Oracle® Database

Utilities

19c

E96081-29

January 2025

Oracle Database Utilities, 19c

E96081-29

Copyright © 2002, 2025, Oracle and/or its affiliates.

Primary Author: Douglas Williams

Contributors: William Beauregard, Steve DiPirro, John Kalogeropoulos, Rod Payne, Rich Phillips, Mike Sakayeda, Jim Stenoish, Roy Swonger

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software, software documentation, data (as defined in the Federal Acquisition Regulation), or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs) and Oracle computer documentation or other Oracle data delivered to or accessed by U.S. Government end users are "commercial computer software," "commercial computer software documentation," or "limited rights data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, reproduction, duplication, release, display, disclosure, modification, preparation of derivative works, and/or adaptation of i) Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs), ii) Oracle computer documentation and/or iii) other Oracle data, is subject to the rights and limitations specified in the license contained in the applicable contract. The terms governing the U.S. Government's use of Oracle cloud services are defined by the applicable contract for such services. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle®, Java, MySQL, and NetSuite are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Inside are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Epyc, and the AMD logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

Contents

2 Oracle Data Pump Export

8 SQL*Loader Command-Line Reference

9 SQL*Loader Control File Reference

10 SQL*Loader Field List Reference

13 SQL*Loader Express

20 ADRCI: ADR Command Interpreter

21 DBVERIFY: Offline Database Verification Utility

22 DBNEWID Utility

23 Using LogMiner to Analyze Redo Log Files

24 Using the Metadata APIs

25 Original Export

26 Original Import

List of Examples

List of Figures

List of Tables

Preface

This document describes how to use Oracle Database utilities for data transfer, data maintenance, and database administration.

Audience

The utilities described in this book are intended for database administrators (DBAs), application programmers, security administrators, system operators, and other Oracle Database users who perform the following tasks:

- Archive data, back up Oracle Database, or move data between different Oracle Databases using the Export and Import utilities (both the original versions and the Oracle Data Pump versions)
- Load data into Oracle Database tables from operating system files, using SQL*Loader
- Load data 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 Oracle Database objects, using the Metadata API
- Query and analyze redo log files (through a SQL interface), using the LogMiner utility
- Use the Automatic Diagnostic Repository Command Interpreter (ADRCI) utility to manage Oracle Database diagnostic data

To use this manual, you need a working knowledge of SQL and of Oracle fundamentals. You can find such information in *Oracle Database Concepts*. In addition, to use SQL*Loader, you must know how to use the file management facilities of your operating system.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Diversity and Inclusion

Oracle is fully committed to diversity and inclusion. Oracle respects and values having a diverse workforce that increases thought leadership and innovation. As part of our initiative to build a more inclusive culture that positively impacts our employees, customers, and partners, we are working to remove insensitive terms from our products and documentation. We are also mindful of the necessity to maintain compatibility with our customers' existing technologies and the need to ensure continuity of service as Oracle's offerings and industry standards evolve. Because of these technical constraints, our effort to remove insensitive terms is ongoing and will take time and external cooperation.

Related Documentation

For more information, refer to the Oracle Database documentation set. In particular, check the following documents:

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

Also refer to My Oracle Support notes that are relevant to Oracle Data Pump tasks, and in particular, refer to recommended proactive patches for your release:

Data Pump Recommended Proactive Patches For 19.10 and Above (Doc ID 2819284.1)

Oracle Data Pump patches are not included in Oracle Database release updates, but instead are provide in bundled patches that contain SQL, PL/SQL packages, and XML stylesheets for Oracle Data Pump. Oracle recommends that you apply the most recent Oracle Data Pump bundle patch for your release. Because these patches do not include Oracle Database binaries, you can apply Oracle Data Pump patches online while the database is running , so long as Oracle Data Pump is not in use at the time.

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.

Syntax Diagrams

Syntax descriptions are provided in this book for various SQL, PL/SQL, or other command-line constructs in graphic form or Backus Naur Form (BNF). See *Oracle Database SQL Language Reference* for information about how to interpret these descriptions.

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

Part I
Oracle Data Pump

Learn about data movement options using Oracle Data Pump Export, Oracle Data Pump Import, legacy mode, performance, and the Oracle Data Pump API `DBMS_DATAPUMP`.

- Overview of Oracle Data Pump
  Oracle Data Pump technology enables very high-speed movement of data and metadata from one database to another.
- Oracle Data Pump Export
  The Oracle Data Pump Export utility is used to unload data and metadata into a set of operating system files, which are called a dump file set.
- Oracle Data Pump Import
  With Oracle Data Pump Import, you can load an export dump file set into a target database, or load a target database directly from a source database with no intervening files.
- Oracle Data Pump Legacy Mode
  With Oracle Data Pump legacy mode, you can use original Export and Import parameters on the Oracle Data Pump Export and Data Pump Import command lines.
- Oracle Data Pump Performance
  Learn how Oracle Data Pump Export and Import is better than that of original Export and Import, and how to enhance performance of export and import operations.
- Using the Oracle Data Pump API
  You can automate data movement operations by using the Oracle Data Pump PL/SQL API `DBMS_DATAPUMP`.

1

Overview of Oracle Data Pump

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

An understanding of the following topics can help you to successfully use Oracle Data Pump to its fullest advantage:

- Oracle Data Pump Components
  Oracle Data Pump is made up of three distinct components: Command-line clients, `expdp` and `impdp`; the `DBMS_DATAPUMP` PL/SQL package (also known as the Data Pump API); and the `DBMS_METADATA` PL/SQL package (also known as the Metadata API).
- How Does Oracle Data Pump Move Data?
  There are several Oracle Data Pump methods that you can use to move data in and out of databases. You can select the method that best fits your use case.
- Using Oracle Data Pump With CDBs
  Oracle Data Pump can migrate all, or portions of, a database from a non-CDB into a PDB, between PDBs within the same or different CDBs, and from a PDB into a non-CDB.
- Cloud Premigration Advisor Tool
  The Cloud Premigration Advisor tool can assist you to migrate a database to the Oracle Cloud.
- Required Roles for Oracle Data Pump Export and Import Operations
  The roles `DATAPUMP_EXP_FULL_DATABASE` and `DATAPUMP_IMP_FULL_DATABASE` are required for many Export and Import operations.
- What Happens During the Processing of an Oracle Data Pump Job?
  Oracle Data Pump jobs use a Data Pump control job table, a Data Pump control job process, and worker processes to perform the work and keep track of progress.
- How to Monitor Status of Oracle Data Pump Jobs
  The Oracle Data Pump Export and Import client utilities can attach to a job in either logging mode or interactive-command mode.
- How to Monitor the Progress of Running Jobs with V$SESSION_LONGOPS
  To monitor table data transfers, you can use the `V$SESSION_LONGOPS` dynamic performance view to monitor Oracle Data Pump jobs.
- File Allocation with Oracle Data Pump
  You can modify how Oracle Data Pump allocates and handles files by using commands in interactive mode.
- Exporting and Importing Between Different Oracle Database Releases
  You can use Oracle Data Pump to migrate all or any portion of an Oracle Database between different releases of the database software.
- Managing SecureFiles Large Object Exports with Oracle Data Pump
  Exports of SecureFiles large objects (LOBs) are affected by the content type, the `VERSION` parameter, and other variables.
- Oracle Data Pump Process Exit Codes
  To check the status of your Oracle Data Pump export and import operations, review the process exit codes in the log file.

- How to Monitor Oracle Data Pump Jobs with Unified Auditing
  To monitor and record specific user database actions, perform auditing on Data Pump jobs with unified auditing.
- Encrypted Data Security Warnings for Oracle Data Pump Operations
  Oracle Data Pump warns you when encrypted data is exported as unencrypted data.
- How Does Oracle Data Pump Handle Timestamp Data?
  Learn about factors that can affect successful completion of export and import jobs that involve the timestamp data types `TIMESTAMP WITH TIMEZONE` and `TIMESTAMP WITH LOCAL TIMEZONE`.
- Character Set and Globalization Support Considerations
  Learn about Globalization support of Oracle Data Pump Export and Import using character set conversion of user data, and data definition language (DDL).
- Oracle Data Pump Behavior with Data-Bound Collation
  Oracle Data Pump supports data-bound collation (DBC).

1.1 Oracle Data Pump Components

Oracle Data Pump is made up of three distinct components: Command-line clients, `expdp` and `impdp`; the `DBMS_DATAPUMP` PL/SQL package (also known as the Data Pump API); and the `DBMS_METADATA` PL/SQL package (also known as the Metadata API).

The Oracle Data Pump clients, `expdp` and `impdp`, start the Oracle Data Pump Export utility and Oracle Data Pump Import utility, respectively.

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 for subsets of a database.

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 re-creation of dictionary metadata.

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

> **Note:**
>
> All Oracle Data Pump Export and Import processing, including the reading and writing of dump files, is done on the system (server) selected by the specified database connect string. **This means that for unprivileged users, the database administrator (DBA) must create directory objects for the Data Pump files that are read and written on that server file system.** (For security reasons, DBAs must ensure that only approved users are allowed access to directory objects.) For privileged users, a default directory object is available.

Starting with Oracle Database 18c, you can include the unified audit trail in either full or partial export and import operations using Oracle Data Pump. There is no change to the user interface. When you perform the export or import operations of a database, the unified audit trail is automatically included in the Oracle Data Pump dump files. See *Oracle Database PL/SQL Packages and Types Reference* for a description of the `DBMS_DATAPUMP` and the

`DBMS_METADATA` packages. See *Oracle Database Security Guide* for information about exporting and importing the unified audit trail using Oracle Data Pump.

Related Topics

- Understanding Dump, Log, and SQL File Default Locations
  Oracle Data Pump is server-based, rather than client-based. Dump files, log files, and SQL files are accessed relative to server-based directory paths.
- *Oracle Database PL/SQL Packages and Types Reference*
- *Oracle Database Security Guide*

1.2 How Does Oracle Data Pump Move Data?

There are several Oracle Data Pump methods that you can use to move data in and out of databases. You can select the method that best fits your use case.

> **Note:**
>
> The `UTL_FILE_DIR` desupport in Oracle Database 18c and later releases affects Oracle Data Pump. This desupport can affect any feature from an earlier release using symbolic links, including (but not restricted to) Oracle Data Pump, BFILEs, and External Tables. If you attempt to use an affected feature configured with symbolic links, then you encounter `ORA-29283: invalid file operation: path traverses a symlink`. Oracle recommends that you instead use directory objects in place of symbolic links.
>
> Data Pump does not load tables with disabled unique indexes. To load data into the table, the indexes must be either dropped or reenabled.

- Using Data File Copying to Move Data
  The fastest method of moving data is to copy the database data files to the target database without interpreting or altering the data. With this method, Data Pump Export is used to unload only structural information (metadata) into the dump file.
- Using Direct Path to Move Data
  After data file copying, direct path is the fastest method of moving data. In this method, the SQL layer of the database is bypassed and rows are moved to and from the dump file with only minimal interpretation.
- Using External Tables to Move Data
  If you do not select data file copying, and the data cannot be moved using direct path, you can use the external tables mechanism.
- Using Conventional Path to Move Data
  Where there are conflicting table attributes, Oracle Data Pump uses conventional path to move data.
- Using Network Link Import to Move Data
  When the Import `NETWORK_LINK` parameter is used to specify a network link for an import operation, the direct path method is used by default. Review supported database link types.
- Using a Parameter File (Parfile) with Oracle Data Pump
  To help to simplify Oracle Data Pump exports and imports, you can create a **parameter** file, also known as a **parfile**.

1.2.1 Using Data File Copying to Move Data

The fastest method of moving data is to copy the database data files to the target database without interpreting or altering the data. With this method, Data Pump Export is used to unload only structural information (metadata) into the dump file.

- The `TRANSPORT_TABLESPACES` parameter is used to specify a transportable tablespace export. Only metadata for the specified tablespaces is exported.
- The `TRANSPORTABLE=ALWAYS` parameter is supplied on a table mode export (specified with the `TABLES` parameter) or a full mode export (specified with the `FULL` parameter) or a full mode network import (specified with the `FULL` and `NETWORK_LINK` parameters).

When an export operation uses data file copying, the corresponding import job always also uses data file copying. During the ensuing import operation, both the data files and the export dump file must be loaded.

> **Note:**
>
> During transportable imports tablespaces are temporarily made read/write and then set back to read-only.The temporary setting change was introduced with Oracle Database 12*c* Release 1 (12.1.0.2) to improve performance. However, be aware that this behavior also causes the SCNs of the import job data files to change. Changing the SCNs for data files can cause issues during future transportable imports of those files.
>
> For example, if a transportable tablespace import fails at any point after the tablespaces have been made read/write (even if they are now read-only again), then the data files become corrupt. *They cannot be recovered.*
>
> Because transportable jobs are not restartable, you must restart the failed job from the beginning. You must delete the corrupt datafiles, and copy fresh versions to the target destination.
>
> When transportable jobs are performed, it is best practice to keep a copy of the data files on the source system until the import job has successfully completed on the target system. If the import job fails for some reason, then keeping copies ensures that you can have uncorrupted copies of the data files.

When data is moved by using data file copying, there are some limitations regarding character set compatibility between the source and target databases.

If the source platform and the target platform are of different endianness, then you must convert the data being transported so that it is in the format of the target platform. You can use the `DBMS_FILE_TRANSFER` PL/SQL package or the `RMAN CONVERT` command to convert the data.

> **See Also:**
>
> - *Oracle Database Backup and Recovery Reference* for information about the RMAN `CONVERT` command
> - *Oracle Database Administrator's Guide* for a description and example (including how to convert the data) of transporting tablespaces between databases

1.2.2 Using Direct Path to Move Data

After data file copying, direct path is the fastest method of moving data. In this method, the SQL layer of the database is bypassed and rows are moved to and from the dump file with only minimal interpretation.

Data Pump automatically uses the direct path method for loading and unloading data unless the structure of a table does not allow it. For example, if a table contains a column of type `BFILE`, then direct path cannot be used to load that table and external tables is used instead.

The following sections describe situations in which direct path cannot be used for loading and unloading.

Situations in Which Direct Path Load Is Not Used

If any of the following conditions exist for a table, then Data Pump uses external tables to load the data for that table, instead of direct path:

- A domain index that is not a `CONTEXT` type index exists for a LOB column.
- A global index on multipartition tables exists during a single-partition load. This case includes object tables that are partitioned.
- A table is in a cluster.
- There is an active trigger on a preexisting table.
- Fine-grained access control is enabled in insert mode on a preexisting table.
- A table contains `BFILE` columns or columns of opaque types.
- A referential integrity constraint is present on a preexisting table.
- A table contains `VARRAY` columns with an embedded opaque type.
- The table has encrypted columns.
- The table into which data is being imported is a preexisting table and at least one of the following conditions exists:
  - There is an active trigger
  - The table is partitioned
  - Fine-grained access control is in insert mode
  - A referential integrity constraint exists
  - A unique index exists
- Supplemental logging is enabled, and the table has at least one LOB column.
- The Data Pump command for the specified table used the `QUERY`, `SAMPLE`, or `REMAP_DATA` parameter.

- A table contains a column (including a `VARRAY` column) with a `TIMESTAMP WITH TIME ZONE` data type, and the version of the time zone data file is different between the export and import systems.

Situations in Which Direct Path Unload Is Not Used

If any of the following conditions exist for a table, then Data Pump uses external tables rather than direct path to unload the data:

- Fine-grained access control for `SELECT` is enabled.
- The table is a queue table.
- The table contains one or more columns of type `BFILE` or opaque, or an object type containing opaque columns.
- The table contains encrypted columns.
- The table contains a column of an evolved type that needs upgrading.
- The Data Pump command for the specified table used the `QUERY`, `SAMPLE`, or `REMAP_DATA` parameter.
- Before the unload operation, the table was altered to contain a column that is NOT NULL, and also has a default value specified.

1.2.3 Using External Tables to Move Data

If you do not select data file copying, and the data cannot be moved using direct path, you can use the external tables mechanism.

The external tables mechanism creates an external table that maps to the dump file data for the database table. The SQL engine is then used to move the data. If possible, use the `APPEND` hint on import to speed the copying of the data into the database. The representation of data for direct path data and external table data is the same in a dump file. Because they are the same, Oracle Data Pump can use the direct path mechanism at export time, but use external tables when the data is imported into the target database. Similarly, Oracle Data Pump can use external tables for the export, but use direct path for the import.

In particular, Oracle Data Pump can use external tables in the following situations:

- Loading and unloading very large tables and partitions in situations where it is advantageous to use parallel SQL capabilities
- 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 a table not created by the import operation (the table exists before the import starts)

> **Note:**
>
> When Oracle Data Pump uses external tables as the data access mechanism, it uses the `ORACLE_DATAPUMP` access driver. However, be aware that the files that Oracle Data Pump creates when it uses external tables are not compatible with files created when you manually create an external table using the SQL `CREATE TABLE ... ORGANIZATION EXTERNAL` statement.

Related Topics

- The ORACLE_DATAPUMP Access Driver
- APPEND Hint
- Loading LOBs with External Tables

1.2.4 Using Conventional Path to Move Data

Where there are conflicting table attributes, Oracle Data Pump uses conventional path to move data.

In situations where there are conflicting table attributes, Oracle Data Pump is not able to load data into a table using either direct path or external tables. In such cases, conventional path is used, which can affect performance.

1.2.5 Using Network Link Import to Move Data

When the Import `NETWORK_LINK` parameter is used to specify a network link for an import operation, the direct path method is used by default. Review supported database link types.

If direct path cannot be used (for example, because one of the columns is a `BFILE`), then SQL is used to move the data using an `INSERT SELECT` statement. (Before Oracle Database 12c Release 2 (12.2.0.1), the default was to use the `INSERT SELECT` statement.) The `SELECT` clause retrieves the data from the remote database over the network link. The `INSERT` clause uses SQL to insert the data into the target database. There are no dump files involved.

When the Export `NETWORK_LINK` parameter is used to specify a network link for an export operation, the data from the remote database is written to dump files on the target database. (Note that to export from a read-only database, the `NETWORK_LINK` parameter is required.)

Because the link can identify a remotely networked database, the terms database link and network link are used interchangeably.

Supported Link Types

The following types of database links are supported for use with Data Pump Export and Import:

- Public fixed user
- Public connected user
- Public shared user (only when used by link owner)
- Private shared user (only when used by link owner)
- Private fixed user (only when used by link owner)

Unsupported Link Types

The following types of database links are not supported for use with Data Pump Export and Import:

- Private connected user
- Current user
- Parallel export or import of metadata for network jobs.

  For conventional jobs, if you need parallel metadata import, then use a dumpfile instead of `NETWORK_LINK`.

> **See Also:**
>
> - The Export NETWORK_LINK parameter for information about performing exports over a database link
> - The Import NETWORK_LINK parameter for information about performing imports over a database link
> - *Oracle Database Administrator's Guide* for information about creating database links and the different types of links

1.2.6 Using a Parameter File (Parfile) with Oracle Data Pump

To help to simplify Oracle Data Pump exports and imports, you can create a **parameter** file, also known as a **parfile**.

Instead of typing in Oracle Data Pump parameters at the command line, when you run an export or import operation, you can prepare a parameter text file (also known as a parfile, after the parameter name) that provides the command-line parameters to the Oracle Data Pump client. You specify that Oracle Data Pump obtains parameters for the command by entering the `PARFILE` parameter, and then specifying the parameter name:

```
PARFILE=[directory_path]file_name
```

When the Oracle Data Pump Export or Import operation starts, the parameter file is opened and read by the client. The default location of the parameter file is the user's current directory.

For example:

```
expdp hr PARFILE=hr.par
```

When you create a parameter file, it makes it easier for you to reuse that file for multiple export or import operations, which can simplify these operations, particularly if you perform them regularly. Creating a parameter file also helps you to avoid typographical errors that can occur from typing long Oracle Data Pump commands on the command line, especially if you use parameters whose values require quotation marks that must be placed precisely. On some systems, if you use a parameter file and the parameter value being specified does not have quotation marks as the first character in the string (for example, `TABLES=scott."Emp"`), then the use of escape characters may not be necessary.

There is no required file name extension, but Oracle examples use `.par` as the extension. Oracle recommends that you also use this file extension convention. Using a consistent parameter file extension makes it easier to identify and use these files.

> **Note:**
>
> The `PARFILE` parameter cannot be specified within a parameter file.

For more information and examples, see the `PARFILE` parameters for Oracle Data Pump Import and Export.

Related Topics

- Oracle Data Pump Export PARFILE
- Oracle Data Pump Import PARFILE

1.3 Using Oracle Data Pump With CDBs

Oracle Data Pump can migrate all, or portions of, a database from a non-CDB into a PDB, between PDBs within the same or different CDBs, and from a PDB into a non-CDB.

- About Using Oracle Data Pump in a Multitenant Environment
  In general, using Oracle Data Pump with PDBs is identical to using Oracle Data Pump with a non-CDB.
- Using Oracle Data Pump to Move Data Into a CDB
  After you create an empty PDB, to move data into the PDB, you can use an Oracle Data Pump full-mode export and import operation.
- Using Oracle Data Pump to Move PDBs Within or Between CDBs
  Learn how to avoid `ORA-65094` user schema errors with Oracle Data Pump export and import operations on PDBs.

1.3.1 About Using Oracle Data Pump in a Multitenant Environment

In general, using Oracle Data Pump with PDBs is identical to using Oracle Data Pump with a non-CDB.

A multitenant container database (CDB) is an Oracle Database that includes zero, one, or many user-created pluggable databases (PDBs). A PDB is a portable set of schemas, schema objects, and non-schema objects that appear to an Oracle Net client as a non-CDB. A non-CDB is an Oracle Database that is not a CDB. Non-CDB architecture Oracle Database was deprecated in Oracle Database 12c Release 1 (12.1). Starting with Oracle Database 21c, non-CDB architecture deployments are desupported.

You can use Oracle Data Pump to migrate all or some of a database in the following scenarios:

- From a non-CDB into a PDB
- Between PDBs within the same or different CDBs
- From a PDB into an earlier release non-CDB

> **Note:**
>
> Oracle Data Pump does not support any operations across the entire CDB. If you are connected to the root or seed database of a CDB, then Oracle Data Pump issues the following warning:
>
> ```
> ORA-39357: Warning: Oracle Data Pump operations are not typically
> needed when connected to the root or seed of a container database.
> ```

1.3.2 Using Oracle Data Pump to Move Data Into a CDB

After you create an empty PDB, to move data into the PDB, you can use an Oracle Data Pump full-mode export and import operation.

You can import data with or without the transportable option. If you use the transportable option on a full mode export or import, then it is referred to as a full transportable export/import.

When the transportable option is used, export and import use both transportable tablespace data movement and conventional data movement; the latter for those tables that reside in non-transportable tablespaces such as `SYSTEM` and `SYSAUX`. Using the transportable option can reduce the export time, and especially, the import time. With the transportable option, table data does not need to be unloaded and reloaded, and index structures in user tablespaces do not need to be recreated.

Note the following requirements when using Oracle Data Pump to move data into a CDB:

- To administer a multitenant environment, you must have the `CDB_DBA` role.
- Full database exports from Oracle Database 11.2.0.2 and earlier can be imported into Oracle Database 12c or later (CDB or non-CDB). However, Oracle recommends that you first upgrade the source database to Oracle Database 11g Release 2 (11.2.0.3 or later), so that information about registered options and components is included in the export.
- When migrating Oracle Database 11g Release 2 (11.2.0.3 or later) to a CDB (or to a non-CDB) using either full database export or full transportable database export, you must set the Oracle Data Pump Export parameter at least to `VERSION=12` to generate a dump file that is ready for import into an Oracle Database 12c or later release. If you do not set `VERSION=12`, then the export file that is generated does not contain complete information about registered database options and components.
- Network-based full transportable imports require use of the `FULL=YES`, `TRANSPORTABLE=ALWAYS`, and `TRANSPORT_DATAFILES=`*`datafile_name`* parameters. When the source database is Oracle Database 11g Release 11.2.0.3 or later, but earlier than Oracle Database 12c Release 1 (12.1), the `VERSION=12` parameter is also required.
- File-based full transportable imports only require use of the `TRANSPORT_DATAFILES=`*`datafile_name`* parameter. Data Pump Import infers the presence of the `TRANSPORTABLE=ALWAYS` and `FULL=YES` parameters.
- As of Oracle Database 12c Release 2 (12.2), in a multitenant container database (CDB) environment, the default Oracle Data Pump directory object, `DATA_PUMP_DIR`, is defined as a unique path for each PDB in the CDB. This unique path is defined whether the `PATH_PREFIX` clause of the `CREATE PLUGGABLE DATABASE` statement is defined or is not defined for relative paths.

- Starting in Oracle Database 19c, the `credential` parameter of `impdp` specifies the name of the credential object that contains the user name and password required to access an object store bucket. You can also specify a default credential using the PDB property named `DEFAULT_CREDENTIAL`. When you run `impdb` with then default credential, you prefix the dump file name with `DEFAULT_CREDENTIAL:` and you do not specify the `credential` parameter.

Example 1-1 Importing a Table into a PDB

To specify a particular PDB for the export/import operation, supply a connect identifier in the connect string when you start Data Pump. For example, to import data to a PDB named `pdb1`, you could enter the following on the Data Pump command line:

```
impdp hr@pdb1 DIRECTORY=dpump_dir1 DUMPFILE=hr.dmp TABLES=employees
```

Example 1-2 Specifying a Credential When Importing Data

This example assumes that you created a credential named `HR_CRED` using `DBMS_CREDENTIAL.CREATE_CREDENTIAL` as follows:

```
BEGIN
  DBMS_CLOUD.CREATE_CREDENTIAL(
    credential_name => 'HR_CRED',
    username => 'atpc_user@example.com',
    password => 'password'
  );
END;
/
```

The following command specifies credential `HR_CRED`, and specifies the file stored in an object store. The URL of the file is `https://example.com/ostore/dnfs/myt.dmp`.

```
impdp hr@pdb1 \
      table_exists_action=replace \
      credential=HR_CRED \
      parallel=16 \
      dumpfile=https://example.com/ostore/dnfs/myt.dmp
```

Example 1-3 Importing Data Using a Default Credential

1. You create a credential named `HR_CRED` using `DBMS_CREDENTIAL.CREATE_CREDENTIAL` as follows:

   ```
   BEGIN
     DBMS_CLOUD.CREATE_CREDENTIAL(
       credential_name => 'HR_CRED',
       username => 'atpc_user@example.com',
       password => 'password'
     );
   END;
   /
   ```

2. You set the PDB property `DEFAULT_CREDENTIAL` as follows:

```
ALTER DATABASE PROPERTY SET DEFAULT_CREDENTIAL = 'ADMIN.HR_CRED'
```

3. The following command specifies the default credential as a prefix to the dump file location `https://example.com/ostore/dnfs/myt.dmp`:

```
impdp hr@pdb1 \
      table_exists_action=replace \
      parallel=16 \
      dumpfile=default_credential:https://example.com/ostore/dnfs/myt.dmp
```

Note that the `credential` parameter is not specified.

> **See Also:**
>
> - *Oracle Database Security Guide* to learn how to configure SSL authentication, which is necessary for object store access
> - Importing a Table to an Object Store Using Oracle Data Pump to learn about using Oracle Data Pump Import to load files to the object store

1.3.3 Using Oracle Data Pump to Move PDBs Within or Between CDBs

Learn how to avoid `ORA-65094` user schema errors with Oracle Data Pump export and import operations on PDBs.

Export and import operations on PDBs are identical to those on non-CDBs, with the exception of how common users are handled.

If you create a common user in a CDB, then a full database or privileged schema export of that user from within any PDB in the CDB results in a standard `CREATE USER C##common name` DDL statement being performed upon import. However, the statement fails because of the common user prefix `C##` on the user name. The following error message is returned:

```
ORA-65094:invalid local user or role name
```

Example 1-4 Avoiding Invalid Local User Error

In the PDB being exported, if you have created local objects in that user's schema, and you want to import them, then either make sure a common user of the same name already exists in the target CDB instance, or use the Oracle Data Pump Import `REMAP_SCHEMA` parameter on the `impdp` command to remap the schema to a valid local user. For example:

```
REMAP_SCHEMA=C##common name:local user name
```

Related Topics

- Full Mode
  You can use Data Pump to carry out a full database export by using the `FULL` parameter.

- Full Import Mode
  To specify a full import with Oracle Data Pump, use the `FULL` parameter.
- Network Considerations for Original Oracle Data Pump Export
  When you use original Export (`exp`) across a network, review protocols and connection qualifier strings.

1.4 Cloud Premigration Advisor Tool

The Cloud Premigration Advisor tool can assist you to migrate a database to the Oracle Cloud.

- What is the Cloud Premigration Advisor Tool (CPAT)
  To determine if your On Premises Oracle Database data is suitable to migrate to an Oracle Cloud, you can use Oracle's Cloud Premigration Advisor Tool (CPAT).

1.4.1 What is the Cloud Premigration Advisor Tool (CPAT)

To determine if your On Premises Oracle Database data is suitable to migrate to an Oracle Cloud, you can use Oracle's Cloud Premigration Advisor Tool (CPAT).

The Cloud Premigration Advisor Tool (CPAT) is a Java application that assists you to analyze your On Premises Oracle Databases to determine whether you can migrate some or all of that database to one of the Oracle Cloud platform options, such as Autonomous Database, or other Cloud database options. The CPAT assists you to evaluate your specific migration scenario, to identify migration options, and assist you to prepare your migration plans from source On Premises Oracle Databases to the target Oracle Cloud database option to which you want to migrate.

How the CPAT Helps You to Avoid Issues

When you use the CPAT tool, and it discovers that there are potential environment issues with a Cloud migration, you are warned ahead of time of what these issues are. As a result, you are less likely to encounter an unforeseen issue with your migration. In addition to warning you about issues, the tool can also provide you with parameters for migration, including parameters for Oracle Data Pump, or other migration tools. These parameters are customized for your specific migration case, so that potential migration issues are either reduced, or avoided entirely.

To identify issues and create customized parameters, CPAT performs several checks on the source database and schema contents. These checks are guided by the target Oracle Cloud database option that you select, and the migration approach that you intend to use. The results of these checks are compiled and presented back to you, either in a machine-readable format (JSON), or a human readable format (plain text or HTML), or both. In addition, the CPAT check results are designed so that they can be used by other Oracle features, such as Oracle Zero Downtime Migration (Oracle ZDM) or Oracle Enterprise Manager.

> **Note:**
>
> CPAT is not itself a migration tool. It is intended to assist you to prepare for migrations. It does not suggest whether a particular migration approach using Oracle GoldenGate or Oracle Data Pump is the best option, but rather provides you with customized support for the option that you choose.

1.5 Required Roles for Oracle Data Pump Export and Import Operations

The roles DATAPUMP_EXP_FULL_DATABASE and DATAPUMP_IMP_FULL_DATABASE are required for many Export and Import operations.

> **Caution:**
>
> Do not run Oracle Data Pump jobs as the SYS user. Either use the schema SYSTEM (or ADMIN in Oracle Autonomous Database) for system management operations, or use a user account that is granted the Data Pump full privileges roles that are described below.

When you run Export or Import operations, the operation can require that the user account that you are using to run the operations on premises or in user-managed cloud services is granted either the DATAPUMP_EXP_FULL_DATABASE role, or the DATAPUMP_IMP_FULL_DATABASE role, or both roles. The corresponding roles for Autonomous Database are DATAPUMP_CLOUD_EXP and DATAPUMP_CLOUD_IMP. These roles are automatically defined for Oracle Database when you run the standard scripts that are part of database creation. (Note that although the names of these roles contain the word FULL, these roles actually apply to any privileged operations in any export or import mode, not only Full mode.)

The DATAPUMP_EXP_FULL_DATABASE role affects only export operations. The DATAPUMP_IMP_FULL_DATABASE role affects import operations and operations that use the Import SQLFILE parameter. These roles allow users performing exports and imports to do the following:

- Perform the operation outside the scope of their schema
- Monitor jobs that were initiated by another user
- Export objects (such as tablespace definitions) and import objects (such as directory definitions) that unprivileged users cannot reference

These are powerful roles. As a database administrator, you should use caution when granting these roles to users.

Although the SYS schema does not have either of these roles assigned to it, all security checks performed by Oracle Data Pump that require these roles also grant access to the SYS schema.

> **Note:**
>
> If you receive an ORA-39181: Only Partial Data Exported Due to Fine Grain Access Control error message, then see My Oracle Support "ORA-39181:Only Partial Table Data Exported Due To Fine Grain Access Control (Doc ID 422480.1)" for information about security during an export of table data with fine-grained access control policies enabled.:
>
> ORA-39181:Only Partial Table Data Exported Due To Fine Grain Access Control (Doc ID 422480.1)

Some Oracle roles require authorization. If you need to use these roles with Oracle Data Pump exports and imports, then you must explicitly enable them by setting the `ENABLE_SECURE_ROLES` parameter to `YES`.

> **See Also:**
>
> *Oracle Database Security Guide* for more information about predefined roles in an Oracle Database installation

1.6 What Happens During the Processing of an Oracle Data Pump Job?

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

- Coordination of an Oracle Data Pump Job
  A Data Pump control process is created to coordinate every Oracle Data Pump Export and Import job.
- Tracking Progress Within a Job
  While the data and metadata are being transferred, a parent job table is used to track the progress within a job.
- Filtering Data and Metadata During an Oracle Data Pump Job
  If you want to filter the types of objects that are exported and imported with Oracle Data Pump, then you can use the `EXCLUDE` and `INCLUDE` parameters.
- Transforming Metadata During an Oracle Data Pump Job
  When you move data from one database to another, you can perform transformations on the metadata by using Oracle Data Pump Import parameters.
- Maximizing Job Performance of Oracle Data Pump
  To increase job performance, you can use the Oracle Data Pump `PARALLEL` parameter to run multiple worker processes in parallel.
- Loading and Unloading Data with Oracle Data Pump
  Learn how Oracle Data Pump child processes operate during data imports and exports.

1.6.1 Coordination of an Oracle Data Pump Job

A Data Pump control process is created to coordinate every Oracle Data Pump Export and Import job.

The Data Pump control process controls the entire job, including communicating with the client processes, creating and controlling a pool of worker processes, and performing logging operations.

1.6.2 Tracking Progress Within a Job

While the data and metadata are being transferred, a parent job table is used to track the progress within a job.

The parent job table is implemented as a user table within the database. The specific function of the parent job table for export and import jobs is as follows:

- For export jobs, the parent job table records the location of database objects within a dump file set. Export builds and maintains the parent job table for the duration of the job. At the end of an export job, the content of the parent job table is written to a file in the dump file set.
- For import jobs, the parent job 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 parent job table is created in the schema of the current user performing the export or import operation. Therefore, that user must have the `CREATE TABLE` system privilege, and a sufficient tablespace quota for creation of the parent job table. The name of the parent job table is the same as the name of the job that created it. Therefore, you cannot explicitly give an Oracle Data Pump job the same name as a preexisting table or view.

For all operations, the information in the parent job table is used to restart a job. (Note that transportable jobs are not restartable.)

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

- Upon successful job completion, the parent job table is dropped. You can override this by setting the Oracle Data Pump `KEEP_MASTER=YES` parameter for the job.
- The parent job table is automatically retained for jobs that do not complete successfully.
- If a job is stopped using the `STOP_JOB` interactive command, then the parent job table is retained for use in restarting the job.
- If a job is killed using the `KILL_JOB` interactive command, then the parent job table is dropped, and the job cannot be restarted.
- If a job terminates unexpectedly, then the parent job table is retained. You can delete it if you do not intend to restart the job.
- If a job stops before it starts running (that is, before any database objects have been copied), then the parent job table is dropped.

Related Topics

- JOB_NAME
  The Oracle Data Pump Export command-line utility `JOB_NAME` parameter identifies the export job in subsequent actions.

1.6.3 Filtering Data and Metadata During an Oracle Data Pump Job

If you want to filter the types of objects that are exported and imported with Oracle Data Pump, then you can use the `EXCLUDE` and `INCLUDE` parameters.

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. You can also specify data-specific filters to restrict the rows that are exported and imported.

Related Topics

- Filtering During Export Operations
  Oracle Data Pump Export provides data and metadata filtering capability. This capability helps you limit the type of information that is exported.
- Filtering During Import Operations
  Oracle Data Pump Import provides data and metadata filtering capability, which can help you limit the type of information that you import.

1.6.4 Transforming Metadata During an Oracle Data Pump Job

When you move data from one database to another, you can perform transformations on the metadata by using Oracle Data Pump Import parameters.

It is often useful to perform transformations on your metadata, so that you can remap storage between tablespaces, or redefine the owner of a particular set of objects. When you move data, you can perform transformations by using the Oracle Data Pump import parameters `REMAP_DATAFILE`, `REMAP_SCHEMA`, `REMAP_TABLE`, `REMAP_TABLESPACE`, `TRANSFORM`, and `PARTITION_OPTIONS`.

1.6.5 Maximizing Job Performance of Oracle Data Pump

To increase job performance, you can use the Oracle Data Pump `PARALLEL` parameter to run multiple worker processes in parallel.

The `PARALLEL` parameter enables you 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, database administrators can choose to restrict the parallelism. The degree of parallelism can be reset at any time during a job. For example, during production hours, you can set `PARALLEL` to 2, so that you restrict a particular job to only two degrees of parallelism. During non-production hours, you can reset the degree of parallelism to 8. The parallelism setting is enforced by the Data Pump control process, which allocates workloads to worker processes that perform the data and metadata processing within an operation. These worker processes operate in parallel. For recommendations on setting the degree of parallelism, refer to the Export `PARALLEL` and Import `PARALLEL` parameter descriptions.

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

Related Topics

- PARALLEL
  The Oracle Data Pump Export command-line utility `PARALLEL` parameter specifies the maximum number of processes of active execution operating on behalf of the export job.
- PARALLEL
  The Oracle Data Pump Import command-line mode `PARALLEL` parameter sets the maximum number of worker processes that can load in parallel.

