.

LogMiner data extraction from redo log files is performed using two mine functions: DBMS_LOGMNR.MINE_VALUE and DBMS_LOGMNR.COLUMN_PRESENT. Support for these mine functions is provided by the REDO_VALUE and UNDO_VALUE columns in the V$LOGMNR_CONTENTS view.

The following is an example of how you could use the MINE_VALUE function to select all updates to hr.employees that increased the salary column to more than twice its original value:

```
SQL> SELECT SQL_REDO FROM V$LOGMNR_CONTENTS
       WHERE
       SEG_NAME = 'EMPLOYEES' AND
       SEG_OWNER = 'HR' AND
       OPERATION = 'UPDATE' AND
```

```
    DBMS_LOGMNR.MINE_VALUE(REDO_VALUE, 'HR.EMPLOYEES.SALARY') >
    2*DBMS_LOGMNR.MINE_VALUE(UNDO_VALUE, 'HR.EMPLOYEES.SALARY');
```

As shown in this example, the MINE_VALUE function takes two arguments:

- The first one specifies whether to mine the redo (REDO_VALUE) or undo (UNDO_VALUE) portion of the data. The redo portion of the data is the data that is in the column after an insert, update, or delete operation; the undo portion of the data is the data that was in the column before an insert, update, or delete operation. It may help to think of the REDO_VALUE as the new value and the UNDO_VALUE as the old value.
- The second argument is a string that specifies the fully qualified name of the column to be mined (in this case, hr.employees.salary). The MINE_VALUE function always returns a string that can be converted back to the original datatype.

The Meaning of NULL Values Returned by the MINE_VALUE Function

If the MINE_VALUE function returns a NULL value, it can mean either:

- The specified column is not present in the redo or undo portion of the data.
- The specified column is present and has a null value.

To distinguish between these two cases, use the DBMS_LOGMNR.COLUMN_PRESENT function which returns a 1 if the column is present in the redo or undo portion of the data. Otherwise, it returns a 0. For example, suppose you wanted to find out the increment by which the values in the salary column were modified and the corresponding transaction identifier. You could issue the following query:

```
SQL> SELECT
       (XIDUSN || '.' || XIDSLT || '.' || XIDSQN) AS XID,
       (DBMS_LOGMNR.MINE_VALUE(REDO_VALUE, 'HR.EMPLOYEES.SALARY') -
       DBMS_LOGMNR.MINE_VALUE(UNDO_VALUE, 'HR.EMPLOYEES.SALARY')) AS INCR_SAL
       FROM V$LOGMNR_CONTENTS
       WHERE
       OPERATION = 'UPDATE' AND
       DBMS_LOGMNR.COLUMN_PRESENT(REDO_VALUE, 'HR.EMPLOYEES.SALARY') = 1 AND
       DBMS_LOGMNR.COLUMN_PRESENT(UNDO_VALUE, 'HR.EMPLOYEES.SALARY') = 1;
```

Usage Rules for the MINE_VALUE and COLUMN_PRESENT Functions

The following usage rules apply to the MINE_VALUE and COLUMN_PRESENT functions:

- They can only be used within a LogMiner session.

- They must be invoked in the context of a select operation from the V$LOGMNR_CONTENTS view.
- They do not support LONG, LONG RAW, CLOB, BLOB, NCLOB, ADT, or COLLECTION datatypes.

> **See Also:** *PL/SQL Packages and Types Reference* for a description of the DBMS_LOGMNR package, which contains the MINE_VALUE and COLUMN_PRESENT functions

Filtering and Formatting Data Returned to V$LOGMNR_CONTENTS

LogMiner can potentially deal with large amounts of information. You can limit the information that is returned to the V$LOGMNR_CONTENTS view, and the speed at which it is returned. The following sections demonstrate how to specify these limits and their impact on the data returned when you query V$LOGMNR_CONTENTS.

- Showing Only Committed Transactions
- Skipping Redo Corruptions
- Filtering Data by Time
- Filtering Data by SCN

In addition, LogMiner offers features for formatting the data that is returned to V$LOGMNR_CONTENTS, as described in the following sections:

- Formatting Reconstructed SQL Statements for Reexecution
- Formatting the Appearance of Returned Data for Readability

You request each of these filtering and formatting features using parameters or options to the DBMS_LOGMNR.START_LOGMNR procedure.

Showing Only Committed Transactions

When you use the COMMITTED_DATA_ONLY option to DBMS_LOGMNR.START_LOGMNR, only rows belonging to committed transactions are shown in the V$LOGMNR_CONTENTS view. This enables you to filter out rolled back transactions, transactions that are in progress, and internal operations.

To enable this option, specify it when you start LogMiner, as follows:

```
SQL> EXECUTE DBMS_LOGMNR.START_LOGMNR(OPTIONS => -
       DBMS_LOGMNR.COMMITTED_DATA_ONLY);
```

When you specify the COMMITTED_DATA_ONLY option, LogMiner groups together all DML operations that belong to the same transaction. Transactions are returned in the order in which they were committed.

> **Note:** If the COMMITTED_DATA_ONLY option is specified and you issue a query, LogMiner stages all redo records within a single transaction in memory until LogMiner finds the commit record for that transaction. Therefore, it is possible to exhaust memory, in which case an "Out of Memory" error will be returned. If this occurs, you must restart LogMiner without the COMMITTED_DATA_ONLY option specified and reissue the query.

The default is for LogMiner to show rows corresponding to all transactions and to return them in the order in which they are encountered in the redo log files.

For example, suppose you start LogMiner without specifying the COMMITTED_DATA_ONLY option and you execute the following query:

```
SQL> SELECT (XIDUSN || '.' || XIDSLT || '.' || XIDSQN) AS XID,
USERNAME, SQL_REDO FROM V$LOGMNR_CONTENTS WHERE USERNAME != 'SYS'
AND SEG_OWNER IS NULL OR SEG_OWNER NOT IN ('SYS', 'SYSTEM');
```

The output is as follows. Both committed and uncommitted transactions are returned and rows from different transactions are interwoven.

```
XID         USERNAME  SQL_REDO

1.15.3045   RON       set transaction read write;
1.15.3045   RON       insert into "HR"."JOBS"("JOB_ID","JOB_TITLE",
                      "MIN_SALARY","MAX_SALARY") values ('9782',
                      'HR_ENTRY',NULL,NULL);
1.18.3046   JANE      set transaction read write;
1.18.3046   JANE      insert into "OE"."CUSTOMERS"("CUSTOMER_ID",
                      "CUST_FIRST_NAME","CUST_LAST_NAME",
                      "CUST_ADDRESS","PHONE_NUMBERS","NLS_LANGUAGE",
                      "NLS_TERRITORY","CREDIT_LIMIT","CUST_EMAIL",
                      "ACCOUNT_MGR_ID") values ('9839','Edgar',
                      'Cummings',NULL,NULL,NULL,NULL,
                       NULL,NULL,NULL);
1.9.3041    RAJIV      set transaction read write;
1.9.3041    RAJIV      insert into "OE"."CUSTOMERS"("CUSTOMER_ID",
                       "CUST_FIRST_NAME","CUST_LAST_NAME","CUST_ADDRESS",
                       "PHONE_NUMBERS","NLS_LANGUAGE","NLS_TERRITORY",
```

```
                        "CREDIT_LIMIT","CUST_EMAIL","ACCOUNT_MGR_ID")
                        values ('9499','Rodney','Emerson',NULL,NULL,NULL,NULL,
                        NULL,NULL,NULL);
1.15.3045    RON        commit;
1.8.3054     RON        set transaction read write;
1.8.3054     RON        insert into "HR"."JOBS"("JOB_ID","JOB_TITLE",
                        "MIN_SALARY","MAX_SALARY") values ('9566',
                        'FI_ENTRY',NULL,NULL);
1.18.3046    JANE       commit;
1.11.3047    JANE       set transaction read write;
1.11.3047    JANE       insert into "OE"."CUSTOMERS"("CUSTOMER_ID",
                        "CUST_FIRST_NAME","CUST_LAST_NAME",
                        "CUST_ADDRESS","PHONE_NUMBERS","NLS_LANGUAGE",
                        "NLS_TERRITORY","CREDIT_LIMIT","CUST_EMAIL",
                        "ACCOUNT_MGR_ID") values ('8933','Ronald',
                        'Frost',NULL,NULL,NULL,NULL,NULL,NULL,NULL);
1.11.3047    JANE       commit;
1.8.3054     RON        commit;
```

Now suppose you start LogMiner, but this time you specify the COMMITTED_DATA_ONLY option. If you execute the previous query again, the output is as follows:

```
1.15.3045   RON        set transaction read write;
1.15.3045   RON        insert into "HR"."JOBS"("JOB_ID","JOB_TITLE",
                       "MIN_SALARY","MAX_SALARY") values ('9782',
                       'HR_ENTRY',NULL,NULL);
1.15.3045    RON        commit;
1.18.3046   JANE       set transaction read write;
1.18.3046   JANE       insert into "OE"."CUSTOMERS"("CUSTOMER_ID",
                       "CUST_FIRST_NAME","CUST_LAST_NAME",
                       "CUST_ADDRESS","PHONE_NUMBERS","NLS_LANGUAGE",
                       "NLS_TERRITORY","CREDIT_LIMIT","CUST_EMAIL",
                       "ACCOUNT_MGR_ID") values ('9839','Edgar',
                       'Cummings',NULL,NULL,NULL,NULL,
                        NULL,NULL,NULL);
1.18.3046    JANE       commit;
1.11.3047    JANE       set transaction read write;
1.11.3047    JANE       insert into "OE"."CUSTOMERS"("CUSTOMER_ID",
                        "CUST_FIRST_NAME","CUST_LAST_NAME",
                        "CUST_ADDRESS","PHONE_NUMBERS","NLS_LANGUAGE",
                        "NLS_TERRITORY","CREDIT_LIMIT","CUST_EMAIL",
                        "ACCOUNT_MGR_ID") values ('8933','Ronald',
                        'Frost',NULL,NULL,NULL,NULL,NULL,NULL,NULL);
1.11.3047    JANE       commit;
1.8.3054     RON        set transaction read write;
```

```
1.8.3054     RON       insert into "HR"."JOBS"("JOB_ID","JOB_TITLE",
                       "MIN_SALARY","MAX_SALARY") values ('9566',
                       'FI_ENTRY',NULL,NULL);
1.8.3054     RON       commit;
```

Because the COMMIT statement for the 1.15.3045 transaction was issued before the COMMIT statement for the 1.18.3046 transaction, the entire 1.15.3045 transaction is returned first. This is true even though the 1.18.3046 transaction started before the 1.15.3045 transaction. None of the 1.9.3041 transaction is returned because a COMMIT statement was never issued for it.

> **See Also:** See Examples Using LogMiner on page 19-48 for a complete example that uses the COMMITTED_DATA_ONLY option

Skipping Redo Corruptions

When you use the SKIP_CORRUPTION option to DBMS_LOGMNR.START_LOGMNR, any corruptions in the redo log files are skipped during select operations from the V$LOGMNR_CONTENTS view. For every corrupt redo record encountered, a row is returned that contains the value CORRUPTED_BLOCKS in the OPERATION column, 1343 in the STATUS column, and the number of blocks skipped in the INFO column.

Be aware that the skipped records may include changes to ongoing transactions in the corrupted blocks; such changes will not be reflected in the data returned from the V$LOGMNR_CONTENTS view.

The default is for the select operation to terminate at the first corruption it encounters in the redo log file.

The following example shows how this option works:

```
-- Add redo log files of interest.
--
SQL> EXECUTE DBMS_LOGMNR.ADD_LOGFILE(-
        logfilename => '/usr/oracle/data/db1arch_1_16_482701534.log' -
        options => DBMS_LOGMNR.NEW);

-- Start LogMiner
--
SQL> EXECUTE DBMS_LOGMNR.START_LOGMNR();

-- Select from the V$LOGMINER_CONTENTS view. This example shows corruptions are
-- in the redo log files.
--
```

```
SQL> SELECT rbasqn, rbablk, rbabyte, operation, status, info
     FROM V$LOGMNR_CONTENTS;

ERROR at line 3:
ORA-00368: checksum error in redo log block
ORA-00353: log corruption near block 6 change 73528 time 11/06/2002 11:30:23
ORA-00334: archived log: /usr/oracle/data/dbarch1_16_482701534.log

-- Restart LogMiner. This time, specify the SKIP_CORRUPTION option.
--
SQL> EXECUTE DBMS_LOGMNR.START_LOGMNR(-
     options => DBMS_LOGMNR.SKIP_CORRUPTION);

-- Select from the V$LOGMINER_CONTENTS view again. The output indicates that
-- corrupted blocks were skipped: CORRUPTED_BLOCKS is in the OPERATION
-- column, 1343 is in the STATUS column, and the number of corrupt blocks
-- skipped is in the INFO column.
--
SQL> SELECT rbasqn, rbablk, rbabyte, operation, status, info
     FROM V$LOGMNR_CONTENTS;

RBASQN  RBABLK RBABYTE  OPERATION        STATUS  INFO
13      2        76     START              0
13      2        76     DELETE             0
13      3       100     INTERNAL           0
13      3       380     DELETE             0
13      0         0     CORRUPTED_BLOCKS   1343  corrupt blocks 4 to 19 skipped
13      20      116     UPDATE             0
```

Filtering Data by Time

To filter data by time, set the STARTTIME and ENDTIME parameters in the DBMS_LOGMNR.START_LOGMNR procedure.

To avoid the need to specify the date format in the call to the DBMS_LOGMNR.START_LOGMNR procedure, you can use the ALTER SESSION SET NLS_DATE_FORMAT statement first, as shown in the following example.

```
SQL> ALTER SESSION SET NLS_DATE_FORMAT = 'DD-MON-YYYY HH24:MI:SS';
SQL> EXECUTE DBMS_LOGMNR.START_LOGMNR( -
       DICTFILENAME => '/oracle/database/dictionary.ora', -
       STARTTIME => '01-Jan-1998 08:30:00', -
       ENDTIME => '01-Jan-1998 08:45:00');
```

The timestamps should not be used to infer ordering of redo records. You can infer the order of redo records by using the SCN.

See Also:

- Examples Using LogMiner on page 19-48 for a complete example of filtering data by time
- *PL/SQL Packages and Types Reference* for information about what happens if you specify starting and ending times and they are not found in the LogMiner redo log file list, and for information about how these parameters interact with the CONTINUOUS_MINE option

Filtering Data by SCN

To filter data by SCN (system change number), use the STARTSCN and ENDSCN parameters to DBMS_LOGMNR.START_LOGMNR, as in this example:

```
EXECUTE DBMS_LOGMNR.START_LOGMNR(-
        STARTSCN => 621047, -
        ENDSCN   => 625695, -
        OPTIONS  => DBMS_LOGMNR.DICT_FROM_ONLINE_CATALOG + -
                    DBMS_LOGMNR.CONTINUOUS_MINE);
```

The STARTSCN and ENDSCN parameters override the STARTTIME and ENDTIME parameters in situations where all are specified.

See Also:

- Examples Using LogMiner on page 19-48 for a complete example of filtering data by SCN
- *PL/SQL Packages and Types Reference* for information about what happens if you specify starting and ending SCN values and they are not found in the LogMiner redo log file list and for information about how these parameters interact with the CONTINUOUS_MINE option

Formatting Reconstructed SQL Statements for Reexecution

By default, a ROWID clause is included in the reconstructed SQL_REDO and SQL_UNDO statements and the statements are ended with a semicolon.

However, you can override the default settings, as follows:

- Specify the NO_ROWID_IN_STMT option when you start LogMiner.

  This excludes the ROWID clause from the reconstructed statements. Because row IDs are not consistent between databases, if you intend to reexecute the SQL_REDO or SQL_UNDO statements against a different database than the one against which they were originally executed, specify the NO_ROWID_IN_STMT option when you start LogMiner.

- Specify the NO_SQL_DELIMITER option when you start LogMiner.

  This suppresses the semicolon from the reconstructed statements. This is helpful for applications that open a cursor and then execute the reconstructed statements.

Note that if the STATUS field of the V$LOGMNR_CONTENTS view contains the value 2 (invalid sql), then the associated SQL statement cannot be executed.

Formatting the Appearance of Returned Data for Readability

Sometimes a query can result in a large number of columns containing reconstructed SQL statements, which can be visually busy and hard to read. LogMiner provides the PRINT_PRETTY_SQL option to address this problem. The PRINT_PRETTY_SQL option to the DBMS_LOGMNR.START_LOGMNR procedure formats the reconstructed SQL statements as follows, which makes them easier to read:

```
insert into "HR"."JOBS"
 values
    "JOB_ID" = '9782',
    "JOB_TITLE" = 'HR_ENTRY',
    "MIN_SALARY" IS NULL,
    "MAX_SALARY" IS NULL;
  update "HR"."JOBS"
  set
    "JOB_TITLE" = 'FI_ENTRY'
  where
    "JOB_TITLE" = 'HR_ENTRY' and
    ROWID = 'AAAHSeAABAAAY+CAAX';

update "HR"."JOBS"
  set
    "JOB_TITLE" = 'FI_ENTRY'
  where
    "JOB_TITLE" = 'HR_ENTRY' and
    ROWID = 'AAAHSeAABAAAY+CAAX';
```

```
delete from "HR"."JOBS"
 where
    "JOB_ID" = '9782' and
    "JOB_TITLE" = 'FI_ENTRY' and
    "MIN_SALARY" IS NULL and
    "MAX_SALARY" IS NULL and
    ROWID = 'AAAHSeAABAAAY+CAAX';
```

SQL statements that are reconstructed when the PRINT_PRETTY_SQL option is enabled are not executable, because they do not use standard SQL syntax.

> **See Also:** Examples Using LogMiner on page 19-48 for a complete example of using the PRINT_PRETTY_SQL option

Reapplying DDL Statements Returned to V$LOGMNR_CONTENTS

Be aware that some DDL statements issued by a user cause Oracle to internally execute one or more other DDL statements. If you want to reapply SQL DDL from the SQL_REDO or SQL_UNDO columns of the V$LOGMNR_CONTENTS view as it was originally applied to the database, you should not execute statements that were executed internally by Oracle.

> **Note:** If you execute DML statements that were executed internally by Oracle you may corrupt your database. See Step 5 of Example 4: Using the LogMiner Dictionary in the Redo Log Files on page 19-58 for an example.

To differentiate between DDL statements that were issued by a user from those that were issued internally by Oracle, query the INFO column of V$LOGMNR_CONTENTS. The value of the INFO column indicates whether the DDL was executed by a user or by Oracle.

If you want to reapply SQL DDL as it was originally applied, you should only reexecute the DDL SQL contained in the SQL_REDO or SQL_UNDO column of V$LOGMNR_CONTENTS if the INFO column contains the value USER_DDL.

Calling DBMS_LOGMNR.START_LOGMNR Multiple Times

Even after you have successfully called DBMS_LOGMNR.START_LOGMNR and selected from the V$LOGMNR_CONTENTS view, you can call DBMS_LOGMNR.START_

LOGMNR again without ending the current LogMiner session and specify different options and time or SCN ranges. The following list presents reasons why you might want to do this:

- You want to limit the amount of redo data that LogMiner has to analyze.
- You want to specify different options. For example, you might decide to specify the PRINT_PRETTY_SQL option or that you only want to see committed transactions (so you specify the COMMITTED_DATA_ONLY option).
- You want to change the time or SCN range to be analyzed.

The following examples illustrate situations where it might be useful to call DBMS_LOGMNR.START_LOGMNR multiple times.

Example 1 Mining Only a Subset of the Data in the Redo Log Files

Suppose the list of redo log files that LogMiner has to mine include those generated for an entire week. However, you want to analyze only what happened from 12:00 to 1:00 each day. You could do this most efficiently by:

1. Calling DBMS_LOGMNR.START_LOGMNR with this time range for Monday.
2. Selecting changes from the V$LOGMNR_CONTENTS view.
3. Repeating Steps 1 and 2 for each day of the week.

If the total amount of redo data is large for the week, then this method would make the whole analysis much faster, because only a small subset of each redo log file in the list would be read by LogMiner.

Example 1 Adjusting the Time Range or SCN Range

Suppose you specify a redo log file list and specify a time (or SCN) range when you start LogMiner. When you query the V$LOGMNR_CONTENTS view, you find that only part of the data of interest is included in the time range you specified. You can call DBMS_LOGMNR.START_LOGMNR again to expand the time range by an hour (or adjust the SCN range).

Example 2 Analyzing Redo Log Files As They Arrive at a Remote Database

Suppose you have written an application to analyze changes or to replicate changes from one database to another database. The source database sends its redo log files to the mining database and drops them into an operating system directory. Your application:

1. Adds all redo log files currently in the directory to the redo log file list

2. Calls DBMS_LOGMNR.START_LOGMNR with appropriate settings and selects from the V$LOGMNR_CONTENTS view
3. Adds additional redo log files that have newly arrived in the directory
4. Repeats Steps 2 and 3, indefinitely

Supplemental Logging

Redo log files are generally used for instance recovery and media recovery. The data needed for such operations is automatically recorded in the redo log files. However, a redo-based application may require that additional columns be logged in the redo log files. The process of logging these additional columns is called **supplemental logging.**

By default, Oracle Database does not provide any supplemental logging, which means that the following LogMiner features are not supported by default:

- Index clusters, chained rows, and migrated rows (For chained rows, supplemental logging is required, regardless of the compatibility level to which the database is set.)
- Direct-path inserts (also require that ARCHIVELOG mode be enabled)
- Extracting the LogMiner dictionary into the redo log files
- DDL tracking
- Generating SQL_REDO and SQL_UNDO with identification key information
- LONG and LOB datatypes

Therefore, to make full use of LogMiner features, you must enable supplemental logging.

The following are examples of situations in which additional columns may be needed:

- An application that applies reconstructed SQL statements to a different database must identify the update statement by a set of columns that uniquely identify the row (for example, a primary key), not by the ROWID shown in the reconstructed SQL returned by the V$LOGMNR_CONTENTS view, because the ROWID of one database will be different and therefore meaningless in another database.
- An application may require that the before-image of the whole row be logged, not just the modified columns, so that tracking of row changes is more efficient.

A **supplemental log group** is the set of additional columns to be logged when supplemental logging is enabled. There are two types of supplemental log groups that determine when columns in the log group are logged:

- **Unconditional supplemental log groups:** The before-images of specified columns are logged any time a row is updated, regardless of whether the update affected any of the specified columns. This is sometimes referred to as an ALWAYS log group.
- **Conditional supplemental log groups:** The before-images of all specified columns are logged only if at least one of the columns in the log group is updated.

Supplemental log groups can be system-generated or user-defined.

In addition to the two types of supplemental logging, there are two levels of supplemental logging, as described in the following sections:

- Database-Level Supplemental Logging on page 19-29
- Table-Level Supplemental Logging on page 19-32

> **See Also:** Querying Views for Supplemental Logging Settings on page 19-41

Database-Level Supplemental Logging

There are two types of database-level supplemental logging: minimal supplemental logging and identification key logging, as described in the following sections. Minimal supplemental logging does not impose significant overhead on the database generating the redo log files. However, enabling database-wide identification key logging can impose overhead on the database generating the redo log files. Oracle recommends that you at least enable minimal supplemental logging for LogMiner.

Minimal Supplemental Logging

Minimal supplemental logging logs the minimal amount of information needed for LogMiner to identify, group, and merge the redo operations associated with DML changes. It ensures that LogMiner (and any product building on LogMiner technology) has sufficient information to support chained rows and various storage arrangements, such as cluster tables. To enable minimal supplemental logging, execute the following statement:

```
SQL> ALTER DATABASE ADD SUPPLEMENTAL LOG DATA;
```

Note: In Oracle Database release 9.0.1, minimal supplemental logging was the default behavior in LogMiner. In release 9.2 and later, the default is no supplemental logging. Supplemental logging must be specifically enabled.

Database-Level Identification Key Logging

Identification key logging is necessary when redo log files will not be mined at the source database instance, for example, when the redo log files will be mined at a logical standby database.