1.6.6 Loading and Unloading Data with Oracle Data Pump

Learn how Oracle Data Pump child processes operate during data imports and exports.

Oracle Data Pump child processes unload and load metadata and table data. For export, all metadata and data are unloaded in parallel, with the exception of jobs that use transportable tablespace. For import, objects must be created in the correct dependency order.

If there are enough objects of the same type to make use of multiple child processes, then the objects are imported by multiple child processes. Some metadata objects have interdependencies, which require one child process to create them serially to satisfy those dependencies. Child processes are created as needed until the number of child processes equals the value supplied for the `PARALLEL` command-line parameter. The number of active child processes can be reset throughout the life of a job. Worker processes can be started on different nodes in an Oracle Real Application Clusters (Oracle RAC) environment.

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

When a child process is assigned the task of loading or unloading a very large table or partition, to make maximum use of parallel execution, it can make use of the external tables access method. In such a case, the child process becomes a parallel execution coordinator. The actual loading and unloading work is divided among some number of parallel input/output (I/O) execution processes allocated from a pool of available processes in an Oracle Real Application Clusters (Oracle RAC) environment.

Related Topics

- PARALLEL
- PARALLEL

1.7 How to Monitor Status of Oracle Data Pump Jobs

The Oracle Data Pump Export and Import client utilities can attach to a job in either logging mode or interactive-command mode.

In logging mode, real-time detailed status about the job is automatically displayed during job execution. The information displayed can include the job and parameter descriptions, an estimate of the amount of data to be processed, a description of the current operation or item being processed, files used during the job, any errors encountered, and the final job state (Stopped or Completed).

In interactive-command mode, job status can be displayed on request. The information displayed can include the job description and state, a description of the current operation or item being processed, files being written, and a cumulative status.

You can also have a log file written during the execution of a job. The log file summarizes the progress of the job, lists any errors encountered during execution of the job, and records the completion status of the job.

As an alternative to determine job status or other information about Oracle Data Pump jobs, you can query the DBA_DATAPUMP_JOBS, USER_DATAPUMP_JOBS, or DBA_DATAPUMP_SESSIONS views. Refer to *Oracle Database Reference* for more information.

Related Topics

- *Oracle Database Reference*

1.8 How to Monitor the Progress of Running Jobs with V$SESSION_LONGOPS

To monitor table data transfers, you can use the V$SESSION_LONGOPS dynamic performance view to monitor Oracle Data Pump jobs.

Oracle 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.

Use of the COMPRESSION, ENCRYPTION, ENCRYPTION_ALGORITHM, ENCRYPTION_MODE, ENCRYPTION_PASSWORD, QUERY, and REMAP_DATA parameters are not reflected in the determination of estimate values.

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 transferred thus far during the job
- TOTALWORK Estimated number of megabytes in the job
- UNITS: Megabytes (MB)
- MESSAGE: A formatted status message that uses the following format:

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

1.9 File Allocation with Oracle Data Pump

You can modify how Oracle Data Pump allocates and handles files by using commands in interactive mode.

- Understanding File Allocation in Oracle Data Pump
  Understanding how Oracle Data Pump allocates and handles files helps you to use Export and Import to their fullest advantage.
- Specifying Files and Adding Additional Dump Files
  For export operations, you can either specify dump files at the time you define the Oracle Data Pump job, or at a later time during the operation.

- Default Locations for Dump, Log, and SQL Files
  Learn about default Oracle Data Pump file locations, and how these locations are affected when you are using Oracle RAC, Oracle Automatic Storage Management, and multitenant architecture.
- Using Substitution Variables with Oracle Data Pump Exports
  If you want to specify multiple dump files during Oracle Data Pump export operations, then use the `DUMPFILE` parameter with a substitution variable in the file name.

1.9.1 Understanding File Allocation in Oracle Data Pump

Understanding how Oracle Data Pump allocates and handles files helps you to use Export and Import to their fullest advantage.

Oracle Data Pump jobs manage the following types of files:

- 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 started using the Oracle Data Pump Import `SQLFILE` parameter. This operation results in all of the `SQL DDL` that Import would execute, based on other parameters, being written to a SQL file.
- Files specified by the `DATA_FILES` parameter during a transportable import.

> **Note:**
>
> If your Oracle Data Pump job generates errors related to Network File Storage (NFS), then consult the installation guide for your platform to determine the correct NFS mount settings.

1.9.2 Specifying Files and Adding Additional Dump Files

For export operations, you can either specify dump files at the time you define the Oracle Data Pump job, or at a later time during the operation.

If you discover that space is running low during an export operation, then you can add additional dump files by using the Oracle 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 overwrite previously existing files. For dump files, you can use the Export `REUSE_DUMPFILES` parameter to specify whether to overwrite a preexisting dump file.

1.9.3 Default Locations for Dump, Log, and SQL Files

Learn about default Oracle Data Pump file locations, and how these locations are affected when you are using Oracle RAC, Oracle Automatic Storage Management, and multitenant architecture.

- Understanding Dump, Log, and SQL File Default Locations
  Oracle Data Pump is server-based, rather than client-based. Dump files, log files, and SQL files are accessed relative to server-based directory paths.

- Understanding How to Use Oracle Data Pump with Oracle RAC
  Using Oracle Data Pump in an Oracle Real Application Clusters (Oracle RAC) environment requires you to perform a few checks to ensure that you are making cluster member nodes available.
- Using Directory Objects When Oracle Automatic Storage Management Is Enabled
  If you use Oracle Data Pump Export or Import with Oracle Automatic Storage Management (Oracle ASM) enabled, then define the directory object used for the dump file.
- The DATA_PUMP_DIR Directory Object and Pluggable Databases
  The default Oracle Data Pump directory object, `DATA_PUMP_DIR`, is defined as a unique path for each PDB in the CDB.

1.9.3.1 Understanding Dump, Log, and SQL File Default Locations

Oracle Data Pump is server-based, rather than client-based. Dump files, log files, and SQL files are accessed relative to server-based directory paths.

Oracle Data Pump requires that directory paths are specified as directory objects. A directory object maps a name to a directory path on the file system. As a database administrator, you must ensure that only approved users are allowed access to the directory object associated with the directory path.

The following example shows a SQL statement that 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 are allowed to specify a directory path location for an input file, then it is possible that you could be able to read data that the server has access to, but to which you should not.
- If you are allowed to specify a directory path location for an output file, then it is possible that you could overwrite a file that normally you do not have privileges to delete.

On Unix, Linux, and Windows operating systems, a default directory object, `DATA_PUMP_DIR`, is created at database creation, or whenever the database dictionary is upgraded. By default, this directory object is available only to privileged users. (The user `SYSTEM` has read and write access to the `DATA_PUMP_DIR` directory, by default.) Oracle can change the definition of the `DATA_PUMP_DIR` directory, either during Oracle Database upgrades, or when patches are applied.

If you are not a privileged user, then before you can run Oracle Data Pump Export or Import, a directory object must be created by a database administrator (DBA), or by any user with the `CREATE ANY DIRECTORY` privilege.

After a directory is created, the user creating the directory object must grant `READ` or `WRITE` permission on the directory to other users. For example, to allow Oracle Database to read and write files on behalf of user `hr` in the directory named by `dpump_dir1`, the DBA must run 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 Oracle Database can read or write files in the corresponding directory on your behalf. Outside of Oracle Database, uou are not given direct access to those files, unless you have the appropriate operating

system privileges. Similarly, Oracle Database requires permission from the operating system to read and write files in the directories.

Oracle 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 file name by a colon.)
2. If a directory object is not specified as part of the file specification, then the directory object named by the `DIRECTORY` parameter is used.
3. If a directory object is not specified as part of the file specification, and if no directory object is named by the `DIRECTORY` parameter, then the value of the environment variable, `DATA_PUMP_DIR`, is used. This environment variable is defined by 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';
   ```

   After this statement is run, 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`, and the log file `export.log`, are written to `'/usr/apps/dumpfiles1'`.

   ```
   %setenv DATA_PUMP_DIR DUMP_FILES1
   %expdp 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 Oracle Data Pump attempts to use the value of the default server-based directory object, `DATA_PUMP_DIR`. This directory object is automatically created, either at database creation, or when the database dictionary is upgraded. To see the path definition for `DATA_PUMP_DIR`, you can use the following SQL query:

   ```
   SQL> SELECT directory_name, directory_path FROM dba_directories
   2 WHERE directory_name='DATA_PUMP_DIR';
   ```

   If you are not a privileged user, then access to the `DATA_PUMP_DIR` directory object must have previously been granted to you by a DBA.

   Do not confuse the default `DATA_PUMP_DIR` directory object with the client-based environment variable of the same name.

1.9.3.2 Understanding How to Use Oracle Data Pump with Oracle RAC

Using Oracle Data Pump in an Oracle Real Application Clusters (Oracle RAC) environment requires you to perform a few checks to ensure that you are making cluster member nodes available.

- To use Oracle Data Pump or external tables in an Oracle RAC configuration, you must ensure that the directory object path is on a cluster-wide file system.

The directory object must point to shared physical storage that is visible to, and accessible from, all instances where Oracle Data Pump or external tables processes (or both) can run.

- The default Oracle Data Pump behavior is that child processes can run on any instance in an Oracle RAC configuration. Therefore, child processes on those Oracle RAC instances must have physical access to the location defined by the directory object, such as shared storage media. If the configuration does not have shared storage for this purpose, but you still require parallelism, then you can use the `CLUSTER=NO` parameter to constrain all child processes to the instance where the Oracle Data Pump job was started.
- Under certain circumstances, Oracle Data Pump uses parallel query child processes to load or unload data. In an Oracle RAC environment, Data Pump does not control where these child processes run. Therefore, these child processes can run on other cluster member nodes in the cluster, regardless of which instance is specified for `CLUSTER` and `SERVICE_NAME` for the Oracle Data Pump job. Controls for parallel query operations are independent of Oracle Data Pump. When parallel query child processes run on other instances as part of an Oracle Data Pump job, they also require access to the physical storage of the dump file set.

1.9.3.3 Using Directory Objects When Oracle Automatic Storage Management Is Enabled

If you use Oracle Data Pump Export or Import with Oracle Automatic Storage Management (Oracle ASM) enabled, then define the directory object used for the dump file.

You must define the directory object used for the dump file so that the Oracle ASM disk group name is used, instead of an operating system directory path.

For log file, use a separate directory object that points to an operating system directory path.

For example, you can create a directory object for the Oracle ASM dump file using this procedure.

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

After you create the directory object, you then 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 assign the necessary privileges for that user:

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

Finally, you then can use use the following Data Pump Export command:

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

Before the command executes, you are prompted for the password.

> **Note:**
>
> If you simply want to copy Data Pump dump files between ASM and disk directories, you can use the `DBMS_FILE_TRANSFER` PL/SQL package.

Related Topics

- *Oracle Database SQL Language Reference*
- *Oracle Database PL/SQL Packages and Types Reference*

1.9.3.4 The DATA_PUMP_DIR Directory Object and Pluggable Databases

The default Oracle Data Pump directory object, `DATA_PUMP_DIR`, is defined as a unique path for each PDB in the CDB.

As of Oracle Database 12c Release 2 (12.2), in a multitenant container database (CDB) environment, the default Oracle Data Pump directory object, `DATA_PUMP_DIR`, is defined as a unique path for each PDB in the CDB, whether or not the `PATH_PREFIX` clause of the `CREATE PLUGGABLE DATABASE` statement is defined for relative paths.

1.9.4 Using Substitution Variables with Oracle Data Pump Exports

If you want to specify multiple dump files during Oracle Data Pump export operations, then use the `DUMPFILE` parameter with a substitution variable in the file name.

When you use substitution variables with file names, instead of or in addition to listing specific file names, then those filenames with a substitution variable are called **dump file templates**.

> **Note:**
>
> In the examples that follow, the substitution variable `%U` is used to explain how Oracle Data Pump uses substitution variables. You can view other available substitution variables under the Import or Export `DUMPFILE` parameter reference topics.

When you use dump file templates, new dump files are created as they are needed. For example, if you are using the substitution variable `%U`, then new dump files are created as needed beginning with `01` for `%U`, and 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, then 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, then they are used to generate dump files in a round-robin fashion. For example, if `expa%U`, `expb%U`, and `expc%U` are all specified for a job having a parallelism of 6, then the initial dump files created are `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 begins by attempting to open the dump files `expa01.dmp`, `expb01.dmp`, and `expc01.dmp`. It is possible for the Data Pump control export table to span multiple dump files. For this reason, until all pieces of the Data Pump control table are found,

dump files continue to be opened by incrementing the substitution variable, and looking up the new file names (For example: `expa02.dmp`, `expb02.dmp`, and `expc02.dmp`). If a dump file does not exist, then 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. After the entire Data Pump control table is found, it is used to determine whether all dump files in the dump file set have been located.

Related Topics

- Oracle Data Pump Export command-line utility DUMPFILE parameter
- Oracle Data Pump Import command-line mode DUMPFILE parameter

1.10 Exporting and Importing Between Different Oracle Database Releases

You can use Oracle Data Pump to migrate all or any portion of an Oracle Database between different releases of the database software.

Typically, you use the Oracle Data Pump Export `VERSION` parameter to migrate between database releases. Using `VERSION` generates an Oracle Data Pump dump file set that is compatible with the specified version.

The default value for `VERSION` is `COMPATIBLE`. This value indicates that exported database object definitions are compatible with the release specified for the `COMPATIBLE` initialization parameter.

In an upgrade situation, when the target release of an Oracle Data Pump-based migration is higher than the source, you typically do not have to specify the `VERSION` parameter. When the target release is higher then the source, all objects in the source database are compatible with the higher target release. However, an exception is when an entire Oracle Database 11g (Release 11.2.0.3 or higher) is exported in preparation for importing into Oracle Database 12c Release 1 (12.1.0.1) or later. In this case, to include a complete set of Oracle Database internal component metadata, explicitly specify `VERSION=12` with `FULL=YES`.

In a downgrade situation, when the target release of an Oracle Data Pump-based migration is lower than the source, set the `VERSION` parameter value to be the same version as the target. An exception is when the target release version is the same as the value of the `COMPATIBLE` initialization parameter on the source system. In that case, you do not need to specify `VERSION`. In general, however, Oracle Data Pump import cannot read dump file sets created by an Oracle Database release that is newer than the current release, unless you explicitly specify the `VERSION` parameter.

Keep the following information in mind when you are exporting and importing between different database releases:

- On an Oracle Data Pump export, if you specify a database version that is older than the current database version, then a dump file set is created that you can import into that older version of the database. For example, if you are running Oracle Database 19c, and you specify `VERSION=12.2` on an export, then the dump file set that is created can be imported into an Oracle Database 12c (Release 12.2) database.

> **Note:**
>
> – Database privileges that are valid only in Oracle Database 12c Release 1 (12.1.0.2) and later (for example, the `READ` privilege on tables, views, materialized views, and synonyms) cannot be imported into Oracle Database 12c Release 1 (12.1.0.1) or earlier. If an attempt is made to do so, then Import reports it as an error, and continues the import operation.
>
> – When you export to a release earlier than Oracle Database 12c Release 2 (12.2.0.1), Oracle Data Pump does not filter out object names longer than 30 bytes. The objects are exported. At import time, if you attempt to create an object with a name longer than 30 bytes, then an error is returned.

- If you specify an Oracle Database release that is older than the current Oracle Database release, then certain features and data types can be unavailable. For example, specifying `VERSION=10.1` causes an error if data compression is also specified for the job, because compression was not supported in Oracle Database 10g release 1 (10.1). Another example: If a user-defined type or Oracle-supplied type in the source Oracle Database release is a later version than the type in the target Oracle Database release, then that type is not loaded, because it does not match any version of the type in the target database.
- Oracle Data Pump Import can always read Oracle Data Pump dump file sets created by older Oracle Database releases.
- When operating across a network link, Oracle Data Pump requires that the source and target Oracle Database releases differ by no more than two versions.

  For example, if one database is Oracle Database 12c, then the other Oracle Database release must be 12c, 11g, or 10g. Oracle Data Pump checks only the major version number (for example, 10g,11g, 12c), not specific Oracle Database release numbers (for example, 12.2, 12.1, 11.1, 11.2, 10.1, or 10.2).
- Importing Oracle Database 11g dump files that contain table statistics into Oracle Database 12c Release 1 (12.1) or later Oracle Database releases can result in an Oracle ORA-39346 error. This error occurs because Oracle Database 11g dump files contain table statistics as metadata. Oracle Database 12c Release 1 (12.1) and later releases require table statistics to be presented as table data. The workaround is to ignore the error during the import operation. After the import operation completes, regather table statistics.
- All forms of `LONG` data types (`LONG`, `LONG RAW`, `LONG VARCHAR`, `LONG VARRAW`) were deprecated in Oracle8i Release 8.1.6. For succeeding releases, the `LONG` data type was provided for backward compatibility with existing applications. In new applications developed with later releases, Oracle strongly recommends that you use `CLOB` and `NCLOB` data types for large amounts of character data.

Related Topics

- Oracle Data Pump Export command-line utility VERSION parameter
- Oracle Data Pump Import command-line mode VERSION parameter

> **See Also:**
>
> - READ and SELECT Object Privileges in *Oracle Database Security Guide* for more information about the `READ` and `READ ANY TABLE` privileges

1.11 Managing SecureFiles Large Object Exports with Oracle Data Pump

Exports of SecureFiles large objects (LOBs) are affected by the content type, the `VERSION` parameter, and other variables.

LOBs are a set of data types that are designed to hold large amounts of data. When you use Oracle Data Pump Export to export SecureFiles LOBs, the export behavior depends on several things, including the Export `VERSION` parameter value, whether a content type (`ContentType`) is present, and whether the LOB is archived and data is cached.

The following scenarios cover different combinations of these variables:

- If a table contains SecureFiles LOBs with a `ContentType`, and the Export `VERSION` parameter is set to a value earlier than `11.2.0.0.0`, then the `ContentType` is not exported.
- If a table contains SecureFiles LOBs with a `ContentType`, and the Export `VERSION` parameter is set to a value of `11.2.0.0.0` or later, then the `ContentType` is exported and restored on a subsequent import.
- If a table contains a SecureFiles LOB that is currently archived, the data is cached, and the Export `VERSION` parameter is set to a value earlier than `11.2.0.0.0`, then the SecureFiles LOB data is exported and the archive metadata is dropped. In this scenario, if `VERSION` is set to `11.1` or later, then the SecureFiles LOB becomes a plain SecureFiles LOB. But if `VERSION` is set to a value earlier than `11.1`, then the SecureFiles LOB becomes a `BasicFiles` LOB.
- If a table contains a SecureFiles LOB that is currently archived, but the data is not cached, and the Export `VERSION` parameter is set to a value earlier than `11.2.0.0.0`, then an `ORA-45001` error is returned.
- If a table contains a SecureFiles LOB that is currently archived, the data is cached, and the Export `VERSION` parameter is set to a value of `11.2.0.0.0` or later, then both the cached data and the archive metadata is exported.

Refer to *Oracle Database SecureFiles and Large Objects Developer's Guide* for more information about SecureFiles LOBs.

Related Topics

- *Oracle Database SecureFiles and Large Objects Developer's Guide*

1.12 Oracle Data Pump Process Exit Codes

To check the status of your Oracle Data Pump export and import operations, review the process exit codes in the log file.

Oracle Data Pump provides the results of export and import operations immediately upon completion. In addition to recording the results in a log file, Oracle Data Pump can also report

the outcome in a process exit code. Use the Oracle Data Pump exit code to check the outcome of an Oracle Data Pump job from the command line or a script:

Table 1-1 Oracle Data Pump Exit Codes

Exit Code	Meaning
`EX_SUCC 0`	The export or import job completed successfully. No errors are displayed to the output device or recorded in the log file, if there is one.
`EX_SUCC_ERR 5`	The export or import job completed successfully, but there were errors encountered during the job. The errors are displayed to the output device and recorded in the log file, if there is one.
`EX_FAIL 1`	The export or import job encountered one or more fatal errors, including the following: • Errors on the command line or in command syntax • Oracle database errors from which export or import cannot recover • Operating system errors (such as malloc) • Invalid parameter values that prevent the job from starting (for example, an invalid directory object specified in the `DIRECTORY` parameter) A fatal error is displayed to the output device but may not be recorded in the log file. Whether it is recorded in the log file can depend on several factors, including: • Was a log file specified at the start of the job? • Did the processing of the job proceed far enough for a log file to be opened?

1.13 How to Monitor Oracle Data Pump Jobs with Unified Auditing

To monitor and record specific user database actions, perform auditing on Data Pump jobs with unified auditing.

To monitor and record specific user database actions, you can perform auditing on Oracle Data Pump jobs. Oracle Data Pump uses unified auditing, in which all audit records are centralized in one place. To set up unified auditing, you create a unified audit policy, or alter an existing audit policy. An audit policy is a named group of audit settings that enable you to audit a particular aspect of user behavior in the database.

To create the policy, use the SQL `CREATE AUDIT POLICY` statement. After creating the audit policy, use the `AUDIT` SQL statement to enable the policy.

To disable the policy, use the `NOAUDIT` SQL statement.

See Also:

- *Oracle Database SQL Language Reference* for more information about the SQL `CREATE AUDIT POLICY`,`ALTER AUDIT POLICY`, `AUDIT`, and `NOAUDIT` statements
- *Oracle Database Security Guide* for more information about using auditing in an Oracle database

1.14 Encrypted Data Security Warnings for Oracle Data Pump Operations

Oracle Data Pump warns you when encrypted data is exported as unencrypted data.

During Oracle Data Pump export operations, you receive an `ORA-39173` warning when Oracle Data Pump encounters encrypted data specified when the export job was started. This `ORA-39173` warning ("ORA-39173: Encrypted data has been stored unencrypted in dump file set") is also written to the the audit record. You can view the ORA-39173 errors encountered during the export operation by checking the `DP_WARNINGS1` column in the unified audit trail. Obtain the audit information by running the following SQL statement:

```
SELECT DP_WARNINGS1 FROM UNIFIED_AUDIT_TRAIL WHERE ACTION_NAME = 'EXPORT'
ORDER BY 1;
```

1.15 How Does Oracle Data Pump Handle Timestamp Data?

Learn about factors that can affect successful completion of export and import jobs that involve the timestamp data types `TIMESTAMP WITH TIMEZONE` and `TIMESTAMP WITH LOCAL TIMEZONE`.

> **Note:**
>
> The information in this section applies only to Oracle Data Pump running on Oracle Database 12c and later.

- TIMESTAMP WITH TIMEZONE Restrictions
  Export and import jobs that have `TIMESTAMP WITH TIME ZONE` data are restricted.
- TIMESTAMP WITH LOCAL TIME ZONE Restrictions
  Moving tables using a transportable mode is restricted.

1.15.1 TIMESTAMP WITH TIMEZONE Restrictions

Export and import jobs that have `TIMESTAMP WITH TIME ZONE` data are restricted.

- Understanding TIMESTAMP WITH TIME ZONE Restrictions
  Carrying out export and import jobs that have `TIMESTAMP WITH TIME ZONE` data requires understanding information about your time zone file data and Oracle Database release.
- Oracle Data Pump Support for TIMESTAMP WITH TIME ZONE Data
  Oracle Data Pump supports `TIMESTAMP WITH TIME ZONE` data during different export and import modes.
- Time Zone File Versions on the Source and Target
  Successful job completion can depend on whether the source and target time zone file versions match.

1.15.1.1 Understanding TIMESTAMP WITH TIME ZONE Restrictions

Carrying out export and import jobs that have `TIMESTAMP WITH TIME ZONE` data requires understanding information about your time zone file data and Oracle Database release.

When you import a dump file, the time zone version of the destination (target) database must be either the same version, or a more recent (higher) version than the time zone version of the source database from which the export was taken. Successful job completion can depend on the following factors:

- The version of the Oracle Database time zone files on the source and target databases.
- The export/import mode and whether the Data Pump version being used supports `TIMESTAMP WITH TIME ZONE` data. (Oracle Data Pump 11.2.0.1 and later releases provide support for `TIMESTAMP WITH TIME ZONE` data.)

To identify the time zone file version of a database, you can run the following SQL statement:

```
SQL> SELECT VERSION FROM V$TIMEZONE_FILE;
```

Related Topics

- Choosing a Time Zone File

1.15.1.2 Oracle Data Pump Support for TIMESTAMP WITH TIME ZONE Data

Oracle Data Pump supports `TIMESTAMP WITH TIME ZONE` data during different export and import modes.

Oracle Data Pump provides support for `TIMESTAMP WITH TIME ZONE` data during different export and import modes when versions of the Oracle Database time zone file are different on the source and target databases. Supported modes include non-transportable mode, transportable tablespace and transportable table mode, and full transportable mode.

Non-transportable Modes

- If the dump file is created with a Data Pump version that supports `TIMESTAMP WITH TIME ZONE` data (11.2.0.1 or later), then the time zone file version of the export system is recorded in the dump file. Oracle Data Pump uses that information to determine whether data conversion is necessary. If the target database knows about the source time zone version, but is actually using a later version, then the data is converted to the later version. `TIMESTAMP WITH TIME ZONE` data cannot be downgraded, so if you attempt to import to a target that is using an earlier version of the time zone file than the source used, the import fails.
- If the dump file was created with an Oracle Data Pump version earlier than Oracle Database 11g release 2 (11.2.0.1), then `TIMESTAMP WITH TIME ZONE` data is not supported. No conversion is done, and corruption may occur.

Transportable Tablespace and Transportable Table Modes

- In transportable tablespace and transportable table modes, if the source and target have different time zone file versions, tables with `TIMESTAMP WITH TIME ZONE` columns are not created. A warning is displayed at the beginning of the job that shows the source and target database time zone file versions. A message is also displayed for each table not created. This is true even if the Oracle Data Pump version used to create the dump file

supports TIMESTAMP WITH TIME ZONE data. (Release 11.2.0.1 and later support TIMESTAMP WITH TIMEZONE data.)

- If the source is earlier than Oracle Database 11g release 2 (11.2.0.1), then the time zone file version must be the same on the source and target database for all transportable jobs, regardless of whether the transportable set uses TIMESTAMP WITH TIME ZONE columns.

Full Transportable Mode

Full transportable exports and imports are supported when the source database is at least Oracle Database 11g release 2 (11.2.0.3) and the target is at least Oracle Database 12c release 1 (12.1) or later.

Oracle Data Pump 11.2.0.1 and later provide support for TIMESTAMP WITH TIME ZONE data. Therefore, in full transportable operations, tables with TIMESTAMP WITH TIME ZONE columns are created. If the source and target database have different time zone file versions, then TIMESTAMP WITH TIME ZONE columns from the source are converted to the time zone file version of the target.

Related Topics

- Limitations on Transportable Tablespaces
- Full Mode
  You can use Data Pump to carry out a full database export by using the FULL parameter.
- Full Import Mode
  To specify a full import with Oracle Data Pump, use the FULL parameter.

1.15.1.3 Time Zone File Versions on the Source and Target

Successful job completion can depend on whether the source and target time zone file versions match.

- If the Oracle Database time zone file version is the same on the source and target databases, then conversion of TIMESTAMP WITH TIME ZONE data is not necessary. The export/import job should complete successfully.

  The exception to this is a transportable tablespace or transportable table export performed using a Data Pump release earlier than 11.2.0.1. In that case, tables in the dump file that have TIMESTAMP WITH TIME ZONE columns are not created on import even though the time zone file version is the same on the source and target.
- If the source time zone file version is not available on the target database, then the job fails. The version of the time zone file on the source may not be available on the target because the source may have had its time zone file updated to a later version but the target has not. For example, if the export is done on Oracle Database 11*g* release 2 (11.2.0.2) with a time zone file version of 17, and the import is done on 11.2.0.2 with only a time zone file of 16 available, then the job fails.

1.15.2 TIMESTAMP WITH LOCAL TIME ZONE Restrictions

Moving tables using a transportable mode is restricted.

If a table is moved using a transportable mode (transportable table, transportable tablespace, or full transportable), and the following conditions exist, then a warning is issued and the table is not created:

- The source and target databases have different database time zones.

- The table contains `TIMESTAMP WITH LOCAL TIME ZONE` data types.

To successfully move a table that was not created because of these conditions, use a non-transportable export and import mode.

1.16 Character Set and Globalization Support Considerations

Learn about Globalization support of Oracle Data Pump Export and Import using character set conversion of user data, and data definition language (DDL).

- Data Definition Language (DDL)
  The Export utility writes dump files using the database character set of the export system.
- Single-Byte Character Sets and Export and Import
  Ensure that the export database and the import database use the same character set.
- Multibyte Character Sets and Export and Import
  During an Oracle Data Pump export and import, the character set conversion depends on the importing Oracle Database character set.

1.16.1 Data Definition Language (DDL)

The Export utility writes dump files using the database character set of the export system.

When the dump file is imported, a character set conversion is required for DDL only if the database character set of the import system is different from the database character set of the export system.

To minimize data loss due to character set conversions, ensure that the import database character set is a superset of the export database character set.

1.16.2 Single-Byte Character Sets and Export and Import

Ensure that the export database and the import database use the same character set.

If the system on which the import occurs uses a 7-bit character set, and you import an 8-bit character set dump file, then some 8-bit characters may be converted to 7-bit equivalents. An indication that this has happened is when accented characters lose the accent mark.

To avoid this unwanted conversion, ensure that the export database and the import database use the same character set.

1.16.3 Multibyte Character Sets and Export and Import

During an Oracle Data Pump export and import, the character set conversion depends on the importing Oracle Database character set.

During character set conversion, any characters in the export file that have no equivalent in the import database character set are replaced with a default character. The import database character set defines the default character.

If the import system has to use replacement characters while converting DDL, then a warning message is displayed and the system attempts to load the converted DDL.

If the import system has to use replacement characters while converting user data, then the default behavior is to load the converted data. However, it is possible to instruct the import system to reject rows of user data that were converted using replacement characters. See the Import `DATA OPTIONS` parameter for details.

To guarantee 100% conversion, the import database character set must be a superset (or equivalent) of the character set used to generate the export file.

> **Caution:**
>
> When the database character set of the export system differs from that of the import system, the import system displays informational messages at the start of the job that show what the database character set is.
>
> When the import database character set is not a superset of the character set used to generate the export file, the import system displays a warning that possible data loss may occur due to character set conversions.

Related Topics

- DATA_OPTIONS

1.17 Oracle Data Pump Behavior with Data-Bound Collation

Oracle Data Pump supports data-bound collation (DBC).

Oracle Data Pump Export always includes all available collation metadata into the created dump file. This includes:

- Current default collations of exported users' schemas
- Current default collations of exported tables, views, materialized views and PL/SQL units (including user-defined types)
- Declared collations of all table and cluster character data type columns

When importing a dump file exported from an Oracle Database 12c Release 2 (12.2) database, Oracle Data Pump Import's behavior depends both on the effective value of the Oracle Data Pump `VERSION` parameter at the time of import, and on whether the data-bound collation (DBC) feature is enabled in the target database. The effective value of the `VERSION` parameter is determined by how it is specified. You can specify the parameter using the following:

- `VERSION=`*`n`*, which means the effective value is the specific version number *`n`*. For example: `VERSION=19`
- `VERSION=LATEST`, which means the effective value is the currently running database version
- `VERSION=COMPATIBLE`, which means the effective value is the same as the value of the database initialization parameter `COMPATIBLE`. This is also true if no value is specified for `VERSION`.

For the DBC feature to be enabled in a database, the initialization parameter `COMPATIBLE` must be set to 12.2 or higher, and the initialization parameter `MAX_STRING_SIZE` must be set to `EXTENDED`.

If the effective value of the Oracle Data Pump Import `VERSION` parameter is 12.2, and DBC is enabled in the target database, then Oracle Data Pump Import generates DDL statements with collation clauses referencing collation metadata from the dump file. Exported objects are created with the original collation metadata that they had in the source database.

No collation syntax is generated if DBC is disabled, or if the Oracle Data Pump Import `VERSION` parameter is set to a value lower than 12.2.

2

Oracle Data Pump Export

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

- What Is Oracle Data Pump Export?
  Oracle Data Pump Export is a utility for unloading data and metadata into a set of operating system files that are called a **dump file set**.
- Starting Oracle Data Pump Export
  Start the Oracle Data Pump Export utility by using the `expdp` command.
- Filtering During Export Operations
  Oracle Data Pump Export provides data and metadata filtering capability. This capability helps you limit the type of information that is exported.
- Parameters Available in Data Pump Export Command-Line Mode
  Use Oracle Data Pump parameters for Export (`expdp`) to manage your data exports.
- Commands Available in Data Pump Export Interactive-Command Mode
  Check which command options are available to you when using Data Pump Export in interactive mode.
- Examples of Using Oracle Data Pump Export
  You can use these common scenario examples to learn how you can create parameter files and use Oracle Data Pump Export to move your data.
- Syntax Diagrams for Oracle Data Pump Export
  You can use syntax diagrams to understand the valid SQL syntax for Oracle Data Pump Export.

2.1 What Is Oracle Data Pump Export?

Oracle Data Pump Export is a utility for unloading data and metadata into a set of operating system files that are called a **dump file set**.

You can import a dump file set only by using the Oracle Data Pump Import utility. You can import the dump file set on the same system, or import it to another system, and load the dump file set 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 Oracle 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, you must create directory objects that define the server locations to which files are written.

Oracle Data Pump Export enables you to specify that you want a job to move a subset of the data and metadata, as determined by the export mode. This subset selection is done by using data filters and metadata filters, which are specified through Oracle Data Pump Export parameters.

> **Note:**
>
> Several system schemas cannot be exported, because they are not user schemas; they contain Oracle-managed data and metadata. Examples of schemas that are not exported include `SYS`, `ORDSYS`, and `MDSYS`. Secondary objects are also not exported, because the `CREATE INDEX` run at import time will recreate them.

Related Topics

- Understanding Dump_ Log_ and SQL File Default Locations
- Filtering During Export Operations
- Export Utility (exp or expdp) does not Export DR${name}$% or DR#{name}$% Secondary Tables of Text Indexes (Doc ID 139388.1)
- Examples of Using Oracle Data Pump Export

2.2 Starting Oracle Data Pump Export

Start the Oracle Data Pump Export utility by using the `expdp` command.

The characteristics of the Oracle Data Pump export operation are determined by the Export parameters that you specify. You can specify these parameters either on the command line, or in a parameter file.

> **Caution:**
>
> Do not start 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.

- Oracle Data Pump Export Interfaces
  You can interact with Oracle Data Pump Export by using a command line, a parameter file, or an interactive-command mode.
- Oracle Data Pump Export Modes
  Export provides different modes for unloading different portions of Oracle Database data.
- Network Considerations for Oracle Data Pump Export
  Learn how Oracle Data Pump Export utility `expdp` identifies instances with connect identifiers in the connection string using Oracle*Net or a net service name, and how they are different from export operations using the `NETWORK_LINK` parameter.

2.2.1 Oracle Data Pump Export Interfaces

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

Choose among the three options:

- Command-Line Interface: Enables you to specify most of the Export parameters directly on the command line.

- 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. If you are using parameters whose values require quotation marks, then Oracle recommends that you use parameter files.
- 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.

Related Topics

- Parameters Available in Data Pump Export Command-Line Mode
  Use Oracle Data Pump parameters for Export (`expdp`) to manage your data exports.
- Commands Available in Data Pump Export Interactive-Command Mode
  Check which command options are available to you when using Data Pump Export in interactive mode.

2.2.2 Oracle Data Pump Export Modes

Export provides different modes for unloading different portions of Oracle Database data.

Specify export modes on the command line, using the appropriate parameter.

> **Note:**
>
> You cannot export several Oracle-managed system schemas for Oracle Database, because they are not user schemas; they contain Oracle-managed data and metadata. Examples of system schemas that are not exported include `SYS`, `ORDSYS`, and `MDSYS`.

- Full Mode
  You can use Data Pump to carry out a full database export by using the `FULL` parameter.
- Schema Mode
  You can specify a schema export with Data Pump by using the `SCHEMAS` parameter. A schema export is the default export mode.
- Table Mode
  You can use Data Pump to carry out a table mode export by specifying the table using the `TABLES` parameter.
- Tablespace Mode
  You can use Data Pump to carry out a tablespace export by specifying tables using the `TABLESPACES` parameter.
- Transportable Tablespace Mode
  You can use Oracle Data Pump to carry out a transportable tablespace export by using the `TRANSPORT_TABLESPACES` parameter.

Related Topics

- Examples of Using Oracle Data Pump Export
  You can use these common scenario examples to learn how you can create parameter files and use Oracle Data Pump Export to move your data.

2.2.2.1 Full Mode

You can use Data Pump to carry out a full database export by using the `FULL` parameter.

In a full database export, the entire database is unloaded. This mode requires that you have the `DATAPUMP_EXP_FULL_DATABASE` role.

Using the Transportable Option During Full Mode Exports

If you specify the `TRANSPORTABLE=ALWAYS` parameter along with the `FULL` parameter, then Data Pump performs a full transportable export. A full transportable export exports all objects and data necessary to create a complete copy of the database. A mix of data movement methods is used:

- Objects residing in transportable tablespaces have only their metadata unloaded into the dump file set; the data itself is moved when you copy the data files to the target database. The data files that must be copied are listed at the end of the log file for the export operation.
- Objects residing in non-transportable tablespaces (for example, `SYSTEM` and `SYSAUX`) have both their metadata and data unloaded into the dump file set, using direct path unload and external tables.

Performing a full transportable export has the following restrictions:

- The user performing a full transportable export requires the `DATAPUMP_EXP_FULL_DATABASE` privilege.
- The default tablespace of the user performing the export must not be set to one of the tablespaces being transported.
- If the database being exported contains either encrypted tablespaces or tables with encrypted columns (either Transparent Data Encryption (TDE) columns or SecureFiles LOB columns), then the `ENCRYPTION_PASSWORD` parameter must also be supplied.
- The source and target databases must be on platforms with the same endianness if there are encrypted tablespaces in the source database.
- If the source platform and the target platform are of different endianness, then you must convert the data being transported so that it is in the format of the target platform. You can use the `DBMS_FILE_TRANSFER` package or the `RMAN CONVERT` command to convert the data.
- A full transportable export is not restartable.
- All objects with storage that are selected for export must have all of their storage segments either entirely within administrative, non-transportable tablespaces (`SYSTEM/SYSAUX`) or entirely within user-defined, transportable tablespaces. Storage for a single object cannot straddle the two kinds of tablespaces.
- When transporting a database over the network using full transportable export, auditing cannot be enabled for tables stored in an administrative tablespace (such as `SYSTEM` and `SYSAUX`) if the audit trail information itself is stored in a user-defined tablespace.
- If both the source and target databases are running Oracle Database 12*c*, then to perform a full transportable export, either the Data Pump `VERSION` parameter must be set to at least 12.0. or the `COMPATIBLE` database initialization parameter must be set to at least 12.0 or later.

Full transportable exports are supported from a source database running release 11.2.0.3. To do so, set the Data Pump `VERSION` parameter to at least 12.0, as shown in the following syntax example, where *`user_name`* is the user performing a full transportable export:

```
> expdp user_name FULL=y DUMPFILE=expdat.dmp DIRECTORY=data_pump_dir
      TRANSPORTABLE=always VERSION=12.0 LOGFILE=export.log
```

Related Topics

- FULL
  The Export command-line FULL parameter specifies that you want to perform a full database mode export
- TRANSPORTABLE
  The Oracle Data Pump Export command-line utility `TRANSPORTABLE` parameter specifies whether the transportable option should be used during a table mode or full mode export.

> **See Also:**
>
> - *Oracle Database Backup and Recovery Reference* for information about the RMAN `CONVERT` command
> - *Oracle Database Administrator's Guide* for an example of performing a full transportable export

2.2.2.2 Schema Mode

You can specify a schema export with Data Pump by using the `SCHEMAS` parameter. A schema export is the default export mode.

If you have the `DATAPUMP_EXP_FULL_DATABASE` role, then you can specify a list of schemas, optionally including the schema definitions themselves and also system privilege grants to those schemas. If you do not have the `DATAPUMP_EXP_FULL_DATABASE` role, then you can export only your own schema.

The `SYS` schema cannot be used as a source schema for export jobs.

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. Also, external type definitions upon which tables in the specified schemas depend are not exported. In such a case, it is expected that the type definitions already exist in the target instance at import time.

Related Topics

- SCHEMAS
  The Oracle Data Pump Export command-line utility `SCHEMAS` parameter specifies that you want to perform a schema-mode export.

2.2.2.3 Table Mode

You can use Data Pump to carry out a table mode export by specifying the table using the `TABLES` parameter.

In table mode, only a specified set of tables, partitions, and their dependent objects are unloaded. Any object required to create the table, such as the owning schema, or types for columns, must already exist.

If you specify the TRANSPORTABLE=ALWAYS parameter with the TABLES parameter, then only object metadata is unloaded. To move the actual data, you copy the data files to the target database. This results in quicker export times. If you are moving data files between releases or platforms, then the data files need to be processed by Oracle Recovery Manager (RMAN).

You must have the DATAPUMP_EXP_FULL_DATABASE role to specify tables that are not in your own schema. 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.

To recover tables and table partitions, you can also use RMAN backups and the RMAN RECOVER TABLE command. During this process, RMAN creates (and optionally imports) a Data Pump export dump file that contains the recovered objects. Refer to *Oracle Database Backup and Recovery Guide* for more information about transporting data across platforms.

Carrying out a table mode export has the following restriction:

- When using TRANSPORTABLE=ALWAYS parameter with the TABLES parameter, the ENCRYPTION_PASSWORD parameter must also be used if the table being exported contains encrypted columns, either Transparent Data Encryption (TDE) columns or SecureFiles LOB columns.

Related Topics

- TABLES
  The Oracle Data Pump Export command-line utility TABLES parameter specifies that you want to perform a table-mode export.
- TRANSPORTABLE
  The Oracle Data Pump Export command-line utility TRANSPORTABLE parameter specifies whether the transportable option should be used during a table mode or full mode export.
- *Oracle Database Backup and Recovery User's Guide*

2.2.2.4 Tablespace Mode

You can use Data Pump to carry out a tablespace export by specifying tables using the TABLESPACES parameter.

In tablespace mode, only the tables contained in a specified set of tablespaces are unloaded. If a table is unloaded, then 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. Privileged users get all tables. Unprivileged users get only the tables in their own schemas.

Related Topics

- TABLESPACES
  The Oracle Data Pump Export command-line utility TABLESPACES parameter specifies a list of tablespace names that you want to be exported in tablespace mode.

2.2.2.5 Transportable Tablespace Mode

You can use Oracle Data Pump to carry out a transportable tablespace export by 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 is exported. The tablespace data files are copied in a separate operation. Then, a transportable tablespace import is performed to import the dump file containing the metadata and to specify the data files to use.

Transportable tablespace mode requires that the specified tables be completely self-contained. That is, all storage segments of all tables (and their indexes) defined within the tablespace set must also be contained within the set. If there are self-containment violations, then Export identifies all of the problems without actually performing the export.

Type definitions for columns of tables in the specified tablespaces are exported and imported. The schemas owning those types must be present in the target instance.

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

> **Note:**
>
> You cannot export transportable tablespaces and then import them into a database at a lower release level. The target database must be at the same or later release level as the source database.

Using Oracle Data Pump to carry out a transportable tablespace export has the following restrictions:

- If any of the tablespaces being exported contains tables with encrypted columns, either Transparent Data Encryption (TDE) columns or SecureFiles LOB columns, then the `ENCRYPTION_PASSWORD` parameter must also be supplied..
- If any of the tablespaces being exported is encrypted, then the use of the `ENCRYPTION_PASSWORD` is optional but recommended. If the `ENCRYPTION_PASSWORD` is omitted in this case, then the following warning message is displayed:

  ```
  ORA-39396: Warning: exporting encrypted data using transportable option
  without password
  ```

  This warning points out that in order to successfully import such a transportable tablespace job, the target database wallet must contain a copy of the same database master key used in the source database when performing the export. Using the `ENCRYPTION_PASSWORD` parameter during the export and import eliminates this requirement.

Related Topics

- How Does Oracle Data Pump Handle Timestamp Data?
  Learn about factors that can affect successful completion of export and import jobs that involve the timestamp data types `TIMESTAMP WITH TIMEZONE` and `TIMESTAMP WITH LOCAL TIMEZONE`.

2.2.3 Network Considerations for Oracle Data Pump Export

Learn how Oracle Data Pump Export utility `expdp` identifies instances with connect identifiers in the connection string using Oracle*Net or a net service name, and how they are different from export operations using the `NETWORK_LINK` parameter.

When you start `expdp`, you can specify a connect identifier in the connect string that can be different from the current instance identified by the current Oracle System ID (SID).

You can specify a connect identifier by using either an Oracle*Net connect descriptor, or using a net service name (usually defined in the `tnsnames.ora` file) that maps to a connect descriptor. To use a connect identifier, you must have Oracle Net Listener running (to start the

default listener, enter `lsnrctl start`). The following example shows this type of connection, in which `inst1` is the connect identifier:

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

Export then prompts you for a password:

```
Password: password
```

The local Export client connects to the database instance defined by the connect identifier `inst1` (a Net service name), retrieves data from `inst1`, and writes it to the dump file `hr.dmp` on `inst1`.

Specifying a connect identifier when you start the Export utility is different from performing an export operation using the `NETWORK_LINK` parameter. When you start an export operation and specify a connect identifier, the local Export client connects to the database instance identified by the connect identifier, retrieves data from that database instance, and writes it to a dump file set on that database instance. By contrast, when you perform an export using the `NETWORK_LINK` parameter, the export is performed using a database link. (A database link is a connection between two physical database servers that allows a client to access them as one logical database.)

Related Topics

- NETWORK_LINK
  The Data Pump Export command-line utility `NETWORK_LINK` parameter enables an export from a (source) database identified by a valid database link. The data from the source database instance is written to a dump file set on the connected database instance.

> **See Also:**
>
> - *Oracle Database Administrator's Guide* for more information about database links
> - *Oracle Database Net Services Administrator's Guide* for more information about connect identifiers and Oracle Net Listener

2.3 Filtering During Export Operations

Oracle Data Pump Export provides data and metadata filtering capability. This capability helps you limit the type of information that is exported.

- Oracle Data Pump Export Data Filters
  You can specify restrictions on the table rows that you export by using Oracle Data Pump Data-specific filtering through the `QUERY` and `SAMPLE` parameters.
- Metadata Filters
  Metadata filtering is implemented through the `EXCLUDE` and `INCLUDE` parameters. The `EXCLUDE` and `INCLUDE` parameters are mutually exclusive.

2.3.1 Oracle Data Pump Export Data Filters

You can specify restrictions on the table rows that you export by using Oracle Data Pump Data-specific filtering through the `QUERY` and `SAMPLE` parameters.

Oracle Data Pump can also implement Data filtering indirectly because of metadata filtering, which can include or exclude table objects along with any associated row data.

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

2.3.2 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, then 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 a list of valid object types, query the following views: `DATABASE_EXPORT_OBJECTS` for full mode, `SCHEMA_EXPORT_OBJECTS` for schema mode, and `TABLE_EXPORT_OBJECTS` for table and tablespace mode. The values listed in the `OBJECT_PATH` column are the valid object types. For example, you could perform the following query:

```
SQL> SELECT OBJECT_PATH, COMMENTS FROM SCHEMA_EXPORT_OBJECTS
  2  WHERE OBJECT_PATH LIKE '%GRANT' AND OBJECT_PATH NOT LIKE '%/%';
```

The output of this query looks similar to the following:

```
OBJECT_PATH
--------------------------------------------------------------------------------
COMMENTS
--------------------------------------------------------------------------------
GRANT
Object grants on the selected tables

OBJECT_GRANT
Object grants on the selected tables

PROCDEPOBJ_GRANT
Grants on instance procedural objects

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
```

Related Topics

- EXCLUDE
  The Data Pump Export command-line utility `EXCLUDE` parameter enables you to filter the metadata that is exported by specifying objects and object types that you want to exclude from the export operation.
- INCLUDE
  The Data Pump Export command-line utility INCLUDE parameter 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.

Related Topics

- EXCLUDE
  The Oracle Data Pump Import command-line mode `EXCLUDE` parameter enables you to filter the metadata that is imported by specifying objects and object types to exclude from the import job.
- INCLUDE
  The Oracle Data Pump Import command-line mode `INCLUDE` parameter enables you to filter the metadata that is imported by specifying objects and object types for the current import mode.

2.4 Parameters Available in Data Pump Export Command-Line Mode

Use Oracle Data Pump parameters for Export (`expdp`) to manage your data exports.

- About Data Pump Export Parameters
  Learn how to use Oracle Data Pump Export parameters in command-line mode, including case sensitivity, quotation marks, escape characters, and information about how to use examples.
- ABORT_STEP
  The Oracle Data Pump Export command-line utility `ABORT_STEP` parameter stops the job after it is initialized.
- ACCESS_METHOD
  The Oracle Data Pump Export command-line utility `ACCESS_METHOD` parameter instructs Export to use a particular method to unload data.
- ATTACH
  The Oracle Data Pump Export command-line utility `ATTACH` parameter attaches a worker or client session to an existing export job, and automatically places you in the interactive-command interface.
- CLUSTER
  The Oracle Data Pump Export command-line utility `CLUSTER` parameter determines whether Data Pump can use Oracle RAC, resources, and start workers on other Oracle RAC instances.

- COMPRESSION
  The Oracle Data Pump Export command-line utility `COMPRESSION` parameter specifies which data to compress before writing to the dump file set.
- COMPRESSION_ALGORITHM
  The Oracle Data Pump Export command-line utility `COMPRESSION_ALGORITHM` parameter specifies the compression algorithm that you want to use when compressing dump file data.
- CONTENT
  The Oracle Data Pump Export command-line utility `CONTENT` parameter enables you to filter what Export unloads: data only, metadata only, or both.
- DATA_OPTIONS
  The Oracle Data Pump Export command-line utility `DATA_OPTIONS` parameter designates how you want certain types of data handled during export operations.
- DIRECTORY
  The Oracle Data Pump Export command-line utility `DIRECTORY` parameter specifies the default location to which Export can write the dump file set and the log file.
- DUMPFILE
  The Oracle Data Pump Export command-line utility `DUMPFILE` parameter specifies the names, and optionally, the directory objects of dump files for an export job.
- ENABLE_SECURE_ROLES
  The Oracle Data Pump Export command-line utility `ENABLE_SECURE_ROLES` parameter prevents inadvertent use of protected roles during exports.
- ENCRYPTION
  The Oracle Data Pump Export command-line utility `ENCRYPTION` parameter specifies whether to encrypt data before writing it to the dump file set.
- ENCRYPTION_ALGORITHM
  The Oracle Data Pump Export command-line utility `ENCRYPTION_ALGORITHM` parameter specifies which cryptographic algorithm should be used to perform the encryption.
- ENCRYPTION_MODE
  The Oracle Data Pump Export command-line utility `ENCRYPTION_MODE` parameter specifies the type of security to use when encryption and decryption are performed.
- ENCRYPTION_PASSWORD
  The Oracle Data Pump Export command-line utility `ENCRYPTION_PASSWORD` parameter specifies a password for encrypting encrypted column data, metadata, or table data in the export dump file. This parameter prevents unauthorized access to an encrypted dump file set.
- ENCRYPTION_PWD_PROMPT
  The Oracle Data Pump Export command-line utility `ENCRYPTION_PWD_PROMPT` specifies whether Oracle Data Pump prompts you for the encryption password.
- ESTIMATE
  The Oracle Data Pump Export command-line utility `ESTIMATE` parameter specifies the method that Export uses to estimate how much disk space each table in the export job will consume (in bytes).
- ESTIMATE_ONLY
  The Oracle Data Pump Export command-line utility `ESTIMATE_ONLY` parameter instructs Export to estimate the space that a job consumes, without actually performing the export operation.

- EXCLUDE
  The Data Pump Export command-line utility `EXCLUDE` parameter enables you to filter the metadata that is exported by specifying objects and object types that you want to exclude from the export operation.
- FILESIZE
  The Oracle Data Pump Export command-line utility `FILESIZE` parameter specifies the maximum size of each dump file.
- FLASHBACK_SCN
  The Oracle Data Pump Export command-line utility `FLASHBACK_SCN` parameter specifies the system change number (SCN) that Export uses to enable the Flashback Query utility.
- FLASHBACK_TIME
  The Oracle Data Pump Export command-line utility `FLASHBACK_TIME` parameter finds the SCN that most closely matches the specified time.
- FULL
  The Export command-line FULL parameter specifies that you want to perform a full database mode export
- HELP
  The Data Pump Export command-line utility `HELP` parameter displays online help for the Export utility.
- INCLUDE
  The Data Pump Export command-line utility INCLUDE parameter 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.
- JOB_NAME
  The Oracle Data Pump Export command-line utility `JOB_NAME` parameter identifies the export job in subsequent actions.
- KEEP_MASTER
  The Oracle Data Pump Export command-line utility `KEEP_MASTER` parameter indicates whether the Data Pump control job table should be deleted or retained at the end of an Oracle Data Pump job that completes successfully.
- LOGFILE
  The Data Pump Export command-line utility LOGFILE parameter specifies the name, and optionally, a directory, for the log file of the export job.
- LOGTIME
  The Oracle Data Pump Export command-line utility `LOGTIME` parameter specifies that messages displayed during export operations are timestamped.
- METRICS
  The Oracle Data Pump Export command-line utility `METRICS` parameter indicates whether you want additional information about the job reported to the Data Pump log file.
- NETWORK_LINK
  The Data Pump Export command-line utility `NETWORK_LINK` parameter enables an export from a (source) database identified by a valid database link. The data from the source database instance is written to a dump file set on the connected database instance.
- NOLOGFILE
  The Data Pump Export command-line utility NOLOGFILE parameter specifies whether to suppress creation of a log file.

- PARALLEL
  The Oracle Data Pump Export command-line utility `PARALLEL` parameter specifies the maximum number of processes of active execution operating on behalf of the export job.
- PARALLEL_THRESHOLD
  The Oracle Data Pump Export command-line utility `PARALLEL_THRESHOLD` parameter specifies the size of the divisor that Data Pump uses to calculate potential parallel DML based on table size
- PARFILE
  The Oracle Data Pump Export command-line utility `PARFILE` parameter specifies the name of an export parameter file.
- QUERY
  The Oracle Data Pump Export command-line utility `QUERY` parameter enables you to specify a query clause that is used to filter the data that gets exported.
- REMAP_DATA
  The Oracle Data Pump Export command-line utility `REMAP_DATA` parameter enables you to specify a remap function that takes as a source the original value of the designated column and returns a remapped value that replaces the original value in the dump file.
- REUSE_DUMPFILES
  The Oracle Data Pump Export command-line utility `REUSE_DUMPFILES` parameter specifies whether to overwrite a preexisting dump file.
- SAMPLE
  The Oracle Data Pump Export command-line utility `SAMPLE` parameter specifies a percentage of the data rows that you want to be sampled and unloaded from the source database.
- SCHEMAS
  The Oracle Data Pump Export command-line utility `SCHEMAS` parameter specifies that you want to perform a schema-mode export.
- SERVICE_NAME
  The Oracle Data Pump Export command-line utility `SERVICE_NAME` parameter specifies a service name that you want to use in conjunction with the `CLUSTER` parameter.
- SOURCE_EDITION
  The Oracle Data Pump Export command-line utility `SOURCE_EDITION` parameter specifies the database edition from which objects are exported.
- STATUS
  The Oracle Data Pump Export command-line utility `STATUS` parameter specifies the frequency at which the job status display is updated.
- TABLES
  The Oracle Data Pump Export command-line utility `TABLES` parameter specifies that you want to perform a table-mode export.
- TABLESPACES
  The Oracle Data Pump Export command-line utility `TABLESPACES` parameter specifies a list of tablespace names that you want to be exported in tablespace mode.
- TRANSPORT_FULL_CHECK
  The Oracle Data Pump Export command-line utility `TRANSPORT_FULL_CHECK` parameter specifies whether to check for dependencies between objects
- TRANSPORT_TABLESPACES
  The Oracle Data Pump Export command-line utility `TRANSPORT_TABLESPACES` parameter specifies that you want to perform an export in transportable-tablespace mode.

- TRANSPORTABLE
  The Oracle Data Pump Export command-line utility `TRANSPORTABLE` parameter specifies whether the transportable option should be used during a table mode or full mode export.
- TTS_CLOSURE_CHECK
  The Oracle Data Pump Export command-line mode `TTS_CLOSURE_CHECK` parameter is used to indicate the degree of closure checking to be performed as part of a Data Pump transportable tablespace operation.
- VERSION
  The Data Pump Export command-line utility `VERSION` parameter specifies the version of database objects that you want to export.
- VIEWS_AS_TABLES
  The Oracle Data Pump Export command-line utility `VIEWS_AS_TABLES` parameter specifies that you want one or more views exported as tables.

Related Topics

- PARFILE
  The Oracle Data Pump Export command-line utility `PARFILE` parameter specifies the name of an export parameter file.
- Understanding Dump, Log, and SQL File Default Locations
  Oracle Data Pump is server-based, rather than client-based. Dump files, log files, and SQL files are accessed relative to server-based directory paths.
- Examples of Using Oracle Data Pump Export
  You can use these common scenario examples to learn how you can create parameter files and use Oracle Data Pump Export to move your data.
- Syntax Diagrams for Oracle Data Pump Export
  You can use syntax diagrams to understand the valid SQL syntax for Oracle Data Pump Export.

2.4.1 About Data Pump Export Parameters

Learn how to use Oracle Data Pump Export parameters in command-line mode, including case sensitivity, quotation marks, escape characters, and information about how to use examples.

Specifying Export Parameters

For parameters that can have multiple values specified, you can specify the values by commas, or by spaces. For example, you can specify `TABLES=employees,jobs` or `TABLES=employees jobs`.

For every parameter you enter, you must enter an equal sign (=), and a value. Data Pump has no other way of knowing that the previous parameter specification is complete and a new parameter specification is beginning. For example, in the following command line, even though `NOLOGFILE` is a valid parameter, Export interprets the string as another dump file name for the `DUMPFILE` parameter:

```
expdp DIRECTORY=dpumpdir DUMPFILE=test.dmp NOLOGFILE TABLES=employees
```

This command results in two dump files being created, `test.dmp` and `nologfile.dmp`.

To avoid this result, specify either `NOLOGFILE=YES` or `NOLOGFILE=NO`.

Case Sensitivity When Specifying Parameter Values

For tablespace names, schema names, table names, and so on, that you enter as parameter values, Oracle Data Pump by default changes values entered as lowercase or mixed-case into uppercase. For example, if you enter `TABLE=hr.employees`, then it is changed to `TABLE=HR.EMPLOYEES`. To maintain case, you must enclose the value within quotation marks. For example, `TABLE="hr.employees"` would preserve the table name in all lower case. The name you enter must exactly match the name stored in the database.

Use of Quotation Marks On the Data Pump Command Line

Some operating systems treat quotation marks as special characters. These operating systems therefore do not pass quotation marks on to an application unless quotation marks are preceded by an escape character, such as the backslash (`\`). This requirement is true both on the command lin, and within parameter files. Some operating systems can require an additional set of single or double quotation marks on the command line around the entire parameter value containing the special characters.

The following examples are provided to illustrate these concepts. Note that your particular operating system can have different requirements. The documentation examples cannot fully anticipate operating environments, which are unique to each user.

In this example, the `TABLES` parameter is specified in a parameter file:

```
TABLES = \"MixedCaseTableName\"
```

If you specify that value on the command line, then some operating systems require that you surround the parameter file name using single quotation marks, as follows:

```
TABLES = '\"MixedCaseTableName\"'
```

To avoid having to supply more quotation marks on the command line, Oracle recommends the use of parameter files. Also, note that if you use a parameter file, and the parameter value being specified does not have quotation marks as the first character in the string (for example, `TABLES=scott."EmP"`), then some operating systems do not require the use of escape characters.

Using the Export Parameter Examples

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

- After you enter the user name and parameters as shown in the example, Export is started, and you are prompted for a password. You are required to enter the password before a database connection is made.
- 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 are granted to the `hr` user for these directory objects.
- Some of the examples require the `DATAPUMP_EXP_FULL_DATABASE` and `DATAPUMP_IMP_FULL_DATABASE` roles. The examples assume that the `hr` user is granted these roles.

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

Unless specifically noted, you can also specify these parameters in a parameter file.

See Also:

- *Oracle Database Sample Schemas*
- Your Oracle operating system-specific documentation for information about how special and reserved characters are handled on your system

2.4.2 ABORT_STEP

The Oracle Data Pump Export command-line utility `ABORT_STEP` parameter stops the job after it is initialized.

Default

Null

Purpose

Used to stop the job after it is initialized. Stopping a job after it is initialized enables you to query the Data Pump control job table that you want to query before any data is exported.

Syntax and Description

```
ABORT_STEP=[n | -1]
```

The possible values correspond to a process order number in the Data Pump control job table. The result of using each number is as follows:

- *n*: If the value is zero or greater, then the export operation is started, and the job is stopped at the object that is stored in the Data Pump control job table with the corresponding process order number.
- `-1`: If the value is negative one (`-1`), then abort the job after setting it up, but before exporting any objects or data.

Restrictions

- None

Example

```
> expdp hr DIRECTORY=dpump_dir1 DUMPFILE=expdat.dmp SCHEMAS=hr ABORT_STEP=-1
```

2.4.3 ACCESS_METHOD

The Oracle Data Pump Export command-line utility `ACCESS_METHOD` parameter instructs Export to use a particular method to unload data.

Default

`AUTOMATIC`

Purpose

Instructs Export to use a particular method to unload data.

Syntax and Description

```
ACCESS_METHOD=[AUTOMATIC | DIRECT_PATH | EXTERNAL_TABLE]
```

The `ACCESS_METHOD` parameter is provided so that you can try an alternative method if the default method does not work for some reason. All methods can be specified for a network export. If the data for a table cannot be unloaded with the specified access method, then the data displays an error for the table and continues with the next work item.

The available options are as follows:

- `AUTOMATIC` — Oracle Data Pump determines the best way to unload data for each table. Oracle recommends that you use `AUTOMATIC` whenever possible because it allows Data Pump to automatically select the most efficient method.
- `DIRECT_PATH` — Oracle Data Pump uses direct path unload for every table.
- `EXTERNAL_TABLE` — Oracle Data Pump uses a SQL `CREATE TABLE AS SELECT` statement to create an external table using data that is stored in the dump file. The `SELECT` clause reads from the table to be unloaded.

Restrictions

- To use the `ACCESS_METHOD` parameter with network exports, you must be using Oracle Database 12c Release 2 (12.2.0.1) or later.
- The `ACCESS_METHOD` parameter for Oracle Data Pump Export is not valid for transportable tablespace jobs.

Example

```
> expdp hr DIRECTORY=dpump_dir1 DUMPFILE=expdat.dmp SCHEMAS=hr
ACCESS_METHOD=EXTERNAL_TABLE
```

2.4.4 ATTACH

The Oracle Data Pump Export command-line utility `ATTACH` parameter attaches a worker or client session to an existing export job, and automatically places you in the interactive-command interface.

Default

The default is the job currently in the user schema, if there is only one.

Purpose

Attaches the worker session to an existing Data Pump control 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 DATAPUMP_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, the only other Data Pump parameter you can specify on the command line is ENCRYPTION_PASSWORD.
- If the job to which you are attaching was initially started using an encryption password, then when you attach to the job, you must again enter the ENCRYPTION_PASSWORD parameter on the command line to respecify that password. The only exception to this requirement is if the job was initially started with the ENCRYPTION=ENCRYPTED_COLUMNS_ONLY parameter. In that case, the encryption password is not needed when attaching to the job.
- You cannot attach to a job in another schema unless it is already running.
- If the dump file set or Data Pump control table for the job have been deleted, then the attach operation fails.
- Altering the Data Pump control table in any way leads to unpredictable results.

Example

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

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

Related Topics

- Commands Available in Data Pump Export Interactive-Command Mode

2.4.5 CLUSTER

The Oracle Data Pump Export command-line utility CLUSTER parameter determines whether Data Pump can use Oracle RAC, resources, and start workers on other Oracle RAC instances.

Default

YES

Purpose

Determines whether Oracle Data Pump can use Oracle Real Application Clusters (Oracle RAC) resources and start workers on other Oracle RAC instances.

Syntax and Description

```
CLUSTER=[YES | NO]
```

To force Oracle Data Pump Export to use only the instance where the job is started and to replicate pre-Oracle Database 11g release 2 (11.2) behavior, specify `CLUSTER=NO`.

To specify a specific, existing service, and constrain worker processes to run only on instances defined for that service, use the `SERVICE_NAME` parameter with the `CLUSTER=YES` parameter.

Use of the `CLUSTER` parameter can affect performance, because there is some additional overhead in distributing the export job across Oracle RAC instances. For small jobs, it can be better to specify `CLUSTER=NO` to constrain the job to run on the instance where it is started. Jobs whose performance benefits the most from using the `CLUSTER` parameter are those involving large amounts of data.

Example

The following is an example of using the `CLUSTER` parameter:

```
> expdp hr DIRECTORY=dpump_dir1 DUMPFILE=hr_clus%U.dmp CLUSTER=NO PARALLEL=3
```

This example starts a schema-mode export (the default) of the `hr` schema. Because `CLUSTER=NO` is specified, the job uses only the instance on which it started. (If you do not specify the `CLUSTER` parameter, then the default value of `Y` is used. With that value, if necessary, workers are started on other instances in the Oracle RAC cluster). The dump files are written to the location specified for the `dpump_dir1` directory object. The job can have up to 3 parallel processes.

Related Topics

- SERVICE_NAME
  The Oracle Data Pump Export command-line utility `SERVICE_NAME` parameter specifies a service name that you want to use in conjunction with the `CLUSTER` parameter.
- Understanding How to Use Oracle Data Pump with Oracle RAC
  Using Oracle Data Pump in an Oracle Real Application Clusters (Oracle RAC) environment requires you to perform a few checks to ensure that you are making cluster member nodes available.

2.4.6 COMPRESSION

The Oracle Data Pump Export command-line utility `COMPRESSION` parameter specifies which data to compress before writing to the dump file set.

Default

```
METADATA_ONLY
```

Purpose

Specifies which data to compress before writing to the dump file set.

Syntax and Description

```
COMPRESSION=[ALL | DATA_ONLY | METADATA_ONLY | NONE]
```

- `ALL` enables compression for the entire export operation. The `ALL` option requires that the Oracle Advanced Compression option is enabled.
- `DATA_ONLY` results in all data being written to the dump file in compressed format. The `DATA_ONLY` option requires that the Oracle Advanced Compression option is enabled.
- `METADATA_ONLY` results in all metadata being written to the dump file in compressed format. This is the default.
- `NONE` disables compression for the entire export operation.

Restrictions

- To make full use of all these compression options, the `COMPATIBLE` initialization parameter must be set to at least 11.0.0.
- The `METADATA_ONLY` option can be used even if the `COMPATIBLE` initialization parameter is set to 10.2.
- Compression of data using `ALL` or `DATA_ONLY` is valid only in the Enterprise Edition of Oracle Database 11g or later, and requires that the Oracle Advanced Compression option is enabled.

Example

The following is an example of using the `COMPRESSION` parameter:

```
> expdp hr DIRECTORY=dpump_dir1 DUMPFILE=hr_comp.dmp COMPRESSION=METADATA_ONLY
```

This command runs a schema-mode export that compresses all metadata before writing it out to the dump file, `hr_comp.dmp`. It defaults to a schema-mode export, because no export mode is specified.

See *Oracle Database Licensing Information* for information about licensing requirements for the Oracle Advanced Compression option.

Related Topics

- Oracle Database Options and Their Permitted Features

2.4.7 COMPRESSION_ALGORITHM

The Oracle Data Pump Export command-line utility `COMPRESSION_ALGORITHM` parameter specifies the compression algorithm that you want to use when compressing dump file data.

Default

```
BASIC
```

Purpose

Specifies the compression algorithm to be used when compressing dump file data.

Syntax and Description

```
COMPRESSION_ALGORITHM = [BASIC | LOW | MEDIUM | HIGH]
```

The parameter options are defined as follows:

- `BASIC`: Offers a good combination of compression ratios and speed; the algorithm used is the same as in previous versions of Oracle Data Pump.
- `LOW`: Least impact on export throughput. This option is suited for environments where CPU resources are the limiting factor.
- `MEDIUM`: Recommended for most environments. This option, like the `BASIC` option, provides a good combination of compression ratios and speed, but it uses a different algorithm than `BASIC`.
- `HIGH`: Best suited for situations in which dump files are copied over slower networks, where the limiting factor is network speed.

You characterize the performance of a compression algorithm by its CPU usage, and by the compression ratio (the size of the compressed output as a percentage of the uncompressed input). These measures vary, based on the size and type of inputs, as well as the speed of the compression algorithms used. The compression ratio generally increases from low to high, with a trade-off of potentially consuming more CPU resources.

Oracle recommends that you run tests with the different compression levels on the data in your environment. Choosing a compression level based on your environment, workload characteristics, and size and type of data is the only way to ensure that the exported dump file set compression level meets your performance and storage requirements.

Restrictions

- To use this feature, database compatibility must be set to 12.0.0 or later.
- This feature requires that you have the Oracle Advanced Compression option enabled.

Example 1

This example performs a schema-mode unload of the `HR` schema, and compresses only the table data using a compression algorithm with a low level of compression. Using this command option can result in fewer CPU resources being used, at the expense of a less than optimal compression ratio.

```
    > expdp hr DIRECTORY=dpump_dir1 DUMPFILE=hr.dmp COMPRESSION=DATA_ONLY
COMPRESSION_ALGORITHM=LOW
```

Example 2

This example performs a schema-mode unload of the `HR` schema, and compresses both metadata and table data using the basic level of compression. Omitting the `COMPRESSION_ALGORITHM` parameter altogether is equivalent to specifying `BASIC` as the value.

```
    > expdp hr DIRECTORY=dpump_dir1 DUMPFILE=hr.dmp COMPRESSION=ALL
COMPRESSION_ALGORITHM=BASIC
```

2.4.8 CONTENT

The Oracle Data Pump Export command-line utility `CONTENT` parameter enables you to filter what Export unloads: data only, metadata only, or both.

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 option 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. Be aware that if you specify `CONTENT=METADATA_ONLY`, then afterward, when the dump file is imported, any index or table statistics imported from the dump file are locked after the import.

Restrictions

- The `CONTENT=METADATA_ONLY` parameter cannot be used with the `TRANSPORT_TABLESPACES` (transportable-tablespace mode) parameter or with the `QUERY` parameter.

Example

The following is an example of using the `CONTENT` parameter:

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

This command executes a schema-mode export that unloads 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.

2.4.9 DATA_OPTIONS

The Oracle Data Pump Export command-line utility `DATA_OPTIONS` parameter designates how you want certain types of data handled during export operations.

Default

There is no default. If this parameter is not used, then the special data handling options it provides do not take effect.

Purpose

The `DATA_OPTIONS` parameter designates how certain types of data should be handled during export operations.

Syntax and Description

```
DATA_OPTIONS= [GROUP_PARTITION_TABLE_DATA | VERIFY_STREAM_FORMAT]
```

- `GROUP_PARTITION_TABLE_DATA`: Tells Oracle Data Pump to unload all table data in one operation rather than unload each table partition as a separate operation. As a result, the definition of the table will not matter at import time because Import will see one partition of data that will be loaded into the entire table.
- `VERIFY_STREAM_FORMAT`: Validates the format of a data stream before it is written to the Oracle Data Pump dump file. The verification checks for a valid format for the stream after it is generated but before it is written to disk. This assures that there are no errors when the dump file is created, which in turn helps to assure that there will not be errors when the stream is read at import time.

Restrictions

The Export `DATA_OPTIONS` parameter requires the job version to be set to `11.0.0` or later. See `VERSION`.

Example

This example shows an export operation in which data for all partitions of a table are unloaded together instead of the default behavior of unloading the data for each partition separately.

```
> expdp hr TABLES=hr.tab1 DIRECTORY=dpump_dir1 DUMPFILE=hr.dmp VERSION=11.2
GROUP_PARTITION_TABLE_DATA
```

See *Oracle XML DB Developer's Guide* for information specific to exporting and importing `XMLType` tables.

Related Topics

- VERSION

2.4.10 DIRECTORY

The Oracle Data Pump Export command-line utility `DIRECTORY` parameter specifies the default location to which Export can write the dump file set and the log file.

Default

```
DATA_PUMP_DIR
```

Purpose

Specifies the default 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. It is not the file path of an actual directory. Privileged users have access to a default directory object named `DATA_PUMP_DIR`. The definition of the `DATA_PUMP_DIR` directory can be changed by Oracle during upgrades, or when patches are applied.

Users with access to the default DATA_PUMP_DIR directory object do not need to use the DIRECTORY parameter.

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

Example

The following is an example of using the DIRECTORY parameter:

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

In this example, the dump file, employees.dump is written to the path that is associated with the directory object dpump_dir1.

Related Topics

- Understanding Dump, Log, and SQL File Default Locations
- Understanding How to Use Oracle Data Pump with Oracle RAC
- *Oracle Database SQL Language Reference*

2.4.11 DUMPFILE

The Oracle Data Pump Export command-line utility DUMPFILE parameter specifies the names, and optionally, the directory objects of dump files for an export job.

Default

```
expdat.dmp
```

Purpose

Specifies the names, and if you choose to do so, the directory objects of dump files for an export job.

Syntax and Description

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

Specifying *directory_object* is optional if you have already specified the directory object by using the DIRECTORY parameter. If you supply a value here, then it must be a directory object that exists, and to which you have access. A database directory object that is specified as part of the DUMPFILE parameter overrides a value specified by the DIRECTORY parameter, or by the default directory object.

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 file name, then Export uses the default file extension of .dmp. The file names can contain a substitution variable. The following table lists the available substitution variables.

Substitution Variable	Meaning
%U	The substitution variable is expanded in the resulting file names into a 2-digit, fixed-width, incrementing integer that starts at 01 and ends at 99. If a file specification contains two substitution variables, then both are incremented at the same time. For example, exp%Uaa%U.dmp resolves to exp01aa01.dmp, exp02aa02.dmp, and so forth.
%d, %D	Specifies the current day of the month from the Gregorian calendar in format DD. Note: This substitution variable cannot be used in an import file name.
%m, %M	Specifies the month in the Gregorian calendar in format MM. Note: This substitution variable cannot be used in an import file name.
%t, %T	Specifies the year, month, and day in the Gregorian calendar in this format: YYYYMMDD. Note: This substitution variable cannot be used in an import file name.
%l, %L	Specifies a system-generated unique file name. The file names can contain a substitution variable (%L), which implies that multiple files can be generated. The substitution variable is expanded in the resulting file names into a 2-digit, fixed-width, incrementing integer starting at 01 and ending at 99 which is the same as (%U). In addition, the substitution variable is expanded in the resulting file names into a 3-digit to 10-digit, variable-width, incrementing integers starting at 100 and ending at 2147483646. The width field is determined by the number of digits in the integer. For example if the current integer is 1, then exp%Laa%L.dmp resolves to: exp01aa01.dmp exp02aa02.dmp and so forth, up until 99. Then, the next file name has 3 digits substituted: exp100aa100.dmp exp101aa101.dmp and so forth, up until 999, where the next file has 4 digits substituted. The substitutions continue up to the largest number substitution allowed, which is 2147483646.
%y, %Y	Specifies the year in this format: YYYY. Note: This substitution variable cannot be used in an import file name.

If the FILESIZE parameter is specified, then each dump file has a maximum of that size and be nonextensible. If more space is required for the dump file set, and a template with a substitution variable was supplied, then a new dump file is automatically created of the size specified by the FILESIZE parameter, if there is room on the device.