Using database identification key logging, you can enable database-wide before-image logging for all updates by specifying one or more of the following options to the SQL ALTER DATABASE ADD SUPPLEMENTAL LOG statement:

- ALL system-generated unconditititional supplemental log group

  This option specifies that when a row is updated, all columns of that row (except for LOBs, LONGS, and ADTs) are placed in the redo log file.

  To enable all column logging at the database level, execute the following statement:

  ```
  SQL> ALTER DATABASE ADD SUPPLEMENTAL LOG DATA (ALL) COLUMNS;
  ```

- PRIMARY KEY system-generated unconditititional supplemental log group

  This option causes the database to place all columns of a row's primary key in the redo log file whenever a row containing a primary key is updated (even if no value in the primary key has changed).

  If a table does not have a primary key, but has one or more non-null unique index key constraints or index keys, then one of the unique index keys is chosen for logging as a means of uniquely identifying the row being updated.

  If the table has neither a primary key nor a non-null unique index key, then all columns except LONG and LOB are supplementally logged; this is equivalent to specifying ALL supplemental logging for that row. Therefore, Oracle recommends that when you use database-level primary key supplemental logging, all or most tables be defined to have primary or unique index keys.

  To enable primary key logging at the database level, execute the following statement:

  ```
  SQL> ALTER DATABASE ADD SUPPLEMENTAL LOG DATA (PRIMARY KEY) COLUMNS;
  ```

- UNIQUE index system-generated conditional supplemental log group

  This option causes the database to place all columns of a row's composite unique index key or bitmap index in the redo log file if any column belonging to the composite unique index key or bitmap index is modified.

  To enable unique index key and bitmap index logging at the database level, execute the following statement:

  ```
  SQL> ALTER DATABASE ADD SUPPLEMENTAL LOG DATA (UNIQUE) COLUMNS;
  ```

- FOREIGN KEY system-generated conditional supplemental log group

  This option causes the database to place all columns of a row's foreign key in the redo log file if any column belonging to the foreign key is modified.

  To enable foreign key logging at the database level, execute the following statement:

  ```
  SQL> ALTER DATABASE ADD SUPPLEMENTAL LOG DATA (FOREIGN KEY) COLUMNS;
  ```

Note: Regardless of whether or not identification key logging is enabled, the SQL statements returned by LogMiner always contain the ROWID clause. You can filter out the ROWID clause by using the NO_ROWID_IN_STMT option to the DBMS_LOGMNR.START_LOGMNR procedure call. See Formatting Reconstructed SQL Statements for Reexecution on page 19-24 for details.

Keep the following in mind when you use identification key logging:

- If the database is open when you enable identification key logging, all DML cursors in the cursor cache are invalidated. This can affect performance until the cursor cache is repopulated.
- When you enable identification key logging at the database level, minimal supplemental logging is enabled implicitly.
- Supplemental logging statements are cumulative. If you issue the following statements, both primary key and unique index key supplemental logging is enabled:

  ```
  SQL> ALTER DATABASE ADD SUPPLEMENTAL LOG DATA (PRIMARY KEY) COLUMNS;
  SQL> ALTER DATABASE ADD SUPPLEMENTAL LOG DATA (UNIQUE INDEX) COLUMNS;
  ```

Disabling Database-Level Supplemental Logging

You disable database-level supplemental logging using the SQL `ALTER DATABASE` statement with the `DROP SUPPLEMENTAL LOGGING` clause. You can drop supplemental logging attributes incrementally. For example, suppose you issued the following statements, in the following order:

```
SQL> ALTER DATABASE ADD SUPPLEMENTAL LOG DATA (PRIMARY KEY) COLUMNS;
SQL> ALTER DATABASE ADD SUPPLEMENTAL LOG DATA (UNIQUE INDEX) COLUMNS;
SQL> ALTER DATABASE DROP SUPPLEMENTAL LOG DATA (PRIMARY KEY) COLUMNS;
SQL> ALTER DATABASE DROP SUPPLEMENTAL LOG DATA;
```

The statements would have the following effects:

- After the first statement, primary key supplemental logging is enabled.
- After the second statement, primary key and unique index key supplemental logging are enabled.
- After the third statement, only unique index key supplemental logging is enabled.
- After the fourth statement, all supplemental logging is not disabled. The following error is returned: `ORA-32589: unable to drop minimal supplemental logging`.

To disable all database supplemental logging, you must first disable any identification key logging that has been enabled, then disable minimal supplemental logging. The following example shows the correct order:

```
SQL> ALTER DATABASE ADD SUPPLEMENTAL LOG DATA (PRIMARY KEY) COLUMNS;
SQL> ALTER DATABASE ADD SUPPLEMENTAL LOG DATA (UNIQUE INDEX) COLUMNS;
SQL> ALTER DATABASE DROP SUPPLEMENTAL LOG DATA (PRIMARY KEY) COLUMNS;
SQL> ALTER DATABASE DROP SUPPLEMENTAL LOG DATA (UNIQUE INDEX) COLUMNS;
SQL> ALTER DATABASE DROP SUPPLEMENTAL LOG DATA;
```

Dropping minimal supplemental log data is allowed only if no other variant of database-level supplemental logging is enabled.

Table-Level Supplemental Logging

Table-level supplemental logging specifies, at the table level, which columns are to be supplementally logged. You can use identification key logging or user-defined conditional and unconditional supplemental log groups to log supplemental information, as described in the following sections.

Table-Level Identification Key Logging

Identification key logging at the table level offers the same options as those provided at the database level: all, primary key, foreign key, and unique index key. However, when you specify identification key logging at the table level, only the specified table is affected. For example, if you enter the following SQL statement (specifying database-level supplemental logging), then whenever a column in any database table is changed, the entire row containing that column (except columns for LOBs, LONGs, and ADTs) will be placed in the redo log file:

```
SQL> ALTER DATABASE ADD SUPPLEMENTAL LOG DATA (ALL) COLUMNS;
```

However, if you enter the following statement (specifying table-level supplemental logging) instead, then only when a column in the `employees` table is changed will the entire row (except for LOB, LONGs, and ADTs) of the table be placed in the redo log file. If a column changes in the `departments` table, only the changed column will be placed in the redo log file.

```
SQL> ALTER TABLE HR.EMPLOYEES ADD SUPPLEMENTAL LOG DATA (ALL) COLUMNS;
```

Keep the following in mind when you use table-level identification key logging:

- If the database is open when you enable identification key logging on a table, all DML cursors for that table in the cursor cache are invalidated. This can affect performance until the cache is repopulated.
- Supplemental logging statements are cumulative. If you issue the following statements, both primary key and unique index key table-level supplemental logging is enabled:

```
SQL> ALTER TABLE HR.EMPLOYEES
       ADD SUPPLEMENTAL LOG DATA (PRIMARY KEY) COLUMNS;
SQL> ALTER TABLE HR.EMPLOYEES
       ADD SUPPLEMENTAL LOG DATA (UNIQUE INDEX) COLUMNS;
```

See Database-Level Identification Key Logging on page 19-30 for a description of each of the identification key logging options.

Table-Level User-Defined Supplemental Log Groups

In addition to table-level identification key logging, Oracle supports user-defined supplemental log groups. With user-defined supplemental log groups, you can specify which columns are supplementally logged. You can specify conditional or unconditional log groups, as follows:

- User-defined unconditional log groups

To enable supplemental logging that uses user-defined unconditional log groups, use the ALWAYS clause as shown in the following example:

```
SQL> ALTER TABLE HR.EMPLOYEES
     ADD SUPPLEMENTAL LOG GROUP emp_parttime (EMPLOYEE_ID, LAST_NAME,
     DEPARTMENT_ID) ALWAYS;
```

This creates a log group named emp_parttime on the hr.employees table that consists of the columns employee_id, last_name, and department_id. These columns will be logged every time an UPDATE statement is executed on the hr.employees table, regardless of whether or not the update affected these columns. (If you want to have the entire row image logged any time an update was made, use table-level ALL identification key logging, as described previously).

Note: LOB, LONG, and ADT columns cannot be supplementally logged.

- User-defined conditional supplemental log groups

  To enable supplemental logging that uses user-defined conditional log groups, omit the ALWAYS clause from your ALTER TABLE statement, as shown in the following example:

  ```
  SQL> ALTER TABLE HR.EMPLOYEES
         ADD SUPPLEMENTAL LOG GROUP emp_fulltime (EMPLOYEE_ID, LAST_NAME,
         DEPARTMENT_ID);
  ```

  This creates a log group named emp_fulltime on table hr.employees. Just like the previous example, it consists of the columns employee_id, last_name, and department_id. But because the ALWAYS clause was omitted, before-images of the columns will be logged only if at least one of the columns is updated.

For both unconditional and conditional user-defined supplemental log groups, you can explicitly specify that a column in the log group be excluded from supplemental logging by specifying the NO LOG option. When you specify a log group and use the NO LOG option, you must specify at least one column in the log group without the NO LOG option, as shown in the following example:

```
SQL> ALTER TABLE HR.EMPLOYEES
       ADD SUPPLEMENTAL LOG GROUP emp_parttime(
       DEPARTMENT_ID NO LOG, EMPLOYEE_ID);
```

This enables you to associate this column with other columns in the named supplemental log group such that any modification to the NO LOG column causes the other columns in the supplemental log group to be placed in the redo log file. This might be useful, for example, if you want to log certain columns in a group if a LONG column changes. You cannot supplementally log the LONG column itself; however, you can use changes to that column to trigger supplemental logging of other columns in the same row.

Usage Notes for User-Defined Supplemental Log Groups

Keep the following in mind when you specify user-defined supplemental log groups:

- A column can belong to more than one supplemental log group. However, the before-image of the columns gets logged only once.
- Redo log files do not contain any information about which supplemental log group a column is part of or whether a column's before-image is being logged because of supplemental log group logging or identification key logging.
- If you specify the same columns to be logged both conditionally and unconditionally, the columns are logged unconditionally.

Tracking DDL Statements in the LogMiner Dictionary

LogMiner automatically builds its own internal dictionary from the LogMiner dictionary that you specify when you start LogMiner (either an online catalog, a dictionary in the redo log files, or a flat file). This dictionary provides a snapshot of the database objects and their definitions.

If your LogMiner dictionary is in the redo log files or is a flat file, you can use the DDL_DICT_TRACKING option to the DBMS_LOGMNR.START_LOGMNR procedure to direct LogMiner to track data definition language (DDL) statements. DDL tracking enables LogMiner to successfully track structural changes made to a database object, such as adding or dropping columns from a table. For example:

```
SQL> EXECUTE DBMS_LOGMNR.START_LOGMNR(OPTIONS => -
       DBMS_LOGMNR.DDL_DICT_TRACKING + DBMS_LOGMNR.DICT_FROM_REDO_LOGS);
```

See Example 5: Tracking DDL Statements in the Internal Dictionary on page 19-69 for a complete example.

With this option set, LogMiner applies any DDL statements seen in the redo log files to its internal dictionary.

> **Note:** In general, it is a good idea to keep supplemental logging and the DDL tracking feature enabled, because if they are not enabled and a DDL event occurs, LogMiner returns some of the redo data as binary data. Also, a metadata version mismatch could occur.

When you enable DDL_DICT_TRACKING, data manipulation language (DML) operations performed on tables created after the LogMiner dictionary was extracted can be shown correctly.

For example, if a table employees is updated through two successive DDL operations such that column gender is added in one operation, and column commission_pct is dropped in the next, LogMiner will keep versioned information for employees for each of these changes. This means that LogMiner can successfully mine redo log files that are from before and after these DDL changes, and no binary data will be presented for the SQL_REDO or SQL_UNDO columns.

Because LogMiner automatically assigns versions to the database metadata, it will detect and notify you of any mismatch between its internal dictionary and the dictionary in the redo log files. If LogMiner detects a mismatch, it generates binary data in the SQL_REDO column of the V$LOGMNR_CONTENTS view, the INFO column contains the string "Dictionary Version Mismatch", and the STATUS column will contain the value 2.

> **Note:** It is important to understand that the LogMiner internal dictionary is not the same as the LogMiner dictionary contained in a flat file, in redo log files, or in the online catalog. LogMiner does update its internal dictionary, but it does not update the dictionary that is contained in a flat file, in redo log files, or in the online catalog.

The following list describes the requirements for specifying the DDL_DICT_TRACKING option with the DBMS_LOGMNR.START_LOGMNR procedure.

- The DDL_DICT_TRACKING option is not valid with the DICT_FROM_ONLINE_CATALOG option.
- The DDL_DICT_TRACKING option requires that the database be open.

- Supplemental logging must be enabled database-wide, or log groups must have been created for the tables of interest.

DDL_DICT_TRACKING and Supplemental Logging Settings

Note the following interactions that occur when various settings of dictionary tracking and supplemental logging are combined:

- If DDL_DICT_TRACKING is enabled, but supplemental logging is not enabled and:
  - A DDL transaction is encountered in the redo log file, then a query of V$LOGMNR_CONTENTS will terminate with the ORA-01347 error.
  - A DML transaction is encountered in the redo log file, LogMiner will not assume that the current version of the table (underlying the DML) in its dictionary is correct, and columns in V$LOGMNR_CONTENTS will be set as follows:
    - The SQL_REDO column will contain binary data.
    - The STATUS column will contain a value of 2 (which indicates that the SQL is not valid).
    - The INFO column will contain the string 'Dictionary Mismatch'.
- If DDL_DICT_TRACKING is not enabled and supplemental logging is not enabled, and the columns referenced in a DML operation match the columns in the LogMiner dictionary, then LogMiner assumes that the latest version in its dictionary is correct, and columns in V$LOGMNR_CONTENTS will be set as follows:
  - LogMiner will use the definition of the object in its dictionary to generate values for the SQL_REDO and SQL_UNDO columns.
  - The status column will contain a value of 3 (which indicates that the SQL is not guaranteed to be accurate).
  - The INFO column will contain the string 'no supplemental log data found'.
- If DDL_DICT_TRACKING is not enabled and supplemental logging is not enabled and there are more modified columns in the redo log file for a table than the LogMiner dictionary definition for the table defines, then:
  - The SQL_REDO and SQL_UNDO columns will contain the string 'Dictionary Version Mismatch'.

- The STATUS column will contain a value of 2 (which indicates that the SQL is not valid).
- The INFO column will contain the string 'Dictionary Mismatch'.

Also be aware that it is possible to get unpredictable behavior if the dictionary definition of a column indicates one type but the column is really another type.

DDL_DICT_TRACKING and Specified Time or SCN Ranges

Because LogMiner must not miss a DDL statement if it is to ensure the consistency of its dictionary, LogMiner may start reading redo log files prior to your requested starting time or SCN (as specified with DBMS_LOGMNR.START_LOGMNR) when the DDL_DICT_TRACKING option is enabled. The actual time or SCN at which LogMiner starts reading redo log files is referred to as the **required starting time** or the **required starting SCN**.

No missing redo log files (based on sequence numbers) are allowed from the required starting time or the required starting SCN.

LogMiner determines where it will start reading redo log data as follows:

- After the dictionary is loaded, the first time that you call DBMS_LOGMNR.START_LOGMNR, LogMiner begins reading as determined by one of the following, whichever causes it to begin earlier:
  - Your requested starting time or SCN value
  - The commit SCN of the dictionary dump
- On subsequent calls to DBMS_LOGMNR.START_LOGMNR, LogMiner begins reading as determined for one of the following, whichever causes it to begin earliest:
  - Your requested starting time or SCN value
  - The start of the earliest DDL transaction where the COMMIT statement has not yet been read by LogMiner
  - The highest SCN read by LogMiner

The following scenario helps illustrate this:

Suppose you create a redo log file list containing five redo log files. Assume that a dictionary is contained in the first redo file, and the changes that you have indicated that you want to see (using DBMS_LOGMNR.START_LOGMNR) are recorded in the third redo log file. You then do the following:

1. Call DBMS_LOGMNR.START_LOGMNR. LogMiner will read:
   a. The first log file to load the dictionary
   b. The second redo log file to pick up any possible DDLs contained within it
   c. The third log file to retrieve the data of interest
2. Call DBMS_LOGMNR.START_LOGMNR again with the same requested range.

   LogMiner will begin with redo log file 3; it no longer needs to read redo log file 2, because it has already processed any DDL statements contained within it.
3. Call DBMS_LOGMNR.START_LOGMNR again, this time specifying parameters that require data to be read from redo log file 5.

   LogMiner will start reading from redo log file 4 to pick up any DDL statements that may be contained within it.

Query the REQUIRED_START_DATE or the REQUIRED_START_SCN columns of the V$LOGMNR_PARAMETERS view to see where LogMiner will actually start reading. Regardless of where LogMiner starts reading, only rows in your requested range will be returned from the V$LOGMINER_CONTENTS view.

Accessing LogMiner Operational Information in Views

LogMiner operational information (as opposed to redo data) is contained in the following views. You can use SQL to query them as you would any other view.

- V$LOGMNR_DICTIONARY

  Shows information about a LogMiner dictionary file that was created using the STORE_IN_FLAT_FILE option to DBMS_LOGMNR.START_LOGMNR. The information shown includes information about the database from which the LogMiner dictionary was created.

- V$LOGMNR_LOGS

  Shows information about specified redo log files, as described in Querying V$LOGMNR_LOGS on page 19-40.

- V$LOGMNR_PARAMETERS

  Shows information about optional LogMiner parameters, including starting and ending system change numbers (SCNs) and starting and ending times.

- V$DATABASE, DBA_LOG_GROUPS, ALL_LOG_GROUPS, USER_LOG_GROUPS, DBA_LOG_GROUP_COLUMNS, ALL_LOG_GROUP_COLUMNS, USER_LOG_GROUP_COLUMNS

  Shows information about the current settings for supplemental logging, as described in Querying Views for Supplemental Logging Settings on page 19-41.

  See Also: *Oracle Database Reference* for detailed information about the contents of these views

Querying V$LOGMNR_LOGS

You can query the V$LOGMNR_LOGS view to determine which redo log files have been manually or automatically added to the list of redo log files for LogMiner to analyze. This view contains one row for each redo log file. It provides valuable information about each of the redo log files including filename, sequence #, SCN and time ranges, and whether it contains all or part of the LogMiner dictionary.

After a successful call to DBMS_LOGMNR.START_LOGMNR, the STATUS column of the V$LOGMNR_LOGS view contains one of the following values:

- 0

  Indicates that the redo log file will be processed during a query of the V$LOGMNR_CONTENTS view.

- 1

  Indicates that this will be the first redo log file to be processed by LogMiner during a select operation against the V$LOGMNR_CONTENTS view.

- 2

  Indicates that the redo log file has been pruned and therefore will not be processed by LogMiner during a query of the V$LOGMNR_CONTENTS view. It has been pruned because it is not needed to satisfy your requested time or SCN range.

- 4

  Indicates that a redo log file (based on sequence number) is missing from the LogMiner redo log file list.

The V$LOGMNR_LOGS view contains a row for each redo log file that is missing from the list, as follows:

- The FILENAME column will contain the consecutive range of sequence numbers and total SCN range gap.

For example: 'Missing log file(s) for thread number 1, sequence number(s) 100 to 102'.

- The INFO column will contain the string 'MISSING_LOGFILE'.

Information about files missing from the redo log file list can be useful for the following reasons:

- The DDL_DICT_TRACKING and CONTINUOUS_MINE options that can be specified when you call DBMS_LOGMNR.START_LOGMNR will not allow redo log files to be missing from the LogMiner redo log file list for the requested time or SCN range. If a call to DBMS_LOGMNR.START_LOGMNR fails, you can query the STATUS column in the V$LOGMNR_LOGS view to determine which redo log files are missing from the list. You can then find and manually add these redo log files and attempt to call DBMS_LOGMNR.START_LOGMNR again.
- Although all other options that can be specified when you call DBMS_LOGMNR.START_LOGMNR allow files to be missing from the LogMiner redo log file list, you may not want to have missing files. You can query the V$LOGMNR_LOGS view before querying the V$LOGMNR_CONTENTS view to ensure that all required files are in the list. If the list is left with missing files and you query the V$LOGMNR_CONTENTS view, a row is returned in V$LOGMNR_CONTENTS with the following column values:
  - In the OPERATION column, a value of 'MISSING_SCN'
  - In the STATUS column, a value of 1291
  - In the INFO column, a string indicating the missing SCN range (for example, 'Missing SCN 100 - 200')

Querying Views for Supplemental Logging Settings

You can query a number of views to determine the current settings for supplemental logging, as described in the following list:

- V$DATABASE view
  - SUPPLEMENTAL_LOG_DATA_FK column

    This column contains one of the following values:

    * NO - if database-level identification key logging with the FOREIGN KEY option is not enabled
    * YES - if database-level identification key logging with the FOREIGN KEY option is enabled

  - SUPPLEMENTAL_LOG_DATA_ALL column

    This column contains one of the following values:

    * NO - if database-level identification key logging with the ALL option is not enabled
    * YES - if database-level identification key logging with the ALL option is enabled

  - SUPPLEMENTAL_LOG_DATA_UI column

    * NO - if database-level identification key logging with the UNIQUE INDEX option is not enabled
    * YES - if database-level identification key logging with the UNIQUE INDEX option is enabled

  - SUPPLEMENTAL_LOG_DATA_MIN column

    This column contains one of the following values:

    * NO - if no database-level supplemental logging is enabled
    * IMPLICIT - if minimal supplemental logging is enabled because database-level identification key logging options is enabled
    * YES - if minimal supplemental logging is enabled because the SQL ALTER DATABASE ADD SUPPLEMENTAL LOG DATA statement was issued

- DBA_LOG_GROUPS, ALL_LOG_GROUPS, and USER_LOG_GROUPS views

  - ALWAYS column

    This column contains one of the following values:

    * ALWAYS - indicates that the columns in this log group will be supplementally logged if any column in the associated row is updated
    * CONDITIONAL - indicates that the columns in this group will be supplementally logged only if a column in the log group is updated

  - GENERATED column

    This column contains one of the following values:

    * GENERATED NAME - if the LOG_GROUP name was system-generated
    * USER NAME - if the LOG_GROUP name was user-defined

  - LOG_GROUP_TYPES column

This column contains one of the following values to indicate the type of logging defined for this log group. USER LOG GROUP indicates that the log group was user-defined (as opposed to system-generated).

* ALL COLUMN LOGGING
* FOREIGN KEY LOGGING
* PRIMARY KEY LOGGING
* UNIQUE KEY LOGGING
* USER LOG GROUP

- DBA_LOG_GROUP_COLUMNS, ALL_LOG_GROUP_COLUMNS, and USER_LOG_GROUP_COLUMNS views
  - The LOGGING_PROPERTY column

    This column contains one of the following values:

    * LOG - indicates that this column in the log group will be supplementally logged
    * NO LOG - indicates that this column in the log group will not be supplementally logged

Steps in a Typical LogMiner Session

This section describes the steps in a typical LogMiner session. Each step is described in its own subsection.

1. Enable Supplemental Logging
2. Extract a LogMiner Dictionary (unless you plan to use the online catalog)
3. Specify Redo Log Files for Analysis
4. Start LogMiner
5. Query V$LOGMNR_CONTENTS
6. End the LogMiner Session

To run LogMiner, you use the DBMS_LOGMNR PL/SQL package. Additionally, you might also use the DBMS_LOGMNR_D package if you choose to extract a LogMiner dictionary rather than use the online catalog.

The DBMS_LOGMNR package contains the procedures used to initialize and run LogMiner, including interfaces to specify names of redo log files, filter criteria, and

session characteristics. The DBMS_LOGMNR_D package queries the database dictionary tables of the current database to create a LogMiner dictionary file.

The LogMiner packages are owned by the SYS schema. Therefore, if you are not connected as user SYS:

- You must include SYS in your call. For example:

  ```
  SQL> EXECUTE SYS.DBMS_LOGMNR.END_LOGMNR;
  ```

- You must have been granted the EXECUTE_CATALOG_ROLE role.

See Also:

- *PL/SQL Packages and Types Reference* for details about syntax and parameters for these LogMiner packages
- *Oracle Database Application Developer's Guide - Fundamentals* for information about executing PL/SQL procedures

Enable Supplemental Logging

Enable the type of supplemental logging you want to use. At the very least, you must enable minimal supplemental logging, as follows:

```
SQL> ALTER DATABASE ADD SUPPLEMENTAL LOG DATA;
```

See Supplemental Logging on page 19-28 for more information.

Extract a LogMiner Dictionary

To use LogMiner, you must supply it with a dictionary by doing one of the following:

- Specify use of the online catalog by using the DICT_FROM_ONLINE_CATALOG option when you start LogMiner. See Using the Online Catalog on page 19-8.
- Extract database dictionary information to the redo log files. See Extracting a LogMiner Dictionary to the Redo Log Files on page 19-9.
- Extract database dictionary information to a flat file. See Extracting the LogMiner Dictionary to a Flat File on page 19-10.

Specify Redo Log Files for Analysis

Before you can start LogMiner, you must specify the redo log files that you want to analyze. To do so, execute the DBMS_LOGMNR.ADD_LOGFILE procedure, as demonstrated in the following steps. You can add and remove redo log files in any order.

Note: If you will be mining in the database instance that is generating the redo log files, you only need to specify the CONTINUOUS_MINE option and one of the following when you start LogMiner:

- The STARTSCN parameter
- The STARTTIME parameter

For more information, see Redo Log File Options on page 19-11.

---

1. Use SQL*Plus to start an Oracle instance, with the database either mounted or unmounted. For example, enter:

```
SQL> STARTUP
```

2. Create a list of redo log files. Specify the NEW option of the DBMS_LOGMNR.ADD_LOGFILE procedure to signal that this is the beginning of a new list. For example, enter the following to specify the /oracle/logs/log1.f redo log file:

```
SQL> EXECUTE DBMS_LOGMNR.ADD_LOGFILE( -
       LOGFILENAME => '/oracle/logs/log1.f', -
       OPTIONS => DBMS_LOGMNR.NEW);
```

3. If desired, add more redo log files by specifying the ADDFILE option of the DBMS_LOGMNR.ADD_LOGFILE procedure. For example, enter the following to add the /oracle/logs/log2.f redo log file:

```
SQL> EXECUTE DBMS_LOGMNR.ADD_LOGFILE( -
       LOGFILENAME => '/oracle/logs/log2.f', -
       OPTIONS => DBMS_LOGMNR.ADDFILE);
```

   The OPTIONS parameter is optional when you are adding additional redo log files. For example, you could simply enter the following:

```
SQL> EXECUTE DBMS_LOGMNR.ADD_LOGFILE( -
       LOGFILENAME=>'/oracle/logs/log2.f');
```

4. If desired, remove redo log files by using the DBMS_LOGMNR.REMOVE_LOGFILE procedure. For example, enter the following to remove the /oracle/logs/log2.f redo log file:

```
SQL> EXECUTE DBMS_LOGMNR.REMOVE_LOGFILE( -
       LOGFILENAME => '/oracle/logs/log2.f');
```

Start LogMiner

After you have created a LogMiner dictionary file and specified which redo log files to analyze, you must start LogMiner. Take the following steps:

1. Execute the DBMS_LOGMNR.START_LOGMNR procedure to start LogMiner.

   Oracle recommends that you specify a LogMiner dictionary option. If you do not, LogMiner cannot translate internal object identifiers and datatypes to object names and external data formats. Therefore, it would return internal object IDs and present data as binary data. Additionally, the MINE_VALUE and COLUMN_PRESENT functions cannot be used without a dictionary.

   If you are specifying the name of a flat file LogMiner dictionary, you must supply a fully qualified filename for the dictionary file. For example, to start LogMiner using /oracle/database/dictionary.ora, issue the following command:

```
SQL> EXECUTE DBMS_LOGMNR.START_LOGMNR( -
       DICTFILENAME =>'/oracle/database/dictionary.ora');
```

   If you are not specifying a flat file dictionary name, then use the OPTIONS parameter to specify either the DICT_FROM_REDO_LOGS or DICT_FROM_ONLINE_CATALOG option.

   If you specify DICT_FROM_REDO_LOGS, LogMiner expects to find a dictionary in the redo log files that you specified with the DBMS_LOGMNR.ADD_LOGFILE procedure. To determine which redo log files contain a dictionary, look at the V$ARCHIVED_LOG view. See Extracting a LogMiner Dictionary to the Redo Log Files on page 19-9 for an example.

> **Note:** If you add additional redo log files after LogMiner has been started, you must restart LogMiner. LogMiner will not retain options that were included in the previous call to DBMS_LOGMNR.START_LOGMNR; you must respecify those options that you want to use. However, LogMiner will retain the dictionary specification from the previous call if you do not specify a dictionary in the current call to DBMS_LOGMNR.START_LOGMNR.

For more information about the DICT_FROM_ONLINE_CATALOG option, see Using the Online Catalog on page 19-8.

2. Optionally, you can filter your query by time or by SCN. See Filtering Data by Time on page 19-23 or Filtering Data by SCN on page 19-24.

3. You can also use the OPTIONS parameter to specify additional characteristics of your LogMiner session. For example, you might decide to use the online catalog as your LogMiner dictionary and to have only committed transactions shown in the V$LOGMNR_CONTENTS view, as follows:

```
SQL> EXECUTE DBMS_LOGMNR.START_LOGMNR(OPTIONS => -
       DBMS_LOGMNR.DICT_FROM_ONLINE_CATALOG + -
       DBMS_LOGMNR.COMMITTED_DATA_ONLY);
```

For more information about DBMS_LOGMNR.START_LOGMNR options, see *PL/SQL Packages and Types Reference*.

You can execute the DBMS_LOGMNR.START_LOGMNR procedure multiple times, specifying different options each time. This can be useful, for example, if you did not get the desired results from a query of V$LOGMNR_CONTENTS, and want to restart LogMiner with different options. Unless you need to respecify the LogMiner dictionary, you do not need to add redo log files if they were already added with a previous call to DBMS_LOGMNR.START_LOGMNR.

Query V$LOGMNR_CONTENTS

At this point, LogMiner is started and you can perform queries against the V$LOGMNR_CONTENTS view. See Filtering and Formatting Data Returned to V$LOGMNR_CONTENTS on page 19-19 for examples of this.

End the LogMiner Session

To properly end a LogMiner session, use the DBMS_LOGMNR.END_LOGMNR procedure, as follows:

```
SQL> EXECUTE DBMS_LOGMNR.END_LOGMNR;
```

This procedure closes all the redo log files and allows all the database and system resources allocated by LogMiner to be released.

If this procedure is not executed, LogMiner retains all its allocated resources until the end of the Oracle session in which it was invoked. It is particularly important to use this procedure to end the LogMiner session if either the DDL_DICT_TRACKING option or the DICT_FROM_REDO_LOGS option was used.

Examples Using LogMiner

This section provides several examples of using LogMiner in each of the following general categories:

- Examples of Mining by Explicitly Specifying the Redo Log Files of Interest
- Examples of Mining Without Specifying the List of Redo Log Files Explicitly
- Example Scenarios

Note: All examples in this section assume that minimal supplemental logging has been enabled:

```
SQL> ALTER DATABASE ADD SUPPLEMENTAL LOG DATA;
```

See Supplemental Logging on page 19-28 for more information.

All examples, except Example 2: Mining the Redo Log Files in a Given SCN Range on page 19-79 and the Example Scenarios on page 19-82, assume that the NLS_DATE_FORMAT parameter has been set as follows:

```
SQL>  ALTER SESSION SET NLS_DATE_FORMAT = 'dd-mon-yyyy hh24:mi:ss';
```

Because LogMiner displays date data using the setting for the NLS_DATE_FORMAT parameter that is active for the user session, this step is optional. However, setting the parameter explicitly lets you predict the date format.

Examples of Mining by Explicitly Specifying the Redo Log Files of Interest

The following examples demonstrate how to use LogMiner when you know which redo log files contain the data of interest. This section contains the following list of examples; these examples are best read sequentially, because each example builds on the example or examples that precede it:

- Example 1: Finding All Modifications in the Last Archived Redo Log File
- Example 2: Grouping DML Statements into Committed Transactions
- Example 3: Formatting the Reconstructed SQL
- Example 4: Using the LogMiner Dictionary in the Redo Log Files
- Example 5: Tracking DDL Statements in the Internal Dictionary
- Example 6: Filtering Output by Time Range

The SQL output formatting may be different on your display than that shown in these examples.

Example 1: Finding All Modifications in the Last Archived Redo Log File

The easiest way to examine the modification history of a database is to mine at the source database and use the online catalog to translate the redo log files. This example shows how to do the simplest analysis using LogMiner.

This example finds all modifications that are contained in the last archived redo log generated by the database (assuming that the database is not an Oracle Real Application Clusters database).

Step 1 Determine which redo log file was most recently archived.

This example assumes that you know that you want to mine the redo log file that was most recently archived.

```
SQL> SELECT NAME FROM V$ARCHIVED_LOG
       WHERE FIRST_TIME = (SELECT MAX(FIRST_TIME) FROM V$ARCHIVED_LOG);

NAME
-------------------------------------------
/usr/oracle/data/db1arch_1_16_482701534.dbf
```

Step 2 Specify the list of redo log files to be analyzed.

Specify the redo log file that was returned by the query in Step 1. The list will consist of one redo log file.

```
SQL> EXECUTE DBMS_LOGMNR.ADD_LOGFILE( -
       LOGFILENAME => '/usr/oracle/data/db1arch_1_16_482701534.dbf', -
       OPTIONS => DBMS_LOGMNR.NEW);
```

Step 3 Start LogMiner.

Start LogMiner and specify the dictionary to use.

```
SQL> EXECUTE DBMS_LOGMNR.START_LOGMNR( -
       OPTIONS => DBMS_LOGMNR.DICT_FROM_ONLINE_CATALOG);
```

Step 4 Query the V$LOGMNR_CONTENTS view.

Note that there are four transactions (two of them were committed within the redo log file being analyzed, and two were not). The output shows the DML statements in the order in which they were executed; thus transactions interleave among themselves.

```
SQL> SELECT username AS USR, (XIDUSN || '.' || XIDSLT || '.' ||  XIDSQN) AS XID,
       SQL_REDO, SQL_UNDO FROM V$LOGMNR_CONTENTS WHERE username IN ('HR', 'OE');

USR    XID          SQL_REDO                          SQL_UNDO
----   ---------  --------------------------------------------------
HR     1.11.1476  set transaction read write;

HR     1.11.1476  insert into "HR"."EMPLOYEES"(     delete from "HR"."EMPLOYEES"
                  "EMPLOYEE_ID","FIRST_NAME",       where "EMPLOYEE_ID" = '306'
                  "LAST_NAME","EMAIL",              and "FIRST_NAME" = 'Nandini'
                  "PHONE_NUMBER","HIRE_DATE",       and "LAST_NAME" = 'Shastry'
                  "JOB_ID","SALARY",                and "EMAIL" = 'NSHASTRY'
                  "COMMISSION_PCT","MANAGER_ID",    and "PHONE_NUMBER" = '1234567890'
                  "DEPARTMENT_ID") values           and "HIRE_DATE" = TO_DATE('10-JAN-2003
                  ('306','Nandini','Shastry',       13:34:43', 'dd-mon-yyyy hh24:mi:ss')
                  'NSHASTRY', '1234567890',         and "JOB_ID" = 'HR_REP' and
                  TO_DATE('10-jan-2003 13:34:43',   "SALARY" = '120000' and
                  'dd-mon-yyyy hh24:mi:ss'),         "COMMISSION_PCT" = '.05' and
                  'HR_REP','120000', '.05',         "DEPARTMENT_ID" = '10' and
                  '105','10');                      ROWID = 'AAAHSkAABAAAY6rAAO';

OE     1.1.1484   set transaction read write;

OE     1.1.1484   update "OE"."PRODUCT_INFORMATION"  update "OE"."PRODUCT_INFORMATION"
                  set "WARRANTY_PERIOD" =            set "WARRANTY_PERIOD" =
                  TO_YMINTERVAL('+05-00') where      TO_YMINTERVAL('+01-00') where
                  "PRODUCT_ID" = '1799' and          "PRODUCT_ID" = '1799' and
```

```
                  "WARRANTY_PERIOD" =                "WARRANTY_PERIOD" =
                  TO_YMINTERVAL('+01-00') and        TO_YMINTERVAL('+05-00') and
                  ROWID = 'AAAHTKAABAAAY9mAAB';      ROWID = 'AAAHTKAABAAAY9mAAB';

OE     1.1.1484   update "OE"."PRODUCT_INFORMATION"  update "OE"."PRODUCT_INFORMATION"
                  set "WARRANTY_PERIOD" =            set "WARRANTY_PERIOD" =
                  TO_YMINTERVAL('+05-00') where      TO_YMINTERVAL('+01-00') where
                  "PRODUCT_ID" = '1801' and          "PRODUCT_ID" = '1801' and
                  "WARRANTY_PERIOD" =                "WARRANTY_PERIOD" =
                  TO_YMINTERVAL('+01-00') and        TO_YMINTERVAL('+05-00') and
                  ROWID = 'AAAHTKAABAAAY9mAAC';      ROWID ='AAAHTKAABAAAY9mAAC';

HR     1.11.1476  insert into "HR"."EMPLOYEES"(     delete from "HR"."EMPLOYEES"
                  "EMPLOYEE_ID","FIRST_NAME",        "EMPLOYEE_ID" = '307' and
                  "LAST_NAME","EMAIL",               "FIRST_NAME" = 'John' and
                  "PHONE_NUMBER","HIRE_DATE",        "LAST_NAME" = 'Silver' and
                  "JOB_ID","SALARY",                 "EMAIL" = 'JSILVER' and
                  "COMMISSION_PCT","MANAGER_ID",     "PHONE_NUMBER" = '5551112222'
                  "DEPARTMENT_ID") values            and "HIRE_DATE" = TO_DATE('10-jan-2003
                  ('307','John','Silver',            13:41:03', 'dd-mon-yyyy hh24:mi:ss')
                   'JSILVER', '5551112222',          and "JOB_ID" ='105' and "DEPARTMENT_ID"
                  TO_DATE('10-jan-2003 13:41:03',    = '50' and ROWID = 'AAAHSkAABAAAY6rAAP';
                  'dd-mon-yyyy hh24:mi:ss'),
                  'SH_CLERK','110000', '.05',
                  '105','50');

OE     1.1.1484   commit;

HR     1.15.1481   set transaction read write;

HR     1.15.1481  delete from "HR"."EMPLOYEES"      insert into "HR"."EMPLOYEES"(
                  where "EMPLOYEE_ID" = '205' and    "EMPLOYEE_ID","FIRST_NAME",
                  "FIRST_NAME" = 'Shelley' and       "LAST_NAME","EMAIL","PHONE_NUMBER",
                  "LAST_NAME" = 'Higgins' and        "HIRE_DATE", "JOB_ID","SALARY",
                  "EMAIL" = 'SHIGGINS' and           "COMMISSION_PCT","MANAGER_ID",
                  "PHONE_NUMBER" = '515.123.8080'    "DEPARTMENT_ID") values
                  and "HIRE_DATE" = TO_DATE(         ('205','Shelley','Higgins',
                  '07-jun-1994 10:05:01',            and     'SHIGGINS','515.123.8080',
                  'dd-mon-yyyy hh24:mi:ss')          TO_DATE('07-jun-1994 10:05:01',
                  and "JOB_ID" = 'AC_MGR'             'dd-mon-yyyy hh24:mi:ss'),
                  and "SALARY"= '12000'              'AC_MGR','12000',NULL,'101','110');
                  and "COMMISSION_PCT" IS NULL
                  and "MANAGER_ID"
                  = '101' and "DEPARTMENT_ID" =
                  '110' and ROWID =
```

```
                 'AAAHSkAABAAAY6rAAM';


OE     1.8.1484   set transaction read write;

OE     1.8.1484   update "OE"."PRODUCT_INFORMATION"  update "OE"."PRODUCT_INFORMATION"
                  set "WARRANTY_PERIOD" =             set "WARRANTY_PERIOD" =
                  TO_YMINTERVAL('+12-06') where       TO_YMINTERVAL('+20-00') where
                  "PRODUCT_ID" = '2350' and           "PRODUCT_ID" = '2350' and
                  "WARRANTY_PERIOD" =                 "WARRANTY_PERIOD" =
                  TO_YMINTERVAL('+20-00') and         TO_YMINTERVAL('+20-00') and
                  ROWID = 'AAAHTKAABAAAY9tAAD';        ROWID ='AAAHTKAABAAAY9tAAD';

HR     1.11.1476  commit;
```

Step 5 End the LogMiner session.

```
SQL> EXECUTE DBMS_LOGMNR.END_LOGMNR();
```

Example 2: Grouping DML Statements into Committed Transactions

As shown in the first example, Example 1: Finding All Modifications in the Last Archived Redo Log File on page 19-49, LogMiner displays all modifications it finds in the redo log files that it analyzes by default, regardless of whether the transaction has been committed or not. In addition, LogMiner shows modifications in the same order in which they were executed. Because DML statements that belong to the same transaction are not grouped together, visual inspection of the output can be difficult. Although you can use SQL to group transactions, LogMiner provides an easier way. In this example, the latest archived redo log file will again be analyzed, but it will return only committed transactions.

Step 1 Determine which redo log file was most recently archived by the database.

This example assumes that you know that you want to mine the redo log file that was most recently archived.

```
SQL> SELECT NAME FROM V$ARCHIVED_LOG
       WHERE FIRST_TIME = (SELECT MAX(FIRST_TIME) FROM V$ARCHIVED_LOG);

NAME
----------------------------------------
/usr/oracle/data/db1arch_1_16_482701534.dbf
```

Step 2 Specify the list of redo log files to be analyzed.

Specify the redo log file that was returned by the query in Step 1. The list will consist of one redo log file.

```
SQL> EXECUTE DBMS_LOGMNR.ADD_LOGFILE( -
       LOGFILENAME => '/usr/oracle/data/db1arch_1_16_482701534.dbf', -
       OPTIONS => DBMS_LOGMNR.NEW);
```

Step 3 Start LogMiner.

Start LogMiner by specifying the dictionary to use and the COMMITTED_DATA_ONLY option.

```
SQL> EXECUTE DBMS_LOGMNR.START_LOGMNR( -
       OPTIONS => DBMS_LOGMNR.DICT_FROM_ONLINE_CATALOG + -
       DBMS_LOGMNR.COMMITTED_DATA_ONLY);
```

Step 4 Query the V$LOGMNR_CONTENTS view.

Although transaction 1.11.1476 was started before transaction 1.1.1484 (as revealed in Example 1: Finding All Modifications in the Last Archived Redo Log File on page 19-49), it committed after transaction 1.1.1484 committed. In this example, therefore, transaction 1.1.1484 is shown in its entirety before transaction 1.11.1476. The two transactions that did not commit within the redo log file being analyzed are not returned.

```
SQL> SELECT username AS USR, (XIDUSN || '.' || XIDSLT || '.' ||  XIDSQN) AS XID, SQL_REDO,
        SQL_UNDO FROM V$LOGMNR_CONTENTS WHERE username IN ('HR', 'OE');
;
USR    XID          SQL_REDO                          SQL_UNDO
----   ---------    --------------------------------- --------------------------------

OE     1.1.1484   set transaction read write;

OE     1.1.1484   update "OE"."PRODUCT_INFORMATION"  update "OE"."PRODUCT_INFORMATION"
                  set "WARRANTY_PERIOD" =            set "WARRANTY_PERIOD" =
                  TO_YMINTERVAL('+05-00') where      TO_YMINTERVAL('+01-00') where
                  "PRODUCT_ID" = '1799' and          "PRODUCT_ID" = '1799' and
                  "WARRANTY_PERIOD" =                "WARRANTY_PERIOD" =
                  TO_YMINTERVAL('+01-00') and        TO_YMINTERVAL('+05-00') and
                  ROWID = 'AAAHTKAABAAAY9mAAB';      ROWID = 'AAAHTKAABAAAY9mAAB';

OE     1.1.1484   update "OE"."PRODUCT_INFORMATION"  update "OE"."PRODUCT_INFORMATION"
                  set "WARRANTY_PERIOD" =            set "WARRANTY_PERIOD" =
                  TO_YMINTERVAL('+05-00') where      TO_YMINTERVAL('+01-00') where
                  "PRODUCT_ID" = '1801' and          "PRODUCT_ID" = '1801' and
```

```
                "WARRANTY_PERIOD" =                "WARRANTY_PERIOD" =
                TO_YMINTERVAL('+01-00') and        TO_YMINTERVAL('+05-00') and
                ROWID = 'AAAHTKAABAAAY9mAAC';      ROWID ='AAAHTKAABAAAY9mAAC';

OE     1.1.1484   commit;

HR     1.11.1476  set transaction read write;

HR     1.11.1476  insert into "HR"."EMPLOYEES"(     delete from "HR"."EMPLOYEES"
                  "EMPLOYEE_ID","FIRST_NAME",       where "EMPLOYEE_ID" = '306'
                  "LAST_NAME","EMAIL",              and "FIRST_NAME" = 'Nandini'
                  "PHONE_NUMBER","HIRE_DATE",       and "LAST_NAME" = 'Shastry'
                  "JOB_ID","SALARY",                and "EMAIL" = 'NSHASTRY'
                  "COMMISSION_PCT","MANAGER_ID",    and "PHONE_NUMBER" = '1234567890'
                  "DEPARTMENT_ID") values           and "HIRE_DATE" = TO_DATE('10-JAN-2003
                  ('306','Nandini','Shastry',       13:34:43', 'dd-mon-yyyy hh24:mi:ss')
                  'NSHASTRY', '1234567890',         and "JOB_ID" = 'HR_REP' and
                  TO_DATE('10-jan-2003 13:34:43',   "SALARY" = '120000' and
                  'dd-mon-yyy hh24:mi:ss'),         "COMMISSION_PCT" = '.05' and
                  'HR_REP','120000', '.05',         "DEPARTMENT_ID" = '10' and
                  '105','10');                      ROWID = 'AAAHSkAABAAAY6rAAO';

HR     1.11.1476  insert into "HR"."EMPLOYEES"(     delete from "HR"."EMPLOYEES"
                  "EMPLOYEE_ID","FIRST_NAME",       "EMPLOYEE_ID" = '307' and
                  "LAST_NAME","EMAIL",              "FIRST_NAME" = 'John' and
                  "PHONE_NUMBER","HIRE_DATE",       "LAST_NAME" = 'Silver' and
                  "JOB_ID","SALARY",                "EMAIL" = 'JSILVER' and
                  "COMMISSION_PCT","MANAGER_ID",    "PHONE_NUMBER" = '5551112222'
                  "DEPARTMENT_ID") values           and "HIRE_DATE" = TO_DATE('10-jan-2003
                  ('307','John','Silver',           13:41:03', 'dd-mon-yyyy hh24:mi:ss')
                   'JSILVER', '5551112222',         and "JOB_ID" ='105' and "DEPARTMENT_ID"
                  TO_DATE('10-jan-2003 13:41:03',   = '50' and ROWID = 'AAAHSkAABAAAY6rAAP';
                  'dd-mon-yyyy hh24:mi:ss'),
                  'SH_CLERK','110000', '.05',
                  '105','50');

HR     1.11.1476  commit;
```

Step 5 End the LogMiner session.

```
SQL> EXECUTE DBMS_LOGMNR.END_LOGMNR();
```

Example 3: Formatting the Reconstructed SQL

As shown in Example 2: Grouping DML Statements into Committed Transactions on page 19-52, using the COMMITTED_DATA_ONLY option with the dictionary in the online redo log file is an easy way to focus on committed transactions. However, one aspect remains that makes visual inspection difficult: the association between the column names and their respective values in an INSERT statement are not apparent. This can be addressed by specifying the PRINT_PRETTY_SQL option. Note that specifying this option will make some of the reconstructed SQL statements nonexecutable.

Step 1 Determine which redo log file was most recently archived.