As each file specification or file template containing a substitution variable is defined, it is instantiated into one fully qualified file name, and Export attempts to create the file. 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 more files are created if file templates with substitution variables were specified.

Although it is possible to specify multiple files using the DUMPFILE parameter, the export job can only require a subset of those files to hold the exported data. The dump file set displayed at the end of the export job shows exactly which files were used. It is this list of files that is required to perform an import operation using this dump file set. Any files that were not used can be discarded.

When you specify the DUMPFILE parameter, it is possible to introduce conflicting file names, regardless of whether substitution variables are used. The following are some examples of expdp commands that would produce file name conflicts. For all these examples, an ORA-27308 created file already exists error is returned:

```
expdp system/manager directory=dpump_dir schemas=hr
DUMPFILE=foo%U.dmp,foo%U.dmp
```

```
expdp system/manager directory=dpump_dir schemas=hr
DUMPFILE=foo%U.dmp,foo%L.dmp
```

```
expdp system/manager directory=dpump_dir schemas=hr
DUMPFILE=foo%U.dmp,foo%D.dmp
```

```
expdp system/manager directory =dpump_dir schemas=hr
DUMPFILE=foo%tK_%t_%u_%y_P,foo%TK_%T_%U_%Y_P
```

Restrictions

- Any resulting dump file names that match preexisting dump file names generate an error, and the preexisting dump files are not overwritten. You can override this behavior by specifying the Export parameter REUSE_DUMPFILES=YES.
- Dump files created on Oracle Database 11g releases with the Oracle Data Pump parameter VERSION=12 can only be imported on Oracle Database 12c Release 1 (12.1) and later.

Example

The following is an example of using the DUMPFILE parameter:

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

The dump file, exp1.dmp, is 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 are given work to perform during this job, dump files named exp201.dmp and exp202.dmp is created, based on the specified substitution variable exp2%U.dmp. Because no directory is specified for them, they are written to the path associated with the directory object, dpump_dir1, that was specified with the DIRECTORY parameter.

Related Topics

- Using Substitution Variables with Oracle Data Pump Exports

2.4.12 ENABLE_SECURE_ROLES

The Oracle Data Pump Export command-line utility ENABLE_SECURE_ROLES parameter prevents inadvertent use of protected roles during exports.

Default

In Oracle Database 19c and later releases, the default value is NO.

Purpose

Some Oracle roles require authorization. If you need to use these roles with Oracle Data Pump exports, then you must explicitly enable them by setting the ENABLE_SECURE_ROLES parameter to YES.

Syntax

```
ENABLE_SECURE_ROLES=[NO|YES]
```

- NO Disables Oracle roles that require authorization.
- YES Enables Oracle roles that require authorization.

Example

```
expdp hr SCHEMAS=hr DIRECTORY=dpump_dir1 DUMPFILE=dpump_dir2:exp1.dmp,
 exp2%U.dmp ENABLE_SECURE_ROLES=YES
```

2.4.13 ENCRYPTION

The Oracle Data Pump Export command-line utility ENCRYPTION parameter specifies whether to encrypt data before writing it to the dump file set.

Default

The default value depends upon the combination of encryption-related parameters that are used. To enable encryption, either the ENCRYPTION or ENCRYPTION_PASSWORD parameter, or both, must be specified.

If only the ENCRYPTION_PASSWORD parameter is specified, then the ENCRYPTION parameter defaults to ALL.

If only the ENCRYPTION parameter is specified and the Oracle encryption wallet is open, then the default mode is TRANSPARENT. If only the ENCRYPTION parameter is specified and the wallet is closed, then an error is returned.

If neither ENCRYPTION nor ENCRYPTION_PASSWORD is specified, then ENCRYPTION defaults to NONE.

Purpose

Specifies whether to encrypt data before writing it to the dump file set.

Syntax and Description

```
ENCRYPTION = [ALL | DATA_ONLY | ENCRYPTED_COLUMNS_ONLY | METADATA_ONLY | NONE]
```

- ALL enables encryption for all data and metadata in the export operation.

- DATA_ONLY specifies that only data is written to the dump file set in encrypted format.
- ENCRYPTED_COLUMNS_ONLY specifies that only encrypted columns are written to the dump file set in encrypted format. This option cannot be used with the ENCRYPTION_ALGORITHM parameter because the columns already have an assigned encryption format and by definition, a column can have only one form of encryption.

  To use the ENCRYPTED_COLUMNS_ONLY option, you must also use the ENCRYPTION_PASSWORD parameter.

  To use the ENCRYPTED_COLUMNS_ONLY option, you must have Oracle Advanced Security Transparent Data Encryption (TDE) enabled. See *Oracle Database Advanced Security Guide* for more information about TDE.
- METADATA_ONLY specifies that only metadata is written to the dump file set in encrypted format.
- NONE specifies that no data is written to the dump file set in encrypted format.

SecureFiles Considerations for Encryption

If the data being exported includes SecureFiles that you want to be encrypted, then you must specify ENCRYPTION=ALL to encrypt the entire dump file set. Encryption of the entire dump file set is the only way to achieve encryption security for SecureFiles during a Data Pump export operation. For more information about SecureFiles, see *Oracle Database SecureFiles and Large Objects Developer's Guide*.

Oracle Database Vault Considerations for Encryption

When an export operation is started, Data Pump determines whether Oracle Database Vault is enabled. If it is, and dump file encryption has not been specified for the job, a warning message is returned to alert you that secure data is being written in an insecure manner (clear text) to the dump file set:

```
ORA-39327: Oracle Database Vault data is being stored unencrypted in dump
file set
```

You can stop the current export operation and start a new one, specifying that you want the output dump file set to be encrypted.

Restrictions

- To specify the ALL, DATA_ONLY, or METADATA_ONLY options, the COMPATIBLE initialization parameter must be set to at least 11.0.0.
- This parameter is valid only in the Enterprise Edition of Oracle Database 11g or later.
- To use the ALL, DATA_ONLY or METADATA_ONLY options without also using an encryption password, you must have the Oracle Advanced Security option enabled. See *Oracle Database Licensing Information* for information about licensing requirements for the Oracle Advanced Security option.

Example

The following example performs an export operation in which only data is encrypted in the dump file:

```
> expdp hr DIRECTORY=dpump_dir1 DUMPFILE=hr_enc.dmp JOB_NAME=enc1
ENCRYPTION=data_only ENCRYPTION_PASSWORD=foobar
```

Related Topics

- *Oracle Database Security Guide*
- SecureFiles LOB Storage
- Oracle Database Options and Their Permitted Features

2.4.14 ENCRYPTION_ALGORITHM

The Oracle Data Pump Export command-line utility `ENCRYPTION_ALGORITHM` parameter specifies which cryptographic algorithm should be used to perform the encryption.

Default

`AES128`

Purpose

Specifies which cryptographic algorithm should be used to perform the encryption.

Syntax and Description

```
ENCRYPTION_ALGORITHM = [AES128 | AES192 | AES256]
```

Restrictions

- To use this encryption feature, the `COMPATIBLE` initialization parameter must be set to at least 11.0.0.
- The `ENCRYPTION_ALGORITHM` parameter requires that you also specify either the `ENCRYPTION` or `ENCRYPTION_PASSWORD` parameter; otherwise an error is returned.
- The `ENCRYPTION_ALGORITHM` parameter cannot be used in conjunction with `ENCRYPTION=ENCRYPTED_COLUMNS_ONLY` because columns that are already encrypted cannot have an additional encryption format assigned to them.
- This parameter is valid only in the Enterprise Edition of Oracle Database 11g or later.
- The `ENCRYPTION _ALGORITHM` parameter does not require that you have the Oracle Advanced Security enabled, but it can be used in conjunction with other encryption-related parameters that do require that option. See *Oracle Database Licensing Information* for information about licensing requirements for the Oracle Advanced Security option.

Example

```
> expdp hr DIRECTORY=dpump_dir1 DUMPFILE=hr_enc3.dmp
ENCRYPTION_PASSWORD=foobar ENCRYPTION_ALGORITHM=AES128
```

Related Topics

- *Oracle Database Security Guide*
- *Oracle Database Licensing Information User Manual*

2.4.15 ENCRYPTION_MODE

The Oracle Data Pump Export command-line utility ENCRYPTION_MODE parameter specifies the type of security to use when encryption and decryption are performed.

Default

The default mode depends on which other encryption-related parameters are used. If only the ENCRYPTION parameter is specified and the Oracle encryption wallet is open, then the default mode is TRANSPARENT. If only the ENCRYPTION parameter is specified and the wallet is closed, then an error is returned.

If the ENCRYPTION_PASSWORD parameter is specified and the wallet is open, then the default is DUAL. If the ENCRYPTION_PASSWORD parameter is specified and the wallet is closed, then the default is PASSWORD.

Purpose

Specifies the type of security to use when encryption and decryption are performed.

Syntax and Description

```
ENCRYPTION_MODE = [DUAL | PASSWORD | TRANSPARENT]
```

DUAL mode creates a dump file set that can later be imported either transparently or by specifying a password that was used when the dual-mode encrypted dump file set was created. When you later import the dump file set created in DUAL mode, you can use either the wallet or the password that was specified with the ENCRYPTION_PASSWORD parameter. DUAL mode is best suited for cases in which the dump file set will be imported on-site using the wallet, but which may also need to be imported offsite where the wallet is not available.

PASSWORD mode requires that you provide a password when creating encrypted dump file sets. You will need to provide the same password when you import the dump file set. PASSWORD mode requires that you also specify the ENCRYPTION_PASSWORD parameter. The PASSWORD mode is best suited for cases in which the dump file set will be imported into a different or remote database, but which must remain secure in transit.

TRANSPARENT mode enables you to create an encrypted dump file set without any intervention from a database administrator (DBA), provided the required wallet is available. Therefore, the ENCRYPTION_PASSWORD parameter is not required. The parameter will, in fact, cause an error if it is used in TRANSPARENT mode. This encryption mode is best suited for cases in which the dump file set is imported into the same database from which it was exported.

Restrictions

- To use DUAL or TRANSPARENT mode, the COMPATIBLE initialization parameter must be set to at least 11.0.0.
- When you use the ENCRYPTION_MODE parameter, you must also use either the ENCRYPTION or ENCRYPTION_PASSWORD parameter. Otherwise, an error is returned.
- When you use the ENCRYPTION=ENCRYPTED_COLUMNS_ONLY, you cannot use the ENCRYPTION_MODE parameter. Otherwise, an error is returned.
- This parameter is valid only in the Enterprise Edition of Oracle Database 11g or later.

- The use of `DUAL` or `TRANSPARENT` mode requires that the Oracle Advanced Security option is enabled. See *Oracle Database Licensing Information* for information about licensing requirements for the Oracle Advanced Security option.

Example

```
> expdp hr DIRECTORY=dpump_dir1 DUMPFILE=hr_enc4.dmp
ENCRYPTION=all ENCRYPTION_PASSWORD=secretwords
ENCRYPTION_ALGORITHM=AES256 ENCRYPTION_MODE=DUAL
```

Related Topics

- *Oracle Database Licensing Information User Manual*

2.4.16 ENCRYPTION_PASSWORD

The Oracle Data Pump Export command-line utility `ENCRYPTION_PASSWORD` parameter specifies a password for encrypting encrypted column data, metadata, or table data in the export dump file. This parameter prevents unauthorized access to an encrypted dump file set.

Default: There is no default; the value is user-provided.

Purpose

Specifies a password for encrypting encrypted column data, metadata, or table data in the export dump file.Using this parameter prevents unauthorized access to an encrypted dump file set.

> **Note:**
>
> Oracle Data Pump encryption functionality changed as of Oracle Database 11g release 1 (11.1). Before release 11.1, the `ENCRYPTION_PASSWORD` parameter applied only to encrypted columns. However, as of release 11.1, the new `ENCRYPTION` parameter provides options for encrypting other types of data. As a result of this change, if you now specify `ENCRYPTION_PASSWORD` without also specifying `ENCRYPTION` and a specific option, then all data written to the dump file is encrypted (equivalent to specifying `ENCRYPTION=ALL`). To re-encrypt only encrypted columns, you must now specify `ENCRYPTION=ENCRYPTED_COLUMNS_ONLY` in addition to `ENCRYPTION_PASSWORD`.

Syntax and Description

```
ENCRYPTION_PASSWORD = password
```

The *`password`* value that is supplied specifies a key for re-encrypting encrypted table columns, metadata, or table data so that they are not written as clear text in the dump file set. If the export operation involves encrypted table columns, but an encryption password is not supplied, then the encrypted columns are written to the dump file set as clear text and a warning is issued.

The password that you enter is echoed to the screen. If you do not want the password shown on the screen as you enter it, then use the `ENCRYPTION_PWD_PROMPT` parameter.

The maximum length allowed for an encryption password is usually 128 bytes. However, the limit is 30 bytes if ENCRYPTION=ENCRYPTED_COLUMNS_ONLY and either the VERSION parameter or database compatibility is set to less than 12.2.

For export operations, this parameter is required if the ENCRYPTION_MODE parameter is set to either PASSWORD or DUAL.

> **Note:**
>
> There is no connection or dependency between the key specified with the Data Pump ENCRYPTION_PASSWORD parameter and the key specified with the ENCRYPT keyword when the table with encrypted columns was initially created. For example, suppose that a table is created as follows, with an encrypted column whose key is xyz:
>
> ```
> CREATE TABLE emp (col1 VARCHAR2(256) ENCRYPT IDENTIFIED BY "xyz");
> ```
>
> When you export the emp table, you can supply any arbitrary value for ENCRYPTION_PASSWORD. It does not have to be xyz.

Restrictions

- This parameter is valid only in the Enterprise Edition of Oracle Database 11g or later.
- The ENCRYPTION_PASSWORD parameter is required for the transport of encrypted tablespaces and tablespaces containing tables with encrypted columns in a full transportable export.
- Oracle Data Pump encryption features require that the Oracle Advanced Security option be enabled. See *Oracle Database Licensing Information* for information about licensing requirements for the Oracle Advanced Security option.
- The ENCRYPTION_PASSWORD parameter is required for the transport of encrypted tablespaces and tablespaces containing tables with encrypted columns in a full transportable export.
- If ENCRYPTION_PASSWORD is specified but ENCRYPTION_MODE is not specified, then it is not necessary to have Oracle Advanced Security Transparent Data Encryption enabled, because ENCRYPTION_MODE defaults to PASSWORD.
- If the requested encryption mode is TRANSPARENT, then the ENCRYPTION_PASSWORD parameter is not valid.
- If ENCRYPTION_MODE is set to DUAL, then to use the ENCRYPTION_PASSWORD parameter, then you must have Oracle Advanced Security Transparent Data Encryption (TDE) enabled. See *Oracle Database Advanced Security Guide* for more information about TDE.
- For network exports, the ENCRYPTION_PASSWORD parameter in conjunction with ENCRYPTION=ENCRYPTED_COLUMNS_ONLY is not supported with user-defined external tables that have encrypted columns. The table is skipped, and an error message is displayed, but the job continues.
- Encryption attributes for all columns must match between the exported table definition and the target table. For example, suppose you have a table, EMP, and one of its columns is named EMPNO. Both of the following scenarios result in an error, because the encryption attribute for the EMP column in the source table does not match the encryption attribute for the EMP column in the target table:

- The EMP table is exported with the EMPNO column being encrypted, but before importing the table you remove the encryption attribute from the EMPNO column.
- The EMP table is exported without the EMPNO column being encrypted, but before importing the table you enable encryption on the EMPNO column.

Example

In the following example, an encryption password, 123456, is assigned to the dump file, dpcd2be1.dmp.

```
> expdp hr TABLES=employee_s_encrypt DIRECTORY=dpump_dir1
DUMPFILE=dpcd2be1.dmp ENCRYPTION=ENCRYPTED_COLUMNS_ONLY
ENCRYPTION_PASSWORD=123456
```

Encrypted columns in the employee_s_encrypt table are not written as clear text in the dpcd2be1.dmp dump file. Afterward, if you want to import the dpcd2be1.dmp file created by this example, then you must supply the same encryption password.

Related Topics

- *Oracle Database Licensing Information User Manual*
- *Oracle Database Advanced Security Guide*

2.4.17 ENCRYPTION_PWD_PROMPT

The Oracle Data Pump Export command-line utility ENCRYPTION_PWD_PROMPT specifies whether Oracle Data Pump prompts you for the encryption password.

Default

NO

Purpose

Specifies whether Data Pump should prompt you for the encryption password.

Syntax and Description

```
ENCRYPTION_PWD_PROMPT=[YES | NO]
```

Specify ENCRYPTION_PWD_PROMPT=YES on the command line to instruct Data Pump to prompt you for the encryption password, rather than you entering it on the command line with the ENCRYPTION_PASSWORD parameter. The advantage to doing this is that the encryption password is not echoed to the screen when it is entered at the prompt. Whereas, when it is entered on the command line using the ENCRYPTION_PASSWORD parameter, it appears in plain text.

The encryption password that you enter at the prompt is subject to the same criteria described for the ENCRYPTION_PASSWORD parameter.

If you specify an encryption password on the export operation, you must also supply it on the import operation.

Restrictions

- Concurrent use of the `ENCRYPTION_PWD_PROMPT` and `ENCRYPTION_PASSWORD` parameters is prohibited.

Example

The following syntax example shows Data Pump first prompting for the user password and then for the encryption password.

```
> expdp hr DIRECTORY=dpump_dir1 DUMPFILE=hr.dmp ENCRYPTION_PWD_PROMPT=YES
.
.
.
Copyright (c) 1982, 2017, Oracle and/or its affiliates.  All rights reserved.

Password:

Connected to: Oracle Database 18c Enterprise Edition Release 18.0.0.0.0 -
Production
Version 18.1.0.0.0

Encryption Password:

Starting "HR"."SYS_EXPORT_SCHEMA_01":  hr/******** directory=dpump_dir1
dumpfile=hr.dmp encryption_pwd_prompt=Y
.
.
.
```

2.4.18 ESTIMATE

The Oracle Data Pump Export command-line utility `ESTIMATE` parameter specifies the method that Export uses to estimate how much disk space each table in the export job will consume (in bytes).

Default

```
STATISTICS
```

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 source 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. (Table analysis can be done with either the SQL `ANALYZE` statement or the `DBMS_STATS` PL/SQL package.)

Restrictions

- If the Data Pump export job involves compressed tables, then when you use `ESTIMATE=BLOCKS`, the default size estimation given for the compressed table is inaccurate. This inaccuracy results because the size estimate does not reflect that the data was stored in a compressed form. To obtain a more accurate size estimate for compressed tables, use `ESTIMATE=STATISTICS`.
- If either the `QUERY` or `REMAP_DATA` parameter is used, then the estimate can also be inaccurate.

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 TABLES=employees ESTIMATE=STATISTICS DIRECTORY=dpump_dir1
 DUMPFILE=estimate_stat.dmp
```

2.4.19 ESTIMATE_ONLY

The Oracle Data Pump Export command-line utility `ESTIMATE_ONLY` parameter instructs Export to estimate the space that a job consumes, without actually performing the export operation.

Default

`NO`

Purpose

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

Syntax and Description

```
ESTIMATE_ONLY=[YES | NO]
```

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

Restrictions

- The `ESTIMATE_ONLY` parameter cannot be used in conjunction with the `QUERY` parameter.

Example

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

```
> expdp hr ESTIMATE_ONLY=YES NOLOGFILE=YES SCHEMAS=HR
```

2.4.20 EXCLUDE

The Data Pump Export command-line utility `EXCLUDE` parameter enables you to filter the metadata that is exported by specifying objects and object types that you want to exclude from the export operation.

Default: There is no default

Purpose

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

Syntax and Description

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

The *object_type* specifies the type of object that you want to exclude. To see a list of valid values for *object_type*, query the following views: `DATABASE_EXPORT_OBJECTS` for full mode, `SCHEMA_EXPORT_OBJECTS` for schema mode, and `TABLE_EXPORT_OBJECTS` for table and tablespace mode. The values listed in the `OBJECT_PATH` column are the valid object types.

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

The *name_clause* is optional. Using this parameter enables selection of specific objects within an object type. It is a SQL expression used as a filter on the object names of that type. It consists of a SQL operator, and the values against which you want to compare the object names of the specified type. The *name_clause* applies only to object types whose instances have names (for example, it is applicable to `TABLE`, but not to `GRANT`). It 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 can set `EXCLUDE=INDEX:"LIKE 'EMP%'"` to exclude all indexes whose names start with `EMP`.

The name that you supply for the *name_clause* must exactly match, including upper and lower casing, an existing object in the database. For example, if the *name_clause* you supply is for a table named `EMPLOYEES`, then there must be an existing table named `EMPLOYEES` using all upper case. If you supplied the *name_clause* as `Employees` or `employees` or any other variation that does not match the existing table, then the table is not found.

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

You can specify more than one `EXCLUDE` statement.

Depending on your operating system, the use of quotation marks when you specify a value for this parameter can also require that you use escape characters. Oracle recommends that you place this parameter in a parameter file, which can reduce the number of escape characters that otherwise can be needed on the command line.

If the *object_type* you specify is `CONSTRAINT`, `GRANT`, or `USER`, then be aware of the effects, as described in the following paragraphs.

Excluding Constraints

The following constraints cannot be explicitly excluded:

- 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

For example, the following `EXCLUDE` statements are interpreted as follows:

- `EXCLUDE=CONSTRAINT` excludes all constraints, except for any constraints needed for successful table creation and loading.
- `EXCLUDE=REF_CONSTRAINT` excludes 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 user schemas.

To exclude a specific user and all objects of that user, specify a command such as the following, where `hr` is the schema name of the user you want to exclude.

```
expdp FULL=YES DUMPFILE=expfull.dmp EXCLUDE=SCHEMA:"='HR'"
```

In this example, the export mode `FULL` is specified. If no mode is specified, then the default mode is used. The default mode is `SCHEMAS`. But if the default mode is used, then in this example, the default causes an error, because if `SCHEMAS` is used, then the command indicates that you want the schema both exported and excluded at the same time.

If you try to exclude a user by using a statement such as `EXCLUDE=USER:"='HR'"`, then only the information used in `CREATE USER hr` DDL statements is excluded, and you can obtain unexpected results.

Restrictions

- The `EXCLUDE` and `INCLUDE` parameters are mutually exclusive.

Example

The following is an example of using the `EXCLUDE` statement.

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

This example results in a schema-mode export (the default export mode) in which all the `hr` schema is exported except its views, packages, and functions.

Related Topics

- Metadata Filters
- Filtering During Export Operations
- INCLUDE
  The Data Pump Export command-line utility INCLUDE parameter 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.

- Parameters Available in Data Pump Export Command-Line Mode

2.4.21 FILESIZE

The Oracle Data Pump Export command-line utility `FILESIZE` parameter specifies the maximum size of each dump file.

Default

`0` (equivalent to the maximum size of 16 terabytes)

Purpose

Specifies the maximum size of each dump file. If the size is reached for any member of the dump file set, then that file is closed and an attempt is made to create a new file, if the file specification contains a substitution variable or if more dump files have been added to the job.

Syntax and Description

```
FILESIZE=integer[B | KB | MB | GB | TB]
```

The *integer* can be immediately followed (do not insert a space) by `B`, `KB`, `MB`, `GB`, or `TB` (indicating bytes, kilobytes, megabytes, gigabytes, and terabytes respectively). Bytes is the default. The actual size of the resulting file can be rounded down slightly to match the size of the internal blocks used in dump files.

Restrictions

- The minimum size for a file is 10 times the default Data Pump block size, which is 4 kilobytes.
- The maximum size for a file is 16 terabytes.

Example

The following example shows setting the size of the dump file to 3 megabytes:

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

In this scenario, if the 3 megabytes allocated was not sufficient to hold all the exported data, then the following error results, and displayed and the job stops:

```
ORA-39095: Dump file space has been exhausted: Unable to allocate 217088 bytes
```

The actual number of bytes that cannot be allocated can vary. Also, this number does not represent the amount of space required complete the entire export operation. It indicates only the size of the current object that was being exported when the job ran out of dump file space. You can correct this problem by first attaching to the stopped job, adding one or more files using the `ADD_FILE` command, and then restarting the operation.

2.4.22 FLASHBACK_SCN

The Oracle Data Pump Export command-line utility `FLASHBACK_SCN` parameter specifies the system change number (SCN) that Export uses to enable the Flashback Query utility.

Default

Default: There is no default

Purpose

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

Syntax and Description

```
FLASHBACK_SCN=scn_value
```

The export operation is performed with data that is consistent up to the specified SCN. If the `NETWORK_LINK` parameter is specified, then the SCN refers to the SCN of the source database.

As of Oracle Database 12c release 2 (12.2) and later releases, the SCN value can be a big SCN (8 bytes). You can also specify a big SCN when you create a dump file for an earlier version that does not support big SCNs, because actual SCN values are not moved.

Restrictions

- `FLASHBACK_SCN` and `FLASHBACK_TIME` are mutually exclusive.
- The `FLASHBACK_SCN` parameter pertains only to the Flashback Query capability of Oracle Database. It is not applicable to Flashback Database, Flashback Drop, or Flashback Data Archive.
- You cannot specify a big SCN for a network export or network import from a version that does not support big SCNs.

Example

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

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

> **Note:**
>
> If you are on a logical standby system and using a network link to access the logical standby primary, then 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.

Related Topics

- Logical Standby Databases in *Oracle Data Guard Concepts and Administration*

2.4.23 FLASHBACK_TIME

The Oracle Data Pump Export command-line utility `FLASHBACK_TIME` parameter finds the SCN that most closely matches the specified time.

Default

There is no default.

Purpose

Finds the system change number (SCN) that most closely matches the specified time. This SCN is used to enable the Flashback utility. The export operation is performed with data that is consistent up to this SCN.

Syntax and Description

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

Because the `TO_TIMESTAMP` value is enclosed in quotation marks, it is best to put this parameter in a parameter file.

Alternatively, you can enter the following parameter setting. This setting initiate a consistent export that is based on current system time:

```
FLASHBACK_TIME=systimestamp
```

Restrictions

- `FLASHBACK_TIME` and `FLASHBACK_SCN` are mutually exclusive.
- The `FLASHBACK_TIME` parameter pertains only to the flashback query capability of Oracle Database. It is not applicable to Flashback Database, Flashback Drop, or Flashback Data Archive.

Example

You can specify the time in any format that the `DBMS_FLASHBACK.ENABLE_AT_TIME` procedure accepts. For example, suppose you have a parameter file, `flashback.par`, with the following contents:

```
DIRECTORY=dpump_dir1
DUMPFILE=hr_time.dmp
FLASHBACK_TIME="TO_TIMESTAMP('27-10-2012 13:16:00', 'DD-MM-YYYY HH24:MI:SS')"
```

You can then issue the following command:

```
> expdp hr PARFILE=flashback.par
```

The export operation is performed with data that is consistent with the SCN that most closely matches the specified time.

> **Note:**
>
> If you are on a logical standby system and using a network link to access the logical standby primary, then the `FLASHBACK_SCN` parameter is ignored, because the logical standby selects the SCNs. See *Oracle Data Guard Concepts and Administration* for information about logical standby databases.
>
> See *Oracle Database Development Guide* for information about using Flashback Query.

Related Topics

- Logical Standby Databases in *Oracle Data Guard Concepts and Administration*
- Using Oracle Flashback Query (SELECT AS OF) in *Oracle Database Development Guide*

2.4.24 FULL

The Export command-line FULL parameter specifies that you want to perform a full database mode export

Default: `NO`

Purpose

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

Syntax and Description

```
FULL=[YES | NO]
```

`FULL=YES` indicates that all data and metadata are to be exported. To perform a full export, you must have the `DATAPUMP_EXP_FULL_DATABASE` role.

Filtering can restrict what is exported using this export mode.

You can perform a full mode export using the transportable option (`TRANSPORTABLE=ALWAYS`). This is referred to as a full transportable export, which exports all objects and data necessary to create a complete copy of the database. See Full Mode.

> **Note:**
>
> Be aware that when you later import a dump file that was created by a full-mode export, the import operation attempts to copy the password for the `SYS` account from the source database. This sometimes fails (for example, if the password is in a shared password file). If it does fail, then after the import completes, you must set the password for the `SYS` account at the target database to a password of your choice.

Restrictions

- To use the `FULL` parameter in conjunction with `TRANSPORTABLE` (a full transportable export), either the Data Pump `VERSION` parameter must be set to at least 12.0. or the `COMPATIBLE` database initialization parameter must be set to at least 12.0 or later.

- A full export does not, by default, export system schemas that contain Oracle-managed data and metadata. Examples of system schemas that are not exported by default include `SYS`, `ORDSYS`, and `MDSYS`.
- Grants on objects owned by the `SYS` schema are never exported.
- A full export operation exports objects from only one database edition; by default it exports the current edition but you can use the Export `SOURCE_EDITION` parameter to specify a different edition.
- If you are exporting data that is protected by a realm, then you must have authorization for that realm.
- The Automatic Workload Repository (AWR) is not moved in a full database export and import operation. (See *Oracle Database Performance Tuning Guide* for information about using Data Pump to move AWR snapshots.)
- The XDB repository is not moved in a full database export and import operation. User created XML schemas are moved.

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 DIRECTORY=dpump_dir2 DUMPFILE=expfull.dmp FULL=YES NOLOGFILE=YES
```

To see a detailed example of how to perform a full transportable export, see *Oracle Database Administrator's Guide*. For information about configuring realms, see *Oracle Database Vault Administrator's Guide*.

Related Topics

- *Oracle Database Performance Tuning Guide*
- *Oracle Database Administrator's Guide*
- *Oracle Database Vault Administrator's Guide*

2.4.25 HELP

The Data Pump Export command-line utility `HELP` parameter displays online help for the Export utility.

Default: `NO`

Purpose

Displays online help for the Export utility.

Syntax and Description

```
HELP = [YES | NO]
```

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

Example

```
> expdp HELP = YES
```

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

2.4.26 INCLUDE

The Data Pump Export command-line utility INCLUDE parameter 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.

Default: There is no default

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] [, ...]
```

The *object_type* specifies the type of object to be included. To see a list of valid values for *object_type*, query the following views: DATABASE_EXPORT_OBJECTS for full mode, SCHEMA_EXPORT_OBJECTS for schema mode, and TABLE_EXPORT_OBJECTS for table and tablespace mode. The values listed in the OBJECT_PATH column are the valid object types.

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

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). It 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.

The name that you supply for the *name_clause* must exactly match an existing object in the database, including upper- and lower- case letters. For example, if the *name_clause* you supply is for a table named EMPLOYEES, then there must be an existing table named EMPLOYEES using all upper-case letters. If the *name_clause* is provided as Employees or employees or any other variation, then the table is not found.

Depending on your operating system, the use of quotation marks when you specify a value for this parameter can also require that you use escape characters. Oracle recommends that you place this parameter in a parameter file, which can reduce the number of escape characters that you otherwise need to enter 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 can then use the `hr.par` file to start an export operation, without having to enter any other parameters on the command line. The `EMPLOYEES` and `DEPARTMENTS` tables, all procedures, and all index names with an EMP prefix, are included in the export.

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

Including Constraints

If the *`object_type`* that you specify is a `CONSTRAINT`, then be aware of the effects of using a constraint..

You cannot include explicitly the following constraints:

- `NOT NULL` constraints
- Constraints that are required for the table to be created and loaded successfully. For example: you cannot include primary key constraints for index-organized tables, or `REF SCOPE` and `WITH ROWID` constraints for tables with `REF` columns.

For example, the following `INCLUDE` statements are interpreted as follows:

- `INCLUDE=CONSTRAINT` includes all (nonreferential) constraints, except for `NOT NULL` constraints, and any constraints needed for successful table creation and loading.
- `INCLUDE=REF_CONSTRAINT` includeS referential integrity (foreign key) constraints.

Restrictions

- The `INCLUDE` and `EXCLUDE` parameters are mutually exclusive.
- Grants on objects owned by the `SYS` schema are never exported.

Example

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

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

Related Topics

- Metadata Filters
- Parameters Available in Data Pump Export Command-Line Mode

2.4.27 JOB_NAME

The Oracle Data Pump Export command-line utility `JOB_NAME` parameter identifies the export job in subsequent actions.

Default

A system-generated name of the form `SYS_EXPORT_`*`EXPORT or SQLFILE_mode_NN`*

Purpose

Use the JOB_NAME parameter when you want to identify the export job in subsequent actions. For example, when you want to use the ATTACH parameter to attach to a job, you use the JOB_NAME parameter to identify the name of the job that you want to attach. You can also use JOB_NAME to identify the job by using the views DBA_DATAPUMP_JOBS or USER_DATAPUMP_JOBS.

Syntax and Description

```
JOB_NAME=jobname_string
```

The *jobname_string* specifies a name of up to 128 bytes for the export job. The bytes must represent printable characters and spaces. If the name includes spaces or other non-alphanumeric characters (for example, hyphens), then the name must be enclosed in single quotation marks. Examples: 'Thursday Export', 'Thursday-Export'. For additional information about job name restrictions, see "Database Object Names and Qualifiers" item 7 in *Oracle Database SQL Language Reference*. The job name is implicitly qualified by the schema of the user performing the export operation. The job name is used as the name of the Data Pump control import job table, which controls the export job.

The default job name is system-generated in the form SYS_EXPORT_*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 DIRECTORY=dpump_dir1 DUMPFILE=exp_job.dmp JOB_NAME=exp_job
NOLOGFILE=YES
```

Related Topics

- Database Object Names and Qualifiers in *Oracle Database SQL Language Reference*

2.4.28 KEEP_MASTER

The Oracle Data Pump Export command-line utility KEEP_MASTER parameter indicates whether the Data Pump control job table should be deleted or retained at the end of an Oracle Data Pump job that completes successfully.

Default

NO

Purpose

Indicates whether the Data Pump control job table should be deleted or retained at the end of an Oracle Data Pump job that completes successfully. The Data Pump control job table is automatically retained for jobs that do not complete successfully.

Syntax and Description

```
KEEP_MASTER=[YES | NO]
```

Restrictions

- None

Example

```
> expdp hr DIRECTORY=dpump_dir1 DUMPFILE=expdat.dmp SCHEMAS=hr KEEP_MASTER=YES
```

2.4.29 LOGFILE

The Data Pump Export command-line utility LOGFILE parameter specifies the name, and optionally, a directory, for the log file of the export job.

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, assuming that you have access to it. This setting 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 file name is overwritten.

Restrictions

- To perform a Data Pump Export using Oracle Automatic Storage Management (Oracle ASM), you must specify a `LOGFILE` parameter that includes a directory object that does not include the Oracle ASM + notation. That is, the log file must be written to a disk file, and not written into the Oracle ASM storage. Alternatively, you can specify `NOLOGFILE=YES`. However, if you specify `NOLOGFILE=YES`, then that setting prevents the writing of the log file.