This example assumes that you know that you want to mine the redo log file that was most recently archived.

```
SQL> SELECT NAME FROM V$ARCHIVED_LOG
       WHERE FIRST_TIME = (SELECT MAX(FIRST_TIME) FROM V$ARCHIVED_LOG);

NAME
-----------------------------------------
/usr/oracle/data/db1arch_1_16_482701534.dbf
```

Step 2 Specify the list of redo log files to be analyzed.

Specify the redo log file that was returned by the query in Step 1. The list will consist of one redo log file.

```
SQL> EXECUTE DBMS_LOGMNR.ADD_LOGFILE( -
       LOGFILENAME => '/usr/oracle/data/db1arch_1_16_482701534.dbf', -
       OPTIONS => DBMS_LOGMNR.NEW);
```

Step 3 Start LogMiner.

Start LogMiner by specifying the dictionary to use and the COMMITTED_DATA_ONLY and PRINT_PRETTY_SQL options.

```
SQL> EXECUTE DBMS_LOGMNR.START_LOGMNR(-
       OPTIONS => DBMS_LOGMNR.DICT_FROM_ONLINE_CATALOG + -
                  DBMS_LOGMNR.COMMITTED_DATA_ONLY + -
                  DBMS_LOGMNR.PRINT_PRETTY_SQL);
```

The DBMS_LOGMNR.PRINT_PRETTY_SQL option changes only the format of the reconstructed SQL, and therefore is useful for generating reports for visual inspection.

Step 4 Query the V$LOGMNR_CONTENTS view for SQL_REDO statements.

```
SQL> SELECT username AS USR, (XIDUSN || '.' || XIDSLT || '.' ||  XIDSQN) AS XID, SQL_REDO
       FROM V$LOGMNR_CONTENTS;

USR    XID          SQL_REDO
----   ---------  --------------------------------------------------

OE     1.1.1484   set transaction read write;

OE     1.1.1484   update "OE"."PRODUCT_INFORMATION"
                    set
                      "WARRANTY_PERIOD" = TO_YMINTERVAL('+05-00')
                    where
                      "PRODUCT_ID" = '1799' and
                      "WARRANTY_PERIOD" = TO_YMINTERVAL('+01-00') and
                      ROWID = 'AAAHTKAABAAAY9mAAB';

OE     1.1.1484   update "OE"."PRODUCT_INFORMATION"
                    set
                      "WARRANTY_PERIOD" = TO_YMINTERVAL('+05-00')
                    where
                      "PRODUCT_ID" = '1801' and
                      "WARRANTY_PERIOD" = TO_YMINTERVAL('+01-00') and
                      ROWID = 'AAAHTKAABAAAY9mAAC';

OE     1.1.1484   commit;

HR     1.11.1476  set transaction read write;

HR     1.11.1476  insert into "HR"."EMPLOYEES"
                   values
                     "EMPLOYEE_ID" = 306,
                     "FIRST_NAME" = 'Nandini',
                     "LAST_NAME" = 'Shastry',
                     "EMAIL" = 'NSHASTRY',
                     "PHONE_NUMBER" = '1234567890',
                     "HIRE_DATE" = TO_DATE('10-jan-2003 13:34:43',
                     'dd-mon-yyyy hh24:mi:ss',
                     "JOB_ID" = 'HR_REP',
                     "SALARY" = 120000,
                     "COMMISSION_PCT" = .05,
                     "MANAGER_ID" = 105,
                     "DEPARTMENT_ID" = 10;

HR     1.11.1476   insert into "HR"."EMPLOYEES"
```

```
                    values
                       "EMPLOYEE_ID" = 307,
                       "FIRST_NAME" = 'John',
                       "LAST_NAME" = 'Silver',
                       "EMAIL" = 'JSILVER',
                       "PHONE_NUMBER" = '5551112222',
                       "HIRE_DATE" = TO_DATE('10-jan-2003 13:41:03',
                       'dd-mon-yyyy hh24:mi:ss'),
                       "JOB_ID" = 'SH_CLERK',
                       "SALARY" = 110000,
                       "COMMISSION_PCT" = .05,
                       "MANAGER_ID" = 105,
                       "DEPARTMENT_ID" = 50;
HR     1.11.1476    commit;
```

Step 5 Query the V$LOGMNR_CONTENTS view for reconstructed SQL_UNDO statements.

```
SQL> SELECT username AS USR, (XIDUSN || '.' || XIDSLT || '.' ||  XIDSQN) AS XID, SQL_UNDO
       FROM V$LOGMNR_CONTENTS;

USR   XID        SQL_UNDO
----   ---------  ----------------------------------------------------


OE     1.1.1484   set transaction read write;

OE     1.1.1484   update "OE"."PRODUCT_INFORMATION"
                    set
                      "WARRANTY_PERIOD" = TO_YMINTERVAL('+01-00')
                    where
                      "PRODUCT_ID" = '1799' and
                      "WARRANTY_PERIOD" = TO_YMINTERVAL('+05-00') and
                      ROWID = 'AAAHTKAABAAAY9mAAB';

OE     1.1.1484   update "OE"."PRODUCT_INFORMATION"
                    set
                      "WARRANTY_PERIOD" = TO_YMINTERVAL('+01-00')
                    where
                      "PRODUCT_ID" = '1801' and
                      "WARRANTY_PERIOD" = TO_YMINTERVAL('+05-00') and
                      ROWID = 'AAAHTKAABAAAY9mAAC';

OE     1.1.1484   commit;
```

```
HR     1.11.1476  set transaction read write;

HR     1.11.1476  delete from "HR"."EMPLOYEES"
                  where
                     "EMPLOYEE_ID" = 306 and
                     "FIRST_NAME" = 'Nandini' and
                     "LAST_NAME" = 'Shastry' and
                     "EMAIL" = 'NSHASTRY' and
                     "PHONE_NUMBER" = '1234567890' and
                     "HIRE_DATE" = TO_DATE('10-jan-2003 13:34:43',
                     'dd-mon-yyyy hh24:mi:ss') and
                     "JOB_ID" = 'HR_REP' and
                     "SALARY" = 120000 and
                     "COMMISSION_PCT" = .05 and
                     "MANAGER_ID" = 105 and
                     "DEPARTMENT_ID" = 10 and
                     ROWID = 'AAAHSkAABAAAY6rAAO';

HR     1.11.1476   delete from "HR"."EMPLOYEES"
                   where
                       "EMPLOYEE_ID" = 307 and
                       "FIRST_NAME" = 'John' and
                       "LAST_NAME" = 'Silver' and
                       "EMAIL" = 'JSILVER' and
                       "PHONE_NUMBER" = '555122122' and
                       "HIRE_DATE" = TO_DATE('10-jan-2003 13:41:03',
                       'dd-mon-yyyy hh24:mi:ss') and
                       "JOB_ID" = 'SH_CLERK' and
                       "SALARY" = 110000 and
                       "COMMISSION_PCT" = .05 and
                       "MANAGER_ID" = 105 and
                       "DEPARTMENT_ID" = 50 and
                       ROWID = 'AAAHSkAABAAAY6rAAP';
HR     1.11.1476    commit;
```

Step 6 End the LogMiner session.

```
SQL> EXECUTE DBMS_LOGMNR.END_LOGMNR();
```

Example 4: Using the LogMiner Dictionary in the Redo Log Files

This example shows how to use the dictionary that has been extracted to the redo log files. When you use the dictionary in the online catalog, you must mine the redo

log files in the same database that generated them. Using the dictionary contained in the redo log files enables you to mine redo log files in a different database.

Step 1 Determine which redo log file was most recently archived by the database.

This example assumes that you know that you want to mine the redo log file that was most recently archived.

```
SQL> SELECT NAME, SEQUENCE# FROM V$ARCHIVED_LOG
       WHERE FIRST_TIME = (SELECT MAX(FIRST_TIME) FROM V$ARCHIVED_LOG);

NAME                                          SEQUENCE#
--------------------------------------------  --------------
/usr/oracle/data/db1arch_1_210_482701534.dbf   210
```

Step 2 Find the redo log files containing the dictionary.

The dictionary may be contained in more than one redo log file. Therefore, you need to determine which redo log files contain the start and end of the dictionary. Query the V$ARCHIVED_LOG view, as follows:

1. Find a redo log file that contains the end of the dictionary extract. This redo log file must have been created before the redo log file that you want to analyze, but should be as recent as possible.

   ```
   SQL> SELECT NAME, SEQUENCE#, DICTIONARY_BEGIN d_beg, DICTIONARY_END d_end
          FROM V$ARCHIVED_LOG
          WHERE SEQUENCE# = (SELECT MAX (SEQUENCE#) FROM V$ARCHIVED_LOG
          WHERE DICTIONARY_END = 'YES' and SEQUENCE# <= 210);

   NAME                                          SEQUENCE#    D_BEG  D_END
   --------------------------------------------  ----------   -----  ------
   /usr/oracle/data/db1arch_1_208_482701534.dbf  208          NO     YES
   ```

2. Find the redo log file that contains the start of the data dictionary extract that matches the end of the dictionary found in the previous step:

   ```
   SQL> SELECT NAME, SEQUENCE#, DICTIONARY_BEGIN d_beg, DICTIONARY_END d_end
          FROM V$ARCHIVED_LOG
          WHERE SEQUENCE# = (SELECT MAX (SEQUENCE#) FROM V$ARCHIVED_LOG
          WHERE DICTIONARY_BEGIN = 'YES' and SEQUENCE# <= 208);

   NAME                                          SEQUENCE#    D_BEG  D_END
   --------------------------------------------  ----------   -----  ------
   /usr/oracle/data/db1arch_1_207_482701534.dbf  207          YES     NO
   ```

3. Specify the list of the redo log files of interest. Add the redo log files that contain the start and end of the dictionary and the redo log file that you want to analyze. You can add the redo log files in any order.

```
SQL> EXECUTE DBMS_LOGMNR.ADD_LOGFILE(-
       LOGFILENAME => '/usr/oracle/data/db1arch_1_210_482701534.dbf', -
           OPTIONS => DBMS_LOGMNR.NEW);
SQL> EXECUTE DBMS_LOGMNR.ADD_LOGFILE(-
       LOGFILENAME => '/usr/oracle/data/db1arch_1_208_482701534.dbf');
SQL> EXECUTE DBMS_LOGMNR.ADD_LOGFILE(-
       LOGFILENAME => '/usr/oracle/data/db1arch_1_207_482701534.dbf');
```

4. Query the V$LOGMNR_LOGS view to display the list of redo log files to be analyzed, including their timestamps.

   In the output, LogMiner flags a missing redo log file. LogMiner lets you proceed with mining, provided that you do not specify an option that requires the missing redo log file for proper functioning.

```
SQL> SELECT FILENAME AS name, LOW_TIME, HIGH_TIME FROM V$LOGMNR_LOGS;
 NAME                                LOW_TIME              HIGH_TIME
-------------------------------------  --------------------  --------------------
/usr/data/db1arch_1_207_482701534.dbf   10-jan-2003 12:01:34  10-jan-2003 13:32:46

/usr/data/db1arch_1_208_482701534.dbf   10-jan-2003 13:32:46  10-jan-2003 15:57:03

Missing logfile(s) for thread number 1, 10-jan-2003 15:57:03  10-jan-2003 15:59:53
sequence number(s) 209 to 209

/usr/data/db1arch_1_210_482701534.dbf   10-jan-2003 15:59:53  10-jan-2003 16:07:41
```

Step 3 Start LogMiner.

Start LogMiner by specifying the dictionary to use and the COMMITTED_DATA_ONLY and PRINT_PRETTY_SQL options.

```
SQL> EXECUTE DBMS_LOGMNR.START_LOGMNR(-
       OPTIONS => DBMS_LOGMNR.DICT_FROM_REDO_LOGS + -
                  DBMS_LOGMNR.COMMITTED_DATA_ONLY + -
                  DBMS_LOGMNR.PRINT_PRETTY_SQL);
```

Step 4 Query the V$LOGMNR_CONTENTS view.

To reduce the number of rows returned by the query, exclude from the query all DML statements done in the `sys` or `system` schemas. (This query specifies a timestamp to exclude transactions that were involved in the dictionary extraction.)

The output shows three transactions: two DDL transactions and one DML transaction. The DDL transactions, 1.2.1594 and 1.18.1602, create the table `oe.product_tracking` and create a trigger on table `oe.product_information`, respectively. In both transactions, the DML statements done to the system tables (tables owned by `sys`) are filtered out because of the query predicate.

The DML transaction, 1.9.1598, updates the `oe.product_information` table. The update operation in this transaction is fully translated. However, the query output also contains some untranslated reconstructed SQL statements. Most likely, these statements were done on the `oe.product_tracking` table that was created after the data dictionary was extracted to the redo log files.

(The next example shows how to run LogMiner with the `DDL_DICT_TRACKING` option so that all SQL statements are fully translated; no binary data is returned.)

```
SQL> SELECT USERNAME AS usr, SQL_REDO FROM V$LOGMNR_CONTENTS
       WHERE SEG_OWNER IS NULL OR SEG_OWNER NOT IN ('SYS', 'SYSTEM') AND
       TIMESTAMP > '10-jan-2003 15:59:53';

USR             XID         SQL_REDO
---             --------    -----------------------------------
SYS             1.2.1594    set transaction read write;
SYS             1.2.1594    create table oe.product_tracking (product_id number not null,
                            modified_time date,
                            old_list_price number(8,2),
                            old_warranty_period interval year(2) to month);
SYS             1.2.1594    commit;

SYS             1.18.1602   set transaction read write;
SYS             1.18.1602   create or replace trigger oe.product_tracking_trigger
                            before update on oe.product_information
                            for each row
                            when (new.list_price <> old.list_price or
                                  new.warranty_period <> old.warranty_period)
                            declare
                            begin
                            insert into oe.product_tracking values
                               (:old.product_id, sysdate,
                                :old.list_price, :old.warranty_period);
                            end;
```

```
SYS             1.18.1602   commit;

OE              1.9.1598    update "OE"."PRODUCT_INFORMATION"
                              set
                                "WARRANTY_PERIOD" = TO_YMINTERVAL('+08-00'),
                                "LIST_PRICE" = 100
                              where
                                "PRODUCT_ID" = 1729 and
                                "WARRANTY_PERIOD" = TO_YMINTERVAL('+05-00') and
                                "LIST_PRICE" = 80 and
                                ROWID = 'AAAHTKAABAAAY9yAAA';

OE              1.9.1598    insert into "UNKNOWN"."OBJ# 33415"
                              values
                                "COL 1" = HEXTORAW('c2121e'),
                                "COL 2" = HEXTORAW('7867010d110804'),
                                "COL 3" = HEXTORAW('c151'),
                                "COL 4" = HEXTORAW('800000053c');

OE              1.9.1598    update "OE"."PRODUCT_INFORMATION"
                              set
                                "WARRANTY_PERIOD" = TO_YMINTERVAL('+08-00'),
                                "LIST_PRICE" = 92
                              where
                                "PRODUCT_ID" = 2340 and
                                "WARRANTY_PERIOD" = TO_YMINTERVAL('+05-00') and
                                "LIST_PRICE" = 72 and
                                ROWID = 'AAAHTKAABAAAY9zAAA';

OE              1.9.1598    insert into "UNKNOWN"."OBJ# 33415"
                              values
                                "COL 1" = HEXTORAW('c21829'),
                                "COL 2" = HEXTORAW('7867010d110808'),
                                "COL 3" = HEXTORAW('c149'),
                                "COL 4" = HEXTORAW('800000053c');

OE              1.9.1598     commit;
```

Step 5 Issue additional queries, if desired.

Display all the DML statements that were executed as part of the CREATE TABLE DDL statement. This includes statements executed by users and internally by Oracle.

Note: If you choose to reapply statements displayed by a query such as the one shown here, reapply DDL statements only. Do not reapply DML statements that were executed internally by Oracle, or you risk corrupting your database. In the following output, the only statement that you should use in a reapply operation is the `CREATE TABLE OE.PRODUCT_TRACKING` statement.

```
SQL> SELECT SQL_REDO FROM V$LOGMNR_CONTENTS
       WHERE XIDUSN  = 1 and XIDSLT = 2 and XIDSQN = 1594;

SQL_REDO
--------------------------------------------------------------------------------
set transaction read write;

insert into "SYS"."OBJ$"
 values
    "OBJ#" = 33415,
    "DATAOBJ#" = 33415,
    "OWNER#" = 37,
    "NAME" = 'PRODUCT_TRACKING',
    "NAMESPACE" = 1,
    "SUBNAME" IS NULL,
    "TYPE#" = 2,
    "CTIME" = TO_DATE('13-jan-2003 14:01:03', 'dd-mon-yyyy hh24:mi:ss'),
    "MTIME" = TO_DATE('13-jan-2003 14:01:03', 'dd-mon-yyyy hh24:mi:ss'),
    "STIME" = TO_DATE('13-jan-2003 14:01:03', 'dd-mon-yyyy hh24:mi:ss'),
    "STATUS" = 1,
    "REMOTEOWNER" IS NULL,
    "LINKNAME" IS NULL,
    "FLAGS" = 0,
    "OID$" IS NULL,
    "SPARE1" = 6,
    "SPARE2" = 1,
    "SPARE3" IS NULL,
    "SPARE4" IS NULL,
    "SPARE5" IS NULL,
    "SPARE6" IS NULL;

insert into "SYS"."TAB$"
 values
    "OBJ#" = 33415,
    "DATAOBJ#" = 33415,
```

```
    "TS#" = 0,
    "FILE#" = 1,
    "BLOCK#" = 121034,
    "BOBJ#" IS NULL,
    "TAB#" IS NULL,
    "COLS" = 5,
    "CLUCOLS" IS NULL,
    "PCTFREE$" = 10,
    "PCTUSED$" = 40,
    "INITRANS" = 1,
    "MAXTRANS" = 255,
    "FLAGS" = 1,
    "AUDIT$" = '--------------------------------------',
    "ROWCNT" IS NULL,
    "BLKCNT" IS NULL,
    "EMPCNT" IS NULL,
    "AVGSPC" IS NULL,
    "CHNCNT" IS NULL,
    "AVGRLN" IS NULL,
    "AVGSPC_FLB" IS NULL,
    "FLBCNT" IS NULL,
    "ANALYZETIME" IS NULL,
    "SAMPLESIZE" IS NULL,
    "DEGREE" IS NULL,
    "INSTANCES" IS NULL,
    "INTCOLS" = 5,
    "KERNELCOLS" = 5,
    "PROPERTY" = 536870912,
    "TRIGFLAG" = 0,
    "SPARE1" = 178,
    "SPARE2" IS NULL,
    "SPARE3" IS NULL,
    "SPARE4" IS NULL,
    "SPARE5" IS NULL,
    "SPARE6" = TO_DATE('13-jan-2003 14:01:05', 'dd-mon-yyyy hh24:mi:ss'),

insert into "SYS"."COL$"
 values
    "OBJ#" = 33415,
    "COL#" = 1,
    "SEGCOL#" = 1,
    "SEGCOLLENGTH" = 22,
    "OFFSET" = 0,
    "NAME" = 'PRODUCT_ID',
    "TYPE#" = 2,
```

```
    "LENGTH" = 22,
    "FIXEDSTORAGE" = 0,
    "PRECISION#" IS NULL,
    "SCALE" IS NULL,
    "NULL$" = 1,
    "DEFLENGTH" IS NULL,
    "SPARE6" IS NULL,
    "INTCOL#" = 1,
    "PROPERTY" = 0,
    "CHARSETID" = 0,
    "CHARSETFORM" = 0,
    "SPARE1" = 0,
    "SPARE2" = 0,
    "SPARE3" = 0,
    "SPARE4" IS NULL,
    "SPARE5" IS NULL,
    "DEFAULT$" IS NULL;

insert into "SYS"."COL$"
 values
    "OBJ#" = 33415,
    "COL#" = 2,
    "SEGCOL#" = 2,
    "SEGCOLLENGTH" = 7,
    "OFFSET" = 0,
    "NAME" = 'MODIFIED_TIME',
    "TYPE#" = 12,
    "LENGTH" = 7,
    "FIXEDSTORAGE" = 0,
    "PRECISION#" IS NULL,
    "SCALE" IS NULL,
    "NULL$" = 0,
    "DEFLENGTH" IS NULL,
    "SPARE6" IS NULL,
    "INTCOL#" = 2,
    "PROPERTY" = 0,
    "CHARSETID" = 0,
    "CHARSETFORM" = 0,
    "SPARE1" = 0,
    "SPARE2" = 0,
    "SPARE3" = 0,
    "SPARE4" IS NULL,
    "SPARE5" IS NULL,
    "DEFAULT$" IS NULL;
```

```
insert into "SYS"."COL$"
 values
    "OBJ#" = 33415,
    "COL#" = 3,
    "SEGCOL#" = 3,
    "SEGCOLLENGTH" = 22,
    "OFFSET" = 0,
    "NAME" = 'OLD_LIST_PRICE',
    "TYPE#" = 2,
    "LENGTH" = 22,
    "FIXEDSTORAGE" = 0,
    "PRECISION#" = 8,
    "SCALE" = 2,
    "NULL$" = 0,
    "DEFLENGTH" IS NULL,
    "SPARE6" IS NULL,
    "INTCOL#" = 3,
    "PROPERTY" = 0,
    "CHARSETID" = 0,
    "CHARSETFORM" = 0,
    "SPARE1" = 0,
    "SPARE2" = 0,
    "SPARE3" = 0,
    "SPARE4" IS NULL,
    "SPARE5" IS NULL,
    "DEFAULT$" IS NULL;

insert into "SYS"."COL$"
 values
    "OBJ#" = 33415,
    "COL#" = 4,
    "SEGCOL#" = 4,
    "SEGCOLLENGTH" = 5,
    "OFFSET" = 0,
    "NAME" = 'OLD_WARRANTY_PERIOD',
    "TYPE#" = 182,
    "LENGTH" = 5,
    "FIXEDSTORAGE" = 0,
    "PRECISION#" = 2,
    "SCALE" = 0,
    "NULL$" = 0,
    "DEFLENGTH" IS NULL,
    "SPARE6" IS NULL,
    "INTCOL#" = 4,
    "PROPERTY" = 0,
```

```
    "CHARSETID" = 0,
    "CHARSETFORM" = 0,
    "SPARE1" = 0,
    "SPARE2" = 2,
    "SPARE3" = 0,
    "SPARE4" IS NULL,
    "SPARE5" IS NULL,
    "DEFAULT$" IS NULL;

insert into "SYS"."CCOL$"
 values
    "OBJ#" = 33415,
    "CON#" = 2090,
    "COL#" = 1,
    "POS#" IS NULL,
    "INTCOL#" = 1,
    "SPARE1" = 0,
    "SPARE2" IS NULL,
    "SPARE3" IS NULL,
    "SPARE4" IS NULL,
    "SPARE5" IS NULL,
    "SPARE6" IS NULL;

insert into "SYS"."CDEF$"
 values
    "OBJ#" = 33415,
    "CON#" = 2090,
    "COLS" = 1,
    "TYPE#" = 7,
    "ROBJ#" IS NULL,
    "RCON#" IS NULL,
    "RRULES" IS NULL,
    "MATCH#" IS NULL,
    "REFACT" IS NULL,
    "ENABLED" = 1,
    "CONDLENGTH" = 24,
    "SPARE6" IS NULL,
    "INTCOLS" = 1,
    "MTIME" = TO_DATE('13-jan-2003 14:01:08', 'dd-mon-yyyy hh24:mi:ss'),
    "DEFER" = 12,
    "SPARE1" = 6,
    "SPARE2" IS NULL,
    "SPARE3" IS NULL,
    "SPARE4" IS NULL,
    "SPARE5" IS NULL,
```

```
    "CONDITION" = '"PRODUCT_ID" IS NOT NULL';

create table oe.product_tracking (product_id number not null,
  modified_time date,
  old_product_description varchar2(2000),
  old_list_price number(8,2),
  old_warranty_period interval year(2) to month);

update "SYS"."SEG$"
  set
    "TYPE#" = 5,
    "BLOCKS" = 5,
    "EXTENTS" = 1,
    "INIEXTS" = 5,
    "MINEXTS" = 1,
    "MAXEXTS" = 121,
    "EXTSIZE" = 5,
    "EXTPCT" = 50,
    "USER#" = 37,
    "LISTS" = 0,
    "GROUPS" = 0,
    "CACHEHINT" = 0,
    "HWMINCR" = 33415,
    "SPARE1" = 1024
  where
    "TS#" = 0 and
    "FILE#" = 1 and
    "BLOCK#" = 121034 and
    "TYPE#" = 3 and
    "BLOCKS" = 5 and
    "EXTENTS" = 1 and
    "INIEXTS" = 5 and
    "MINEXTS" = 1 and
    "MAXEXTS" = 121 and
    "EXTSIZE" = 5 and
    "EXTPCT" = 50 and
    "USER#" = 37 and
    "LISTS" = 0 and
    "GROUPS" = 0 and
    "BITMAPRANGES" = 0 and
    "CACHEHINT" = 0 and
    "SCANHINT" = 0 and
    "HWMINCR" = 33415 and
    "SPARE1" = 1024 and
    "SPARE2" IS NULL and
```

```
    ROWID = 'AAAAAIAABAAAdMOAAB';

insert into "SYS"."CON$"
 values
    "OWNER#" = 37,
    "NAME" = 'SYS_C002090',
    "CON#" = 2090,
    "SPARE1" IS NULL,
    "SPARE2" IS NULL,
    "SPARE3" IS NULL,
    "SPARE4" IS NULL,
    "SPARE5" IS NULL,
    "SPARE6" IS NULL;

commit;
```

Step 6 End the LogMiner session.

```
SQL> EXECUTE DBMS_LOGMNR.END_LOGMNR();
```

Example 5: Tracking DDL Statements in the Internal Dictionary

By using the DBMS_LOGMNR.DDL_DICT_TRACKING option, this example ensures that the LogMiner internal dictionary is updated with the DDL statements encountered in the redo log files.