Example

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

```
> expdp 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 can be different in the log file than they are when displayed on the client output screen.

Related Topics

- STATUS
- Using Directory Objects When Oracle Automatic Storage Management Is Enabled

2.4.30 LOGTIME

The Oracle Data Pump Export command-line utility LOGTIME parameter specifies that messages displayed during export operations are timestamped.

Default

No timestamps are recorded

Purpose

Specifies that messages displayed during export operations are timestamped. You can use the timestamps to figure out the elapsed time between different phases of a Data Pump operation. Such information can be helpful in diagnosing performance problems and estimating the timing of future similar operations.

Syntax and Description

```
LOGTIME=[NONE | STATUS | LOGFILE | ALL]
```

The available options are defined as follows:

- NONE: No timestamps on status or log file messages (same as default)
- STATUS: Timestamps on status messages only
- LOGFILE: Timestamps on log file messages only
- ALL: Timestamps on both status and log file messages

Restrictions

If the file specified by LOGFILE exists and it is not identified as a Data Pump LOGFILE, such as using more than one dot in the filename (specifically, a compound suffix), then it cannot be overwritten. You must specify a different filename.

Example

The following example records timestamps for all status and log file messages that are displayed during the export operation:

```
> expdp hr DIRECTORY=dpump_dir1 DUMPFILE=expdat.dmp SCHEMAS=hr LOGTIME=ALL
```

The output looks similar to the following:

```
10-JUL-12 10:12:22.300: Starting "HR"."SYS_EXPORT_SCHEMA_01":  hr/********
directory=dpump_dir1 dumpfile=expdat.dmp schemas=hr logtime=all
10-JUL-12 10:12:22.915: Estimate in progress using BLOCKS method...
10-JUL-12 10:12:24.422: Processing object type SCHEMA_EXPORT/TABLE/TABLE_DATA
10-JUL-12 10:12:24.498: Total estimation using BLOCKS method: 128 KB
10-JUL-12 10:12:24.822: Processing object type SCHEMA_EXPORT/USER
10-JUL-12 10:12:24.902: Processing object type SCHEMA_EXPORT/SYSTEM_GRANT
10-JUL-12 10:12:24.926: Processing object type SCHEMA_EXPORT/ROLE_GRANT
10-JUL-12 10:12:24.948: Processing object type SCHEMA_EXPORT/DEFAULT_ROLE
10-JUL-12 10:12:24.967: Processing object type SCHEMA_EXPORT/TABLESPACE_QUOTA
10-JUL-12 10:12:25.747: Processing object type SCHEMA_EXPORT/PRE_SCHEMA/
PROCACT_SCHEMA
10-JUL-12 10:12:32.762: Processing object type SCHEMA_EXPORT/SEQUENCE/SEQUENCE
10-JUL-12 10:12:46.631: Processing object type SCHEMA_EXPORT/TABLE/TABLE
10-JUL-12 10:12:58.007: Processing object type SCHEMA_EXPORT/TABLE/GRANT/
OWNER_GRANT/OBJECT_GRANT
10-JUL-12 10:12:58.106: Processing object type SCHEMA_EXPORT/TABLE/COMMENT
10-JUL-12 10:12:58.516: Processing object type SCHEMA_EXPORT/PROCEDURE/
PROCEDURE
10-JUL-12 10:12:58.630: Processing object type SCHEMA_EXPORT/PROCEDURE/
ALTER_PROCEDURE
10-JUL-12 10:12:59.365: Processing object type SCHEMA_EXPORT/TABLE/INDEX/INDEX
10-JUL-12 10:13:01.066: Processing object type SCHEMA_EXPORT/TABLE/CONSTRAINT/
CONSTRAINT
10-JUL-12 10:13:01.143: Processing object type SCHEMA_EXPORT/TABLE/INDEX/
STATISTICS/INDEX_STATISTICS
10-JUL-12 10:13:02.503: Processing object type SCHEMA_EXPORT/VIEW/VIEW
10-JUL-12 10:13:03.288: Processing object type SCHEMA_EXPORT/TABLE/CONSTRAINT/
REF_CONSTRAINT
10-JUL-12 10:13:04.067: Processing object type SCHEMA_EXPORT/TABLE/TRIGGER
10-JUL-12 10:13:05.251: Processing object type SCHEMA_EXPORT/TABLE/STATISTICS/
TABLE_STATISTICS
10-JUL-12 10:13:06.172: . . exported
"HR"."EMPLOYEES"                            17.05 KB     107 rows
10-JUL-12 10:13:06.658: . . exported
"HR"."COUNTRIES"                            6.429 KB      25 rows
10-JUL-12 10:13:06.691: . . exported
"HR"."DEPARTMENTS"                          7.093 KB      27 rows
10-JUL-12 10:13:06.723: . . exported
"HR"."JOBS"                                 7.078 KB      19 rows
10-JUL-12 10:13:06.758: . . exported
"HR"."JOB_HISTORY"                          7.164 KB      10 rows
10-JUL-12 10:13:06.794: . . exported
"HR"."LOCATIONS"                            8.398 KB      23 rows
10-JUL-12 10:13:06.824: . . exported
"HR"."REGIONS"                              5.515 KB       4 rows
10-JUL-12 10:13:07.500: Master table "HR"."SYS_EXPORT_SCHEMA_01" successfully
loaded/unloaded
10-JUL-12 10:13:07.503:
******************************************************************************
```

2.4.31 METRICS

The Oracle Data Pump Export command-line utility `METRICS` parameter indicates whether you want additional information about the job reported to the Data Pump log file.

Default

`NO`

Purpose

Indicates whether additional information about the job should be reported to the Data Pump log file.

Syntax and Description

```
METRICS=[YES | NO]
```

When `METRICS=YES` is used, the number of objects and the elapsed time are recorded in the Data Pump log file.

Restrictions

None

Example

```
> expdp hr DIRECTORY=dpump_dir1 DUMPFILE=expdat.dmp SCHEMAS=hr METRICS=YES
```

2.4.32 NETWORK_LINK

The Data Pump Export command-line utility `NETWORK_LINK` parameter enables an export from a (source) database identified by a valid database link. The data from the source database instance is written to a dump file set on the connected database instance.

Default: There is no default

Purpose

Enables an export from a (source) database identified by a valid database link. The data from the source database instance is written to a dump file set on the connected database instance.

Syntax and Description

```
NETWORK_LINK=source_database_link
```

The `NETWORK_LINK` parameter initiates an export using a database link. This export setting means that the system to which the `expdp` client is connected contacts the source database referenced by the *`source_database_link`*, retrieves data from it, and writes the data to a dump file set back on the connected system.

The *`source_database_link`* provided must be the name of a database link to an available database. If the database on that instance does not already have a database link, then you or your DBA must create one using the SQL `CREATE DATABASE LINK` statement.

If the source database is read-only, then the user on the source database must have a locally managed temporary tablespace assigned as the default temporary tablespace. Otherwise, the job will fail.

The following types of database links are supported for use with Data Pump Export:

- Public fixed user
- Public connected user
- Public shared user (only when used by link owner)
- Private shared user (only when used by link owner)
- Private fixed user (only when used by link owner)

> **Caution:**
>
> If an export operation is performed over an unencrypted network link, then all data is exported as clear text, even if it is encrypted in the database. See *Oracle Database Security Guide* for more information about network security.

Restrictions

- The following types of database links are not supported for use with Data Pump Export:
  - Private connected user
  - Current user
- When operating across a network link, Data Pump requires that the source and target databases differ by no more than two versions. For example, if one database is Oracle Database 12c, then the other database must be 12c, 11g, or 10g. Note that Data Pump checks only the major version number (for example, 10g,11g, 12c), not specific release numbers (for example, 12.1, 12.2, 11.1, 11.2, 10.1 or 10.2).
- When transporting a database over the network using full transportable export, auditing cannot be enabled for tables stored in an administrative tablespace (such as `SYSTEM` and `SYSAUX`) if the audit trail information itself is stored in a user-defined tablespace.
- Metadata cannot be imported in parallel when the `NETWORK_LINK` parameter is also used

Example

The following is a syntax example of using the `NETWORK_LINK` parameter. Replace the variable *`source_database_link`* with the name of a valid database link that must already exist.

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

> **See Also:**
>
> - *Oracle Database Administrator's Guide* for more information about database links
> - *Oracle Database SQL Language Reference* for more information about the `CREATE DATABASE LINK` statement
> - *Oracle Database Administrator's Guide* for more information about locally managed tablespaces

2.4.33 NOLOGFILE

The Data Pump Export command-line utility NOLOGFILE parameter specifies whether to suppress creation of a log file.

Default: `NO`

Purpose

Specifies whether to suppress creation of a log file.

Syntax and Description

```
NOLOGFILE=[YES | NO]
```

Specify `NOLOGFILE=YES` 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=YES`, then you run the risk of losing important progress and error information.

Example

The following is an example of using the `NOLOGFILE` parameter:

```
> expdp hr DIRECTORY=dpump_dir1 DUMPFILE=hr.dmp NOLOGFILE=YES
```

This command results in a schema-mode export (the default), in which no log file is written.

2.4.34 PARALLEL

The Oracle Data Pump Export command-line utility `PARALLEL` parameter specifies the maximum number of processes of active execution operating on behalf of the export job.

Default

```
1
```

Purpose

Specifies the maximum number of processes of active execution operating on behalf of the export job. This execution set consists of a combination of worker processes and parallel input/output (I/O) server processes. The Data Pump 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 that you specify for *integer* should be less than, or equal to, the number of files in the dump file set (or you should specify either the %U or %L substitution variables in the dump file specifications). Because each active worker processor I/O server process writes exclusively to one file at a time, an insufficient number of files can have adverse effects. For example, some of the worker processes can 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 cannot obtain a file for output, then the export operation is 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 decreases the number of worker processes that are running at any given time. Also, any ongoing work must reach an orderly completion point before the decrease takes effect. Therefore, it can take a while to see any effect from decreasing the value. Idle worker processes are not deleted until the job exits.

If there is work that can be performed in parallel, then increasing the parallelism takes effect immediately .

Using PARALLEL During An Export In An Oracle RAC Environment

In an Oracle Real Application Clusters (Oracle RAC) environment, if an export operation has PARALLEL=1, then all Oracle Data Pump processes reside on the instance where the job is started. Therefore, the directory object can point to local storage for that instance.

If the export operation has PARALLEL set to a value greater than 1, then Oracle Data Pump processes can reside on instances other than the one where the job was started. Therefore, the directory object must point to shared storage that is accessible by all Oracle RAC cluster members.

Restrictions

- This parameter is valid only in the Enterprise Edition of Oracle Database 11g or later.
- To export a table or table partition in parallel (using parallel query, or PQ, worker processes), you must have the DATAPUMP_EXP_FULL_DATABASE role.
- Transportable tablespace metadata cannot be exported in parallel.
- Metadata cannot be exported in parallel when the NETWORK_LINK parameter is also used.
- The following objects cannot be exported in parallel:
  - TRIGGER
  - VIEW
  - OBJECT_GRANT
  - SEQUENCE
  - CONSTRAINT

- REF_CONSTRAINT

Example

The following is an example of using the PARALLEL parameter:

```
> expdp hr DIRECTORY=dpump_dir1 LOGFILE=parallel_export.log
JOB_NAME=par4_job DUMPFILE=par_exp%u.dmp PARALLEL=4
```

This results in a schema-mode export (the default) of the hr schema, in which up to four files can be created in the path pointed to by the directory object, dpump_dir1.

Related Topics

- DUMPFILE
- Commands Available in Data Pump Export Interactive-Command Mode
- Performing a Parallel Full Database Export

2.4.35 PARALLEL_THRESHOLD

The Oracle Data Pump Export command-line utility PARALLEL_THRESHOLD parameter specifies the size of the divisor that Data Pump uses to calculate potential parallel DML based on table size

Default

250MB

Purpose

PARALLEL_THRESHOLD should only be used with export or import jobs of a single unpartitioned table, or one partition of a partitioned table. When you specify PARALLEL in the job, you can specify PARALLEL_THRESHOLD to modify the size of the divisor that Oracle Data Pump uses to determine if a table should be exported or imported using parallel data manipulation statements (PDML) during imports and exports. If you specify a lower value than the default, then it enables a smaller table size to use the Oracle Data Pump parallel algorithm. For example, if you have a 100MB table and you want it to use PDML of 5, to break it into five units, then you specify PARALLEL_THRESHOLD=20M. Note that the database, the optimizer, and the execution plan produced by the optimizer for the SQL determine the actual degree of parallelism used to load or unload the object specified in the job.

Syntax and Description

The parameter value specifies the threshold size in bytes:

```
PARALLEL_THRESHOLD=size-in-bytes
```

For a single table export or import, if you want a higher degree of parallelism, then you may want to set PARALLEL_THRESHOLD to lower values, to take advantage of parallelism for a smaller table or table partition. However, the benefit of this resource allocation can be limited by the performance of the I/O of the file systems to which you are loading or unloading. Also, if the job involves more than one object, for both tables and metadata objects, then the PQ allocation request specified by PARALLEL with PARALLEL_THRESHOLD is of limited value. The actual amount of PQ processes allocated to a table is impacted by how many operations Oracle Data Pump is

running concurrently, where the amount of parallelism has to be shared. The database, the optimizer, and the execution plan produced by the optimizer for the SQL determine the actual degree of parallelism used to load or unload the object specified in the job.

You can use this parameter to assist with particular data movement issues. For example:

- When you want to use Oracle Data Pump to load a large table from one database into a larger table in another database. One possible use case: Uploading weekly sales data from an OLTP database into a reporting or business analytics data warehouse database.
- When you want to export a single large table, but you have not gathered RDBMS stats recently. The default size is determined from the table's statistics. However, suppose that the statistics are old (or have never been run). In that case, the value used by Oracle Data Pump could underrepresent the table's actual size. To compensate for a case such as this, you can specify a smaller `parallel_threshold` value, so that the algorithm for the degree of parallelism (table size divided by threshold amount) can yield a more reasonable degree of parallelism value.

Restrictions

`PARALLEL_THRESHOLD` is used only in conjunction when the `PARALLEL` parameter is specified with a value greater than 1.

Example

The following is an example of using the `PARALLEL_THRESHOLD` parameter to export the table `table_to_use_PDML`, where the size of the divisor for PQ processes is set to 1 KB, the variables *`user`* and *`user-password`* are the user and password of the user running Export (`expdp`), and the job name is `parathresh_example`.

```
expdp user/user-password \
    directory=dpump_dir \
    dumpfile=parathresh_example.dmp
    tables=table_to_use_PDML \
    parallel=8 \
    parallel_threshold=1K \
    job_name=parathresh_example
```

2.4.36 PARFILE

The Oracle Data Pump Export command-line utility `PARFILE` parameter specifies the name of an export parameter file.

Default

There is no default

Purpose

Specifies the name of an export parameter file, also known as a **parfile**.

Syntax and Description

```
PARFILE=[directory_path]file_name
```

A parameter file enables you to specify Oracle Data Pump parameters within a file. You can then specify that file on the command line, instead of entering all of the individual commands.

Using a parameter file can be useful if you use the same parameter combination many times. The use of parameter files is also highly recommended when you use parameters whose values require the use of quotation marks.

A directory object is not specified for the parameter file. You do not specify a directory object, because the parameter file is opened and read by the `expdp` client, unlike dump files, log files, and SQL files which are created and written by the server. The default location of the parameter file is the user's current directory.

Within a parameter file, a comma is implicit at every newline character so you do not have to enter commas at the end of each line. If you have a long line that wraps, such as a long table name, then enter the backslash continuation character (`\`) at the end of the current line to continue onto the next line.

The contents of the parameter file are written to the Data Pump log file.

Restrictions

The `PARFILE` parameter cannot be specified within a parameter file.

Example

Suppose the content of an example parameter file, `hr.par`, is as follows:

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

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

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

Related Topics

- About Data Pump Export Parameters
  Learn how to use Oracle Data Pump Export parameters in command-line mode, including case sensitivity, quotation marks, escape characters, and information about how to use examples.

2.4.37 QUERY

The Oracle Data Pump Export command-line utility `QUERY` parameter enables you to specify a query clause that is used to filter the data that gets exported.

Default

There is no default.

Purpose

Enables you to specify a query clause that is used to filter the data that gets exported.

Syntax and Description

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

The *query_clause* is typically a SQL WHERE clause for fine-grained row selection, but could be any SQL clause. For example, you can use an ORDER BY clause to speed up a migration from a heap-organized table to an index-organized table. If a schema and table name are not supplied, then 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 query can be specified per table.

If the NETWORK_LINK parameter is specified along with the QUERY parameter, then any objects specified in the *query_clause* that are on the remote (source) node must be explicitly qualified with the NETWORK_LINK value. Otherwise, Data Pump assumes that the object is on the local (target) node; if it is not, then an error is returned and the import of the table from the remote (source) system fails.

For example, if you specify NETWORK_LINK=dblink1, then the *query_clause* of the QUERY parameter must specify that link, as shown in the following example:

```
QUERY=(hr.employees:"WHERE last_name IN(SELECT last_name
FROM hr.employees@dblink1)")
```

Depending on your operating system, when you specify a value for this parameter that the uses quotation marks, it can also require that you use escape characters. Oracle recommends that you place this parameter in a parameter file, which can reduce the number of escape characters that might otherwise be needed on the command line. .

To specify a schema other than your own in a table-specific query, you must be granted access to that specific table.

Restrictions

- The QUERY parameter cannot be used with the following parameters:
  - CONTENT=METADATA_ONLY
  - ESTIMATE_ONLY
  - TRANSPORT_TABLESPACES
- When the QUERY parameter is specified for a table, Data Pump uses external tables to unload the target table. External tables uses a SQL CREATE TABLE AS SELECT statement. The value of the QUERY parameter is the WHERE clause in the SELECT portion of the CREATE TABLE statement. If the QUERY parameter includes references to another table with columns whose names match the table being unloaded, and if those columns are used in the query, then you will need to use a table alias to distinguish between columns in the table being unloaded and columns in the SELECT statement with the same name. The table alias used by Data Pump for the table being unloaded is KU$.

  For example, suppose you want to export a subset of the sh.sales table based on the credit limit for a customer in the sh.customers table. In the following example, KU$ is used to qualify the cust_id field in the QUERY parameter for unloading sh.sales. As a result, Data Pump exports only rows for customers whose credit limit is greater than $10,000.

  ```
  QUERY='sales:"WHERE EXISTS (SELECT cust_id FROM customers c
     WHERE cust_credit_limit > 10000 AND ku$.cust_id = c.cust_id)"'
  ```

In the following query, KU$ is not used for a table alias. The result is that all rows are unloaded:

```
QUERY='sales:"WHERE EXISTS (SELECT cust_id FROM customers c
   WHERE cust_credit_limit > 10000 AND cust_id = c.cust_id)"'
```

- The maximum length allowed for a QUERY string is 4000 bytes, which includes quotation marks. This restriction means that the actual maximum length allowed is 3998 bytes.

Example

The following is an example of using the QUERY parameter:

```
> expdp hr PARFILE=emp_query.par
```

The contents of the emp_query.par file are as follows:

```
QUERY=employees:"WHERE department_id > 10 AND salary > 10000"
NOLOGFILE=YES
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 are unloaded. For the employees table, only rows that meet the query criteria are unloaded.

Related Topics

- About Data Pump Export Parameters

2.4.38 REMAP_DATA

The Oracle Data Pump Export command-line utility REMAP_DATA parameter enables you to specify a remap function that takes as a source the original value of the designated column and returns a remapped value that replaces the original value in the dump file.

Default

There is no default

Purpose

The REMAP_DATA parameter enables you to specify a remap function that takes as a source the original value of the designated column, and returns a remapped value that will replace the original value in the dump file. A common use for this option is to mask data when moving from a production system to a test system. For example, a column of sensitive customer data, such as credit card numbers, could be replaced with numbers generated by a REMAP_DATA function. Replacing the sensitive data with numbers enables the data to retain its essential formatting and processing characteristics, without exposing private data to unauthorized personnel.

The same function can be applied to multiple columns being dumped. This function is useful when you want to guarantee consistency in remapping both the child and parent column in a referential constraint.

Syntax and Description

```
REMAP_DATA=[schema.]tablename.column_name:[schema.]pkg.function
```

The description of each syntax element, in the order in which they appear in the syntax, is as follows:

schema: the schema containing the table that you want to be remapped. By default, this is the schema of the user doing the export.

tablename: the table whose column you want to be remapped.

column_name: the column whose data you want to be remapped.

schema : the schema containing the PL/SQL package that you have created that contains the remapping function. As a default, this is the schema of the user doing the export.

pkg: the name of the PL/SQL package you have created that contains the remapping function.

function: the name of the function within the PL/SQL that will be called to remap the column table in each row of the specified table.

Restrictions

- The data types and sizes of the source argument and the returned value must both match the data type and size of the designated column in the table.
- Remapping functions should not perform commits or rollbacks except in autonomous transactions.
- The use of synonyms as values for the `REMAP_DATA` parameter is not supported. For example, if the `regions` table in the `hr` schema had a synonym of `regn`, an error would be returned if you specified `regn` as part of the `REMAP_DATA` specification.
- Remapping LOB column data of a remote table is not supported.
- Columns of the following types are not supported by`REMAP_DATA`: User Defined Types, attributes of User Defined Types, `LONG`s, `REF`s, `VARRAY`s, Nested Tables, `BFILE`s, and `XMLtype`.

Example

The following example assumes a package named `remap` has been created that contains functions named `minus10` and `plusx`. These functions change the values for `employee_id` and `first_name` in the `employees` table.

```
> expdp hr DIRECTORY=dpump_dir1 DUMPFILE=remap1.dmp TABLES=employees
REMAP_DATA=hr.employees.employee_id:hr.remap.minus10
REMAP_DATA=hr.employees.first_name:hr.remap.plusx
```

2.4.39 REUSE_DUMPFILES

The Oracle Data Pump Export command-line utility `REUSE_DUMPFILES` parameter specifies whether to overwrite a preexisting dump file.

Default

`NO`

Purpose

Specifies whether to overwrite a preexisting dump file.

Syntax and Description

```
REUSE_DUMPFILES=[YES | NO]
```

Normally, Data Pump Export will return an error if you specify a dump file name that already exists. The `REUSE_DUMPFILES` parameter allows you to override that behavior and reuse a dump file name. For example, if you performed an export and specified `DUMPFILE=hr.dmp` and `REUSE_DUMPFILES=YES`, then `hr.dmp` is overwritten if it already exists. Its previous contents are then lost, and it instead contains data for the current export.

Example

The following export operation creates a dump file named `enc1.dmp`, even if a dump file with that name already exists.

```
> expdp hr DIRECTORY=dpump_dir1 DUMPFILE=enc1.dmp
TABLES=employees REUSE_DUMPFILES=YES
```

2.4.40 SAMPLE

The Oracle Data Pump Export command-line utility `SAMPLE` parameter specifies a percentage of the data rows that you want to be sampled and unloaded from the source database.

Default

There is no default.

Purpose

Specifies a percentage of the data rows that you want to be sampled and unloaded from the source database.

Syntax and Description

```
SAMPLE=[[schema_name.]table_name:]sample_percent
```

This parameter allows you to export subsets of data by specifying the percentage of data to be sampled and exported. The *`sample_percent`* indicates the probability that a row will be selected as part of the sample. It does not mean that the database will retrieve exactly that amount of rows from the table. The value you supply for *`sample_percent`* can be anywhere from .000001 up to, but not including, 100.

You can apply the *`sample_percent`* to specific tables. In the following example, 50% of the `HR.EMPLOYEES` table is exported:

```
SAMPLE="HR"."EMPLOYEES":50
```

If you specify a schema, then you must also specify a table. However, you can specify a table without specifying a schema. In that scenario, the current user is assumed. If no table is specified, then the *`sample_percent`* value applies to the entire export job.

You can use this parameter with the Data Pump Import `PCTSPACE` transform, so that the size of storage allocations matches the sampled data subset. (See the Import `TRANSFORM` parameter).

Restrictions

- The `SAMPLE` parameter is not valid for network exports.

Example

In the following example, the value `70` for `SAMPLE` is applied to the entire export job because no table name is specified.

```
> expdp hr DIRECTORY=dpump_dir1 DUMPFILE=sample.dmp SAMPLE=70
```

Related Topics

- TRANSFORM

2.4.41 SCHEMAS

The Oracle Data Pump Export command-line utility `SCHEMAS` parameter specifies that you want to perform a schema-mode export.

Default

The 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 `DATAPUMP_EXP_FULL_DATABASE` role, then you can specify a single schema other than your own or a list of schema names. The `DATAPUMP_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. Filtering can further restrict what is exported using schema mode.

Restrictions

- If you do not have the `DATAPUMP_EXP_FULL_DATABASE` role, then you can specify only your own schema.
- The `SYS` schema cannot be used as a source schema for export jobs.

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 `DATAPUMP_EXP_FULL_DATABASE` role was previously assigned to it for the purpose of these examples.

```
> expdp 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.

Related Topics

- Filtering During Export Operations

2.4.42 SERVICE_NAME

The Oracle Data Pump Export command-line utility SERVICE_NAME parameter specifies a service name that you want to use in conjunction with the CLUSTER parameter.

Default

There is no default.

Purpose

Specifies a service name that you want to use in conjunction with the CLUSTER parameter.

Syntax and Description

```
SERVICE_NAME=name
```

You can use the SERVICE_NAME parameter with the CLUSTER=YES parameter to specify an existing service associated with a resource group that defines a set of Oracle Real Application Clusters (Oracle RAC) instances belonging to that resource group. Typically, the resource group is a subset of all the Oracle RAC instances.

The service name is only used to determine the resource group, and the instances defined for that resource group. The instance where the job is started is always used, regardless of whether it is part of the resource group.

If CLUSTER=NO is also specified, then the SERVICE_NAME parameter is ignored

Suppose you have an Oracle RAC configuration containing instances A, B, C, and D. Also suppose that a service named my_service exists with a resource group consisting of instances A, B, and C only. In such a scenario, the following is true:

- If you start an Oracle Data Pump job on instance A, and specify CLUSTER=YES (or accept the default, which is Y), and you do not specify the SERVICE_NAME parameter, then Oracle Data Pump creates workers on all instances: A, B, C, and D, depending on the degree of parallelism specified.
- If you start a Data Pump job on instance A, and specify CLUSTER=YES, and SERVICE_NAME=my_service, then workers can be started on instances A, B, and C only.
- If you start a Data Pump job on instance D, and specify CLUSTER=YES, and SERVICE_NAME=my_service, then workers can be started on instances A, B, C, and D. Even though instance D is not in my_service it is included because it is the instance on which the job was started.
- If you start a Data Pump job on instance A, and specify CLUSTER=NO, then any SERVICE_NAME parameter that you specify is ignored. All processes start on instance A.

Example

The following is an example of using the SERVICE_NAME parameter:

```
> expdp hr DIRECTORY=dpump_dir1 DUMPFILE=hr_svname2.dmp SERVICE_NAME=sales
```

This example starts a schema-mode export (the default mode) of the hr schema. Even though CLUSTER=YES is not specified on the command line, it is the default behavior, so the job uses all

instances in the resource group associated with the service name `sales`. A dump file named `hr_svname2.dmp` is written to the location specified by the `dpump_dir1` directory object.

Related Topics

- CLUSTER

2.4.43 SOURCE_EDITION

The Oracle Data Pump Export command-line utility `SOURCE_EDITION` parameter specifies the database edition from which objects are exported.

Default: the default database edition on the system

Purpose

Specifies the database edition from which objects are exported.

Syntax and Description

`SOURCE_EDITION=`*`edition_name`*

If `SOURCE_EDITION=`*`edition_name`* is specified, then the objects from that edition are exported. Data Pump selects all inherited objects that have not changed, and all actual objects that have changed.

If this parameter is not specified, then the default edition is used. If the specified edition does not exist or is not usable, then an error message is returned.

Restrictions

- This parameter is only useful if there are two or more versions of the same versionable objects in the database.
- The job version must be `11.2` or later.

Example

The following is an example of using the `SOURCE_EDITION` parameter:

```
> expdp hr DIRECTORY=dpump_dir1 DUMPFILE=exp_dat.dmp SOURCE_EDITION=exp_edition
EXCLUDE=USER
```

This example assumes the existence of an edition named `exp_edition` on the system from which objects are being exported. Because no export mode is specified, the default of schema mode will be used. The `EXCLUDE=user` parameter excludes only the definitions of users, not the objects contained within users' schemas.

Related Topics

- VERSION
- CREATE EDITION in *Oracle Database SQL Language Reference*
- Editions in*Oracle Database Development Guide*

2.4.44 STATUS

The Oracle Data Pump Export command-line utility `STATUS` parameter specifies the frequency at which the job status display is updated.

Default

`0`

Purpose

Specifies the frequency at which the job status display is updated.

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, then no additional information is displayed beyond information about the 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 DIRECTORY=dpump_dir1 SCHEMAS=hr,sh STATUS=300
```

This example exports the `hr` and `sh` schemas, and displays the status of the export every 5 minutes (60 seconds x 5 = 300 seconds).

2.4.45 TABLES

The Oracle Data Pump Export command-line utility `TABLES` parameter specifies that you want to perform a table-mode export.

Default

There is no default.

Purpose

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

Syntax and Description

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

Filtering can restrict what is exported using this mode. 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, then 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.

If an entire partitioned table is exported, then it is imported in its entirety as a partitioned table. The only case in which this is not true is if `PARTITION_OPTIONS=DEPARTITION` is specified during import.

The table name that you specify can be preceded by a qualifying schema name. The schema defaults to that of the current user. To specify a schema other than your own, you must have the `DATAPUMP_EXP_FULL_DATABASE` role.

Use of the wildcard character (`%`) to specify table names and partition names is supported.

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, then 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 are preceded by an escape character. The following examples show of how case-sensitivity can be preserved in the different Export modes.

  – In command-line mode:

    ```
    TABLES='\"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 (`#`), then the Data Pump 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, then Data Pump 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, if the parameter file contains the following line, then the Data Pump Export utility exports all three tables, because `emp#` is enclosed in quotation marks:

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

> **Note:**
>
> Some operating systems use single quotation marks as escape characters, rather than double quotation marks, and others the reverse. Refer to your 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 be able to use such characters in the name and have them ignored by the shell, and used by Export.

Using the Transportable Option During Table-Mode Export

To use the transportable option during a table-mode export, specify the `TRANSPORTABLE=ALWAYS` parameter with the `TABLES` parameter. Metadata for the specified tables, partitions, or subpartitions is exported to the dump file. To move the actual data, you copy the data files to the target database.

If only a subset of a table's partitions are exported and the `TRANSPORTABLE=ALWAYS` parameter is used, then on import each partition becomes a non-partitioned table.

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 restriction means that if you subsequently import the dump file, and the type does not already exist in the destination database, then the table creation fails.
- 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`, then it is not valid to use `TABLES=regn`. If you attempt to use the synonym, then an error is returned.
- The export of tables that include a wildcard character (`%`) in the table name is not supported if the table has partitions.
- The length of the table name list specified for the `TABLES` parameter is limited to a maximum of 4 MB, unless you are using the `NETWORK_LINK` parameter to an Oracle Database release 10.2.0.3 or earlier, or to a read-only database. In such cases, the limit is 4 KB.
- You can only specify partitions from one table if `TRANSPORTABLE=ALWAYS` is also set on the export.

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 DIRECTORY=dpump_dir1 DUMPFILE=tables.dmp
TABLES=employees,jobs,departments
```

The following example assumes that user `hr` has the `DATAPUMP_EXP_FULL_DATABASE` role. It shows the use of the `TABLES` parameter to export partitions.

```
> expdp hr DIRECTORY=dpump_dir1 DUMPFILE=tables_part.dmp
TABLES=sh.sales:sales_Q1_2012,sh.sales:sales_Q2_2012
```

This example exports the partitions, `sales_Q1_2012` and `sales_Q2_2012`, from the table `sales` in the schema `sh`.

Related Topics

- Filtering During Export Operations
- TRANSPORTABLE
  The Oracle Data Pump Export command-line utility `TRANSPORTABLE` parameter specifies whether the transportable option should be used during a table mode or full mode export.
- REMAP_TABLE
  The Oracle Data Pump Import command-line mode `REMAP_TABLE` parameter enables you to rename tables during an import operation.
- Using Data File Copying to Move Data

2.4.46 TABLESPACES

The Oracle Data Pump Export command-line utility `TABLESPACES` parameter specifies a list of tablespace names that you want to be exported in tablespace mode.

Default

There is no default.

Purpose

Specifies a list of tablespace names that you want 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, then its dependent objects are also unloaded. Both object metadata and data are unloaded. If any part of a table resides in the specified set, then that table and all of its dependent objects are exported. Privileged users get all tables. Unprivileged users obtain only the tables in their own schemas

Filtering can restrict what is exported using this mode.

Restrictions

The length of the tablespace name list specified for the `TABLESPACES` parameter is limited to a maximum of 4 MB, unless you are using the `NETWORK_LINK` to an Oracle Database release 10.2.0.3 or earlier, or to a read-only database. In such cases, the limit is 4 KB.

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 DIRECTORY=dpump_dir1 DUMPFILE=tbs.dmp
TABLESPACES=tbs_4, tbs_5, tbs_6
```

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

Related Topics

- Filtering During Export Operations

2.4.47 TRANSPORT_FULL_CHECK

The Oracle Data Pump Export command-line utility TRANSPORT_FULL_CHECK parameter specifies whether to check for dependencies between objects

Default

NO

Purpose

Specifies whether 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=[YES | NO]
```

If TRANSPORT_FULL_CHECK=YES, then the Data Pump 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, then 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=NO 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, then the export operation is terminated.

There are other checks performed as well. For instance, Data Pump 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.

There are two current command line parameters that control full closure check:

```
TTS_FULL_CHECK=[YES|NO]
TRANSPORT_FULL_CHECK=[YES|NO]
```

`[TTS|TRANSPORT]_FULL_CHECK=YES` is interpreted as `TTS_CLOSURE_CHECK=FULL`.`[TTS|TRANSPORT]_FULL_CHECK=NO` is interpreted as `TTS_CLOSURE_CHECK=ON`.

Example

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

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

2.4.48 TRANSPORT_TABLESPACES

The Oracle Data Pump Export command-line utility `TRANSPORT_TABLESPACES` parameter specifies that you want to perform an export in transportable-tablespace mode.

Default

There is no default.

Purpose

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

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.

The log file for the export lists the data files that are used in the transportable set, the dump files, and any containment violations.

The `TRANSPORT_TABLESPACES` parameter exports metadata for all objects within the specified tablespaces. If you want to perform a transportable export of only certain tables, partitions, or subpartitions, then you must use the `TABLES` parameter with the `TRANSPORTABLE=ALWAYS` parameter.

> **Note:**
>
> You cannot export transportable tablespaces and then import them into a database at a lower release level. The target database must be at the same or later release level as the source database.

Restrictions

- Transportable tablespace jobs are no longer restricted to a degree of parallelism of 1.

- Transportable tablespace mode requires that you have the DATAPUMP_EXP_FULL_DATABASE role.
- The default tablespace of the user performing the export must not be set to one of the tablespaces being transported.
- The SYSTEM and SYSAUX tablespaces are not transportable in transportable tablespace mode.
- All tablespaces in the transportable set must be set to read-only.
- If the Data Pump Export VERSION parameter is specified along with the TRANSPORT_TABLESPACES parameter, then the version must be equal to or greater than the Oracle Database COMPATIBLE initialization parameter.
- The TRANSPORT_TABLESPACES parameter cannot be used in conjunction with the QUERY parameter.
- Transportable tablespace jobs do not support the ACCESS_METHOD parameter for Data Pump Export.

Example

The following is an example of using the TRANSPORT_TABLESPACES parameter in a file-based job (rather than network-based). The tablespace tbs_1 is the tablespace being moved. This example assumes that tablespace tbs_1 exists and that it has been set to read-only. This example also assumes that the default tablespace was changed before this export command was issued.

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

See *Oracle Database Administrator's Guide* for detailed information about transporting tablespaces between databases

Related Topics

- Transportable Tablespace Mode
- Using Data File Copying to Move Data
- How Does Oracle Data Pump Handle Timestamp Data?
- Transporting Tablespaces Between Databases in *Oracle Database Administrator's Guide*

2.4.49 TRANSPORTABLE

The Oracle Data Pump Export command-line utility TRANSPORTABLE parameter specifies whether the transportable option should be used during a table mode or full mode export.

Default

NEVER

Purpose

Specifies whether the transportable option should be used during a table mode export (specified with the TABLES parameter) or a full mode export (specified with the FULL parameter).

Syntax and Description

```
TRANSPORTABLE = [ALWAYS | NEVER]
```

The definitions of the allowed values are as follows:

`ALWAYS` - Instructs the export job to use the transportable option. If transportable is not possible, then the job fails.

In a table mode export, using the transportable option results in a transportable tablespace export in which metadata for only the specified tables, partitions, or subpartitions is exported.

In a full mode export, using the transportable option results in a full transportable export which exports all objects and data necessary to create a complete copy of the database.

`NEVER` - Instructs the export job to use either the direct path or external table method to unload data rather than the transportable option. This is the default.

> **Note:**
>
> To export an entire tablespace in transportable mode, use the `TRANSPORT_TABLESPACES` parameter.

- If only a subset of a table's partitions are exported and the `TRANSPORTABLE=ALWAYS` parameter is used, then on import each partition becomes a non-partitioned table.
- If only a subset of a table's partitions are exported and the `TRANSPORTABLE` parameter is *not* used at all or is set to `NEVER` (the default), then on import:
  - If `PARTITION_OPTIONS=DEPARTITION` is used, then each partition included in the dump file set is created as a non-partitioned table.
  - If `PARTITION_OPTIONS` is not used, then the complete table is created. That is, all the metadata for the complete table is present, so that the table definition looks the same on the target system as it did on the source. But only the data that was exported for the specified partitions is inserted into the table.

Restrictions

- The `TRANSPORTABLE` parameter is only valid in table mode exports and full mode exports.
- To use the `TRANSPORTABLE` parameter, the `COMPATIBLE` initialization parameter must be set to at least 11.0.0.
- To use the `FULL` parameter in conjunction with `TRANSPORTABLE` (to perform a full transportable export), the Data Pump `VERSION` parameter must be set to at least 12.0. If the `VERSION` parameter is not specified, then the `COMPATIBLE` database initialization parameter must be set to at least 12.0 or later.
- The user performing a transportable export requires the `DATAPUMP_EXP_FULL_DATABASE` privilege.
- Tablespaces associated with tables, partitions, and subpartitions must be read-only.
- A full transportable export uses a mix of data movement methods. Objects residing in a transportable tablespace have only their metadata unloaded; data is copied when the data files are copied from the source system to the target system. The data files that must be copied are listed at the end of the log file for the export operation. Objects residing in non-

transportable tablespaces (for example, SYSTEM and SYSAUX) have both their metadata and data unloaded into the dump file set. (See *Oracle Database Administrator's Guide* for more information about performing full transportable exports.)

- The default tablespace of the user performing the export must not be set to one of the tablespaces being transported.

Example

The following example assumes that the sh user has the DATAPUMP_EXP_FULL_DATABASE role and that table sales2 is partitioned and contained within tablespace tbs2. (The tbs2 tablespace must be set to read-only in the source database.)

```
> expdp sh DIRECTORY=dpump_dir1 DUMPFILE=tto1.dmp
TABLES=sh.sales2 TRANSPORTABLE=ALWAYS
```

After the export completes successfully, you must copy the data files to the target database area. You could then perform an import operation using the PARTITION_OPTIONS and REMAP_SCHEMA parameters to make each of the partitions in sales2 its own table.