Step 1 Determine which redo log file was most recently archived by the database.

This example assumes that you know that you want to mine the redo log file that was most recently archived.

```
SQL> SELECT NAME, SEQUENCE# FROM V$ARCHIVED_LOG
       WHERE FIRST_TIME = (SELECT MAX(FIRST_TIME) FROM V$ARCHIVED_LOG);

NAME                                          SEQUENCE#
--------------------------------------------  --------------
/usr/oracle/data/db1arch_1_210_482701534.dbf   210
```

Step 2 Find the dictionary in the redo log files.

Because the dictionary may be contained in more than one redo log file, you need to determine which redo log files contain the start and end of the data dictionary. Query the V$ARCHIVED_LOG view, as follows:

1. Find a redo log that contains the end of the data dictionary extract. This redo log file must have been created before the redo log files that you want to analyze, but should be as recent as possible.

```
SQL> SELECT NAME, SEQUENCE#, DICTIONARY_BEGIN d_beg, DICTIONARY_END d_end
       FROM V$ARCHIVED_LOG
       WHERE SEQUENCE# = (SELECT MAX (SEQUENCE#) FROM V$ARCHIVED_LOG
       WHERE DICTIONARY_END = 'YES' and SEQUENCE# < 210);


NAME                                          SEQUENCE#    D_BEG  D_END
--------------------------------------------  ----------   -----  ------
/usr/oracle/data/db1arch_1_208_482701534.dbf  208          NO     YES
```

2. Find the redo log file that contains the start of the data dictionary extract that matches the end of the dictionary found by the previous SQL statement:

```
SQL> SELECT NAME, SEQUENCE#, DICTIONARY_BEGIN d_beg, DICTIONARY_END d_end
       FROM V$ARCHIVED_LOG
       WHERE SEQUENCE# = (SELECT MAX (SEQUENCE#) FROM V$ARCHIVED_LOG
       WHERE DICTIONARY_BEGIN = 'YES' and SEQUENCE# <= 208);

NAME                                          SEQUENCE#    D_BEG  D_END
--------------------------------------------  ----------   -----  ------
/usr/oracle/data/db1arch_1_208_482701534.dbf  207          YES     NO
```

Step 3 Make sure you have a complete list of redo log files.

To successfully apply DDL statements encountered in the redo log files, ensure that all files are included in the list of redo log files to mine. The missing log file corresponding to sequence# 209 must be included in the list. Determine the names of the redo log files that you need to add to the list by issuing the following query:

```
SQL> SELECT NAME FROM V$ARCHIVED_LOG
       WHERE SEQUENCE# >= 207 AND SEQUENCE# <= 210
       ORDER BY SEQUENCE# ASC;

NAME
--------------------------------------------
/usr/oracle/data/db1arch_1_207_482701534.dbf
/usr/oracle/data/db1arch_1_208_482701534.dbf
/usr/oracle/data/db1arch_1_209_482701534.dbf
/usr/oracle/data/db1arch_1_210_482701534.dbf
```

Step 4 Specify the list of the redo log files of interest.

Include the redo log files that contain the beginning and end of the dictionary, the redo log file that you want to mine, and any redo log files required to create a list without gaps. You can add the redo log files in any order.

```
SQL> EXECUTE DBMS_LOGMNR.ADD_LOGFILE(-
       LOGFILENAME => '/usr/oracle/data/db1arch_1_210_482701534.dbf', -
           OPTIONS => DBMS_LOGMNR.NEW);
SQL> EXECUTE DBMS_LOGMNR.ADD_LOGFILE(-
       LOGFILENAME => '/usr/oracle/data/db1arch_1_209_482701534.dbf');
SQL> EXECUTE DBMS_LOGMNR.ADD_LOGFILE(-
       LOGFILENAME => '/usr/oracle/data/db1arch_1_208_482701534.dbf');
SQL> EXECUTE DBMS_LOGMNR.ADD_LOGFILE(-
       LOGFILENAME => '/usr/oracle/data/db1arch_1_207_482701534.dbf');
```

Step 5 Start LogMiner.

Start LogMiner by specifying the dictionary to use and the DDL_DICT_TRACKING, COMMITTED_DATA_ONLY, and PRINT_PRETTY_SQL options.

```
SQL> EXECUTE DBMS_LOGMNR.START_LOGMNR(-
       OPTIONS => DBMS_LOGMNR.DICT_FROM_REDO_LOGS + -
                  DBMS_LOGMNR.DDL_DICT_TRACKING + -
                  DBMS_LOGMNR.COMMITTED_DATA_ONLY + -
                  DBMS_LOGMNR.PRINT_PRETTY_SQL);
```

Step 6 Query the V$LOGMNR_CONTENTS view.

To reduce the number of rows returned, exclude from the query all DML statements done in the sys or system schemas. (This query specifies a timestamp to exclude transactions that were involved in the dictionary extraction.)

The query returns all the reconstructed SQL statements correctly translated and the insert operations on the oe.product_tracking table that occurred because of the trigger execution.

```
SQL> SELECT USERNAME AS usr,(XIDUSN || '.' || XIDSLT || '.' || XIDSQN) as XID, SQL_REDO FROM
       V$LOGMNR_CONTENTS
       WHERE SEG_OWNER IS NULL OR SEG_OWNER NOT IN ('SYS', 'SYSTEM') AND
       TIMESTAMP > '10-jan-2003 15:59:53';

USR             XID         SQL_REDO
-----------     --------    ---------------------------------
SYS             1.2.1594    set transaction read write;
SYS             1.2.1594    create table oe.product_tracking (product_id number not null,
                            modified_time date,
```

```
                        old_list_price number(8,2),
                        old_warranty_period interval year(2) to month);
SYS             1.2.1594    commit;

SYS             1.18.1602   set transaction read write;
SYS             1.18.1602   create or replace trigger oe.product_tracking_trigger
                            before update on oe.product_information
                            for each row
                            when (new.list_price <> old.list_price or
                                  new.warranty_period <> old.warranty_period)
                            declare
                            begin
                            insert into oe.product_tracking values
                               (:old.product_id, sysdate,
                                :old.list_price, :old.warranty_period);
                            end;
SYS             1.18.1602   commit;

OE              1.9.1598    update "OE"."PRODUCT_INFORMATION"
                              set
                                "WARRANTY_PERIOD" = TO_YMINTERVAL('+08-00'),
                                "LIST_PRICE" = 100
                              where
                                "PRODUCT_ID" = 1729 and
                                "WARRANTY_PERIOD" = TO_YMINTERVAL('+05-00') and
                                "LIST_PRICE" = 80 and
                                ROWID = 'AAAHTKAABAAAY9yAAA';
OE              1.9.1598    insert into "OE"."PRODUCT_TRACKING"
                              values
                                "PRODUCT_ID" = 1729,
                                "MODIFIED_TIME" = TO_DATE('13-jan-2003 16:07:03',
                                'dd-mon-yyyy hh24:mi:ss'),
                                "OLD_LIST_PRICE" = 80,
                                "OLD_WARRANTY_PERIOD" = TO_YMINTERVAL('+05-00');

OE              1.9.1598    update "OE"."PRODUCT_INFORMATION"
                              set
                                "WARRANTY_PERIOD" = TO_YMINTERVAL('+08-00'),
                                "LIST_PRICE" = 92
                              where
                                "PRODUCT_ID" = 2340 and
                                "WARRANTY_PERIOD" = TO_YMINTERVAL('+05-00') and
                                "LIST_PRICE" = 72 and
                                ROWID = 'AAAHTKAABAAAY9zAAA';
```

```
OE              1.9.1598    insert into "OE"."PRODUCT_TRACKING"
                              values
                                "PRODUCT_ID" = 2340,
                                "MODIFIED_TIME" = TO_DATE('13-jan-2003 16:07:07',
                                'dd-mon-yyyy hh24:mi:ss'),
                                "OLD_LIST_PRICE" = 72,
                                "OLD_WARRANTY_PERIOD" = TO_YMINTERVAL('+05-00');

OE              1.9.1598     commit;
```

Step 7 End the LogMiner session.

```
SQL> EXECUTE DBMS_LOGMNR.END_LOGMNR();
```

Example 6: Filtering Output by Time Range

In the previous two examples, rows were filtered by specifying a timestamp-based predicate (timestamp > '10-jan-2003 15:59:53') in the query. However, a more efficient way to filter out redo records based on timestamp values is by specifying the time range in the DBMS_LOGMNR.START_LOGMNR procedure call, as shown in this example.

Step 1 Create a list of redo log files to mine.

Suppose you want to mine redo log files generated since a given time. The following procedure creates a list of redo log files based on a specified time. The subsequent SQL EXECUTE statement calls the procedure and specifies the starting time as 2 p.m. on Jan-13-2003.

```
--
-- my_add_logfiles
-- Add all archived logs generated after a specified start_time.
--
CREATE OR REPLACE PROCEDURE my_add_logfiles (in_start_time  IN DATE) AS
  CURSOR  c_log IS
    SELECT NAME FROM V$ARCHIVED_LOG
      WHERE FIRST_TIME >= in_start_time;

count      pls_integer := 0;
my_option  pls_integer := DBMS_LOGMNR.NEW;

BEGIN
  FOR c_log_rec IN c_log
  LOOP
    DBMS_LOGMNR.ADD_LOGFILE(LOGFILENAME => c_log_rec.name,
                            OPTIONS => my_option);
```

```
    my_option := DBMS_LOGMNR.ADDFILE;
    DBMS_OUTPUT.PUT_LINE('Added logfile ' || c_log_rec.name);
  END LOOP;
END;
/

SQL> EXECUTE my_add_logfiles(in_start_time => '13-jan-2003 14:00:00');
```

Step 2 Query the V$LOGMNR_LOGS to see the list of redo log files.

This example includes the size of the redo log files in the output.

```
SQL> SELECT FILENAME name, LOW_TIME start_time, FILESIZE bytes
       FROM V$LOGMNR_LOGS;

NAME                                START_TIME            BYTES
----------------------------------- --------------------  ----------------
/usr/orcl/arch1_310_482932022.dbf    13-jan-2003 14:02:35  23683584
/usr/orcl/arch1_311_482932022.dbf    13-jan-2003 14:56:35  2564096
/usr/orcl/arch1_312_482932022.dbf    13-jan-2003 15:10:43  23683584
/usr/orcl/arch1_313_482932022.dbf    13-jan-2003 15:17:52  23683584
/usr/orcl/arch1_314_482932022.dbf    13-jan-2003 15:23:10  23683584
/usr/orcl/arch1_315_482932022.dbf    13-jan-2003 15:43:22  23683584
/usr/orcl/arch1_316_482932022.dbf    13-jan-2003 16:03:10  23683584
/usr/orcl/arch1_317_482932022.dbf    13-jan-2003 16:33:43  23683584
/usr/orcl/arch1_318_482932022.dbf    13-jan-2003 17:23:10  23683584
```

Step 3 Adjust the list of redo log files.

Suppose you realize that you want to mine just the redo log files generated between 3 p.m. and 4 p.m.

You could use the query predicate (`timestamp > '13-jan-2003 15:00:00' and timestamp < '13-jan-2003 16:00:00'`) to accomplish this. However, the query predicate is evaluated on each row returned by LogMiner, and the internal mining engine does not filter rows based on the query predicate. Thus, although you only wanted to get rows out of redo log files `arch1_311_482932022.dbf` to `arch1_315_482932022.dbf`, your query would result in mining all redo log files registered to the LogMiner session.

Furthermore, although you could use the query predicate and manually remove the redo log files that do not fall inside the time range of interest, the simplest solution is to specify the time range of interest in the `DBMS_LOGMNR.START_LOGMNR` procedure call.

Although this does not change the list of redo log files, LogMiner will mine only those redo log files that fall in the time range specified.

```
SQL> EXECUTE DBMS_LOGMNR.START_LOGMNR(-
       STARTTIME => '13-jan-2003 15:00:00', -
       ENDTIME   => '13-jan-2003 16:00:00', -
       OPTIONS   => DBMS_LOGMNR.DICT_FROM_ONLINE_CATALOG + -
                    DBMS_LOGMNR.COMMITTED_DATA_ONLY + -
                    DBMS_LOGMNR.PRINT_PRETTY_SQL);
```

Step 4 Query the V$LOGMNR_CONTENTS view.

```
SQL> SELECT TIMESTAMP, (XIDUSN || '.' || XIDSLT || '.' || XIDSQN) AS XID,
 SQL_REDO FROM V$LOGMNR_CONTENTS WHERE SEG_OWNER = 'OE';

TIMESTAMP              XID          SQL_REDO
---------------------  -----------  --------------------------------
13-jan-2003 15:29:31   1.17.2376    update "OE"."PRODUCT_INFORMATION"
                                      set
                                        "WARRANTY_PERIOD" = TO_YMINTERVAL('+05-00')
                                      where
                                        "PRODUCT_ID" = 3399 and
                                        "WARRANTY_PERIOD" = TO_YMINTERVAL('+02-00') and
                                        ROWID = 'AAAHTKAABAAAY9TAAE';
13-jan-2003 15:29:34   1.17.2376      insert into "OE"."PRODUCT_TRACKING"
                                        values
                                        "PRODUCT_ID" = 3399,
                                        "MODIFIED_TIME" = TO_DATE('13-jan-2003 15:29:34',
                                        'dd-mon-yyyy hh24:mi:ss'),
                                        "OLD_LIST_PRICE" = 815,
                                        "OLD_WARRANTY_PERIOD" = TO_YMINTERVAL('+02-00');

13-jan-2003 15:52:43   1.15.1756      update "OE"."PRODUCT_INFORMATION"
                                        set
                                          "WARRANTY_PERIOD" = TO_YMINTERVAL('+05-00')
                                        where
                                          "PRODUCT_ID" = 1768 and
                                          "WARRANTY_PERIOD" = TO_YMINTERVAL('+02-00') and
                                          ROWID = 'AAAHTKAABAAAY9UAAB';

13-jan-2003 15:52:43   1.15.1756      insert into "OE"."PRODUCT_TRACKING"
                                        values
                                        "PRODUCT_ID" = 1768,
                                        "MODIFIED_TIME" = TO_DATE('13-jan-2003 16:52:43',
                                        'dd-mon-yyyy hh24:mi:ss'),
                                        "OLD_LIST_PRICE" = 715,
                                        "OLD_WARRANTY_PERIOD" = TO_YMINTERVAL('+02-00');
```

Step 5 End the LogMiner session.

```
SQL> EXECUTE DBMS_LOGMNR.END_LOGMNR();
```

Examples of Mining Without Specifying the List of Redo Log Files Explicitly

The previous set of examples explicitly specified the redo log file or files to be mined. However, if you are mining in the same database that generated the redo log files, then you can mine the appropriate list of redo log files by just specifying the time (or SCN) range of interest. To mine a set of redo log files without explicitly specifying them, use the `DBMS_LOGMNR.CONTINUOUS_MINE` option to the `DBMS_LOGMNR.START_LOGMNR` procedure, and specify either a time range or an SCN range of interest.

This section contains the following list of examples; these examples are best read in sequential order, because each example builds on the example or examples that precede it:

- Example 1: Mining Redo Log Files in a Given Time Range
- Example 2: Mining the Redo Log Files in a Given SCN Range
- Example 3: Using Continuous Mining to Include Future Values in a Query

The SQL output formatting may be different on your display than that shown in these examples.

Example 1: Mining Redo Log Files in a Given Time Range

This example is similar to Example 4: Using the LogMiner Dictionary in the Redo Log Files on page 19-58, except the list of redo log files are not specified explicitly. This example assumes that you want to use the data dictionary extracted to the redo log files.

Step 1 Determine the timestamp of the redo log file that contains the start of the data dictionary.

```
SQL> SELECT NAME, FIRST_TIME FROM V$ARCHIVED_LOG
       WHERE SEQUENCE# = (SELECT MAX(SEQUENCE#) FROM V$ARCHIVED_LOG
       WHERE DICTIONARY_BEGIN = 'YES');

NAME                                          FIRST_TIME
--------------------------------------------  --------------------
/usr/oracle/data/db1arch_1_207_482701534.dbf  10-jan-2003 12:01:34
```

Step 2 Display all the redo log files that have been generated so far.

This step is not required, but is included to demonstrate that the CONTINUOUS_MINE option works as expected, as will be shown in Step 4.

```
SQL> SELECT FILENAME name FROM V$LOGMNR_LOGS
       WHERE LOW_TIME > '10-jan-2003 12:01:34';

NAME
--------------------------------------------
/usr/oracle/data/db1arch_1_207_482701534.dbf
/usr/oracle/data/db1arch_1_208_482701534.dbf
/usr/oracle/data/db1arch_1_209_482701534.dbf
/usr/oracle/data/db1arch_1_210_482701534.dbf
```

Step 3 Start LogMiner.

Start LogMiner by specifying the dictionary to use and the COMMITTED_DATA_ONLY, PRINT_PRETTY_SQL, and CONTINUOUS_MINE options.

```
SQL> EXECUTE DBMS_LOGMNR.START_LOGMNR(-
       STARTTIME => '10-jan-2003 12:01:34', -
         ENDTIME => SYSDATE, -
                OPTIONS => DBMS_LOGMNR.DICT_FROM_REDO_LOGS + -
                    DBMS_LOGMNR.COMMITTED_DATA_ONLY + -
                    DBMS_LOGMNR.PRINT_PRETTY_SQL + -
                    DBMS_LOGMNR.CONTINUOUS_MINE);
```

Step 4 Query the V$LOGMNR_LOGS view.

This step shows that the DBMS_LOGMNR.START_LOGMNR procedure with the CONTINUOUS_MINE option includes all of the redo log files that have been generated so far, as expected. (Compare the output in this step to the output in Step 2.)

```
SQL> SELECT FILENAME name FROM V$LOGMNR_LOGS;

NAME
-------------------------------------------------
/usr/oracle/data/db1arch_1_207_482701534.dbf
/usr/oracle/data/db1arch_1_208_482701534.dbf
/usr/oracle/data/db1arch_1_209_482701534.dbf
/usr/oracle/data/db1arch_1_210_482701534.dbf
```

Step 5 Query the V$LOGMNR_CONTENTS view.

To reduce the number of rows returned by the query, exclude all DML statements done in the sys or system schema. (This query specifies a timestamp to exclude transactions that were involved in the dictionary extraction.)

Note that all reconstructed SQL statements returned by the query are correctly translated.

```
SQL> SELECT USERNAME AS usr,(XIDUSN || '.' || XIDSLT || '.' || XIDSQN) as XID,
       SQL_REDO FROM V$LOGMNR_CONTENTS
       WHERE SEG_OWNER IS NULL OR SEG_OWNER NOT IN ('SYS', 'SYSTEM') AND
       TIMESTAMP > '10-jan-2003 15:59:53';

USR             XID         SQL_REDO
-----------     --------    ----------------------------------
SYS             1.2.1594    set transaction read write;
SYS             1.2.1594    create table oe.product_tracking (product_id number not null,
                            modified_time date,
                            old_list_price number(8,2),
                            old_warranty_period interval year(2) to month);
SYS             1.2.1594    commit;

SYS             1.18.1602   set transaction read write;
SYS             1.18.1602   create or replace trigger oe.product_tracking_trigger
                            before update on oe.product_information
                            for each row
                            when (new.list_price <> old.list_price or
                                  new.warranty_period <> old.warranty_period)
                            declare
                            begin
                            insert into oe.product_tracking values
                               (:old.product_id, sysdate,
                                :old.list_price, :old.warranty_period);
                            end;
SYS             1.18.1602   commit;

OE              1.9.1598    update "OE"."PRODUCT_INFORMATION"
                              set
                                "WARRANTY_PERIOD" = TO_YMINTERVAL('+08-00'),
                                "LIST_PRICE" = 100
                              where
                                "PRODUCT_ID" = 1729 and
                                "WARRANTY_PERIOD" = TO_YMINTERVAL('+05-00') and
                                "LIST_PRICE" = 80 and
                                ROWID = 'AAAHTKAABAAAY9yAAA';
OE              1.9.1598    insert into "OE"."PRODUCT_TRACKING"
                              values
                                "PRODUCT_ID" = 1729,
                                "MODIFIED_TIME" = TO_DATE('13-jan-2003 16:07:03',
```

```
                                'dd-mon-yyyy hh24:mi:ss'),
                                "OLD_LIST_PRICE" = 80,
                                "OLD_WARRANTY_PERIOD" = TO_YMINTERVAL('+05-00');

OE              1.9.1598    update "OE"."PRODUCT_INFORMATION"
                              set
                                "WARRANTY_PERIOD" = TO_YMINTERVAL('+08-00'),
                                "LIST_PRICE" = 92
                              where
                                "PRODUCT_ID" = 2340 and
                                "WARRANTY_PERIOD" = TO_YMINTERVAL('+05-00') and
                                "LIST_PRICE" = 72 and
                                ROWID = 'AAAHTKAABAAAY9zAAA';

OE              1.9.1598    insert into "OE"."PRODUCT_TRACKING"
                              values
                                "PRODUCT_ID" = 2340,
                                "MODIFIED_TIME" = TO_DATE('13-jan-2003 16:07:07',
                                'dd-mon-yyyy hh24:mi:ss'),
                                "OLD_LIST_PRICE" = 72,
                                "OLD_WARRANTY_PERIOD" = TO_YMINTERVAL('+05-00');

OE              1.9.1598     commit;
```

Step 6 End the LogMiner session.

```
SQL> EXECUTE DBMS_LOGMNR.END_LOGMNR();
```

Example 2: Mining the Redo Log Files in a Given SCN Range

This example shows how to specify an SCN range of interest and mine the redo log files that satisfy that range. You can use LogMiner to see all committed DML statements whose effects have not yet been made permanent in the datafiles.

Note that in this example (unlike the other examples) it is not assumed that you have set the NLS_DATE_FORMAT parameter.