```
> impdp system PARTITION_OPTIONS=DEPARTITION
TRANSPORT_DATAFILES=oracle/dbs/tbs2 DIRECTORY=dpump_dir1
DUMPFILE=tto1.dmp REMAP_SCHEMA=sh:dp
```

Related Topics

- Transporting Databases in *Oracle Database Administrator's Guide*
- Full Mode
- Using Data File Copying to Move Data

2.4.50 TTS_CLOSURE_CHECK

The Oracle Data Pump Export command-line mode TTS_CLOSURE_CHECK parameter is used to indicate the degree of closure checking to be performed as part of a Data Pump transportable tablespace operation.

Default

There is no default.

Purpose

Specifies the level of closure check that you want to be performed as part of the transportable export operation. The TTS_CLOSURE_CHECK parameter can also be used to indicate that tablespaces can remain read-write during a test mode transportable tablespace operation. This option is used to obtain the timing requirements of the export operation. It is for testing purposes only. The dump file is unavailable for import.

Syntax and Description

```
TTS_CLOSURE_CHECK = [ ON | OFF | FULL | TEST_MODE ]
```

The TTS_CLOSURE_CHECK parameter supports the following options:

- ON - indicates self-containment closure check be performed
- OFF - indicates no closure check be performed
- FULL - indicates full bidirectional closure check be performed

- TEST_MODE - indicates that tablespaces are not required to be in read-only mode

ON,OFF, and FULL options are mutually exclusive. TEST_MODE is an Oracle Data Pump Export option only.

Example

```
TTS_CLOSURE_CHECK=FULL
```

2.4.51 VERSION

The Data Pump Export command-line utility VERSION parameter specifies the version of database objects that you want to export.

Default: COMPATIBLE

Purpose

Specifies the version of database objects that you want to export. Only database objects and attributes that are compatible with the specified release are exported. You can use the VERSION parameter to create a dump file set that is compatible with a previous release of Oracle Database. You cannot use Data Pump Export with releases of Oracle Database before Oracle Database 10*g* release 1 (10.1). Data Pump Export only works with Oracle Database 10*g* release 1 (10.1) or later. The VERSION parameter simply allows you to identify the version of objects that you export.

On Oracle Database 11*g* release 2 (11.2.0.3) or later, you can specify the VERSION parameter as VERSION=12 with FULL=Y to generate a full export dump file that is ready for import into Oracle Database 12*c*. The export with the later release target VERSION value includes information from registered database options and components. The dump file set specifying a later release version can only be imported into Oracle Database 12c Release 1 (12.1.0.1) and later. For example, if VERSION=12 is used with FULL=Y and also with TRANSPORTABLE=ALWAYS, then a full transportable export dump file is generated that is ready for import into Oracle Database 12*c*. For more information, refer to the FULL export parameter option.

Syntax and Description

```
VERSION=[COMPATIBLE | LATEST | version_string]
```

The legal values for the VERSION parameter are as follows:

- COMPATIBLE - This value is the default value. The version of the metadata corresponds to the database compatibility level as specified on the COMPATIBLE initialization parameter.

  Note: Database compatibility must be set to 9.2 or later.
- LATEST - The version of the metadata and resulting SQL DDL corresponds to the database release, regardless of its compatibility level.
- *version_string* - A specific database release (for example, 11.2.0). In Oracle Database 11*g*, this value cannot be lower than 9.2.

Database objects or attributes that are incompatible with the release specified for VERSION are not exported. For example, tables containing new data types that are not supported in the specified release are not exported.

Restrictions

- Exporting a table with archived LOBs to a database release earlier than 11.2 is not allowed.

- If the Data Pump Export VERSION parameter is specified with the TRANSPORT_TABLESPACES parameter, then the value for VERSION must be equal to or greater than the Oracle Database COMPATIBLE initialization parameter.
- If the Data Pump VERSION parameter is specified as any value earlier than 12.1, then the Data Pump dump file excludes any tables that contain VARCHAR2 or NVARCHAR2 columns longer than 4000 bytes, and any RAW columns longer than 2000 bytes.
- Dump files created on Oracle Database 11*g* releases with the Data Pump parameter VERSION=12 can only be imported on Oracle Database 12*c* Release 1 (12.1) and later.

Example

The following example shows an export for which the version of the metadata corresponds to the database release:

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

Related Topics

- Full Mode
- Exporting and Importing Between Different Oracle Database Releases

2.4.52 VIEWS_AS_TABLES

The Oracle Data Pump Export command-line utility VIEWS_AS_TABLES parameter specifies that you want one or more views exported as tables.

Default

There is no default.

> **Caution:**
>
> The VIEWS_AS_TABLES parameter unloads view data in unencrypted format, and creates an unencrypted table. If you are unloading sensitive data, then Oracle strongly recommends that you enable encryption on the export operation, and that you ensure the table is created in an encrypted tablespace. You can use the REMAP_TABLESPACE parameter to move the table to such a tablespace.

Purpose

Specifies that you want one or more views exported as tables.

Syntax and Description

```
VIEWS_AS_TABLES=[schema_name.]view_name[:table_name], ...
```

Oracle Data Pump exports a table with the same columns as the view, and with row data obtained from the view. Oracle Data Pump also exports objects dependent on the view, such as grants and constraints. Dependent objects that do not apply to tables (for example, grants of the UNDER object privilege) are not exported. You can use the VIEWS_AS_TABLES parameter

by itself, or use it with the TABLES parameter. Either way you use the parameter, Oracle Data Pump performs a table-mode export.

The syntax elements are defined as follows:

schema_name: The name of the schema in which the view resides. If a schema name is not supplied, then it defaults to the user performing the export.

view_name: The name of the view that you want exported as a table.

table_name: The name of a table that you want to serve as the source of the metadata for the exported view. By default, Oracle Data Pump automatically creates a temporary "template table" with the same columns and data types as the view, but with no rows. If the database is read-only, then this default creation of a template table fails. In such a case, you can specify a table name.

If the export job contains multiple views with explicitly specified template tables, then the template tables must all be different. For example, in the following job (in which two views use the same template table) one of the views is skipped:

```
expdp scott/password directory=dpump_dir dumpfile=a.dmp
views_as_tables=v1:emp,v2:emp
```

An error message is returned reporting the omitted object.

Template tables are automatically dropped after the export operation is completed. While they exist, you can perform the following query to view their names (which all begin with KU$VAT):

```
SQL> SELECT * FROM user_tab_comments WHERE table_name LIKE 'KU$VAT%';
TABLE_NAME                     TABLE_TYPE
------------------------------ -----------
COMMENTS
--------------------------------------------------
KU$VAT_63629                   TABLE
Data Pump metadata template table for view SCOTT.EMPV
```

Restrictions

- The VIEWS_AS_TABLES parameter cannot be used with the TRANSPORTABLE=ALWAYS parameter.
- Tables that you want to serve as the source of the metadata for the exported view must be in the same schema as the view.
- Tables that you want to serve as the source of the metadata for the exported view must be non-partitioned relational tables with heap organization.
- Tables that you want to serve as the source of the metadata for the exported view cannot be nested tables.
- Tables created using the VIEWS_AS_TABLES parameter do not contain any hidden or invisible columns that were part of the specified view.
- Views that you want exported as tables must exist, and must be relational views with only scalar, non-LOB columns. If you specify an invalid or non-existent view, then the view is skipped, and an error message is returned.
- The VIEWS_AS_TABLES parameter does not support tables that have columns with a data type of LONG.

Example

The following example exports the contents of view `scott.view1` to a dump file named `scott1.dmp`.

```
> expdp scott/password views_as_tables=view1 directory=data_pump_dir
dumpfile=scott1.dmp
```

The dump file contains a table named `view1` with rows obtained from the view.

2.5 Commands Available in Data Pump Export Interactive-Command Mode

Check which command options are available to you when using Data Pump Export in interactive mode.

- About Oracle Data Pump Export Interactive Command Mode
  Learn about commands you can use with Oracle Data Pump Export in interactive command mode while your current job is running.
- ADD_FILE
  The Oracle Data Pump Export interactive command mode `ADD_FILE` parameter adds additional files or substitution variables to the export dump file set.
- CONTINUE_CLIENT
  The Oracle Data Pump Export interactive command mode `CONTINUE_CLIENT` parameter changes the Export mode from interactive-command mode to logging mode.
- EXIT_CLIENT
  The Oracle Data Pump Export interactive command mode `EXIT_CLIENT` parameter stops the export client session, exits Export, and discontinues logging to the terminal, but leaves the current job running.
- FILESIZE
  The Oracle Data Pump Export interactive command mode `FILESIZE` parameter redefines the maximum size of subsequent dump files.
- HELP
  The Oracle Data Pump Export interactive command mode `HELP` parameter provides information about Data Pump Export commands available in interactive-command mode.
- KILL_JOB
  The Oracle Data Pump Export interactive command mode `KILL_JOB` parameter detaches all currently attached worker client sessions, and then terminates the current job. It exits Export, and returns to the terminal prompt.
- PARALLEL
  The Export Interactive-Command Mode `PARALLEL` parameter enables you to increase or decrease the number of active processes (child and parallel child processes) for the current job.
- START_JOB
  The Oracle Data Pump Export interactive command mode `START_JOB` parameter starts the current job to which you are attached.

- STATUS
  The Oracle Data Pump Export interactive command `STATUS` parameter displays status information about the export, and enables you to set the display interval for logging mode status.
- STOP_JOB
  The Oracle Data Pump Export interactive command mode `STOP_JOB` parameter stops the current job. It stops the job either immediately, or after an orderly shutdown, and exits Export.

2.5.1 About Oracle Data Pump Export Interactive Command Mode

Learn about commands you can use with Oracle Data Pump Export in interactive command mode while your current job is running.

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

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. `ATTACH` 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.

The following table lists the activities that you can perform for the current job from the Data Pump Export prompt in interactive-command mode.

Table 2-1 Supported Activities in Data Pump Export's Interactive-Command Mode

Activity	Command Used
Add additional dump files.	`ADD_FILE`
Exit interactive mode and enter logging mode.	`CONTINUE_CLIENT`
Stop the export client session, but leave the job running.	EXIT_CLIENT
Redefine the default size to be used for any subsequent dump files.	`FILESIZE`
Display a summary of available commands.	HELP
Detach all currently attached client sessions and terminate the current job.	KILL_JOB
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 11*g* or later.	PARALLEL
Restart a stopped job to which you are attached.	START_JOB
Display detailed status for the current job and/or set status interval.	STATUS
Stop the current job for later restart.	STOP_JOB

2.5.2 ADD_FILE

The Oracle Data Pump Export interactive command mode `ADD_FILE` parameter adds additional files or substitution variables to the export dump file set.

Purpose

Adds additional files or substitution variables to the export dump file set.

Syntax and Description

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

Each file name can have a different directory object. If no directory object is specified, then the default is assumed.

The *`file_name`* must not contain any directory path information. However, it can include a substitution variable, `%U`, which indicates that multiple files can be generated using the specified file name as a template.

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

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 for the job is assumed. A different directory object, `dpump_dir2`, is specified for the dump file named `hr3.dmp`.

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

Related Topics

- File Allocation with Oracle Data Pump

2.5.3 CONTINUE_CLIENT

The Oracle Data Pump Export interactive command mode `CONTINUE_CLIENT` parameter changes the Export mode from interactive-command mode to logging mode.

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` also causes the client to attempt to start the job.

Example

```
Export> CONTINUE_CLIENT
```

2.5.4 EXIT_CLIENT

The Oracle Data Pump Export interactive command mode `EXIT_CLIENT` parameter stops the export client session, exits Export, and discontinues logging to the terminal, but leaves the current job running.

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` 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

```
Export> EXIT_CLIENT
```

2.5.5 FILESIZE

The Oracle Data Pump Export interactive command mode `FILESIZE` parameter redefines the maximum size of subsequent dump files.

Purpose

Redefines the maximum size of subsequent dump files. If the size is reached for any member of the dump file set, then that file is closed and an attempt is made to create a new file, if the file specification contains a substitution variable or if additional dump files have been added to the job.

Syntax and Description

```
FILESIZE=integer[B | KB | MB | GB | TB]
```

The *integer* can be immediately followed (do not insert a space) by `B`, `KB`, `MB`, `GB`, or `TB` (indicating bytes, kilobytes, megabytes, gigabytes, and terabytes 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.

A file size of 0 is equivalent to the maximum file size of 16 TB.

Restrictions

- The minimum size for a file is ten times the default Oracle Data Pump block size, which is 4 kilobytes.
- The maximum size for a file is 16 terabytes.

Example

```
Export> FILESIZE=100MB
```

2.5.6 HELP

The Oracle Data Pump Export interactive command mode `HELP` parameter provides information about Data Pump Export commands available in interactive-command mode.

Purpose

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

Syntax and Description

```
HELP
```

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

Example

```
Export> HELP
```

2.5.7 KILL_JOB

The Oracle Data Pump Export interactive command mode `KILL_JOB` parameter detaches all currently attached worker client sessions, and then terminates the current job. It exits Export, and returns to the terminal prompt.

Purpose

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

Syntax and Description

```
KILL_JOB
```

A job that is terminated 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 terminated by the current user and are then detached. After all child clients are detached, the job's process structure is immediately run down and the Data Pump control job table and dump files are deleted. Log files are not deleted.

Example

```
Export> KILL_JOB
```

2.5.8 PARALLEL

The Export Interactive-Command Mode `PARALLEL` parameter enables you to increase or decrease the number of active processes (child and parallel child processes) for the current job.

Purpose

Enables you to increase or decrease the number of active processes (child and parallel child processes) for the current job.

Syntax and Description

```
PARALLEL=integer
```

`PARALLEL` is available as both a command-line parameter, and as an interactive-command mode parameter. You set it to the desired number of parallel processes (child and parallel child 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, then child processes are idled but not deleted until the job exits.

Restrictions

- This parameter is valid only in the Enterprise Edition of Oracle Database 11g or later releases.
- Transportable tablespace metadata cannot be imported in parallel.
- Metadata cannot be imported in parallel when the `NETWORK_LINK` parameter is used.

In addition, the following objects cannot be imported in parallel:

- `TRIGGER`
- `VIEW`
- `OBJECT_GRANT`
- `SEQUENCE`
- `CONSTRAINT`
- `REF_CONSTRAINT`

Example

```
Export> PARALLEL=10
```

Related Topics

- PARALLEL

2.5.9 START_JOB

The Oracle Data Pump Export interactive command mode `START_JOB` parameter starts the current job to which you are attached.

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 running when you issue the command. 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 Data Pump control job table have not been altered in any way.

Exports done in transportable-tablespace mode are not restartable.

Example

```
Export> START_JOB
```

2.5.10 STATUS

The Oracle Data Pump Export interactive command `STATUS` parameter displays status information about the export, and enables you to set the display interval for logging mode status.

Purpose

Displays cumulative status of the job, a description of the current operation, and an estimated completion percentage. It also allows you to reset the display interval for logging mode status.

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, then 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 displays the current job status, and changes the logging mode display interval to five minutes (300 seconds):

```
Export> STATUS=300
```

2.5.11 STOP_JOB

The Oracle Data Pump Export interactive command mode `STOP_JOB` parameter stops the current job. It stops the job either immediately, or after an orderly shutdown, and exits Export.

Purpose

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

Syntax and Description

```
STOP_JOB[=IMMEDIATE]
```

If the Data Pump control job table and dump file set are not disturbed when or after the `STOP_JOB` command is issued, then 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 Data Pump control job process will not wait for the child 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

```
Export> STOP_JOB=IMMEDIATE
```

2.6 Examples of Using Oracle Data Pump Export

You can use these common scenario examples to learn how you can create parameter files and use Oracle Data Pump Export to move your data.

- Performing a Table-Mode Export
  This example shows a table-mode export, specified using the `TABLES` parameter.
- Data-Only Unload of Selected Tables and Rows
  This example shows data-only unload of selected tables and rows.
- Estimating Disk Space Needed in a Table-Mode Export
  This example shows how to estimate the disk space needed in a table-mode export.
- Performing a Schema-Mode Export
  This example shows you how to perform a schema-mode export.
- Performing a Parallel Full Database Export
  To learn how to perform a parallel full database export, use this example to understand the syntax.
- Using Interactive Mode to Stop and Reattach to a Job
  This example shows you how to use interactive mode to stop and reattach to a job.

2.6.1 Performing a Table-Mode Export

This example shows a table-mode export, specified using the TABLES parameter.

In this example, the Data Pump export command performs a table export of the tables employees and jobs from the human resources (hr) schema.

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

Example 2-1 Performing a Table-Mode Export

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

2.6.2 Data-Only Unload of Selected Tables and Rows

This example shows data-only unload of selected tables and rows.

The example shows the contents of a parameter file (exp.par), which you can use to perform a data-only unload of all the tables in the human resources (hr) schema, except for the tables countries and regions. Rows in the employees table are unloaded that have a department_id other than 50. The rows are ordered by employee_id.

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

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

This export performs a schema-mode export (the default mode), but the CONTENT parameter effectively limits the export to an unload of just the table data. 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.

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

```
DIRECTORY=dpump_dir1
DUMPFILE=dataonly.dmp
CONTENT=DATA_ONLY
EXCLUDE=TABLE:"IN ('COUNTRIES', 'REGIONS')"
QUERY=employees:"WHERE department_id !=50 ORDER BY employee_id"
```

2.6.3 Estimating Disk Space Needed in a Table-Mode Export

This example shows how to estimate the disk space needed in a table-mode export.

In this example, the ESTIMATE_ONLY parameter is used to estimate the space that is 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.

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.

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

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

2.6.4 Performing a Schema-Mode Export

This example shows you how to perform a schema-mode export.

The example 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 DUMPFILE=dpump_dir1:expschema.dmp LOGFILE=dpump_dir1:expschema.log
```

2.6.5 Performing a Parallel Full Database Export

To learn how to perform a parallel full database export, use this example to understand the syntax.

The example shows a full database Export that can use 3 parallel processes (worker or parallel query worker processes).

Example 2-5 Parallel Full Export

```
> expdp hr FULL=YES 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 export is a full database export, all data and metadata in the database is exported. Dump files `full101.dmp`, `full201.dmp`, `full102.dmp`, and so on, are created in a round-robin fashion in the directories pointed to by the `dpump_dir1` and `dpump_dir2` directory objects. For best performance, Oracle recommends that you place the dump files on separate input/output (I/O) channels. Each file is up to 2 gigabytes in size, as necessary. Initially, up to three files are created. If needed, more files are created. The job and Data Pump control process table has a name of `expfull`. The log file is written to `expfull.log` in the `dpump_dir1` directory.

2.6.6 Using Interactive Mode to Stop and Reattach to a Job

This example shows you how to use interactive mode to stop and reattach to a job.

To start this example, reexecute the parallel full export described here:

Performing a Parallel Full Database Export

While the export is running, press Ctrl+C. This keyboard command starts the interactive-command interface of Data Pump Export. In the interactive interface, logging to the terminal stops, and the Export prompt is displayed.

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.

Example 2-6 Stopping and Reattaching to a Job

At the Export prompt, 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.

To reattach to the job you just stopped, enter the following command:

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

2.7 Syntax Diagrams for Oracle Data Pump Export

You can use syntax diagrams to understand the valid SQL syntax for Oracle Data Pump Export.

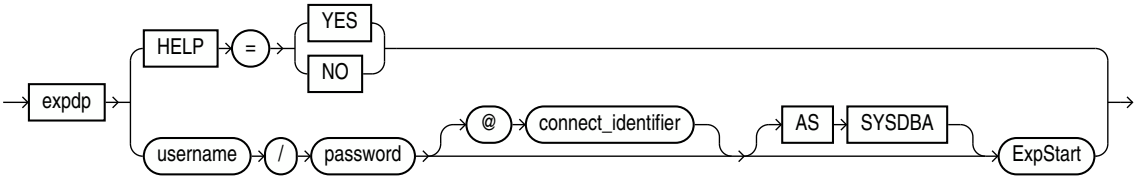
How to Read Graphic Syntax Diagrams

Syntax diagrams are drawings that illustrate valid SQL syntax. To read a diagram, trace it from left to right, in the direction shown by the arrows.

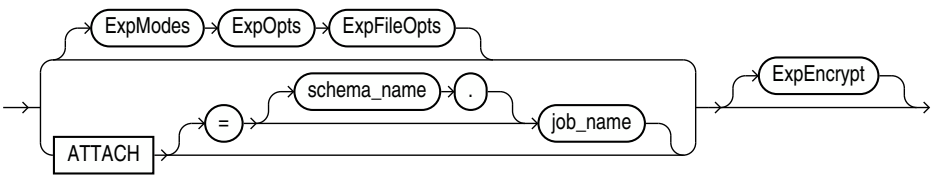
For more information about standard SQL syntax notation, see:

How to Read Syntax Diagrams in *Oracle Database SQL Language Reference*

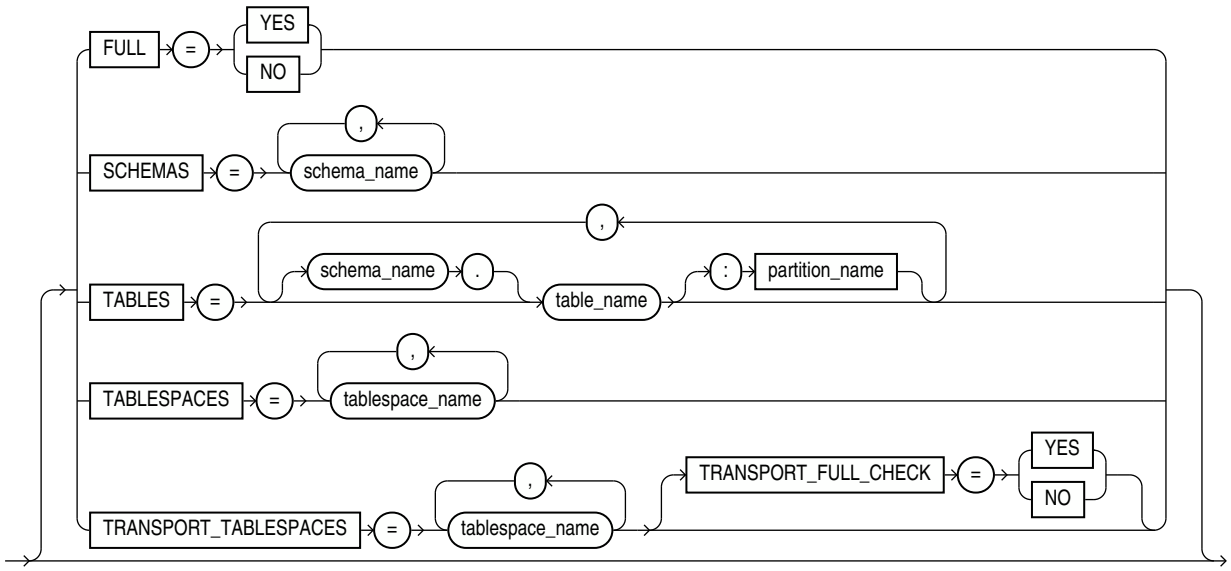
ExpInit



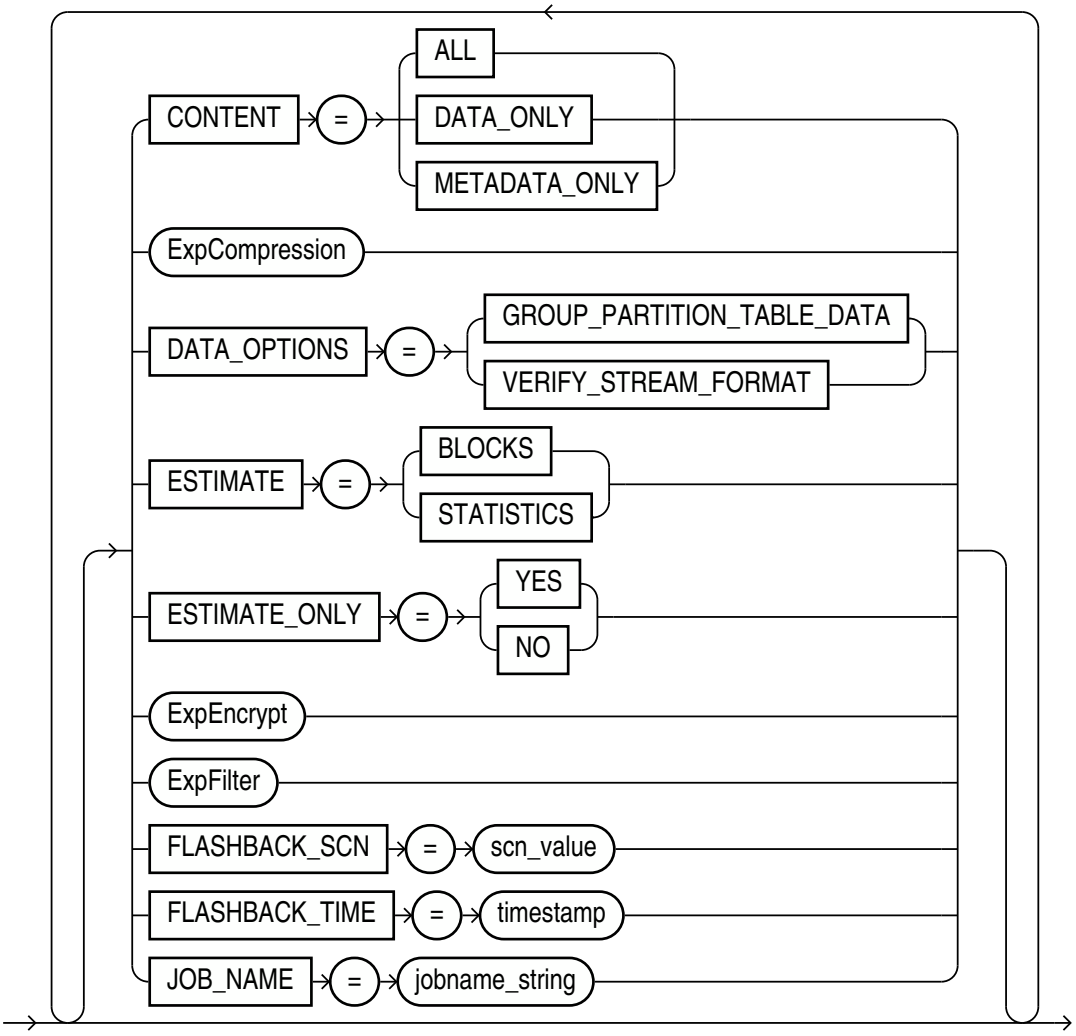
ExpStart



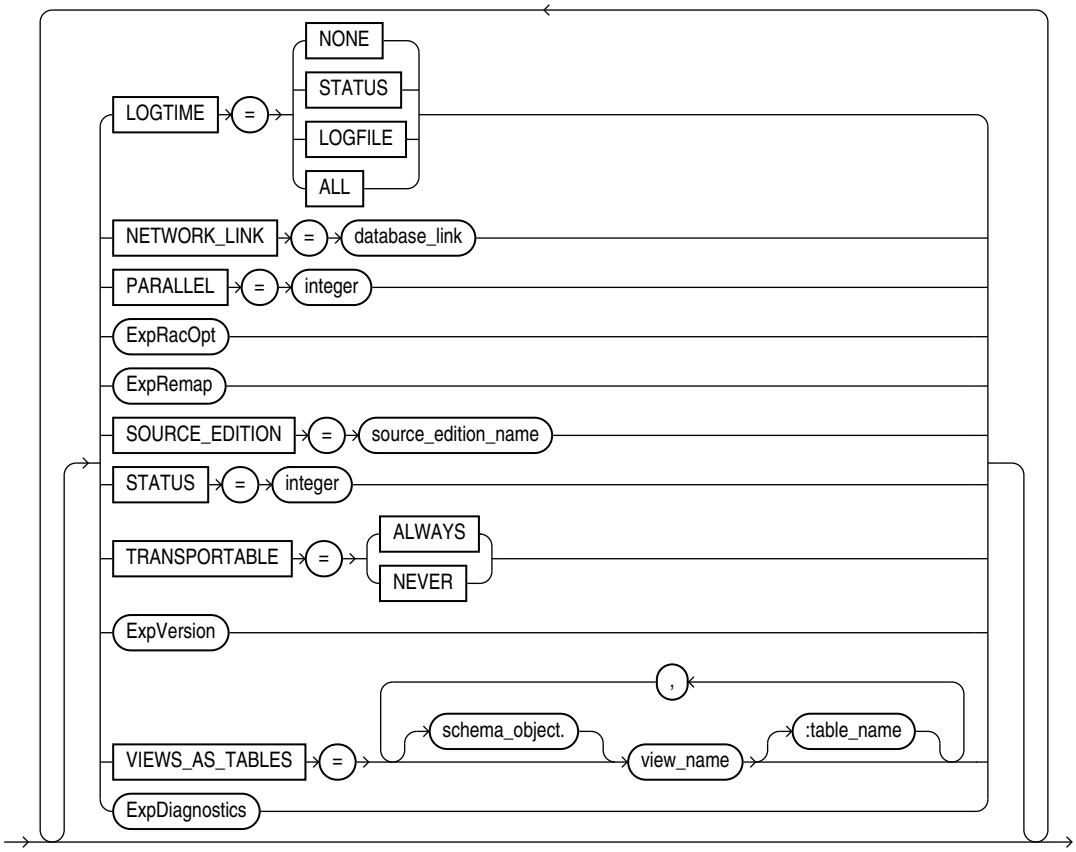
ExpModes


FULL
=
YES
NO
SCHEMAS
=
,
schema_name
TABLES
=
,
schema_name
.
table_name
:
partition_name
TABLESPACES
=
,
tablespace_name
TRANSPORT_TABLESPACES
=
,
tablespace_name
TRANSPORT_FULL_CHECK
=
YES
NO

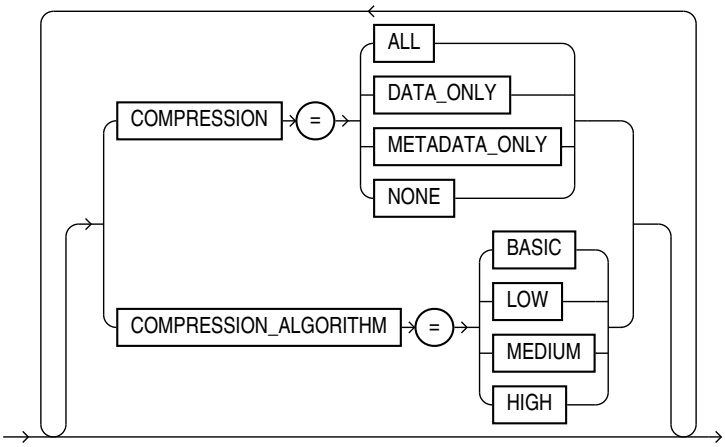
ExpOpts


CONTENT
=
ALL
DATA_ONLY
METADATA_ONLY
ExpCompression
DATA_OPTIONS
=
GROUP_PARTITION_TABLE_DATA
VERIFY_STREAM_FORMAT
ESTIMATE
=
BLOCKS
STATISTICS
ESTIMATE_ONLY
=
YES
NO
ExpEncrypt
ExpFilter
FLASHBACK_SCN
=
scn_value
FLASHBACK_TIME
=
timestamp
JOB_NAME
=
jobname_string

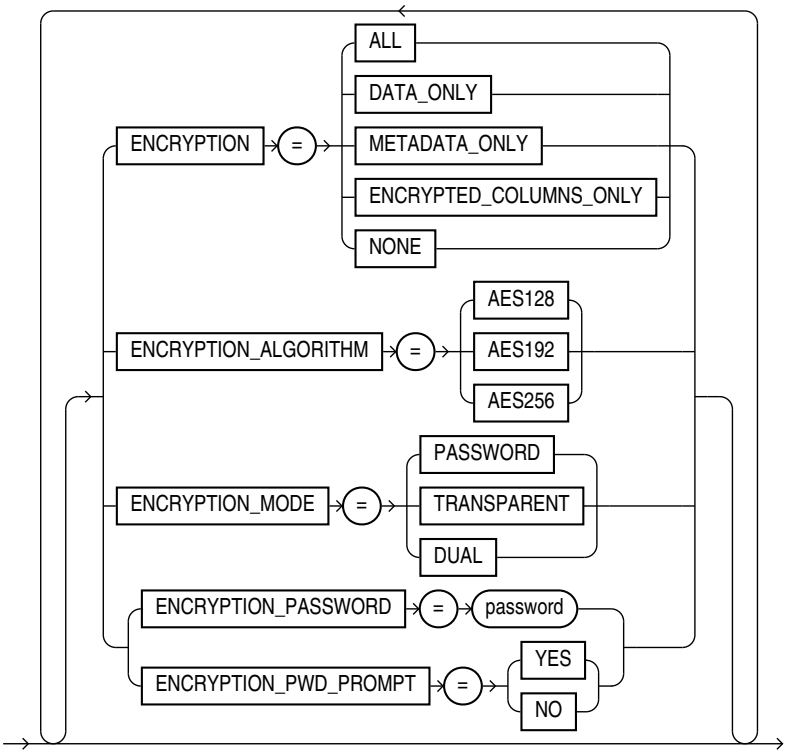
ExpOpts_Cont



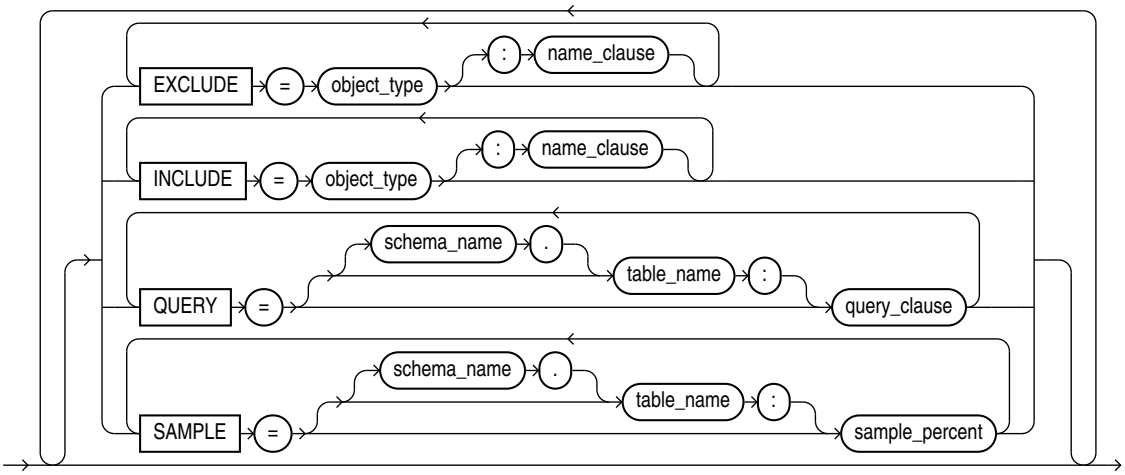
ExpCompression



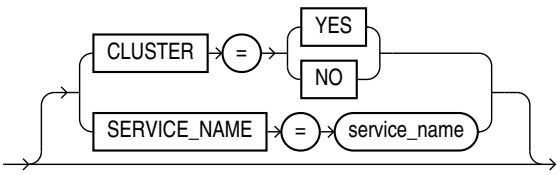
ExpEncrypt



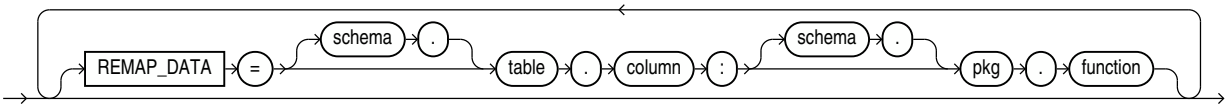
ExpFilter



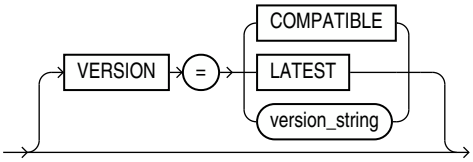
ExpRacOpt



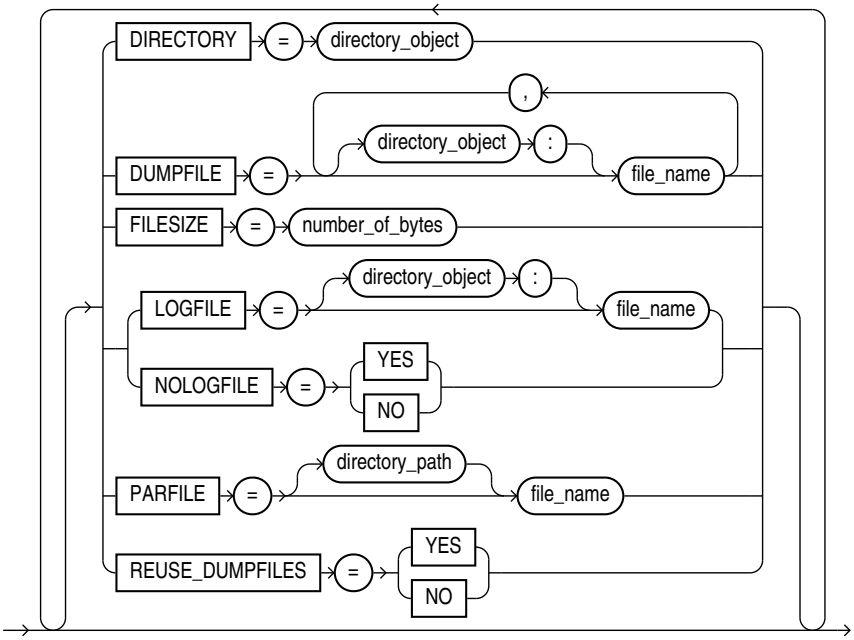
ExpRemap



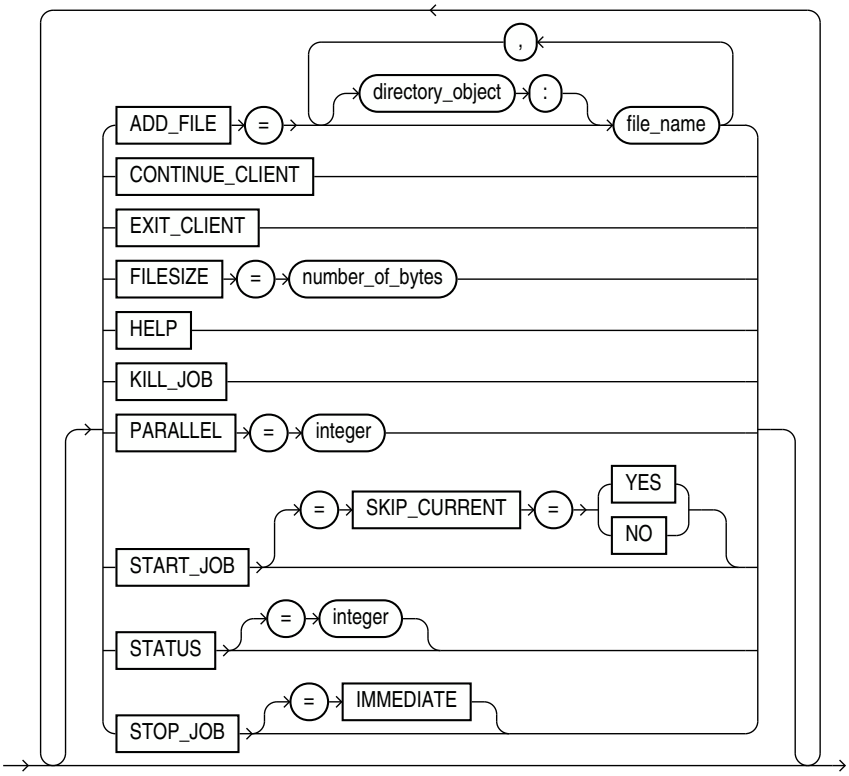
ExpVersion



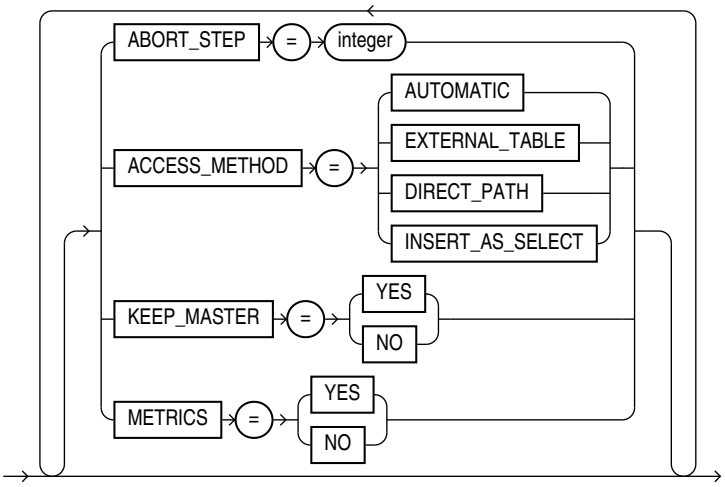
ExpFileOpts



ExpDynOpts



ExpDiagnostics



3
Oracle Data Pump Import

With Oracle Data Pump Import, you can load an export dump file set into a target database, or load a target database directly from a source database with no intervening files.

- What Is Oracle Data Pump Import?
  Oracle Data Pump Import is a utility for loading an Oracle export dump file set into a target system.
- Starting Oracle Data Pump Import
  Start the Oracle Data Pump Import utility by using the `impdp` command.
- Filtering During Import Operations
  Oracle Data Pump Import provides data and metadata filtering capability, which can help you limit the type of information that you import.
- Parameters Available in Oracle Data Pump Import Command-Line Mode
  Use Oracle Data Pump parameters for Import (`impdp`) to manage your data imports.
- Commands Available in Oracle Data Pump Import Interactive-Command Mode
  In interactive-command mode, the current job continues running, but logging to the terminal is suspended, and the Import prompt (`Import>`) is displayed.
- Examples of Using Oracle Data Pump Import
  You can use these common scenario examples to learn how you can use Oracle Data Pump Import to move your data.
- Syntax Diagrams for Oracle Data Pump Import
  You can use syntax diagrams to understand the valid SQL syntax for Oracle Data Pump Import.

3.1 What Is Oracle Data Pump Import?

Oracle Data Pump Import is a utility for loading an Oracle export dump file set into a target system.

An export 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 Oracle Data Pump import operation, the Import utility uses these files to locate each database object in the dump file set.

You can also use Import to load a target database directly from a source database with no intervening dump files. This type of import is called a **network import**.

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 by using data filters and metadata filters, which are implemented through Import commands.

3.2 Starting Oracle Data Pump Import

Start the Oracle Data Pump Import utility by 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 start 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.
> - Be aware that if you are performing a Data Pump Import into a table or tablespace created with the `NOLOGGING` clause enabled, then a redo log file may still be generated. The redo that is generated in such a case is generally for maintenance of the Data Pump control table, or related to underlying recursive space transactions, data dictionary changes, and index maintenance for indices on the table that require logging.
> - If the timezone version used by the export database is older than the version used by the import database, then loading columns with data type `TIMESTAMP WITH TIMEZONE` takes longer than it would otherwise. This additional time is required because the database must check to determine if the new timezone rules change the values being loaded.

- Oracle Data Pump Import Interfaces
  You can interact with Oracle Data Pump Import by using a command line, a parameter file, or an interactive-command mode.
- Oracle Data Pump Import Modes
  The import mode that you use for Oracle Data Pump determines what is imported.
- Network Considerations for Oracle Data Pump Import
  Learn how Oracle Data Pump Import utility `impdp` identifies instances with connect identifiers in the connection string using Oracle*Net or a net service name, and how they are different from import operations using the NETWORK_LINK parameter.

3.2.1 Oracle Data Pump Import Interfaces

You can interact with Oracle 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.
- 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. The use of parameter files is recommended if you are using parameters whose values require quotation marks.
- 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.

Related Topics

- Parameters Available in Oracle Data Pump Import Command-Line Mode
  Use Oracle Data Pump parameters for Import (`impdp`) to manage your data imports.
- Commands Available in Oracle Data Pump Import Interactive-Command Mode
  In interactive-command mode, the current job continues running, but logging to the terminal is suspended, and the Import prompt (`Import>`) is displayed.

3.2.2 Oracle Data Pump Import Modes

The import mode that you use for Oracle Data Pump determines what is imported.

- About Oracle Data Pump Import Modes
  Learn how Oracle Data Pump Import modes operate during the import.
- Full Import Mode
  To specify a full import with Oracle Data Pump, use the `FULL` parameter.
- Schema Mode
  To specify a schema import with Oracle Data Pump, use the `SCHEMAS` parameter.
- Table Mode
  To specify a table mode import with Oracle Data Pump, use the `TABLES` parameter.
- Tablespace Mode
  To specify a tablespace mode import with Oracle Data Pump, use the `TABLESPACES` parameter.
- Transportable Tablespace Mode
  To specify a transportable tablespace mode import with Oracle Data Pump, use the `TRANSPORT_TABLESPACES` parameter.

3.2.2.1 About Oracle Data Pump Import Modes

Learn how Oracle Data Pump Import modes operate during the import.

The Oracle Data Pump import mode that you specify for the import applies to the source of the operation. If you specify the `NETWORK_LINK` parameter, then that source is either a dump file set, or another database.

When the source of the import operation is a dump file set, specifying a mode is optional. If you do not specify a mode, 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.

> **Note:**
>
> When you import a dump file that was created by a full-mode export, the import operation attempts to copy the password for the `SYS` account from the source database. This copy sometimes fails (For example, if the password is in a shared password file). If it does fail, then after the import completes, you must set the password for the `SYS` account at the target database to a password of your choice.

3.2.2.2 Full Import Mode

To specify a full import with Oracle Data Pump, use 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 mode is the default for file-based imports. If the source is another database containing schemas other than your own, then you must have the `DATAPUMP_IMP_FULL_DATABASE` role.

Cross-schema references are not imported for non-privileged users. For example, a trigger defined on a table within the schema of the importing user, but residing in another user schema, is not imported.

The `DATAPUMP_IMP_FULL_DATABASE` role is required on the target database. If the `NETWORK_LINK` parameter is used for a full import, then the `DATAPUMP_EXP_FULL_DATABASE` role is required on the source database

Using the Transportable Option During Full Mode Imports

You can use the transportable option during a full-mode import to perform a full transportable import.

Network-based full transportable imports require use of the `FULL=YES`, `TRANSPORTABLE=ALWAYS`, and `TRANSPORT_DATAFILES=`*`datafile_name`* parameters.

File-based full transportable imports only require use of the `TRANSPORT_DATAFILES=`*`datafile_name`* parameter. Data Pump Import infers the presence of the `TRANSPORTABLE=ALWAYS` and `FULL=Y` parameters.

There are several requirements when performing a full transportable import:

- Either you must also specify the `NETWORK_LINK` parameter, or the dump file set being imported must have been created using the transportable option during export.
- If you are using a network link, then the database specified on the `NETWORK_LINK` parameter must be Oracle Database 11g release 2 (11.2.0.3) or later, and the Oracle Data Pump `VERSION` parameter must be set to at least 12. (In a non-network import, `VERSION=12` is implicitly determined from the dump file.)
- If the source platform and the target platform are of different endianness, then you must convert the data being transported so that it is in the format of the target platform. To convert the data, you can use either the `DBMS_FILE_TRANSFER` package or the `RMAN CONVERT` command.
- If the source and target platforms do not have the same endianness, then a full transportable import of encrypted tablespaces is not supported in network mode or in dump file mode

For a detailed example of performing a full transportable import, see *Oracle Database Administrator's Guide*.

Related Topics

- FULL
  The Oracle Data Pump Import command-line mode `FULL` parameter specifies that you want to perform a full database import.

- TRANSPORTABLE
  The optional Oracle Data Pump Import command-line mode `TRANSPORTABLE` parameter specifies either that transportable tables are imported with `KEEP_READ_ONLY`, or `NO_BITMAP_REBUILD`.
- Transporting Tablespaces Between Databases in *Oracle Database Administrator's Guide*

3.2.2.3 Schema Mode

To specify a schema import with Oracle Data Pump, use the `SCHEMAS` parameter.

In a schema import, only objects owned by the specified schemas are loaded. The source can be a full, table, tablespace, or a schema-mode export dump file set, or another database. If you have the `DATAPUMP_IMP_FULL_DATABASE` role, then you can specify a list of schemas, 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 non-privileged users unless the other schema is remapped to the current schema. For example, a trigger defined on a table within the importing user's schema, but residing in another user's schema, is not imported.

Related Topics

- SCHEMAS
  The Oracle Data Pump Import command-line mode `SCHEMAS` parameter specifies that you want a schema-mode import to be performed.

3.2.2.4 Table Mode

To specify a table mode import with Oracle Data Pump, use the `TABLES` parameter.

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 `DATAPUMP_IMP_FULL_DATABASE` role to specify tables that are not in your own schema.

You can use the transportable option during a table-mode import by specifying the `TRANPORTABLE=ALWAYS` parameter with the `TABLES` parameter. If you use this option, then you must also use the `NETWORK_LINK` parameter.

To recover tables and table partitions, you can also use `RMAN` backups, and the `RMAN RECOVER TABLE` command. During this process, `RMAN` creates (and optionally imports) an Oracle Data Pump export dump file that contains the recovered objects.

Related Topics

- TABLES
  The Oracle Data Pump Import command-line mode `TABLES` parameter specifies that you want to perform a table-mode import.
- TRANSPORTABLE
  The optional Oracle Data Pump Import command-line mode `TRANSPORTABLE` parameter specifies either that transportable tables are imported with `KEEP_READ_ONLY`, or `NO_BITMAP_REBUILD`.
- Using Data File Copying to Move Data
  The fastest method of moving data is to copy the database data files to the target database without interpreting or altering the data. With this method, Data Pump Export is used to unload only structural information (metadata) into the dump file.

- *Oracle Database Backup and Recovery User's Guide*

3.2.2.5 Tablespace Mode

To specify a tablespace mode import with Oracle Data Pump, use the `TABLESPACES` parameter.

A tablespace-mode import is specified using the `TABLESPACES` parameter. In tablespace mode, all objects contained within the specified set of tablespaces are loaded, along with the dependent objects. The source can be a full, schema, tablespace, or table-mode export dump file set, or another database. For unprivileged users, objects not remapped to the current schema will not be processed.

Related Topics

- TABLESPACES
  The Oracle Data Pump Import command-line mode `TABLESPACES` parameter specifies that you want to perform a tablespace-mode import.

3.2.2.6 Transportable Tablespace Mode

To specify a transportable tablespace mode import with Oracle Data Pump, use the `TRANSPORT_TABLESPACES` parameter.

In transportable tablespace mode, the metadata from another database is loaded by using either a database link (specified with the `NETWORK_LINK` parameter), or by specifying a dump file that contains the metadata. The actual data files, 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.

When transportable jobs are performed, Oracle recommends that you keep a copy of the data files on the source system until the import job has successfully completed on the target system. With a copy of the data files, if the import job should fail for some reason, then you still have uncorrupted copies of the data files.

Using this mode requires the `DATAPUMP_IMP_FULL_DATABASE` role.

> **Note:**
>
> You cannot export transportable tablespaces and then import them into a database at a lower release level. The target database must be at the same or later release level as the source database.

Related Topics

- How Does Oracle Data Pump Handle Timestamp Data?
  Learn about factors that can affect successful completion of export and import jobs that involve the timestamp data types `TIMESTAMP WITH TIMEZONE` and `TIMESTAMP WITH LOCAL TIMEZONE`.
- Using Data File Copying to Move Data
  The fastest method of moving data is to copy the database data files to the target database without interpreting or altering the data. With this method, Data Pump Export is used to unload only structural information (metadata) into the dump file.

3.2.3 Network Considerations for Oracle Data Pump Import

Learn how Oracle Data Pump Import utility `impdp` identifies instances with connect identifiers in the connection string using Oracle*Net or a net service name, and how they are different from import operations using the NETWORK_LINK parameter.

When you start `impdp`, you can specify a connect identifier in the connect string that can be different from the current instance identified by the current Oracle System ID (SID).

You can specify a connect identifier by using either an Oracle*Net connect descriptor, or by using a net service name (usually defined in the `tnsnames.ora` file) that maps to a connect descriptor. Use of a connect identifier requires that you have Oracle Net Listener running (to start the default listener, enter `lsnrctl start`). The following example shows this type of connection, in which `inst1` is the connect identifier:

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

Import then prompts you for a password:

```
Password: password
```

The local Import client connects to the database instance identified by the connect identifier `inst1` (a net service name), and imports the data from the dump file `hr.dmp` to `inst1`.

Specifying a connect identifier when you start the Import utility is different from performing an import operation using the `NETWORK_LINK` parameter. When you start an import operation and specify a connect identifier, the local Import client connects to the database instance identified by the connect identifier and imports the data from the dump file named on the command line to that database instance.

By contrast, when you perform an import using the `NETWORK_LINK` parameter, the import is performed using a database link, and there is no dump file involved. (A database link is a connection between two physical database servers that allows a client to access them as one logical database.)

Related Topics

- NETWORK_LINK
  The Oracle Data Pump Import command-line mode `NETWORK_LINK` parameter enables an import from a source database identified by a valid database link.

> **See Also:**
>
> - *Oracle Database Administrator's Guide* for more information about database links
> - *Oracle Database Net Services Administrator's Guide* for more information about connect identifiers and Oracle Net Listener

3.3 Filtering During Import Operations

Oracle Data Pump Import provides data and metadata filtering capability, which can help you limit the type of information that you import.

- Data Filters
  Data-specific filtering is implemented through the `QUERY` and `SAMPLE` parameters, which specify restrictions on the table rows that are to be imported.
- Metadata Filters
  Metadata filtering is implemented through the `EXCLUDE` and `INCLUDE` parameters.

3.3.1 Data Filters

Data-specific filtering is implemented through the `QUERY` and `SAMPLE` parameters, which specify restrictions on the table rows that are to be imported.

Data filtering can also occur indirectly because of metadata filtering, which can include or exclude table objects along with any associated row data.

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, then the filter parameter supplied for the specific table takes precedence.

3.3.2 Metadata Filters

Metadata filtering is implemented through the `EXCLUDE` and `INCLUDE` parameters.

Data Pump Import provides much greater metadata filtering capability than was provided by the original Import utility. 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, then 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 a list of valid object types, query the following views: `DATABASE_EXPORT_OBJECTS` for full mode, `SCHEMA_EXPORT_OBJECTS` for schema mode, and `TABLE_EXPORT_OBJECTS` for table and tablespace mode. The values listed in the `OBJECT_PATH` column are the valid object types. Note that full object path names are determined by the export mode, not by the import mode.

Related Topics

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

- EXCLUDE
  The Oracle Data Pump Import command-line mode `EXCLUDE` parameter enables you to filter the metadata that is imported by specifying objects and object types to exclude from the import job.
- INCLUDE
  The Oracle Data Pump Import command-line mode `INCLUDE` parameter enables you to filter the metadata that is imported by specifying objects and object types for the current import mode.

3.4 Parameters Available in Oracle Data Pump Import Command-Line Mode

Use Oracle Data Pump parameters for Import (`impdp`) to manage your data imports.

- About Import Command-Line Mode
  Learn how to use Oracle Data Pump Import parameters in command-line mode, including case sensitivity, quotation marks, escape characters, and information about how to use examples.
- ABORT_STEP
  The Oracle Data Pump Import command-line mode `ABORT_STEP` parameter stops the job after it is initialized. Stopping the job enables the Data Pump control job table to be queried before any data is imported.
- ACCESS_METHOD
  The Oracle Data Pump Import command-line mode `ACCESS_METHOD` parameter instructs Import to use a particular method to load data
- ATTACH
  The Oracle Data Pump Import command-line mode `ATTACH` parameter attaches the client session to an existing import job and automatically places you in interactive-command mode.
- CLUSTER
  The Oracle Data Pump Import command-line mode `CLUSTER` parameter determines whether Data Pump can use Oracle Real Application Clusters (Oracle RAC) resources, and start workers on other Oracle RAC instances.
- CONTENT
  The Oracle Data Pump Import command-line mode `CONTENT` parameter enables you to filter what is loaded during the import operation.
- CREDENTIAL
  The Oracle Data Pump Import command-line mode `CREDENTIAL` parameter specifies the credential object name owned by the database user that Import uses to process files in the dump file set imported into cloud storage.
- DATA_OPTIONS
  The Oracle Data Pump Import command-line mode `DATA_OPTIONS` parameter designates how you want certain types of data to be handled during import operations.
- DIRECTORY
  The Oracle Data Pump Import command-line mode `DIRECTORY` parameter specifies the default location in which the import job can find the dump file set, and create log and SQL files.

- DUMPFILE
  The Oracle Data Pump Import command-line mode `DUMPFILE` parameter specifies the names, and optionally, the directory objects of the dump file set that Export created.
- ENABLE_SECURE_ROLES
  The Oracle Data Pump Import command-line utility `ENABLE_SECURE_ROLES` parameter prevents inadvertent use of protected roles during exports.
- ENCRYPTION_PASSWORD
  The Oracle Data Pump Import command-line mode `ENCRYPTION_PASSWORD` parameter specifies a password for accessing encrypted column data in the dump file set. This prevents unauthorized access to an encrypted dump file set.
- ENCRYPTION_PWD_PROMPT
  The Oracle Data Pump Import command-line mode `ENCRYPTION_PWD_PROMPT` parameter specifies whether Data Pump should prompt you for the encryption password.
- ESTIMATE
  The Oracle Data Pump Import command-line mode `ESTIMATE` parameter instructs the source system in a network import operation to estimate how much data is generated during the import.
- EXCLUDE
  The Oracle Data Pump Import command-line mode `EXCLUDE` parameter enables you to filter the metadata that is imported by specifying objects and object types to exclude from the import job.
- FLASHBACK_SCN
  The Oracle Data Pump Import command-line mode `FLASHBACK_SCN` specifies the system change number (SCN) that Import uses to enable the Flashback utility.
- FLASHBACK_TIME
  The Oracle Data Pump Import command-line mode `FLASHBACK_TIME` parameter specifies the system change number (SCN) that Import uses to enable the Flashback utility.
- FULL
  The Oracle Data Pump Import command-line mode `FULL` parameter specifies that you want to perform a full database import.
- HELP
  The Oracle Data Pump Import command-line mode `HELP` parameter displays online help for the Import utility.
- INCLUDE
  The Oracle Data Pump Import command-line mode `INCLUDE` parameter enables you to filter the metadata that is imported by specifying objects and object types for the current import mode.
- INDEX_THRESHOLD
- JOB_NAME
  The Oracle Data Pump Import command-line mode `JOB_NAME` parameter is used to identify the import job in subsequent actions.
- KEEP_MASTER
  The Oracle Data Pump Import command-line mode `KEEP_MASTER` parameter indicates whether the Data Pump control job table should be deleted or retained at the end of an Oracle Data Pump job that completes successfully.
- LOGFILE
  The Oracle Data Pump Import command-line mode `LOGFILE` parameter specifies the name, and optionally, a directory object, for the log file of the import job.

- LOGTIME
  The Oracle Data Pump Import command-line mode `LOGTIME` parameter specifies that you want to have messages displayed with timestamps during import.
- MASTER_ONLY
  The Oracle Data Pump Import command-line mode `MASTER_ONLY` parameter indicates whether to import just the Data Pump control job table, and then stop the job so that the contents of the Data Pump control job table can be examined.
- METRICS
  The Oracle Data Pump Import command-line mode `METRICS` parameter indicates whether additional information about the job should be reported to the log file.
- NETWORK_LINK
  The Oracle Data Pump Import command-line mode `NETWORK_LINK` parameter enables an import from a source database identified by a valid database link.
- NOLOGFILE
  The Oracle Data Pump Import command-line mode `NOLOGFILE` parameter specifies whether to suppress the default behavior of creating a log file.
- PARALLEL
  The Oracle Data Pump Import command-line mode `PARALLEL` parameter sets the maximum number of worker processes that can load in parallel.
- PARALLEL_THRESHOLD
  The Oracle Data Pump Import command-line utility `PARALLEL_THRESHOLD` parameter specifies the size of the divisor that Data Pump uses to calculate potential parallel DML based on table size.
- PARFILE
  The Oracle Data Pump Import command-line mode `PARFILE` parameter specifies the name of an import parameter file.
- PARTITION_OPTIONS
  The Oracle Data Pump Import command-line mode `PARTITION_OPTIONS` parameter specifies how you want table partitions created during an import operation.
- QUERY
  The Oracle Data Pump Import command-line mode `QUERY` parameter enables you to specify a query clause that filters the data that is imported.
- REMAP_DATA
  The Oracle Data Pump Import command-line mode `REMAP_DATA` parameter enables you to remap data as it is being inserted into a new database.
- REMAP_DATAFILE
  The Oracle Data Pump Import command-line mode `REMAP_DATAFILE` parameter changes the name of the source data file to the target data file name in all SQL statements where the source data file is referenced.
- REMAP_DIRECTORY
  The Oracle Data Pump Import command-line mode `REMAP_DIRECTORY` parameter lets you remap directories when you move databases between platforms.
- REMAP_SCHEMA
  The Oracle Data Pump Import command-line mode `REMAP_SCHEMA` parameter loads all objects from the source schema into a target schema.
- REMAP_TABLE
  The Oracle Data Pump Import command-line mode `REMAP_TABLE` parameter enables you to rename tables during an import operation.

- REMAP_TABLESPACE
  The Oracle Data Pump Import command-line mode `REMAP_TABLESPACE` parameter remaps all objects selected for import with persistent data in the source tablespace to be created in the target tablespace.
- SCHEMAS
  The Oracle Data Pump Import command-line mode `SCHEMAS` parameter specifies that you want a schema-mode import to be performed.
- SERVICE_NAME
  The Oracle Data Pump Import command-line mode `SERVICE_NAME` parameter specifies a service name that you want to use in conjunction with the `CLUSTER` parameter.
- SKIP_UNUSABLE_INDEXES
  The Oracle Data Pump Import command-line mode `SKIP_UNUSABLE_INDEXES` parameter specifies whether Import skips loading tables that have indexes that were set to the Index Unusable state (by either the system or the user).
- SOURCE_EDITION
  The Oracle Data Pump Import command-line mode `SOURCE_EDITION` parameter specifies the database edition on the remote node from which objects are fetched.
- SQLFILE
  The Oracle Data Pump Import command-line mode `SQLFILE` parameter specifies a file into which all the SQL DDL that Import prepares to execute is written, based on other Import parameters selected.
- STATUS
  The Oracle Data Pump Import command-line mode `STATUS` parameter specifies the frequency at which the job status is displayed.
- STREAMS_CONFIGURATION
  The Oracle Data Pump Import command-line mode `STREAMS_CONFIGURATION` parameter specifies whether to import any GoldenGate Replication metadata that may be present in the export dump file.
- TABLE_EXISTS_ACTION
  The Oracle Data Pump Import command-line mode `TABLE_EXISTS_ACTION` parameter specifies for Import what to do if the table it is trying to create already exists.
- REUSE_DATAFILES
  The Oracle Data Pump Import command-line mode `REUSE_DATAFILES` parameter specifies whether you want the import job to reuse existing data files for tablespace creation.
- TABLES
  The Oracle Data Pump Import command-line mode `TABLES` parameter specifies that you want to perform a table-mode import.
- TABLESPACES
  The Oracle Data Pump Import command-line mode `TABLESPACES` parameter specifies that you want to perform a tablespace-mode import.
- TARGET_EDITION
  The Oracle Data Pump Import command-line mode `TARGET_EDITION` parameter specifies the database edition into which you want objects imported.
- TRANSFORM
  The Oracle Data Pump Import command-line mode `TRANSFORM` parameter enables you to alter object creation DDL for objects being imported.

- TRANSPORT_DATAFILES
  The Oracle Data Pump Import command-line mode `TRANSPORT_DATAFILES` parameter specifies a list of data files that are imported into the target database when `TRANSPORTABLE=ALWAYS` is set during the export.
- TRANSPORT_FULL_CHECK
  The Oracle Data Pump Import command-line mode `TRANSPORT_FULL_CHECK` parameter specifies whether to verify that the specified transportable tablespace set is being referenced by objects in other tablespaces.
- TRANSPORT_TABLESPACES
  The Oracle Data Pump Import command-line mode `TRANSPORT_TABLESPACES` parameter specifies that you want to perform an import in transportable-tablespace mode over a database link.
- TRANSPORTABLE
  The optional Oracle Data Pump Import command-line mode `TRANSPORTABLE` parameter specifies either that transportable tables are imported with `KEEP_READ_ONLY`, or `NO_BITMAP_REBUILD`.
- VERSION
  The Oracle Data Pump Import command-line mode `VERSION` parameter specifies the version of database objects that you want to import.
- VIEWS_AS_TABLES (Network Import)
  The Oracle Data Pump Import command-line mode `VIEWS_AS_TABLES` (Network Import) parameter specifies that you want one or more views to be imported as tables.

Related Topics

- PARFILE
  The Oracle Data Pump Import command-line mode `PARFILE` parameter specifies the name of an import parameter file.
- Understanding Dump, Log, and SQL File Default Locations
  Oracle Data Pump is server-based, rather than client-based. Dump files, log files, and SQL files are accessed relative to server-based directory paths.
- Examples of Using Oracle Data Pump Import
  You can use these common scenario examples to learn how you can use Oracle Data Pump Import to move your data.
- Syntax Diagrams for Oracle Data Pump Import
  You can use syntax diagrams to understand the valid SQL syntax for Oracle Data Pump Import.

3.4.1 About Import Command-Line Mode

Learn how to use Oracle Data Pump Import parameters in command-line mode, including case sensitivity, quotation marks, escape characters, and information about how to use examples.

Before using Oracle Data Pump import parameters, read the following sections:

- Specifying Import Parameter
- Use of Quotation Marks On the Data Pump Command Line

Many of the descriptions include an example of how to use the parameter. For background information on setting up the necessary environment to run the examples, see:

- Using the Import Parameter Examples

Specifying Import Parameters

For parameters that can have multiple values specified, the values can be separated by commas or by spaces. For example, you could specify `TABLES=employees,jobs` or `TABLES=employees jobs`.

For every parameter you enter, you must enter an equal sign (=) and a value. Data Pump has no other way of knowing that the previous parameter specification is complete and a new parameter specification is beginning. For example, in the following command line, even though `NOLOGFILE` is a valid parameter, it would be interpreted as another dump file name for the `DUMPFILE` parameter:

```
impdp DIRECTORY=dpumpdir DUMPFILE=test.dmp NOLOGFILE TABLES=employees
```

This would result in two dump files being created, `test.dmp` and `nologfile.dmp`.

To avoid this, specify either `NOLOGFILE=YES` or `NOLOGFILE=NO`.

Case Sensitivity When Specifying Parameter Values

For tablespace names, schema names, table names, and so on that you enter as parameter values, Oracle Data Pump by default changes values entered as lowercase or mixed-case into uppercase. For example, if you enter `TABLE=hr.employees`, then it is changed to `TABLE=HR.EMPLOYEES`. To maintain case, you must enclose the value within quotation marks. For example, `TABLE="hr.employees"` would preserve the table name in all lower case. The name you enter must exactly match the name stored in the database.

Use of Quotation Marks On the Data Pump Command Line

Some operating systems treat quotation marks as special characters and will therefore not pass them to an application unless they are preceded by an escape character, such as the backslash (\). This is true both on the command line and within parameter files. Some operating systems may require an additional set of single or double quotation marks on the command line around the entire parameter value containing the special characters.

The following examples are provided to illustrate these concepts. Be aware that they may not apply to your particular operating system and that this documentation cannot anticipate the operating environments unique to each user.

Suppose you specify the `TABLES` parameter in a parameter file, as follows:

```
TABLES = \"MixedCaseTableName\"
```

If you were to specify that on the command line, then some operating systems would require that it be surrounded by single quotation marks, as follows:

```
TABLES = '\"MixedCaseTableName\"'
```

To avoid having to supply additional quotation marks on the command line, Oracle recommends the use of parameter files. Also, note that if you use a parameter file and the parameter value being specified does not have quotation marks as the first character in the string (for example, `TABLES=scott."EmP"`), then the use of escape characters may not be necessary on some systems.

Using the Import Parameter Examples

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

- After you enter the username and parameters as shown in the example, Import is started and you are prompted for a password. You must supply a password before a database connection is made.
- 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.
- 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` user for these directory objects.
- Some of the examples require the `DATAPUMP_EXP_FULL_DATABASE` and `DATAPUMP_IMP_FULL_DATABASE` roles. The examples assume that the `hr` user has been granted these roles.

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

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

> **See Also:**
>
> *Oracle Database Sample Schemas*
>
> Your Oracle operating system-specific documentation for information about how special and reserved characters are handled on your system.

3.4.2 ABORT_STEP

The Oracle Data Pump Import command-line mode `ABORT_STEP` parameter stops the job after it is initialized. Stopping the job enables the Data Pump control job table to be queried before any data is imported.

Default

Null

Purpose

Stops the job after it is initialized. Stopping the job enables the Data Pump control job table to be queried before any data is imported.

Syntax and Description

```
ABORT_STEP=[n | -1]
```

The possible values correspond to a process order number in the Data Pump control job table. The result of using each number is as follows:

- *n*: If the value is zero or greater, then the import operation is started. The job is stopped at the object that is stored in the Data Pump control job table with the corresponding process order number.

- -1 The import job uses a NETWORK_LINK: Abort the job after setting it up but before importing any objects.
- -1 The import job does not use NETWORK_LINK: Abort the job after loading the master table and applying filters.

Restrictions

- None

Example

```
> impdp hr SCHEMAS=hr DIRECTORY=dpump_dir1 LOGFILE=schemas.log
DUMPFILE=expdat.dmp ABORT_STEP=-1
```

3.4.3 ACCESS_METHOD

The Oracle Data Pump Import command-line mode ACCESS_METHOD parameter instructs Import to use a particular method to load data

Default

AUTOMATIC

Purpose

Instructs Import to use a particular method to load data.

Syntax and Description

```
ACCESS_METHOD=[AUTOMATIC | DIRECT_PATH | EXTERNAL_TABLE | CONVENTIONAL |
INSERT_AS_SELECT]
```

The ACCESS_METHOD parameter is provided so that you can try an alternative method if the default method does not work for some reason. If the data for a table cannot be loaded with the specified access method, then the data displays an error for the table and continues with the next work item.

The available options are:

- AUTOMATIC: This access method is the default. Data Pump determines the best way to load data for each table. Oracle recommends that you use AUTOMATIC whenever possible, because it enables Data Pump to automatically select the most efficient method.
- DIRECT_PATH: Data Pump uses direct path load for every table.
- EXTERNAL_TABLE: Data Pump creates an external table over the data stored in the dump file, and uses a SQL INSERT AS SELECT statement to load the data into the table. Data Pump applies the APPEND hint to the INSERT statement.
- CONVENTIONAL: Data Pump creates an external table over the data stored in the dump file and reads rows from the external table one at a time. Every time it reads a row, Data Pump executes an insert statement that loads that row into the target table. This method takes a long time to load data, but it is the only way to load data that cannot be loaded by direct path and external tables.
- INSERT_AS_SELECT: Data Pump loads tables by executing a SQL INSERT AS SELECT statement that selects data from the remote database and inserts it into the target table.

This option is available only for network mode imports. It is used to disable use of DIRECT_PATH when data is moved over the network.

Restrictions

- The valid options for network mode import are AUTOMATIC, DIRECT_PATH and INSERT_AS_SELECT .
- The only valid options when importing from a dump file are AUTOMATIC, DIRECT_PATH, EXTERNAL_TABLE and CONVENTIONAL.
- To use the ACCESS_METHOD parameter with network imports, you must be using Oracle Database 12c Release 2 (12.2.0.1) or later
- The ACCESS_METHOD parameter for Oracle Data Pump Import is not valid for transportable tablespace jobs.

Example

The following example enables Oracle Data Pump to load data for multiple partitions of the pre-existing table SALES at the same time.

```
impdp hr SCHEMAS=hr DIRECTORY=dpump_dir1 LOGFILE=schemas.log
DUMPFILE=expdat.dmp ACCESS_METHOD=CONVENTIONAL
```

3.4.4 ATTACH

The Oracle Data Pump Import command-line mode ATTACH parameter attaches the client session to an existing import job and automatically places you in interactive-command mode.

Default

If there is only one running job, then the current job in user's schema.

Purpose

This command 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. To do this, you must have the DATAPUMP_IMP_FULL_DATABASE role.

A *job_name* does not have to be specified if only one running job is associated with your schema, and if the job is active. If the job you are attaching to is stopped, then you must supply the job name. To see a list of Oracle 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, the only other Data Pump parameter you can specify on the command line is ENCRYPTION_PASSWORD.

- If the job you are attaching to was initially started using an encryption password, then when you attach to the job, you must again enter the `ENCRYPTION_PASSWORD` parameter on the command line to re-specify that password.
- You cannot attach to a job in another schema unless it is already running.
- If the dump file set or master table for the job have been deleted, then the attach operation fails.
- Altering the master table in any way can lead to unpredictable results.

Example

The following is an example of using the `ATTACH` parameter.

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

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

Related Topics

- Commands Available in Oracle Data Pump Import Interactive-Command Mode
  In interactive-command mode, the current job continues running, but logging to the terminal is suspended, and the Import prompt (`Import>`) is displayed.

3.4.5 CLUSTER

The Oracle Data Pump Import command-line mode `CLUSTER` parameter determines whether Data Pump can use Oracle Real Application Clusters (Oracle RAC) resources, and start workers on other Oracle RAC instances.

Default

`YES`

Purpose

Determines whether Oracle Data Pump can use Oracle Real Application Clusters (Oracle RAC) resources, and start workers on other Oracle RAC instances.

Syntax and Description

```
CLUSTER=[YES | NO]
```

To force Data Pump Import to use only the instance where the job is started and to replicate pre-Oracle Database 11g Release 2 (11.2) behavior, specify `CLUSTER=NO`.

To specify a specific, existing service, and constrain worker processes to run only on instances defined for that service, use the `SERVICE_NAME` parameter with the `CLUSTER=YES` parameter.

Using the `CLUSTER` parameter can affect performance, because there is some additional overhead in distributing the import job across Oracle RAC instances. For small jobs, it can be better to specify `CLUSTER=NO`, so that the job is constrained to run on the instance where it is started. Jobs that obtain the most performance benefits from using the `CLUSTER` parameter are those involving large amounts of data.

Example

```
> impdp hr DIRECTORY=dpump_dir1 SCHEMAS=hr CLUSTER=NO PARALLEL=3
NETWORK_LINK=dbs1
```

This example performs a schema-mode import of the `hr` schema. Because `CLUSTER=NO` is used, the job uses only the instance where it is started. Up to 3 parallel processes can be used. The `NETWORK_LINK` value of `dbs1` would be replaced with the name of the source database from which you were importing data. (Note that there is no dump file generated, because this is a network import.)

In this example, the `NETWORK_LINK` parameter is only used as part of the example. It is not required when using the `CLUSTER` parameter.

Related Topics

- SERVICE_NAME
  The Oracle Data Pump Import command-line mode `SERVICE_NAME` parameter specifies a service name that you want to use in conjunction with the `CLUSTER` parameter.
- Understanding How to Use Oracle Data Pump with Oracle RAC
  Using Oracle Data Pump in an Oracle Real Application Clusters (Oracle RAC) environment requires you to perform a few checks to ensure that you are making cluster member nodes available.

3.4.6 CONTENT

The Oracle Data Pump Import command-line mode `CONTENT` parameter enables you to filter what is loaded during the import operation.

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. It does not load table row data. Be aware that if you specify `CONTENT=METADATA_ONLY`, then any index or table statistics imported from the dump file are locked after the import operation is complete.

Restrictions

- The `CONTENT=METADATA_ONLY` parameter and value cannot be used in conjunction with the `TRANSPORT_TABLESPACES` (transportable-tablespace mode) parameter or the `QUERY` parameter.
- The `CONTENT=ALL` and `CONTENT=DATA_ONLY` parameter and values cannot be used in conjunction with the `SQLFILE` parameter.

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.

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

This command runs a full import that loads only the metadata in the expfull.dmp dump file. It runs a full import, because a full import is the default for file-based imports in which no import mode is specified.

Related Topics

- FULL
  The Export command-line FULL parameter specifies that you want to perform a full database mode export

3.4.7 CREDENTIAL

The Oracle Data Pump Import command-line mode CREDENTIAL parameter specifies the credential object name owned by the database user that Import uses to process files in the dump file set imported into cloud storage.

Default

none.

Purpose

Specifies the credential object name owned by the database user that Import uses to process files in the dump file set imported into Oracle Cloud Infrastructure cloud storage.

Syntax and Description

CREDENTIAL=*credential_object_name*
The import operation reads and processes files in the dump file set stored in the cloud the same as files stored on local file systems.

If the CREDENTIAL parameter is specified, then the value for the DUMPFILE parameter is a list of comma-delimited strings that Import treats as URI values. Starting with Oracle Database 19c, the URI files in the dump file set can include templates that contain the Data Pump substitution variables, such as %U, %L, and so on. For example: *urlpath*exp%U.dmp.

> **Note:**
>
> Substitution variables are only allowed in the filename portion of the URI.

The DUMPFILE parameter enables you to specify an optional directory object, using the format *directory_object _name*:*file_name*. However, if you specify the CREDENTIAL parameter, then Import does not attempt to look for a directory object name in the strings passed for DUMPFILE. Instead, the strings are treated as URI strings.

The DIRECTORY parameter is still used as the location of log files and SQL files. Also, you can still specify directory object names as part of the file names for LOGFILE and SQLFILE.

Oracle Data Pump import is no longer constrained to using the default_credential value in Oracle Autonomous Database. The Import CREDENTIAL parameter now accepts any Oracle Cloud Infrastructure (OCI) Object Storage credential created in the Oracle Autonomous Database that is added to the database using the DBMS_CLOUD.CREATE_CREDENTIAL() procedure. Oracle Data Pump validates if the credential exists, and if the user has access to read the credential. Any errors are returned back to the impdp client.

Restrictions

The credential parameter cannot be an OCI resource principal, Azure service principal, Amazon Resource Name (ARN), or a Google service account.

Example: Using the Import CREDENTIAL Parameter

The following is an example of using the Import CREDENTIAL parameter. You can create the dump files used in this example by running the example provided for the Export DUMPFILE parameter, and then uploading the dump files into your cloud storage.