Step 1 Determine the SCN of the last checkpoint taken.

```
SQL> SELECT CHECKPOINT_CHANGE#, CURRENT_SCN FROM V$DATABASE;

CHECKPOINT_CHANGE#  CURRENT_SCN
------------------  ---------------
          56453576         56454208
```

Step 2 Start LogMiner and specify the CONTINUOUS_MINE option.

```
SQL> EXECUTE DBMS_LOGMNR.START_LOGMNR(-
```

```
        STARTSCN => 56453576, -
        ENDSCN   => 56454208, -
        OPTIONS  => DBMS_LOGMNR.DICT_FROM_ONLINE_CATALOG + -
                    DBMS_LOGMNR.COMMITTED_DATA_ONLY + -
                    DBMS_LOGMNR.PRINT_PRETTY_SQL + -
                    DBMS_LOGMNR.CONTINUOUS_MINE);
```

Step 3 Display the list of archived redo log files added by LogMiner.

```
SQL> SELECT FILENAME name, LOW_SCN, NEXT_SCN FROM V$LOGMNR_LOGS;

NAME                                          LOW_SCN   NEXT_SCN
--------------------------------------------  --------  --------
/usr/oracle/data/db1arch_1_215_482701534.dbf   56316771  56453579
```

Note that the redo log file that LogMiner added does not contain the whole SCN range. When you specify the CONTINUOUS_MINE option, LogMiner adds only archived redo log files when you call the DBMS_LOGMNR.START_LOGMNR procedure. LogMiner will add the rest of the SCN range contained in the online redo log files automatically, as needed during the query execution. Use the following query to determine whether the redo log file added is the latest archived redo log file produced.

```
SQL> SELECT NAME FROM V$ARCHIVED_LOG
        WHERE SEQUENCE# = (SELECT MAX(SEQUENCE#) FROM V$ARCHIVED_LOG);

NAME
--------------------------------------------
/usr/oracle/data/db1arch_1_215_482701534.dbf
```

Step 4 Query the V$LOGMNR_CONTENTS view for changes made to the user tables.

The following query does not return the SET TRANSACTION READ WRITE and COMMIT statements associated with transaction 1.6.1911 because these statements do not have a segment owner (SEG_OWNER) associated with them.

Note that the default NLS_DATE_FORMAT, 'DD-MON-RR', is used to display the column MODIFIED_TIME of type DATE.

```
SQL> SELECT SCN, (XIDUSN || '.' || XIDSLT || '.' ||  XIDSQN) as XID, SQL_REDO
       FROM V$LOGMNR_CONTENTS
       WHERE SEG_OWNER NOT IN ('SYS', 'SYSTEM');


SCN        XID        SQL_REDO
```

```
---------- ---------- -------------
56454198   1.6.1911   update "OE"."PRODUCT_INFORMATION"
                        set
                          "WARRANTY_PERIOD" = TO_YMINTERVAL('+05-00')
                        where
                          "PRODUCT_ID" = 2430 and
                          "WARRANTY_PERIOD" = TO_YMINTERVAL('+02-00') and
                          ROWID = 'AAAHTKAABAAAY9AAAC';

56454199   1.6.1911   insert into "OE"."PRODUCT_TRACKING"
                        values
                          "PRODUCT_ID" = 2430,
                          "MODIFIED_TIME" = TO_DATE('17-JAN-03', 'DD-MON-RR'),
                          "OLD_LIST_PRICE" = 175,
                          "OLD_WARRANTY_PERIOD" = TO_YMINTERVAL('+02-00');

56454204   1.6.1911    update "OE"."PRODUCT_INFORMATION"
                         set
                           "WARRANTY_PERIOD" = TO_YMINTERVAL('+05-00')
                         where
                           "PRODUCT_ID" = 2302 and
                           "WARRANTY_PERIOD" = TO_YMINTERVAL('+02-00') and
                           ROWID = 'AAAHTKAABAAAY9QAAA';
56454206   1.6.1911    insert into "OE"."PRODUCT_TRACKING"
                         values
                           "PRODUCT_ID" = 2302,
                           "MODIFIED_TIME" = TO_DATE('17-JAN-03', 'DD-MON-RR'),
                           "OLD_LIST_PRICE" = 150,
                           "OLD_WARRANTY_PERIOD" = TO_YMINTERVAL('+02-00');
```

Step 5 End the LogMiner session.

```
SQL> EXECUTE DBMS_LOGMNR.END_LOGMNR();
```

Example 3: Using Continuous Mining to Include Future Values in a Query

To specify that a query not finish until some future time occurs or SCN is reached, use the `CONTINUOUS_MINE` option and set either the `ENDTIME` or `ENDSCAN` option in your call to the `DBMS_LOGMNR.START_LOGMNR` procedure to a time in the future or to an SCN value that has not yet been reached.

This examples assumes that you want to monitor all changes made to the table `hr.employees` from now until 5 hours from now, and that you are using the dictionary in the online catalog.

Step 1 Start LogMiner.

```
SQL> EXECUTE DBMS_LOGMNR.START_LOGMNR(-
        STARTTIME => SYSDATE, -
        ENDTIME   => SYSDATE + 5/24, -
        OPTIONS   => DBMS_LOGMNR.CONTINUOUS_MINE  + -
        DBMS_LOGMNR.DICT_FROM_ONLINE_CATALOG);
```

Step 2 Query the V$LOGMNR_CONTENTS view.

This select operation will not complete until it encounters the first redo log file record that is generated after the time range of interest (5 hours from now). You can end the select operation prematurely by entering Ctrl+C.

This example specifies the SET ARRAYSIZE statement so that rows are displayed as they are entered in the redo log file. If you do not specify the SET ARRAYSIZE statement, rows are not returned until the SQL internal buffer is full.

```
SQL> SET ARRAYSIZE 1;
SQL> SELECT USERNAME AS usr, SQL_REDO FROM V$LOGMNR_CONTENTS
       WHERE  SEG_OWNER = 'HR' AND TABLE_NAME = 'EMPLOYEES';
```

Step 3 End the LogMiner session.

```
SQL> EXECUTE DBMS_LOGMNR.END_LOGMNR();
```

Example Scenarios

The examples in this section demonstrate how to use LogMiner for typical scenarios. This section includes the following examples:

- Scenario 1: Using LogMiner to Track Changes Made by a Specific User
- Scenario 2: Using LogMiner to Calculate Table Access Statistics

Scenario 1: Using LogMiner to Track Changes Made by a Specific User

This example shows how to see all changes made to the database in a specific time range by a single user: joedevo. Connect to the database and then take the following steps:

1. Create the LogMiner dictionary file.

   To use LogMiner to analyze joedevo's data, you must either create a LogMiner dictionary file before any table definition changes are made to tables that joedevo uses or use the online catalog at LogMiner startup. See Extract a LogMiner Dictionary on page 19-44 for examples of creating LogMiner

dictionaries. This example uses a LogMiner dictionary that has been extracted to the redo log files.

2. Add redo log files.

   Assume that joedevo has made some changes to the database. You can now specify the names of the redo log files that you want to analyze, as follows:

```
SQL> EXECUTE DBMS_LOGMNR.ADD_LOGFILE( -
       LOGFILENAME => 'log1orc1.ora', -
       OPTIONS => DBMS_LOGMNR.NEW);
```

   If desired, add additional redo log files, as follows:

```
SQL> EXECUTE DBMS_LOGMNR.ADD_LOGFILE( -
       LOGFILENAME => 'log2orc1.ora', -
       OPTIONS => DBMS_LOGMNR.ADDFILE);
```

3. Start LogMiner and limit the search to the specified time range:

```
SQL> EXECUTE DBMS_LOGMNR.START_LOGMNR( -
      DICTFILENAME => 'orcldict.ora', -
      STARTTIME => TO_DATE('01-Jan-1998 08:30:00','DD-MON-YYYY HH:MI:SS'), -
      ENDTIME => TO_DATE('01-Jan-1998 08:45:00', 'DD-MON-YYYY HH:MI:SS'));
```

4. Query the V$LOGMNR_CONTENTS view.

   At this point, the V$LOGMNR_CONTENTS view is available for queries. You decide to find all of the changes made by user joedevo to the salary table. Execute the following SELECT statement:

```
SQL> SELECT SQL_REDO, SQL_UNDO FROM V$LOGMNR_CONTENTS
       WHERE USERNAME = 'joedevo' AND SEG_NAME = 'salary';
```

   For both the SQL_REDO and SQL_UNDO columns, two rows are returned (the format of the data display will be different on your screen). You discover that joedevo requested two operations: he deleted his old salary and then inserted a new, higher salary. You now have the data necessary to undo this operation.

```
SQL_REDO                                 SQL_UNDO
--------                                 --------
delete from SALARY                       insert into SALARY(NAME, EMPNO, SAL)
where EMPNO = 12345                       values ('JOEDEVO', 12345, 500)
and NAME='JOEDEVO'
and SAL=500;

insert into SALARY(NAME, EMPNO, SAL)  delete from SALARY
```

```
values('JOEDEVO',12345, 2500)          where EMPNO = 12345
                                       and NAME = 'JOEDEVO'
2 rows selected                        and SAL = 2500;
```

5. End the LogMiner session.

   Use the DBMS_LOGMNR.END_LOGMNR procedure to finish the LogMiner session properly:

```
SQL> DBMS_LOGMNR.END_LOGMNR( );
```

Scenario 2: Using LogMiner to Calculate Table Access Statistics

In this example, assume you manage a direct marketing database and want to determine how productive the customer contacts have been in generating revenue for a 2-week period in January. Assume that you have already created the LogMiner dictionary and added the redo log files that you want to search (as demonstrated in the previous example). Take the following steps:

1. Start LogMiner and specify a range of times:

```
SQL> EXECUTE DBMS_LOGMNR.START_LOGMNR( -
      STARTTIME => TO_DATE('07-Jan-2003 08:30:00','DD-MON-YYYY HH:MI:SS'), -
      ENDTIME => TO_DATE('21-Jan-2003 08:45:00','DD-MON-YYYY HH:MI:SS'), -
      DICTFILENAME => '/usr/local/dict.ora');
```

2. Query the V$LOGMNR_CONTENTS view to determine which tables were modified in the time range you specified, as shown in the following example. (This query filters out system tables that traditionally have a $ in their name.)

```
SQL> SELECT SEG_OWNER, SEG_NAME, COUNT(*) AS Hits FROM
       V$LOGMNR_CONTENTS WHERE SEG_NAME NOT LIKE '%$' GROUP BY
       SEG_OWNER, SEG_NAME ORDER BY Hits DESC;
```

3. The following data is displayed. (The format of your display may be different.)

```
SEG_OWNER          SEG_NAME          Hits
---------          --------          ----
CUST               ACCOUNT            384
UNIV               EXECDONOR          325
UNIV               DONOR              234
UNIV               MEGADONOR           32
HR                 EMPLOYEES           12
SYS                DONOR               12
```

   The values in the Hits column show the number of times that the named table had an insert, delete, or update operation performed on it during the 2-week

period specified in the query. In this example, the `cust.account` table was modified the most during the specified 2-week period, and the `hr.employees` and `sys.donor` tables were modified the least during the same time period.

4. End the LogMiner session.

   Use the `DBMS_LOGMNR.END_LOGMNR` procedure to finish the LogMiner session properly:

   ```
   SQL> DBMS_LOGMNR.END_LOGMNR( );
   ```

Supported Datatypes, Storage Attributes, and Database and Redo Log File Versions

The following sections provide information about datatype and storage attribute support and the versions of the database and redo log files supported:

- Supported Datatypes and Table Storage Attributes
- Unsupported Datatypes and Table Storage Attributes
- Supported Databases and Redo Log File Versions

Supported Datatypes and Table Storage Attributes

LogMiner supports the following datatypes and table storage attributes:

- CHAR
- NCHAR
- VARCHAR2 and VARCHAR
- NVARCHAR2
- NUMBER
- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE
- TIMESTAMP WITH LOCAL TIME ZONE
- INTERVAL YEAR TO MONTH
- INTERVAL DAY TO SECOND

- RAW
- CLOB
- NCLOB
- BLOB
- LONG
- LONG RAW
- BINARY_FLOAT
- BINARY_DOUBLE
- Index-organized tables (IOTs) without overflows and without LOB columns
- Function-based indexes

Support for LOB and LONG datatypes in redo log files is available only for redo log files generated on a release 9.2 or later Oracle database.

Unsupported Datatypes and Table Storage Attributes

LogMiner does not support these datatypes and table storage attributes:

- BFILE datatype
- Simple and nested abstract datatypes (ADTs)
- Collections (nested tables and VARRAYs)
- Object refs
- XMLTYPE datatype
- Index-organized tables (IOTs) with LOB columns
- Tables using table compression

Supported Databases and Redo Log File Versions

LogMiner runs only on databases of release 8.1 or later, but you can use it to analyze redo log files from release 8.0 databases. However, the information that LogMiner is able to retrieve from a redo log file depends on the version of the log, not the version of the database in use. For example, redo log files for Oracle9*i* can be augmented to capture additional information when supplemental logging is enabled. This allows LogMiner functionality to be used to its fullest advantage. Redo log files created with older releases of Oracle will not have that additional

data and may therefore have limitations on the operations and datatypes supported by LogMiner.

> **See Also:** Steps in a Typical LogMiner Session on page 19-43 and Supplemental Logging on page 19-28

20

Original Export and Import

This chapter describes how to use the original Export and Import utilities, invoked with the `exp` and `imp` command, respectively. These are called the original Export and Import utilities to differentiate them from the new Oracle Data Pump Export and Import utilities available as of Oracle Database 10*g*. These new utilities are invoked with the `expdp` and `impdp` commands, respectively. In general, Oracle recommends that you use the new Data Pump Export and Import utilities because they support all Oracle Database 10*g* features. Original Export and Import do not support all Oracle Database 10*g* features.

However, you should still use the original Export and Import utilities in the following situations:

- You want to import files that were created using the original Export utility (`exp`).
- You want to export files that will be imported using the original Import utility (`imp`). An example of this would be if you wanted to export data from Oracle Database 10*g* and then import it into an earlier database release.

See Also: Part I, "Oracle Data Pump"

This chapter discusses the following topics:

- Effect of Schema and Database Triggers on Import Operations
- Export and Import Modes
- Export Parameters
- Import Parameters
- Example Export Sessions
- Example Import Sessions
- Using Export and Import to Move a Database Between Platforms
- Warning, Error, and Completion Messages
- Exit Codes for Inspection and Display
- Network Considerations
- Character Set and Globalization Support Considerations
- Materialized Views and Snapshots
- Transportable Tablespaces
- Read-Only Tablespaces
- Dropping a Tablespace
- Reorganizing Tablespaces
- Support for Fine-Grained Access Control
- Using Instance Affinity with Export and Import
- Reducing Database Fragmentation
- Using Storage Parameters with Export and Import
- Information Specific to Export
- Information Specific to Import
- Using Export and Import to Partition a Database Migration
- Using Different Releases and Versions of Export

What Are the Export and Import Utilities?

The Export and Import utilities provide 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 dump file. The Import utility reads the object definitions and table data from the dump file.

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 20-48 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 20-105
- Part II of this manual for information about the SQL*Loader utility
- *Oracle Database Advanced Replication* for information about how to use the Export and Import utilities to facilitate certain aspects of Oracle Advanced Replication, such as offline instantiation

Before Using Export and Import

Before you begin using Export and Import, 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 and Import, you must run the script `catexp.sql` or `catalog.sql` (which runs `catexp.sql`) after the database has been created or migrated to Oracle Database 10*g*.

The `catexp.sql` or `catalog.sql` script needs to be run only once on a database. The script performs the following tasks to prepare the database for export and import operations:

- Creates the necessary export and import views in the data dictionary
- Creates the EXP_FULL_DATABASE role
- Assigns all necessary privileges to the EXP_FULL_DATABASE and IMP_FULL_DATABASE roles
- Assigns EXP_FULL_DATABASE and IMP_FULL_DATABASE to the DBA role
- Records the version of `catexp.sql` that has been installed

Ensuring Sufficient Disk Space for Export Operations

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:** *Oracle Database Reference* for more information about dictionary views

Verifying Access Privileges for Export and Import Operations

To use Export and Import, you must have the CREATE SESSION privilege on an Oracle database. This privilege belongs to the CONNECT role established during database creation. To export tables owned by another user, you must have the EXP_FULL_DATABASE role enabled. This role is granted to all database administrators (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

You can perform an import operation even if you did not create the export file. However, keep in mind that if the export file was created by a user with the EXP_FULL_DATABASE role, then you must have the IMP_FULL_DATABASE role to import it. Both of these roles are typically assigned to database administrators (DBAs).

Invoking Export and Import

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

- Command-line entries
- Parameter files
- Interactive mode

Before you use one of these methods, be sure to read the descriptions of the available parameters. See Export Parameters on page 20-22 and Import Parameters on page 20-39.

Invoking Export and 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 Export or Import as SYSDBA, except in the following situations:

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

To invoke Export or Import as SYSDBA, use the following syntax (substitute exp for imp if you are using Export). Add 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, you will be prompted 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 and Import use would not understand the quotation marks and would remove them.

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. Note that the examples could use imp to invoke Import rather than exp to invoke Export.

Parameter Files

The information in this section applies to both Export and Import, but the examples show use of the Export command, exp.

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.dmp
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.

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 20-22 for descriptions of the Export parameters
- Import Parameters on page 20-39
- Exporting and Importing with Oracle Net on page 20-75 for information about how to specify an export from a remote database

Interactive Mode

If you prefer to be prompted for the value of each parameter, you can use the following syntax to start Export (or Import, if you specify `imp`) in interactive mode:

```
exp username/password
```

Commonly used parameters are displayed with a request for you to enter a value. The command-line interactive method does not provide prompts for all functionality and is provided only for backward compatibility. If you want to use an interactive interface, Oracle recommends that you use the Oracle Enterprise Manager Export or Import Wizard.

If you do not specify a *username*/*password* combination on the command line, then you are prompted for this information.

Restrictions When Using Export's Interactive Method

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.

Getting Online Help

Export and Import both provide online help. Enter `exp help=y` on the command line to invoke Export help or `imp help=y` to invoke Import help.

Importing Objects into Your Own Schema

Table 20–1 lists the privileges required to import objects into your own schema. All of these privileges initially belong to the `RESOURCE` role.

Table 20–1 Privileges Required to Import Objects into Your Own Schema

Object	Required Privilege (Privilege Type, If Applicable)
Clusters	`CREATE CLUSTER` (System) or `UNLIMITED TABLESPACE` (System). The user must also be assigned a tablespace quota.
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) or `UNLIMITED TABLESPACE` (System). The user must also be assigned a tablespace quota.
Integrity constraints	`ALTER TABLE` (Object)
Libraries	`CREATE ANY LIBRARY` (System)
Packages	`CREATE PROCEDURE` (System)
Private synonyms	`CREATE SYNONYM` (System)
Sequences	`CREATE SEQUENCE` (System)

Table 20–1 (Cont.) Privileges Required to Import Objects into Your Own Schema

Object	Required Privilege (Privilege Type, If Applicable)
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) or UNLIMITED TABLESPACE (System). The user must also be assigned a tablespace quota.
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 20–2 shows the required conditions for the authorizations to be valid on the target system.

Table 20–2 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 IMP_FULL_DATABASE role must be enabled. The parameter FULL specifies that the following 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

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 JavaBeans (EJB) 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.
- The table compression attribute of tables and partitions is preserved during export and import. However, the import process does not use the direct path API, hence the data will not be stored in the compressed format when imported. Use the new Data Pump Export and Import utilities to enable compression during import.

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, function-based, and domain indexes

The order of import is as follows: new tables are created, data is imported and indexes are built, triggers are imported, integrity constraints are enabled on the new tables, and any bitmap, function-based, 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.

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 exists 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 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 recommends that you re-create them after the import in case they define actions that would impede progress of the import.

Export and Import Modes

The Export and Import utilities support four modes of operation:

- Full: Exports and imports a full database. Only users with the EXP_FULL_DATABASE and IMP_FULL_DATABASE roles can use this mode. Use the FULL parameter to specify this mode.
- Tablespace: Enables 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: Enables you to export and 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 OWNER parameter to specify this mode in Export, and use the FROMUSER parameter to specify this mode in Import.
- Table: Enables you to export and 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.

See Table 20–3 for a list of objects that are exported and imported in each 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.

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.

You can use conventional path Export or direct path Export to export in any mode except tablespace mode.The differences between conventional path Export and direct path Export are described in Conventional Path Export Versus Direct Path Export on page 20-84.

See Also:

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

Table 20–3 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 function-based indexes	Yes[1]	Yes	Yes	Yes
Cluster definitions	No	Yes	Yes	Yes

Table 20–3 (Cont.) Objects Exported and Imported in Each Mode

Object	Table Mode	User Mode	Full Database Mode	Tablespace Mode
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
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

Table 20–3 (Cont.) Objects Exported and Imported in Each Mode

Object	Table Mode	User Mode	Full Database Mode	Tablespace Mode
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
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 20–3 (Cont.) Objects Exported and Imported in Each Mode

Object	Table Mode	User Mode	Full Database Mode	Tablespace Mode
Table data	Yes	Yes	Yes	Yes
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
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 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 about how to specify table-level and partition-level Exports, see TABLES on page 20-35.

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 20-44 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.

Export Parameters

This section contains descriptions of the Export command-line parameters.

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 LOBs, LONG, 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. For direct path Exports, use the RECORDLENGTH parameter to specify the size of the buffer that Export uses for writing to the export file.

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: Neither LOB data nor subpartition data is compressed. Rather, 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 20–4 shows a sequence of events by two users: user1 exports partitions in a table and user2 updates data in that table.

Table 20–4 Sequence of Events During Updates by Two Users

TIme Sequence	User1	User2
1	Begins export of TAB:P1	No activity
2	No activity	Updates TAB:P2 Updates TAB:P1 Commits transaction

Table 20–4 (Cont.) Sequence of Events During Updates by Two Users

TIme Sequence	User1	User2
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 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.

Note: Rollback segments will be deprecated in a future Oracle database release. Oracle recommends that you use automatic undo management instead.

See Also: OBJECT_CONSISTENT on page 20-31

CONSTRAINTS

Default: `y`

Specifies whether or not the Export utility exports table constraints.

DIRECT

Default: `n`

Specifies the use of direct 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 20-84.

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 individually for each table.

FILE

Default: `expdat.dmp`

Specifies the names of the export dump files. The default extension is .dmp, but you can specify any extension. Because Export supports multiple export files, 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 20–5, 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 20-26 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 terminate, 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 20–5 shows that the maximum size for dump files depends on the operating system you are using and on the release of the Oracle database that you are using.

Table 20–5 Maximum Size for Dump Files

Operating System	Release of Oracle Database	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

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:** *Oracle Database Application Developer's Guide - Fundamentals* for more information about using flashback

The following is an example of specifying an SCN. When the export is performed, the data will be consistent as of SCN `3482971`.

```
> exp system/password FILE=exp.dmp FLASHBACK_SCN=3482971
```

FLASHBACK_TIME

Default: none

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

You can specify the time in any format that the DBMS_FLASHBACK.ENABLE_AT_TIME procedure accepts. This means that you can specify it in either of the following ways:

```
> exp system/password FILE=exp.dmp FLASHBACK_TIME="TIMESTAMP '2002-05-01 11:00:00'"

> exp system/password FILE=exp.dmp FLASHBACK_TIME="TO_TIMESTAMP('12-02-2001 14:35:00',
'DD-MM-YYYY HH24:MI:SS')"
```

Also, the old format, as shown in the following example, will continue to be accepted to ensure backward compatibility:

```
> exp system/password FILE=exp.dmp FLASHBACK_TIME="'2002-05-01 11:00:00'"
```

See Also:

- *Oracle Database Application Developer's Guide - Fundamentals* for more information about using flashback
- *PL/SQL Packages and Types Reference* for more information about the DBMS_FLASHBACK package

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.

Points to Consider for Full Database Exports and Imports

A full database export and import can be a good way to replicate or clean up a database. However, to avoid problems be sure to keep the following points in mind:

- 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 recommends that you re-create them after the import in case they define actions that would impede progress of the import.

- If possible, before beginning, make a physical copy of the exported database and the database into which you intend to import. This ensures that any mistakes are reversible.
- Before you begin the export, it is advisable to produce a report that includes the following information:
  - A list of tablespaces and datafiles
  - A list of rollback segments
  - A count, by user, of each object type such as tables, indexes, and so on

  This information lets you ensure that tablespaces have already been created and that the import was successful.
- If you are creating a completely new database from an export, remember to create an extra rollback segment in `SYSTEM` and to make it available in your initialization parameter file (`init.ora`) before proceeding with the import.
- When you perform the import, ensure you are pointing at the correct instance. This is very important because on some UNIX systems, just the act of entering a subshell can change the database against which an import operation was performed.
- Do not perform a full import on a system that has more than one database unless you are certain that all tablespaces have already been created. A full import creates any undefined tablespaces using the same datafile names as the exported database. This can result in problems in the following situations:
  - If the datafiles belong to any other database, they will become corrupted. This is especially true if the exported database is on the same system, because its datafiles will be reused by the database into which you are importing.
  - If the datafiles have names that conflict with existing operating system files.

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 20-24

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 database administrator (DBA), multiple users can be listed.

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.

PARFILE

Default: none

Specifies a filename for a file that contains a list of Export parameters. For more information about using a parameter file, see Invoking Export and Import on page 20-5.

QUERY

Default: none

This parameter enables 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 When Using the QUERY Parameter

- The QUERY parameter cannot be specified for full, user, or tablespace-mode exports.
- The QUERY parameter must be applicable to all specified tables.
- The QUERY parameter cannot be specified in a direct path Export (DIRECT=y)
- The QUERY parameter 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.

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.

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:

- *Oracle Database Concepts*
- *Oracle 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 terminated.

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 20-49 and Importing Statistics on page 20-94.

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 enables you to tune plan generation for queries in a nonproduction database using statistics from a production database.

See Also: *Oracle 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, LBACSYS, 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 20-61 for several examples of partition-level Exports.

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. Different operating systems also have other restrictions on table naming.

TABLESPACES

Default: none

The TABLESPACES parameter specifies that all tables in the specified 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 20-79
- *Oracle Database Administrator's Guide*
- *Oracle Database Concepts*

TRIGGERS

Default: y

Specifies whether or not the Export utility exports triggers.

TTS_FULL_CHECK

Default: n

When TTS_FULL_CHECK is set to y, 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 the reverse.

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 and Import on page 20-5 for more information.

See Also:

- *Oracle Database Heterogeneous Connectivity Administrator's Guide*
- The user's guide for your Oracle Net protocol for information about specifying a connect string for Oracle Net

VOLSIZE

Default: none

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 on your platform.

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).

Import Parameters

This section contains descriptions of the Import command-line parameters.

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 LOBs or LONG, 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.

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 LOBs, LONG, BFILE, REF, ROWID, or UROWID 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 20-101

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 20-55

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 SQL 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 individually set for each table.

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 size allowed 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 about the maximum size of dump files, see Table 20–5.

FROMUSER

Default: none

A comma-delimited 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 function-based 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 20-54). The user that you specify with TOUSER must exist in the target database prior to the import operation; otherwise an error is returned.

If you do not specify TOUSER, Import will do the following:

- 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: y

Specifies whether to import the entire export dump 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 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 about using a parameter file, see Parameter Files on page 20-7.

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.

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.

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:

- *Oracle Database Concepts*
- *Oracle 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 terminated.

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 dump 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: the value of the Oracle database configuration parameter, SKIP_UNUSABLE_INDEXES, as specified in the initialization parameter file

Both Import and the Oracle database provide a SKIP_UNUSABLE_INDEXES parameter. The Import SKIP_UNUSABLE_INDEXES parameter is specified at the Import command line. The Oracle database SKIP_UNUSABLE_INDEXES parameter is specified as a configuration parameter in the initialization parameter file. It is important to understand how they affect each other.

If you do not specify a value for SKIP_UNUSABLE_INDEXES at the Import command line, then Import uses the database setting for the SKIP_UNUSABLE_INDEXES configuration parameter, as specified in the initialization parameter file.

If you do specify a value for SKIP_UNUSABLE_INDEXES at the Import command line, it overrides the value of the SKIP_UNUSABLE_INDEXES configuration parameter in the initialization parameter file.

A value of y means that Import will skip building indexes that were set to the Index Unusable state (by either system or user). Other indexes (not previously set to Index Unusable) continue to be updated as rows are inserted.

This parameter enables 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. If the SKIP_UNUSABLE_INDEXES parameter is not specified, row insertions that attempt to update unusable indexes will fail.

See Also: The ALTER SESSION statement in the *Oracle Database 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. This requires that the original export operation that created the dump file must have generated the necessary ANALYZE statements (that is, the export was not performed with STATISTICS=NONE). These ANALYZE statements are

included in the dump file and used by the import operation for recalculation of the table's statistics.

See Also:

- *Oracle Database Concepts* for more information about the optimizer and the statistics it uses
- Importing Statistics on page 20-94

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: *Oracle Streams Replication Administrator's Guide*

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: *Oracle Streams Replication Administrator's Guide*

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 20–3 on page 20-16. 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 Using Pattern Matching to Import Various Tables on page 20-71.

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 imports 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 20-55 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=jobs TOID_NOVALIDATE=typ1
imp scott/tiger TABLE=salaries TOID_NOVALIDATE=(fred.typ0,sally.typ2,typ3)
```

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 `typ1` defaults to `scott.typ1`.

Note that `TOID_NOVALIDATE` deals only with table column types. It has no effect on table types.

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 user names whose schemas will be targets for Import. The user names must exist prior to the import operation; otherwise an error is returned. 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 20-43 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 20-55.

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 Also:

- *Oracle Database Heterogeneous Connectivity Administrator's Guide*
- The user's guide for your Oracle Net protocol for information about specifying a connect string for Oracle Net

VOLSIZE

Default: none

Specifies the maximum number of bytes in a dump 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 on your platform.

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 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. Some examples use vertical ellipses to indicate sections of example output that were too long to include.

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

Information is displayed about the release of Export you are using and the release of Oracle Database that you are connected to. Status messages are written out as the entire database is exported. A final completion message is returned when the export completes 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

Information is displayed about the release of Export you are using and the release of Oracle Database that you are connected to. Then, status messages similar to the following are shown:

```
.
.
. 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
.
.
.
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

Information is displayed about the release of Export you are using and the release of Oracle Database that you are connected to. Then, status messages similar to the following are shown:

```
.
.
.
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

Information is displayed about the release of Export you are using and the release of Oracle Database that you are connected to. Then, status messages similar to the following are shown:

```
.
.
.

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

Information is displayed about the release of Export you are using and the release of Oracle Database that you are connected to. Then, status messages similar to the following are shown:

```
.
.
```

```
.
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

Information is displayed about the release of Export you are using and the release of Oracle Database that you are connected to. Then, status messages similar to the following are shown:

```
.
.
.
```

```
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

Information is displayed about the release of Export you are using and the release of Oracle Database that you are connected to. Then, status messages similar to the following are shown:

```
.
.
.
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

Information is displayed about the release of Export you are using and the release of Oracle Database that you are connected to. Then, status messages similar to the following are shown:

```
.
.
.
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.
```

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 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

Information is displayed about the release of Import you are using and the release of Oracle Database that you are connected to. Then, status messages similar to the following are shown:

```
.
.
.
Export file created by EXPORT:V10.00.00 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

Information is displayed about the release of Import you are using and the release of Oracle Database that you are connected to. Then, status messages similar to the following are shown:

```
.
.
.
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 database administrator (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

Information is displayed about the release of Import you are using and the release of Oracle Database that you are connected to. Then, status messages similar to the following are shown:

```
.
.
.
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
```

Export Messages

Information is displayed about the release of Export you are using and the release of Oracle Database that you are connected to. Then, status messages similar to the following are shown:

```
.
.
.
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

Information is displayed about the release of Import you are using and the release of Oracle Database that you are connected to. Then, status messages similar to the following are shown:

```
.
.
.
Export file created by EXPORT:V10.00.00 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. Partition P1 has three subpartitions: P1_SP1, P1_SP2, and P1_SP3. Partition 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
```

Export Messages

Information is displayed about the release of Export you are using and the release of Oracle Database that you are connected to. Then, status messages similar to the following are shown:

When the command executes, the following Export messages are displayed:

```
.
.
.
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

Information is displayed about the release of Import you are using and the release of Oracle Database that you are connected to. Then, status messages similar to the following are shown:

```
.
.
.
. 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
.
.
.

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 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 SQL 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 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

Information is displayed about the release of Import you are using and the release of Oracle Database that you are connected to. Then, status messages similar to the following are shown:

```
.
.
.
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 Export and Import to Move a Database Between Platforms

The Export and Import utilities are the only method that Oracle supports for moving an existing Oracle database from one hardware platform to another. This includes moving between UNIX and NT systems and also moving between two NT systems running on different platforms.

The following steps present a general overview of how to move a database between platforms.

1. As a DBA user, issue the following SQL query to get the exact name of all tablespaces. You will need this information later in the process.

   ```
   SQL> SELECT tablespace_name FROM dba_tablespaces;
   ```

2. As a DBA user, perform a full export from the source database, for example:

   ```
   > exp system/manager FULL=y FILE=expdat.dmp
   ```

   See Also: Points to Consider for Full Database Exports and Imports on page 20-29

3. Move the dump file to the target database server. If you use FTP, be sure to copy it in binary format (by entering `binary` at the FTP prompt) to avoid file corruption.

4. Create a database on the target server.

   See Also: *Oracle Database Administrator's Guide* for information about how to create a database

5. Before importing the dump file, you must first create your tablespaces, using the information obtained in Step 1. Otherwise, the import will create the corresponding datafiles in the same file structure as at the source database, which may not be compatible with the file structure on the target system.

6. As a DBA user, perform a full import with the `IGNORE` parameter enabled:

   ```
   > imp system/manager FULL=y IGNORE=y FILE=expdat.dmp
   ```

   Using `IGNORE=y` instructs Oracle to ignore any creation errors during the import and permit the import to complete.

7. Perform a full backup of your new database.

Warning, Error, and Completion Messages

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

Log File

You can capture all Export and Import messages in a log file, either by using the LOG parameter or, for those systems that permit it, by redirecting the output to a file. A log of detailed information is written about successful unloads and loads and any errors that may have occurred.

Warning Messages

Export and Import do 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 and Import also issue warnings when invalid objects are encountered.

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 or 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. 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 or import completes without errors, a message to that effect is displayed, for example:

```
Export terminated successfully without warnings
```

If one or more recoverable errors occurs but the job continues to completion, a message similar to the following is displayed:

```
Export terminated successfully with warnings
```

If a nonrecoverable error occurs, the job terminates immediately and displays a message stating so, for example:

```
Export terminated unsuccessfully
```

Exit Codes for Inspection and Display

Export and Import provide the results of an operation immediately upon completion. Depending on the platform, the outcome may be reported in a process exit code and the results recorded in the log file. This enables you to check the outcome from the command line or script. Table 20–6 shows the exit codes that get returned for various results.

Table 20–6 Exit Codes for Export and Import

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

For UNIX, the exit codes are as follows:

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

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 Export or 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:

- *Oracle Net Services Administrator's Guide*
- *Oracle Database Heterogeneous Connectivity Administrator's Guide*

Character Set and Globalization Support Considerations

The following sections describe the globalization support behavior of Export and Import with respect to character set conversion of user data and data definition language (DDL).

User Data

The Export utility always exports user data, including Unicode data, in the character sets of the Export server. (Character sets are specified at database creation.) If the character sets of the source database are different than the character sets of the import database, a single conversion is performed to automatically convert the data to the character sets of the Import server.

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: *Oracle Database Globalization Support Guide*

Data Definition Language (DDL)

Up to three character set conversions may be required for data definition language (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 the character set used by the import user session.

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.

Single-Byte Character Sets and Export and Import

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.

Multibyte Character Sets and Export and Import

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: *Oracle Database Globalization Support Guide*

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.

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.

See Also: *Oracle Database Advanced 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.

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 and related items are exported with the schema name explicitly given in the DDL statements. To import them into a different schema, use the FROMUSER and TOUSER parameters. This does not apply to snapshot logs, which cannot be imported into a different schema.

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 and Import provide the following parameters to enable movement of transportable tablespace metadata.

- TABLESPACES
- TRANSPORT_TABLESPACE

See TABLESPACES on page 20-37 and TRANSPORT_TABLESPACE on page 20-38 for more information about using these parameters during an export operation.

See TABLESPACES on page 20-53 and TRANSPORT_TABLESPACE on page 20-55 for information about using these parameters during an import operation.

See Also:

- *Oracle Database Administrator's Guide* for details about managing transportable tablespaces
- *Oracle Database Concepts* for an introduction to transportable tablespaces

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.

   When you revoke a role, it does not have a cascade effect. 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.)

Support for Fine-Grained Access Control

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

- To restore the fine-grained access control policies, the user who imports from an export file containing such tables must have the EXECUTE privilege on the DBMS_RLS package, so that the security policies on the tables 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 user has privileges to read will be exported.

  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.
- 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 EXP_FULL_DATABASE role enabled or who has been granted the EXEMPT ACCESS POLICY privilege, can perform direct path Exports on tables having fine-grained access control.

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

Using Instance Affinity with Export and Import

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.

See Also:

- *Oracle Database Administrator's Guide*
- *Oracle Database Reference*
- *Oracle Database Upgrade Guide*

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 after all users are logged off.
3. Delete the database. See your Oracle operating system-specific documentation for information about 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: *Oracle Database Administrator's Guide* for more information about creating databases

Using Storage Parameters with Export and Import

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 20-80 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 20-23.

Information Specific to Export

The material presented in this section is specific to the original Export utility. The following topics are discussed:

- Conventional Path Export Versus Direct Path Export
- Invoking a Direct Path Export
- Exporting from a Read-Only Database
- Considerations When Importing Database Objects

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 (that is, the SQL command-processing layer) 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.

Invoking a Direct Path Export

To use direct path Export, specify the `DIRECT=y` parameter on the command line or in the parameter file. The default is `DIRECT=n`, which extracts the table data using the conventional path. The rest of this section discusses the following topics:

- Security Considerations for Direct Path Exports
- Performance Considerations for Direct Path Exports

- Restrictions for Direct Path Exports

Note: When you export a table in direct path, be sure that no other transaction is updating the same table, and that the size of the rollback segment is sufficient. Otherwise, you may receive the following error:

ORA-01555 snapshot too old; rollback segment number *string* with name "*string*" too small

This will cause the export to terminate unsuccessfully.

Security Considerations for Direct Path Exports

Oracle 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 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 20-81
- *Oracle Database Application Developer's Guide - Fundamentals*

Performance Considerations 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

An export file that is created using direct path Export will take the same amount of time to import as an export file created using conventional path Export.

Restrictions for Direct Path Exports

Keep the following restrictions in mind when you are using direct path mode:

- 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.
- 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.
- You cannot use direct path when exporting in tablespace mode (TRANSPORT_TABLESPACES=Y).
- The QUERY parameter cannot be specified in a direct path Export.
- A direct path Export can only export data when the NLS_LANG environment variable of the session invoking the export is equal to the database character set. If NLS_LANG is not set or if it is different than the database character set, a warning is displayed and the export is discontinued. The default value for the NLS_LANG environment variable is AMERICAN_AMERICA.US7ASCII.

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 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:** *Oracle Data Guard Concepts and Administration* for more information about setting up this environment

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 might 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.

Note: Oracle also recommends that you convert existing LONG columns to LOB columns. LOB columns are subject to far fewer restrictions than LONG columns. Further, LOB functionality is enhanced in every release, whereas LONG functionality has been static for several releases.

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 mode 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 Locally Managed Tablespaces

If the data you are exporting contains offline locally managed 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 create the offline locally managed 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-mode 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.

Exporting 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 mode 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 mode, user mode, 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 mode 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:** *Oracle Streams Advanced Queuing User's Guide and Reference*

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.

Information Specific to Import

The material in this section is specific to the original Import utility. The following topics are discussed:

- Error Handling During an Import Operation
- Controlling Index Creation and Maintenance
- Importing Statistics
- Tuning Considerations for Import Operations
- Considerations When Importing Database Objects

Error Handling During an Import Operation

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:

- *Oracle Database Application Developer's Guide - Fundamentals*
- *Oracle 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 20-40.

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.

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 the Import. Other indexes (not previously set to Index Unusable) continue to be updated as rows are inserted. This approach saves on index updates during the 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.

Importing Statistics

If statistics are requested at export time and analyzer statistics are available for a table, Export will include the ANALYZE statement used 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 20-35 and the Import parameter STATISTICS on page 20-49.

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 execute 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. The importer might want to import only unquestionable statistics, not 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 import 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.

Tuning Considerations for Import Operations

This section discusses some ways to possibly improve the performance of an import operation. The information is categorized as follows:

- Changing System-Level Options
- Changing Initialization Parameters
- Changing Import Options
- Dealing with Large Amounts of LOB Data
- Dealing with Large Amounts of LONG Data

Changing System-Level Options

The following suggestions about system-level options may help to improve performance of an import operation:

- Create and use one large rollback segment and take all other rollback segments offline. Generally a rollback segment that is one half the size of the largest table being imported should be big enough. It can also help if the rollback segment is created with the minimum number of two extents, of equal size.

> **Note:** Rollback segments will be deprecated in a future Oracle Database release. Oracle recommends that you use automatic undo management instead.

- Put the database in NOARCHIVELOG mode until the import is complete. This will reduce the overhead of creating and managing archive logs.
- Create several large redo files and take any small redo log files offline. This will result in fewer log switches being made.
- If possible, have the rollback segment, table data, and redo log files all on separate disks. This will reduce I/O contention and increase throughput.
- If possible, do not run any other jobs at the same time that may compete with the import operation for system resources.
- Make sure there are no statistics on dictionary tables.
- Set TRACE_LEVEL_CLIENT=OFF in the sqlnet.ora file.
- If possible, increase the value of DB_BLOCK_SIZE when you re-create the database. The larger the block size, the smaller the number of I/O cycles needed. *This change is permanent, so be sure to carefully consider all effects it will have before making it.*

Changing Initialization Parameters

The following suggestions about settings in your initialization parameter file may help to improve performance of an import operation.

- Set LOG_CHECKPOINT_INTERVAL to a number that is larger than the size of the redo log files. This number is in operating system blocks (512 on most UNIX systems). This reduces checkpoints to a minimum (at log switching time).
- Increase the value of SORT_AREA_SIZE. The amount you increase it depends on other activity taking place on the system and on the amount of free memory available. (If the system begins swapping and paging, the value is probably set too high.)
- Increase the value for DB_BLOCK_BUFFERS and SHARED_POOL_SIZE.

Changing Import Options

The following suggestions about usage of import options may help to improve performance. Be sure to also read the individual descriptions of all the available options in Import Parameters on page 20-39.

- Set `COMMIT=N`. This causes Import to commit after each object (table), not after each buffer. This is why one large rollback segment is needed. (Because rollback segments will be deprecated in future releases, Oracle recommends that you use automatic undo management instead.)
- Specify a large value for `BUFFER` or `RECORDLENGTH`, depending on system activity, database size, and so on. A larger size reduces the number of times that the export file has to be accessed for data. Several megabytes is usually enough. Be sure to check your system for excessive paging and swapping activity, which can indicate that the buffer size is too large.
- Consider setting `INDEXES=N` because indexes can be created at some point after the import, when time is not a factor. If you choose to do this, you need to use the `INDEXFILE` parameter to extract the DLL for the index creation or to rerun the import with `INDEXES=Y` and `ROWS=N`.

Dealing with Large Amounts of LOB Data

Keep the following in mind when you are importing large amounts of LOB data:

Eliminating indexes significantly reduces total import time. This is because LOB data requires special consideration during an import because the LOB locator has a primary key that cannot be explicitly dropped or ignored during an import.

Make sure there is enough space available in large contiguous chunks to complete the data load.

Dealing with Large Amounts of LONG Data

Keep in mind that importing a table with a `LONG` column may cause a higher rate of I/O and disk usage, resulting in reduced performance of the import operation. There are no specific parameters that will improve performance during an import of large amounts of LONG data, although some of the more general tuning suggestions made in this section may help overall performance.

See Also: Importing LONG Columns on page 20-102

Considerations When Importing Database Objects

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

Importing Object Identifiers

The Oracle database 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 20-53 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 ROWIDs for the target database. You should execute the following statement to reset the ROWIDs to their proper values:

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

> **See Also:** *Oracle Database 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. The error occurs when the user seeks subsequent access to the file data. 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 20-40

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 result in 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: *Oracle Streams Advanced Queuing User's Guide and Reference*

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.

Note: Oracle recommends that you convert existing LONG columns to LOB columns. LOB columns are subject to far fewer restrictions than LONG columns. Further, LOB functionality is enhanced in every release, whereas LONG functionality has been static for several releases.

Importing LOB Columns When Triggers Are Present

As of Oracle Database 10*g*, LOB handling has been improved to ensure that triggers work properly and that performance remains high when LOBs are being loaded. To achieve these improvements, the Import utility automatically changes all LOBs that were empty at export time to be NULL after they are imported.

If you have applications that expect the LOBs to be empty rather than NULL, then after the import you can issue a SQL UPDATE statement for each LOB column. Depending on whether the LOB column type was a BLOB or a CLOB, the syntax would be one of the following:

```
UPDATE <tablename> SET <lob column> = EMPTY_BLOB() WHERE <lob column> = IS NULL;
UPDATE <tablename> SET <lob column> = EMPTY_CLOB() WHERE <lob column> = IS NULL;
```

It is important to note that once the import is performed, there is no way to distinguish between LOB columns that are NULL versus those that are empty. Therefore, if that information is important to the integrity of your data, be sure you know which LOB columns are NULL and which are empty before you perform the import.

Importing Views

Views are exported in dependency order. In some cases, Export must determine the ordering, rather than obtaining the order from the 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.

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=schema`*n* `FILE=schema`*n*
   b. `imp dba/password FILE=schema`*n* `FROMUSER=schema`*n* `TOUSER=schema`*n* `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.

Whenever you are moving data between different releases of the Oracle database, 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 earliest version of the source or target database.

  For example, to create an export file for an import into a later 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 an earlier release database, use a version of the Export utility that is equal to the version of the target database.

  – 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 20-107.

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 later release of the Oracle database.
- The Import utility cannot read export dump files created by the Export utility of a later maintenance release or version. For example, a release 9.2 export dump file cannot be imported by a release 9.0.1 Import utility.
- Whenever a lower version of the Export utility runs with a later version of the Oracle database, categories of database objects that did not exist in the earlier version are excluded from the export.
- 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 20–7 shows some examples of which Export and Import releases to use when moving data between different releases of the Oracle database.

Table 20–7 Using Different Releases of Export and Import

Export from->Import to	Use Export Release	Use Import Release
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
9.0.1 -> 8.1.6	8.1.6	8.1.6
9.0.1 -> 9.0.2	9.0.1	9.0.2
9.0.2 -> 10.1.0	9.0.2	10.1.0
10.1.0 -> 9.0.2	9.0.2	9.0.2

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.
- Function-based indexes 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 are not 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 CLOBs, 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.

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, "Backus-Naur Form Syntax"

This appendix explains the symbols and conventions of the BNF variant used in text descriptions of the syntax diagrams.

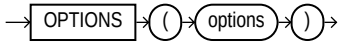
A

SQL*Loader Syntax Diagrams

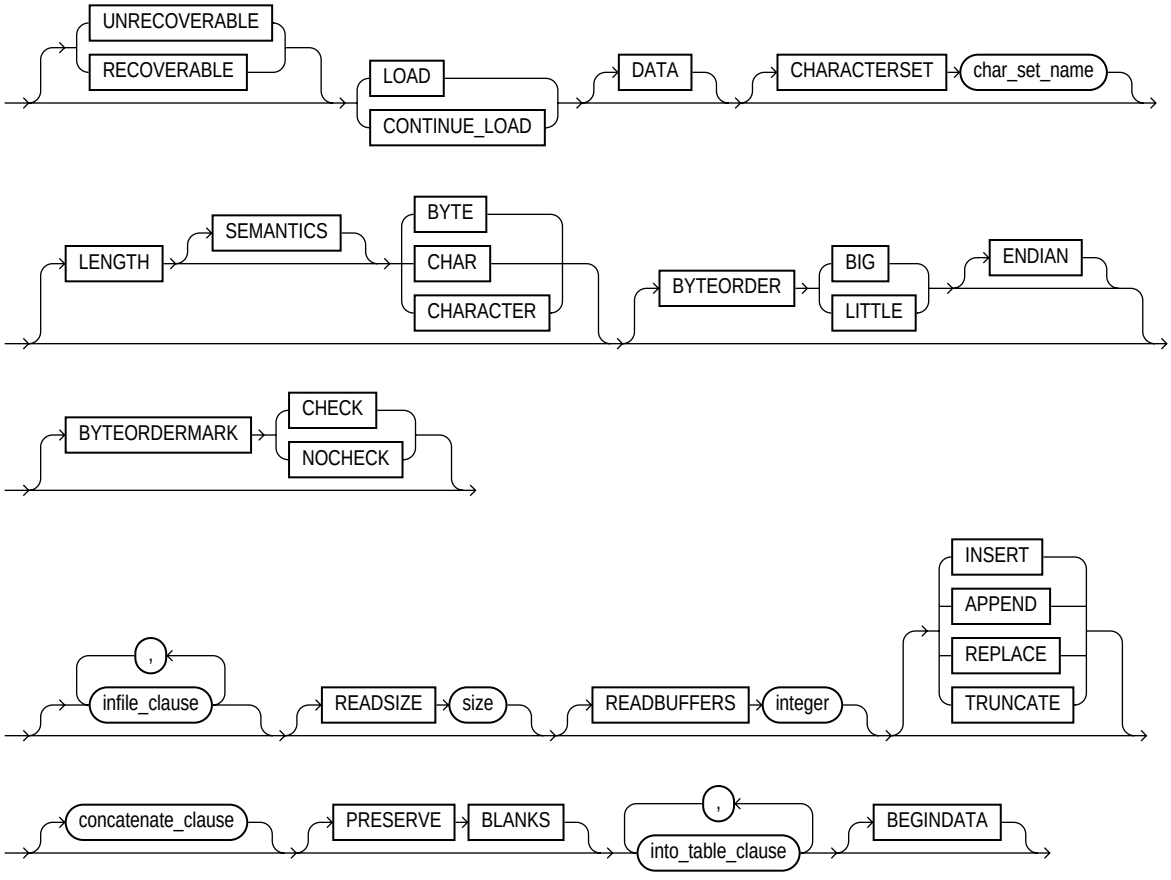
The SQL*Loader syntax diagrams (sometimes called railroad diagrams or DDL diagrams) use standard SQL syntax notation. For more information about the syntax notation used in this appendix, see the *Oracle Database SQL Reference*.

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.

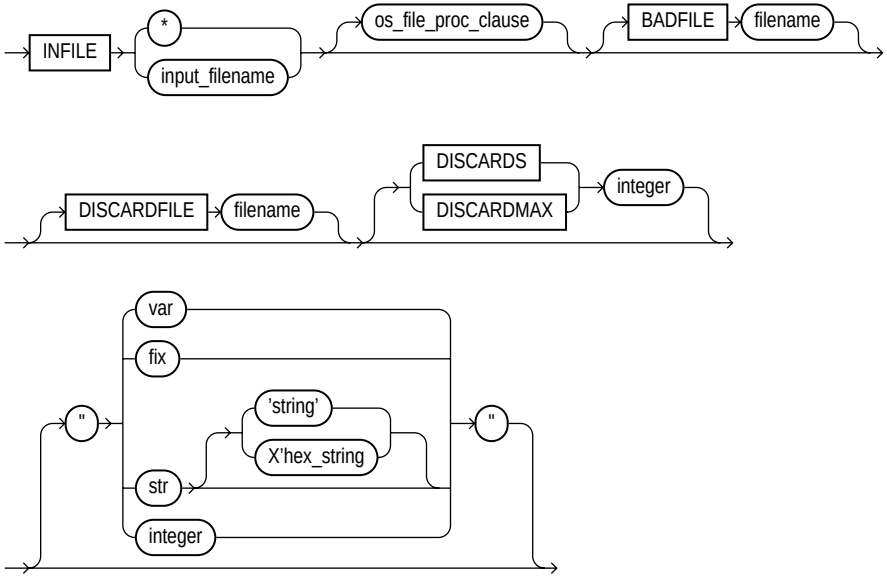
Options Clause



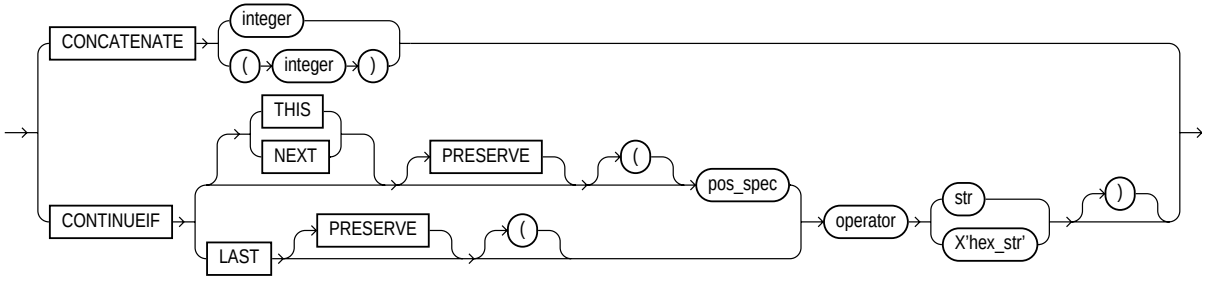
Load Statement



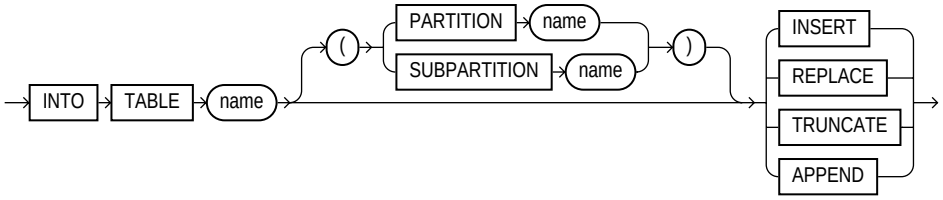
infile_clause

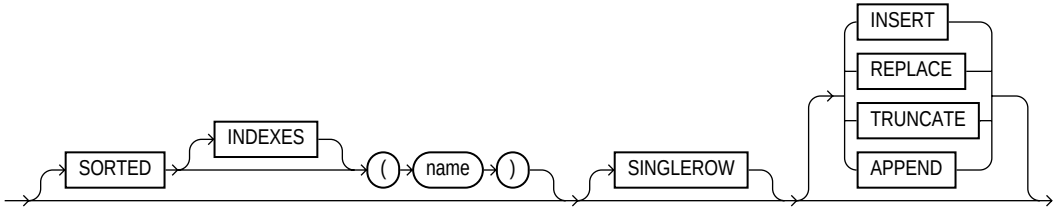


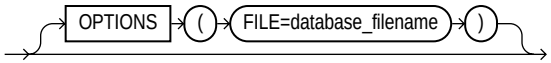
concatenate_clause

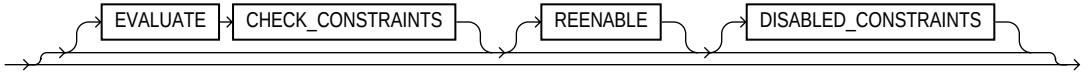


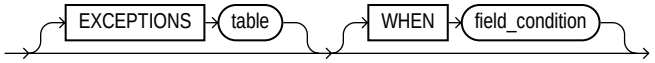
into_table_clause

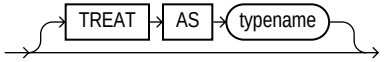


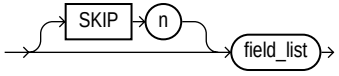




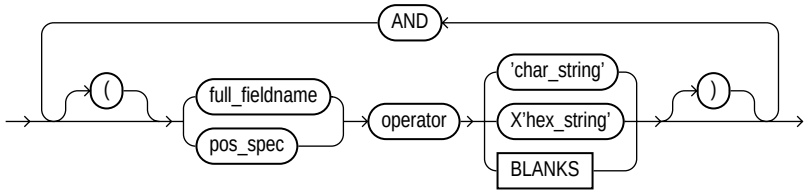




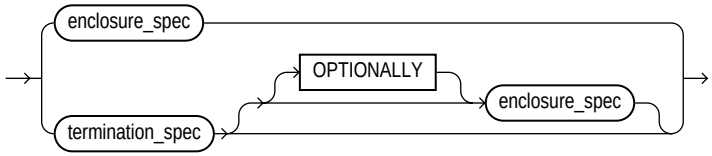




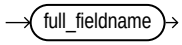
field_condition



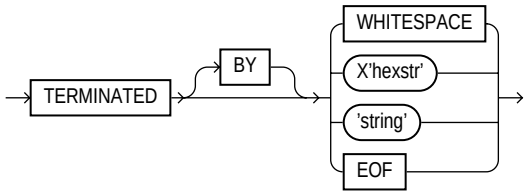
delim_spec



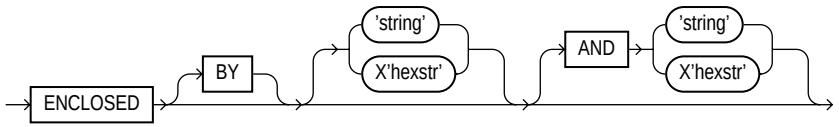
full_fieldname



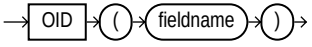
termination_spec



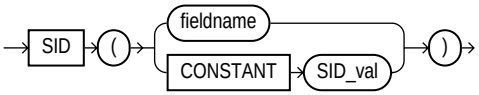
enclosure_spec



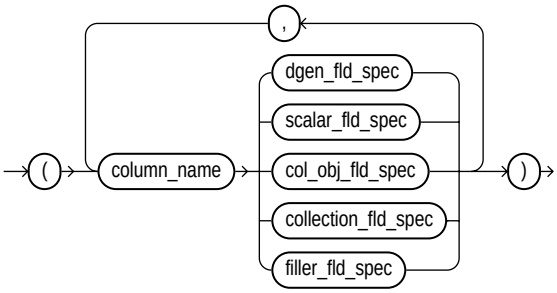
oid_spec



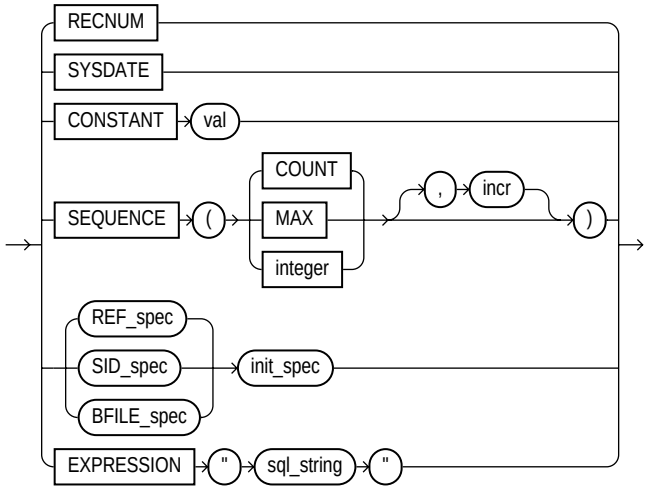
sid_spec



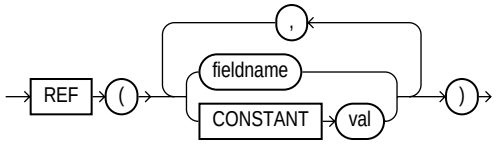
field_list



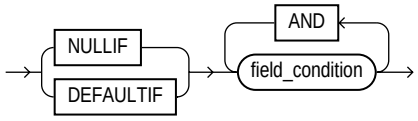
dgen_fld_spec



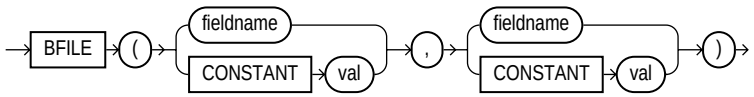
ref_spec



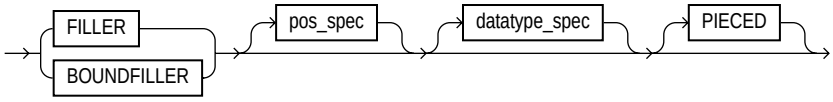
init_spec



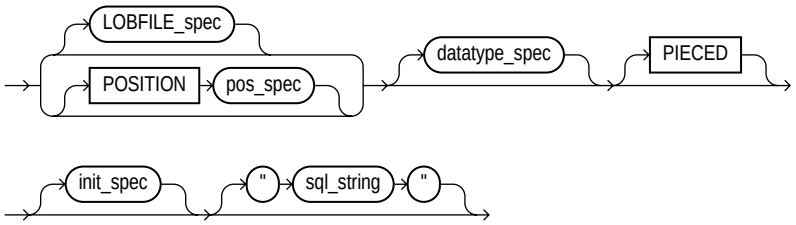
bfile_spec



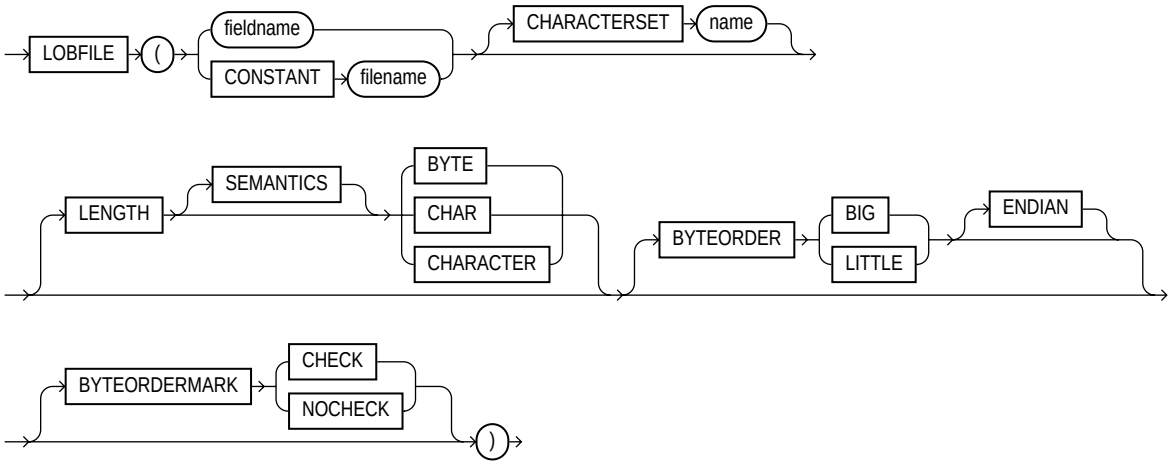
filler_fld_spec



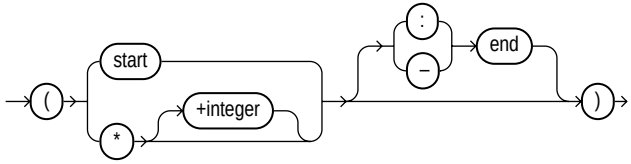
scalar_fld_spec



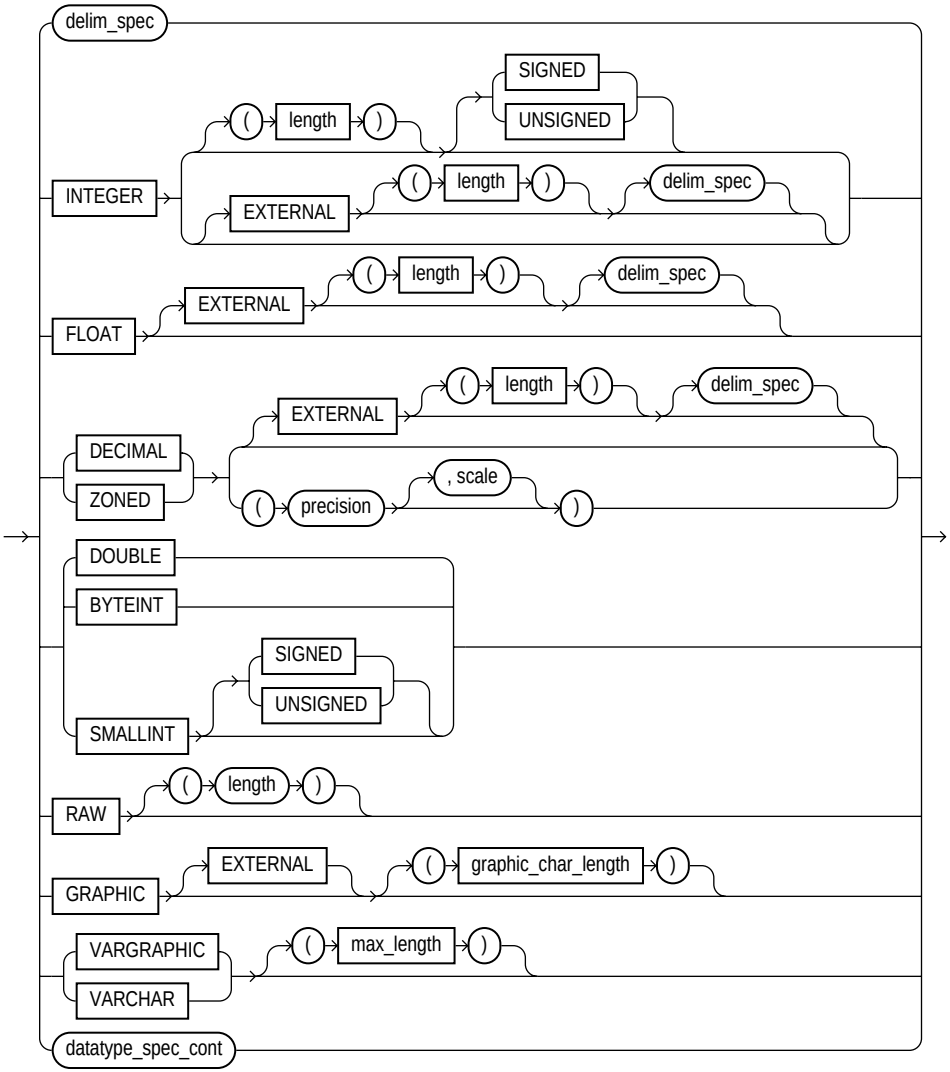
lobfile_spec



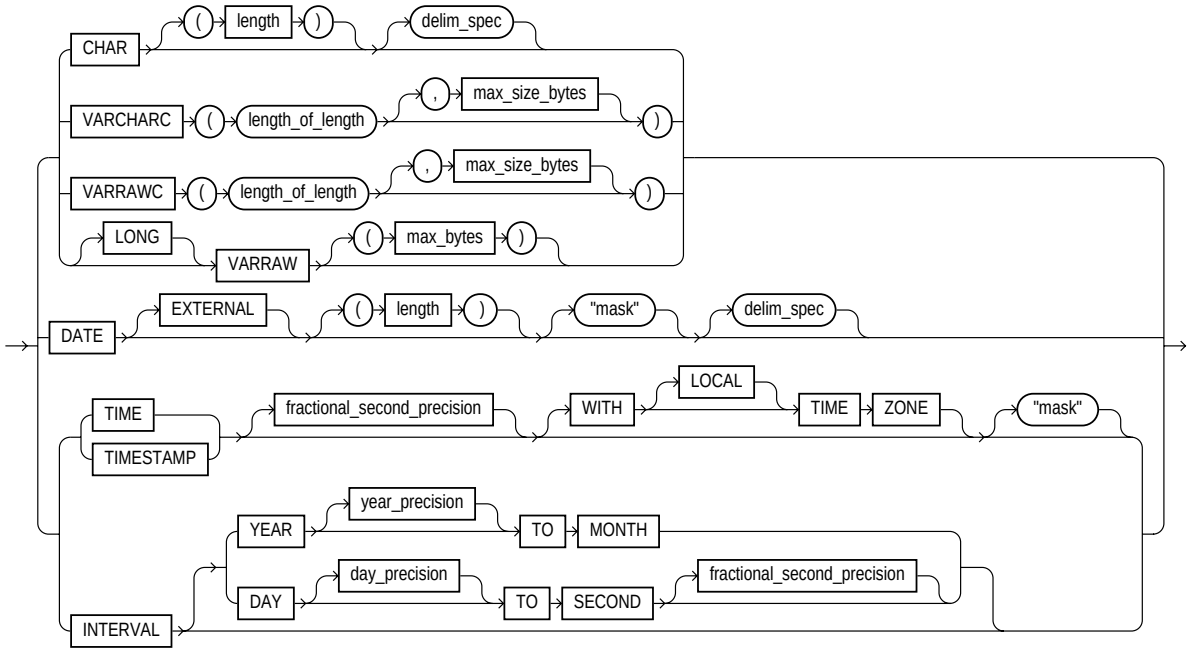
pos_spec



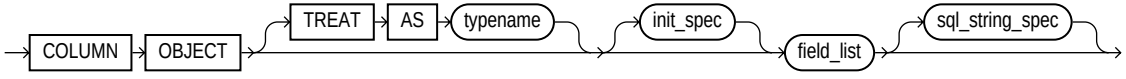
datatype_spec



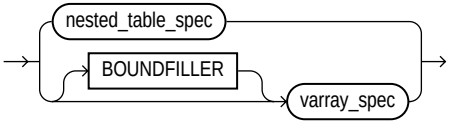
datatype_spec_cont



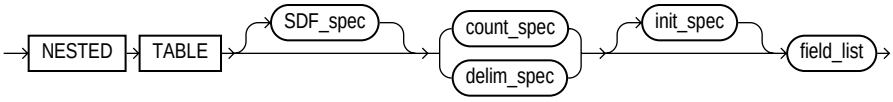
col_obj_fld_spec



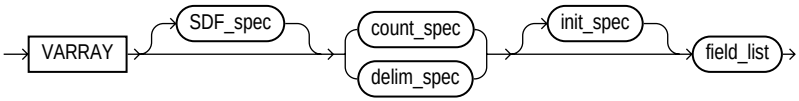
collection_fld_spec



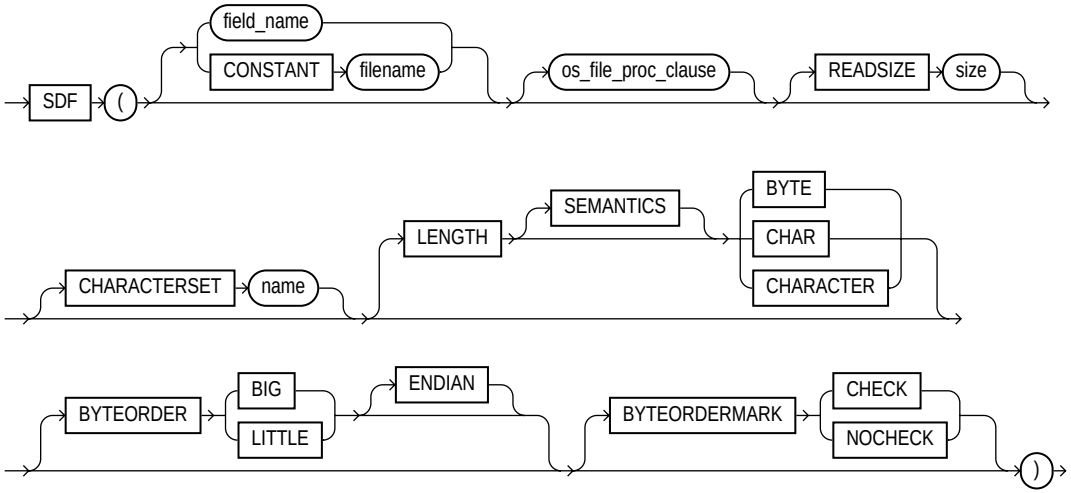
nested_table_spec



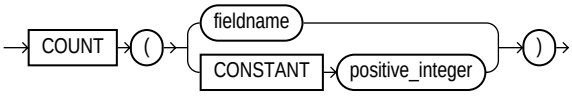
varray_spec



sdf_spec



count_spec



B

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 B–1.

Table B–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

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z