```
> impdp hr/your_password DIRECTORY=dpump_dir1
CREDENTIAL=user_accessible_credential
      DUMPFILE='https://objectstorage.example.com/exp1.dmp',
  'https://objectstorage.example.com/exp201.dmp',
     'https://objectstorage.example.com/exp202.dmp'
```

The import job looks in the specified cloud storage for the dump files. The log file is written to the path associated with the directory object, dpump_dir1, that was specified with the DIRECTORY parameter.

Example: Specifying a User-Defined Credential

The following example creates a new user-defined credential in the Oracle Autonomous Database, and uses the same credential in an impdp command:

```
BEGIN
 DBMS_CLOUD.CREATE_CREDENTIAL(
 credential_name => 'MY_CRED_NAME',
 username => 'adwc_user@example.com',
 password => 'Auth token' ); END;

> impdp admin/password@ADWC1_high
 directory=data_pump_dir
 credential=MY_cred_name …
```

3.4.8 DATA_OPTIONS

The Oracle Data Pump Import command-line mode DATA_OPTIONS parameter designates how you want certain types of data to be handled during import operations.

Default

There is no default. If this parameter is not used, then the special data handling options it provides simply do not take effect.

Purpose

The DATA_OPTIONS parameter designates how you want certain types of data to be handled during import operations.

Syntax and Description

```
DATA_OPTIONS = [DISABLE_APPEND_HINT | SKIP_CONSTRAINT_ERRORS |
REJECT_ROWS_WITH_REPL_CHAR | GROUP_PARTITION_TABLE_DATA |
TRUST_EXISTING_TABLE_PARTITIONS |
VALIDATE_TABLE_DATA | ENABLE_NETWORK_COMPRESSION |
CONTINUE_LOAD_ON_FORMAT_ERROR]
```

- DISABLE_APPEND_HINT: Specifies that you do not want the import operation to use the APPEND hint while loading the data object. Disabling the APPEND hint can be useful to address duplicate data. For example, you can use DISABLE_APPEND_HINT when there is a small set of data objects to load that exists already in the database, and some other application can be concurrently accessing one or more of the data objects.

  DISABLE_APPEND_HINT: Changes the default behavior, so that the APPEND hint is not used for loading data objects. When not set, the default is to use the APPEND hint for loading data objects.

- GROUP_PARTITION_TABLE_DATA: Tells Oracle Data Pump to import the table data in all partitions of a table as one operation. The default behavior is to import each table partition as a separate operation. If you know that the data for a partition will not move, then choose this parameter to accelerate the import of partitioned table data. There are cases when Oracle Data Pump attempts to load only one partition at a time. It does this when the table already exists, or when there is a risk that the data for one partition might be moved to another partition.

- REJECT_ROWS_WITH_REPL_CHAR: Specifies that you want the import operation to reject any rows that experience data loss because the default replacement character was used during character set conversion.

  If REJECT_ROWS_WITH_REPL_CHAR is not set, then the default behavior is to load the converted rows with replacement characters.

- SKIP_CONSTRAINT_ERRORS: Affects how non-deferred constraint violations are handled while a data object (table, partition, or subpartition) is being loaded.

  If deferred constraint violations are encountered, then SKIP_CONSTRAINT_ERRORS has no effect on the load. Deferred constraint violations always cause the entire load to be rolled back.

  The SKIP_CONSTRAINT_ERRORS option specifies that you want the import operation to proceed even if non-deferred constraint violations are encountered. It logs any rows that cause non-deferred constraint violations, but does not stop the load for the data object experiencing the violation.

  SKIP_CONSTRAINT_ERRORS: Prevents roll back of the entire data object when non-deferred constraint violations are encountered.

  If SKIP_CONSTRAINT_ERRORS is not set, then the default behavior is to roll back the entire load of the data object on which non-deferred constraint violations are encountered.

- TRUST_EXISTING_TABLE_PARTITIONS: Tells Data Pump to load partition data in parallel into existing tables.

Use this option when you are using Data Pump to create the table from the definition in the export database before the table data import is started. Typically, you use this parameter as part of a migration when the metadata is static, and you can move it before the databases are taken off line to migrate the data. Moving the metadata separately minimizes downtime. If you use this option, and if other attributes of the database are the same (for example, character set), then the data from the export database goes to the same partitions in the import database.

You can create the table outside of Oracle Data Pump. However, if you create tables as a separate option from using Oracle Data Pump, then the partition attributes and partition names must be identical to the export database.

> **Note:**
>
> This option can be used for import no matter the source version of the export.

- `VALIDATE_TABLE_DATA`: Directs Oracle Data Pump to validate the number and date data types in table data columns.

  If the import encounters invalid data, then an `ORA-39376` error is written to the `.log` file. The error text includes the column name. The default is to do no validation. Use this option if the source of the Oracle Data Pump dump file is not trusted.

- `ENABLE_NETWORK_COMPRESSION`: Used for network imports in which the Oracle Data Pump `ACCESS_METHOD` parameter is set to `DIRECT_PATH` to load remote table data.

  When `ENABLE_NETWORK_COMPRESSION` is specified, Oracle Data Pump compresses data on the remote node before it is sent over the network to the target database, where it is decompressed. This option is useful if the network connection between the remote and local database is slow, because it reduces the amount of data sent over the network.

  Setting `ACCESS_METHOD=AUTOMATIC` enables Oracle Data Pump to set `ENABLE_NETWORK_COMPRESSION` automatically during the import if Oracle Data Pump uses `DIRECT_PATH` for a network import.

  The `ENABLE_NETWORK_COMPRESSION` option is ignored if Oracle Data Pump is importing data from a dump file, if the remote data base is earlier than Oracle Database 12c Release 2 (12.2), or if an `INSERT_AS_SELECT` statement is being used to load data from the remote database.

- `CONTINUE_LOAD_ON_FORMAT_ERROR`: Directs Oracle Data Pump to skip forward to the start of the next granule when a stream format error is encountered while loading table data.

  Stream format errors typically are the result of corrupt dump files. If Oracle Data Pump encounters a stream format error, and the original export database is not available to export the table data again, then you can use `CONTINUE_LOAD_ON_FORMAT_ERROR`. If Oracle Data Pump skips over data, then not all data from the source database is imported, which potentially skips hundreds or thousands of rows.

Restrictions

- If you use `DISABLE_APPEND_HINT`, then it can take longer for data objects to load.
- If you use `SKIP_CONSTRAINT_ERRORS`, and if a data object has unique indexes or constraints defined on it at the time of the load, then the `APPEND` hint is not used for loading that data object. Therefore, loading such data objects can take longer when the `SKIP_CONSTRAINT_ERRORS` option is used.

- Even if SKIP_CONSTRAINT_ERRORS is specified, it is not used unless a data object is being loaded using the external table access method.

Example

This example shows a data-only table mode import with SKIP_CONSTRAINT_ERRORS enabled:

```
> impdp hr TABLES=employees CONTENT=DATA_ONLY
DUMPFILE=dpump_dir1:table.dmp DATA_OPTIONS=skip_constraint_errors
```

If any non-deferred constraint violations are encountered during this import operation, then they are logged. The import continues on to completion.

3.4.9 DIRECTORY

The Oracle Data Pump Import command-line mode DIRECTORY parameter specifies the default location in which the import job can find the dump file set, and create log and SQL files.

Default

DATA_PUMP_DIR

Purpose

Specifies the default 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. It is not the file path of an actual directory. Privileged users have access to a default directory object named DATA_PUMP_DIR. The definition of the DATA_PUMP_DIR directory can be changed by Oracle during upgrades, or when patches are applied.

Users with access to the default DATA_PUMP_DIR directory object do not need to use the DIRECTORY parameter.

A directory object specified on the DUMPFILE, LOGFILE, or SQLFILE parameter overrides any directory object that you specify for the DIRECTORY parameter. You must have Read access to the directory used for the dump file set. You must have 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.

```
> impdp 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. Refer to *Oracle Database SQL Language Reference* for more information about the CREATE DIRECTORY command.

Related Topics

- Understanding Dump, Log, and SQL File Default Locations
  Oracle Data Pump is server-based, rather than client-based. Dump files, log files, and SQL files are accessed relative to server-based directory paths.
- Understanding How to Use Oracle Data Pump with Oracle RAC
  Using Oracle Data Pump in an Oracle Real Application Clusters (Oracle RAC) environment requires you to perform a few checks to ensure that you are making cluster member nodes available.
- CREATE DIRECTORY in *Oracle Database SQL Language Reference*

3.4.10 DUMPFILE

The Oracle Data Pump Import command-line mode DUMPFILE parameter specifies the names, and optionally, the directory objects of the dump file set that Export created.

Default

expdat.dmp

Purpose

Specifies the names, and, if you choose, the directory objects or default credential of the dump file set that was created by Export.

Syntax and Description

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

Or

```
DUMPFILE=[DEFAULT_CREDENTIAL:]URI_file [, ...]
```

The *directory_object* is optional if one is already established by the DIRECTORY parameter. If you do supply a value, then it must be a directory object that already exists, and to which you have access. A database directory object that is specified as part of the DUMPFILE parameter overrides a value specified by the DIRECTORY parameter.

The *file_name* is the name of a file in the dump file set. The file names can also be templates that contain the substitution variable %U. The Import process checks each file that matches the template to locate all files that are part of the dump file set, until no match is found. Sufficient information is contained within the files for Import to locate the entire set, provided that the file specifications defined in the DUMPFILE parameter encompass the entire set. The files are not required to have the same names, locations, or order used at export time.

The possible substitution variables are described in the following table.

Substitution Variable	Description
%U	If %U is used, then the%U expands to a 2-digit incrementing integer starting with 01.

Substitution Variable	Description
%l, %L	Specifies a system-generated unique file name. The file names can contain a substitution variable (%L), which implies that multiple files may be generated. The substitution variable is expanded in the resulting file names into a 2-digit, fixed-width, incrementing integer starting at 01 and ending at 99 which is the same as (%U). In addition, the substitution variable is expanded in the resulting file names into a 3-digit to 10-digit, variable-width, incrementing integers starting at 100 and ending at 2147483646. The width field is determined by the number of digits in the integer. For example if the current integer is 1, then exp%Laa%L.dmp resolves to the following sequence order exp01aa01.dmp exp02aa02.dmp The 2-digit increment continues increasing, up to 99. Then, the next file names substitute a 3-digit increment: exp100aa100.dmp exp101aa101.dmp The 3-digit increments continue up until 999. Then, the next file names substitute a 4-digit increment. The substitutions continue up to the largest number substitution allowed, which is 2147483646.

Restrictions

- Dump files created on Oracle Database 11g releases with the Oracle Data Pump parameter VERSION=12 can only be imported on Oracle Database 12c Release 1 (12.1) and later.

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.

```
> impdp 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 looks there for the file. It also looks in dpump_dir1 for dump files of the form exp2*nn*.dmp. The log file is written to dpump_dir1.

If you use the alternative DEFAULT_CREDENTIAL keyword syntax for the Import DUMPFILE parameter, then a default credential with user access must already exist. The import operation uses the default credential to read and process files in the dump file set that is stored in the cloud at the specified *URI_file* location.

The variable *URI_file* represents the name of a URI file in the dump file set. The file name cannot be the same as templates that contain the Data Pump substitution variables, such as %U, %L, and so on.

The DUMPFILE parameter DEFAULT_CREDENTIAL keyword syntax is mutually exclusive to the *directory_object* syntax. Only one form can be used in the same command line.

Example of Using the Import DUMPFILE with User-Defined Credentials

This example specifies the default location in which the import job can find the dump file set, and create log and SQL files, and specifies the credential object name owned by the database user that Import uses to process files in the dump file set that were previously imported into cloud storage.

```
> impdp admin/password@ADWC1_high
      directory=data_pump_dir
      credential=MY_cred_name …
```

Example of Using the Import DUMPFILE parameter with DEFAULT_CREDENTIAL Keywords.

You can create the dump files used in this example by running the example provided for the Export `DUMPFILE` parameter.

```
> impdp hr/your_password DIRECTORY=dpump_dir1
      DUMPFILE='DEFAULT_CREDENTIAL:https://objectstorage.example.com/
exp1.dmp',
  'DEFAULT_CREDENTIAL:https://objectstorage.example.com/exp201.dmp',
     'DEFAULT_CREDENTIAL:https://objectstorage.example.com/exp202.dmp'
```

The import job looks in the specified *URI_file* location for the dump files using the default credential which has already been setup for the user. The log file is written to the path associated with the directory object, *dpump_dir1* that was specified with the `DIRECTORY` parameter.

Example of Using the Import DUMPFILE parameter with User-Defined Credentials

This example specifies the default location in which the import job can find the dump file set, and create log and SQL files, and specifies the credential object name owned by the database user that Import uses to process files in the dump file set that were previously imported into cloud storage.

```
> impdp impdp admin/password@ADWC1_high  DIRECTORY=data_pump_dir
      DUMPFILE='MY_cred_name:https://objectstorage.example.com/exp1.dmp',
  'MY_cred_name:https://objectstorage.example.com/exp201.dmp',
     'MY_cred_name:https://objectstorage.example.com/exp202.dmp'
```

Related Topics

- DUMPFILE
- File Allocation with Oracle Data Pump
- Performing a Data-Only Table-Mode Import

3.4.11 ENABLE_SECURE_ROLES

The Oracle Data Pump Import command-line utility `ENABLE_SECURE_ROLES` parameter prevents inadvertent use of protected roles during exports.

Default

In Oracle Database 19c and later releases, the default value is `NO`.

Purpose

Some Oracle roles require authorization. If you need to use these roles with Oracle Data Pump imports, then you must explicitly enable them by setting the `ENABLE_SECURE_ROLES` parameter to `YES`.

Syntax

```
ENABLE_SECURE_ROLES=[NO|YES]
```

- `NO` Disables Oracle roles that require authorization.
- `YES` Enables Oracle roles that require authorization.

Example

```
impdp hr SCHEMAS=hr DIRECTORY=dpump_dir1 DUMPFILE=dpump_dir2:imp1.dmp,
 imp2%U.dmp ENABLE_SECURE_ROLES=YES
```

3.4.12 ENCRYPTION_PASSWORD

The Oracle Data Pump Import command-line mode `ENCRYPTION_PASSWORD` parameter specifies a password for accessing encrypted column data in the dump file set. This prevents unauthorized access to an encrypted dump file set.

Default

There is no default; the value is user-supplied.

Purpose

Specifies a password for accessing encrypted column data in the dump file set. This prevents unauthorized access to an encrypted dump file set.

This parameter is also required for the transport of keys associated with encrypted tablespaces, and tables with encrypted columns during a full transportable export or import operation.

The password that you enter is echoed to the screen. If you do not want the password shown on the screen as you enter it, then use the `ENCRYPTION_PWD_PROMPT` parameter.

Syntax and Description

```
ENCRYPTION_PASSWORD = password
```

This parameter is required on an import operation if an encryption password was specified on the export operation. The password that is specified must be the same one that was specified on the export operation.

Restrictions

- The export operation using this parameter requires the Enterprise Edition release of Oracle Database 11g or later, It is not possible to use `ENCRYPTION_PASSWORD` for an export from Standard Edition, so you cannot use this this parameter for a migration from Standard Edition to Enterprise Edition. You can use this parameter for migrations from Enterprise Edition to Standard Edition.
- Data Pump encryption features require that you have the Oracle Advanced Security option enabled. Refer to *Oracle Database Licensing Information* for information about licensing requirements for the Oracle Advanced Security option.
- The `ENCRYPTION_PASSWORD` parameter is not valid if the dump file set was created using the transparent mode of encryption.
- The `ENCRYPTION_PASSWORD` parameter is required for network-based full transportable imports where the source database has encrypted tablespaces or tables with encrypted columns.
- Encryption attributes for all columns must match between the exported table definition and the target table. For example, suppose you have a table, `EMP`, and one of its columns is named `EMPNO`. Both of the following situations would result in an error because the encryption attribute for the `EMP` column in the source table would not match the encryption attribute for the `EMP` column in the target table:
  - The `EMP` table is exported with the `EMPNO` column being encrypted, but before importing the table you remove the encryption attribute from the `EMPNO` column.
  - The `EMP` table is exported without the `EMPNO` column being encrypted, but before importing the table you enable encryption on the `EMPNO` column.

Example

In the following example, the encryption password, `123456`, must be specified, because it was specified when the `dpcd2be1.dmp` dump file was created.

```
> impdp hr TABLES=employee_s_encrypt DIRECTORY=dpump_dir
  DUMPFILE=dpcd2be1.dmp ENCRYPTION_PASSWORD=123456
```

During the import operation, any columns in the `employee_s_encrypt` table encrypted during the export operation are decrypted before being imported.

Related Topics

- *Oracle Database Licensing Information User Manual*

3.4.13 ENCRYPTION_PWD_PROMPT

The Oracle Data Pump Import command-line mode `ENCRYPTION_PWD_PROMPT` parameter specifies whether Data Pump should prompt you for the encryption password.

Default

`NO`

Purpose

Specifies whether Oracle Data Pump should prompt you for the encryption password.

Syntax and Description

```
ENCRYPTION_PWD_PROMPT=[YES | NO]
```

Specify `ENCRYPTION_PWD_PROMPT=YES` on the command line to instruct Oracle Data Pump to prompt you for the encryption password. If you do not specify the value to `YES`, then you must enter the encryption password on the command line with the `ENCRYPTION_PASSWORD` parameter. The advantage to setting the parameter to `YES` is that the encryption password is not echoed to the screen when it is entered at the prompt. By contrast, if you enter the password on the command line using the `ENCRYPTION_PASSWORD` parameter, then the password appears in plain text.

The encryption password that you enter at the prompt is subject to the same criteria described for the `ENCRYPTION_PASSWORD` parameter.

If you specify an encryption password on the export operation, then you must also supply it on the import operation.

Restrictions

Concurrent use of the `ENCRYPTION_PWD_PROMPT` and `ENCRYPTION_PASSWORD` parameters is prohibited.

Example

The following example shows Oracle Data Pump first prompting for the user password, and then for the encryption password.

```
> impdp hr DIRECTORY=dpump_dir1 DUMPFILE=hr.dmp ENCRYPTION_PWD_PROMPT=YES
.
.
.
Copyright (c) 1982, 2017, Oracle and/or its affiliates.  All rights reserved.

Password:

Connected to: Oracle Database 18c Enterprise Edition Release 18.0.0.0.0 -
Development
Version 18.1.0.0.0

Encryption Password:

Master table "HR"."SYS_IMPORT_FULL_01" successfully loaded/unloaded
Starting "HR"."SYS_IMPORT_FULL_01":  hr/******** directory=dpump_dir1
dumpfile=hr.dmp encryption_pwd_prompt=Y
.
.
.
```

3.4.14 ESTIMATE

The Oracle Data Pump Import command-line mode `ESTIMATE` parameter instructs the source system in a network import operation to estimate how much data is generated during the import.

Default

`STATISTICS`

Purpose

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

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 source 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. (Table analysis can be done with either the SQL `ANALYZE` statement or the `DBMS_STATS` PL/SQL package.)

You can use the estimate that is generated to determine a percentage of the import job that is completed throughout the import.

Restrictions

- The Import `ESTIMATE` parameter is valid only if the `NETWORK_LINK` parameter is also specified.
- 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.
- The estimate may be inaccurate if either the `QUERY` or `REMAP_DATA` parameter is used.

Example

In the following syntax example, you replace the variable *`source_database_link`* with the name of a valid link to the source database.

```
> impdp 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.

3.4.15 EXCLUDE

The Oracle Data Pump Import command-line mode `EXCLUDE` parameter enables you to filter the metadata that is imported by specifying objects and object types to exclude from the import job.

Default

There is no default

Purpose

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

Syntax and Description

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

The *object_type* specifies the type of object to be excluded. To see a list of valid values for *object_type*, query the following views: `DATABASE_EXPORT_OBJECTS` for full mode, `SCHEMA_EXPORT_OBJECTS` for schema mode, and `TABLE_EXPORT_OBJECTS` for table and tablespace mode. The values listed in the `OBJECT_PATH` column are the valid object types.

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, then 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`). It 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`.

The name that you supply for the *name_clause* must exactly match, including upper and lower casing, an existing object in the database. For example, if the *name_clause* you supply is for a table named `EMPLOYEES`, then there must be an existing table named `EMPLOYEES` using all upper case. If the *name_clause* were supplied as `Employees` or `employees` or any other variation, then the table would not be found.

More than one `EXCLUDE` statement can be specified.

Depending on your operating system, the use of quotation marks when you specify a value for this parameter may also require that you use escape characters. Oracle recommends that you place this parameter in a parameter file, which can reduce the number of escape characters that might otherwise be needed 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:

- 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 excludes all constraints, except for any constraints needed for successful table creation and loading.
- EXCLUDE=REF_CONSTRAINT excludes 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 command such as the following, where hr is the schema name of the user you want to exclude.

```
impdp FULL=YES DUMPFILE=expfull.dmp EXCLUDE=SCHEMA:"='HR'"
```

Note that in this example, the FULL import mode is specified. If no mode is specified, then SCHEMAS is used, because that is the default mode. However, with this example, if you do not specify FULL, and instead use SCHEMAS, followed by the EXCLUDE=SCHEMA argument, then that causes an error, because in that case you are indicating that you want the schema both to be imported and excluded at the same time.

If you try to exclude a user by using a statement such as EXCLUDE=USER:"= 'HR'", then only CREATE USER hr DDL statements are excluded, which can return unexpected results.

Restrictions

- The EXCLUDE and INCLUDE parameters are mutually exclusive.

Example

Assume the following is in a parameter file, exclude.par, being used by a DBA or some other user with the DATAPUMP_IMP_FULL_DATABASE role. (To run the example, you must first create this file.)

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

You then issue the following command:

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

You can create the expfull.dmp dump file used in this command by running the example provided for the Export FULL parameter. in the FULL reference topic. All data from the expfull.dmp dump file is loaded, except for functions, procedures, packages, and indexes whose names start with emp.

Related Topics

- FULL
- Metadata Filters
- Filtering During Import Operations
- About Import Command-Line Mode

3.4.16 FLASHBACK_SCN

The Oracle Data Pump Import command-line mode `FLASHBACK_SCN` specifies the system change number (SCN) that Import uses to enable the Flashback utility.

Default

There is no default

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 up to the specified *scn_number*.

Starting with Oracle Database 12c Release 2 (12.2), the SCN value can be a big SCN (8 bytes). See the following restrictions for more information about using big SCNs.

Restrictions

- The `FLASHBACK_SCN` parameter is valid only when the `NETWORK_LINK` parameter is also specified.
- The `FLASHBACK_SCN` parameter pertains only to the Flashback Query capability of Oracle Database. It is not applicable to Flashback Database, Flashback Drop, or Flashback Data Archive.
- `FLASHBACK_SCN` and `FLASHBACK_TIME` are mutually exclusive.
- You cannot specify a big SCN for a network export or network import from a version that does not support big SCNs.

Example

The following is a syntax example of using the `FLASHBACK_SCN` parameter.

```
> impdp hr DIRECTORY=dpump_dir1 FLASHBACK_SCN=123456
NETWORK_LINK=source_database_link
```

When using this command, replace the variables *123456* and *source_database_link* with the SCN and the name of a source database from which you are importing data.

> **Note:**
>
> If you are on a logical standby system, then 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.

Related Topics

- Logical Standby Databases in *Oracle Data Guard Concepts and Administration*

3.4.17 FLASHBACK_TIME

The Oracle Data Pump Import command-line mode `FLASHBACK_TIME` parameter specifies the system change number (SCN) that Import uses to enable the Flashback utility.

Default

There is no default

Purpose

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

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 up to this SCN. Because the `TO_TIMESTAMP` value is enclosed in quotation marks, it would be best to put this parameter in a parameter file.

> **Note:**
>
> If you are on a logical standby system, then 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.
- The `FLASHBACK_TIME` parameter pertains only to the flashback query capability of Oracle Database. It is not applicable to Flashback Database, Flashback Drop, or Flashback Data Archive.
- `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, suppose you have a parameter file, `flashback_imp.par`, that contains the following:

```
FLASHBACK_TIME="TO_TIMESTAMP('27-10-2012 13:40:00', 'DD-MM-YYYY HH24:MI:SS')"
```

You could then issue the following command:

```
> impdp hr DIRECTORY=dpump_dir1 PARFILE=flashback_imp.par
NETWORK_LINK=source_database_link
```

The import operation will be performed with data that is consistent with the SCN that most closely matches the specified time.

> **Note:**
>
> See *Oracle Database Development Guide* for information about using flashback

Related Topics

- About Import Command-Line Mode
  Learn how to use Oracle Data Pump Import parameters in command-line mode, including case sensitivity, quotation marks, escape characters, and information about how to use examples.
- Logical Standby Databases in *Oracle Data Guard Concepts and Administration*
- Using Oracle Flashback Query (SELECT AS OF) in *Oracle Database Development Guide*

3.4.18 FULL

The Oracle Data Pump Import command-line mode `FULL` parameter specifies that you want to perform a full database import.

Default

`YES`

Purpose

Specifies that you want to perform a full database import.

Syntax and Description

```
FULL=YES
```

A value of `FULL=YES` indicates that all data and metadata from the source is imported. The source can be a dump file set for a file-based import or it can be another database, specified with the `NETWORK_LINK` parameter, for a network import.

If you are importing from a file and do not have the `DATAPUMP_IMP_FULL_DATABASE` role, then only schemas that map to your own schema are imported.

If the `NETWORK_LINK` parameter is used and the user executing the import job has the `DATAPUMP_IMP_FULL_DATABASE` role on the target database, then that user must also have the `DATAPUMP_EXP_FULL_DATABASE` role on the source database.

Filtering can restrict what is imported using this import mode.

FULL is the default mode, and does not need to be specified on the command line when you are performing a file-based import, but if you are performing a network-based full import then you must specify FULL=Y on the command line.

You can use the transportable option during a full-mode import to perform a full transportable import.

Restrictions

- The Automatic Workload Repository (AWR) is not moved in a full database export and import operation. (See *Oracle Database Performance Tuning Guide* for information about using Data Pump to move AWR snapshots.)
- The XDB repository is not moved in a full database export and import operation. User created XML schemas are moved.
- If the target is Oracle Database 12c Release 1 (12.1.0.1) or later, and the source is Oracle Database 11g Release 2 (11.2.0.3) or later, then Full imports performed over a network link require that you set VERSION=12

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.

```
> impdp hr DUMPFILE=dpump_dir1:expfull.dmp FULL=YES
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.

Related Topics

- Transporting Automatic Workload Repository Data in *Oracle Database Performance Tuning Guide*
- Transporting Databases in *Oracle Database Administrator's Guide*
- FULL

3.4.19 HELP

The Oracle Data Pump Import command-line mode HELP parameter displays online help for the Import utility.

Default

NO

Purpose

Displays online help for the Import utility.

Syntax and Description

```
HELP=YES
```

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

Example

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

```
> impdp HELP = YES
```

3.4.20 INCLUDE

The Oracle Data Pump Import command-line mode INCLUDE parameter enables you to filter the metadata that is imported by specifying objects and object types for the current import mode.

Default

There is no default

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] [, ...]
```

The variable *object_type* in the syntax specifies the type of object that you want to include. To see a list of valid values for *object_type*, query the following views:

- Full mode: DATABASE_EXPORT_OBJECTS
- Schema mode: SCHEMA_EXPORT_OBJECTS
- Table and Tablespace mode: TABLE_EXPORT_OBJECTS

In the query result, the values listed in the OBJECT_PATH column are the valid object types. (See "Metadata Filters" for an example of how to perform such a query.)

Only object types in the source (and their dependents) that you explicitly specify in the INCLUDE statement are imported.

The variable *name_clause* in the syntax is optional. It enables you to perform 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). It must be separated from the object type with a colon, and enclosed in double quotation marks. You must use double quotation marks, because single quotation marks are required to delimit the name strings.

The name string that you supply for the *name_clause* must exactly match an existing object in the database, including upper and lower case. For example, if the *name_clause* that you supply is for a table named EMPLOYEES, then there must be an existing table named EMPLOYEES, using all upper case characters. If the *name_clause* is supplied as Employees, or employees, or uses any other variation from the existing table names string, then the table is not found.

You can specify more than one INCLUDE statement.

Depending on your operating system, when you specify a value for this parameter with the use of quotation marks, you can also be required to use escape characters. Oracle recommends that you place this parameter in a parameter file, which can reduce the number of escape characters that you otherwise must use in the command line..

To see a list of valid paths for use with the INCLUDE parameter, query the following views:

- Full mode: DATABASE_EXPORT_OBJECTS
- Schema mode: SCHEMA_EXPORT_OBJECTS
- Table and Tablespace mode: TABLE_EXPORT_OBJECTS

Restrictions

- The INCLUDE and EXCLUDE parameters are mutually exclusive.

Example

Assume the following is in a parameter file named imp_include.par. This parameter file is being used by a DBA or some other user that is granted the role DATAPUMP_IMP_FULL_DATABASE:

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

With the aid of this parameter file, you can then issue the following command:

```
> impdp system 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.

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 DATAPUMP_IMP_FULL_DATABASE role), the schema definition is not imported, because the USER object type was not specified in an INCLUDE statement.

Related Topics

- Metadata Filters
- About Import Command-Line Mode
- FULL

3.4.21 INDEX_THRESHOLD

The Oracle Data Pump Import command-line mode INDEX_THRESHOLD parameter sets a size threshold for creating large indexes with a degree of parallelism greater than 1. It is used in conjunction with the ONESTEP_INDEX parameter.

Default

150M

Purpose

Sets the index size threshold. Indexes of a size equal to and above the threshold can be created with a degree of parallelism (DOP) greater than 1 if the parameter `ONESTEP_INDEX=FALSE`.

Syntax and Description

```
INDEX_THRESHOLD=string value
```

Indexes on tables that are smaller than the threshold value will be created with a degree of parallelism (DOP) of 1. An index on a table equal to or larger than the threshold will incur some additional overhead to determine the optimal DOP for creating that index, and the index will be created with that DOP unless it is limited by the ' setting for the job itself, as discussed for the `ONESTEP_INDEX` parameter.

Indexes on tables that are smaller than the threshold value will be created with a DOP of 1. An index on a table equal to or larger than the threshold will incur some additional overhead to determine the optimal DOP for creating that index. The index will be created with that optimal DOP unless it is limited by the `PARALLEL` setting for the job itself..as discussed for the `ONESTEP_INDEX` parameter.

The default works well in almost all cases. It can be adjusted by experimentation in database configurations that warrant it.

An invalid threshold will produce an error.

> **Note:**
>
> You can also call the parameter by using the subprogram `dbms_datapump.set_parameter(handle,parameter_name,value)`.

Restrictions

The `INDEX_THRESHOLD` value must be specified as a string, such as 1000B, 100k, 200kb, 100M, 200mb, 100G, 200gb, 100t, 200TB.

Example

The following is an example of using the `ONESTEP_INDEX` and `INDEX_THRESHOLD` parameters with their default settings. This command specifies that indexes equal to or larger than 150MB can be created with a DOP greater than 1. It balances large index creation DOP with the overall import job DOP for importing all objects in the job, up to the `PARALLEL=16` setting for the job.

```
> impdp hr DIRECTORY=dpump_dir1 LOGFILE=parallel_import.log
JOB_NAME=imp_par3 DUMPFILE=par_exp%L.dmp PARALLEL=16 ONESTEP_INDEX=FALSE
INDEX_THRESHOLD=150M
```

Related Topics

- SET_PARAMETER Procedures in *Oracle Database PL/SQL Packages and Types Reference*

3.4.22 JOB_NAME

The Oracle Data Pump Import command-line mode `JOB_NAME` parameter is used to identify the import job in subsequent actions.

Default

A system-generated name of the form `SYS_IMPORT or SQLFILE_mode_NN`

Purpose

Use the `JOB_NAME` parameter when you want to identify the import job in subsequent actions. For example, when you want to use the `ATTACH` parameter to attach to a job, you use the `JOB_NAME` parameter to identify the job that you want to attach. You can also use `JOB_NAME` to identify the job by using the views `DBA_DATAPUMP_JOBS` or `USER_DATAPUMP_JOBS`.

Syntax and Description

```
JOB_NAME=jobname_string
```

The variable *jobname_string* specifies a name of up to 128 bytes for the import job. The bytes must represent printable characters and spaces. If the string includes spaces, then the name must be enclosed in single quotation marks (for example, `'Thursday Import'`). For additional information about job name restrictions, see "Database Object Names and Qualifiers" item 7 in *Oracle Database SQL Language Reference*. The job name is implicitly qualified by the schema of the user performing the import operation. The job name is used as the name of the Data Pump control import job table, which controls the export job.

The default job name is system-generated in the form `SYS_IMPORT_mode_NN` or `SYS_SQLFILE_mode_NN`, where *NN* expands to a 2-digit incrementing integer, starting at `01`. For example, `SYS_IMPORT_TABLESPACE_02`' is a default job name.

Example

The following is an example of using the `JOB_NAME` parameter. You can create the `expfull.dmp` dump file that is used in this example by running the example provided in the Export `FULL` parameter.

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

Related Topics

- Database Object Names and Qualifiers in *Oracle Database SQL Language Reference*
- FULL

3.4.23 KEEP_MASTER

The Oracle Data Pump Import command-line mode `KEEP_MASTER` parameter indicates whether the Data Pump control job table should be deleted or retained at the end of an Oracle Data Pump job that completes successfully.

Default

`NO`

Purpose

Indicates whether the Data Pump control job table should be deleted or retained at the end of an Oracle Data Pump job that completes successfully. The Data Pump control job table is automatically retained for jobs that do not complete successfully.

Syntax and Description

```
KEEP_MASTER=[YES | NO]
```

Restrictions

- None

Example

```
> impdp hr SCHEMAS=hr DIRECTORY=dpump_dir1 LOGFILE=schemas.log
DUMPFILE=expdat.dmp KEEP_MASTER=YES
```

3.4.24 LOGFILE

The Oracle Data Pump Import command-line mode `LOGFILE` parameter specifies the name, and optionally, a directory object, for the log file of the import job.

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`*, then it must be one that was previously established by the DBA, and to which you have access. This parameter 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, then it is 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 you specify the `NOLOGFILE` parameter. As with the dump file set, the log file is relative to the server, and not the client.

> **Note:**
>
> Oracle 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 can be different in the log file than they are when displayed on the client output screen.

Restrictions

- To perform an Oracle Data Pump Import using Oracle Automatic Storage Management (Oracle ASM), you must specify a LOGFILE parameter that includes a directory object that does not include the Oracle ASM + notation. That is, the log file must be written to a disk file, and not written into the Oracle ASM storage. Alternatively, you can specify NOLOGFILE=YES. However, this prevents the writing of the log file.

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.

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

Because no directory object is specified on the LOGFILE parameter, the log file is written to the directory object specified on the DIRECTORY parameter.

Related Topics

- STATUS
- Using Directory Objects When Oracle Automatic Storage Management Is Enabled
- FULL

3.4.25 LOGTIME

The Oracle Data Pump Import command-line mode LOGTIME parameter specifies that you want to have messages displayed with timestamps during import.

Default

No timestamps are recorded

Purpose

Specifies that you want to have messages displayed with timestamps during import.. You can use the timestamps to figure out the elapsed time between different phases of a Data Pump operation. Such information can be helpful in diagnosing performance problems and estimating the timing of future similar operations.

Syntax and Description

```
LOGTIME=[NONE | STATUS | LOGFILE | ALL]
```

The available options are defined as follows:

- NONE: No timestamps on status or log file messages (same as default)
- STATUS: Timestamps on status messages only
- LOGFILE: Timestamps on log file messages only
- ALL: Timestamps on both status and log file messages

Restrictions

If the file specified by LOGFILE exists and it is not identified as a Data Pump LOGFILE, such as using more than one dot in the filename (specifically, a compound suffix), then it cannot be overwritten. You must specify a different filename.

Example

The following example records timestamps for all status and log file messages that are displayed during the import operation:

```
> impdp hr DIRECTORY=dpump_dir1 DUMPFILE=expdat.dmp SCHEMAS=hr LOGTIME=ALL
TABLE_EXISTS_ACTION=REPLACE
```

For an example of what the LOGTIME output looks like, see the Export LOGTIME parameter.

Related Topics

- LOGTIME

3.4.26 MASTER_ONLY

The Oracle Data Pump Import command-line mode MASTER_ONLY parameter indicates whether to import just the Data Pump control job table, and then stop the job so that the contents of the Data Pump control job table can be examined.

Default

NO

Purpose

Indicates whether to import just the Data Pump control job table and then stop the job so that the contents of the Data Pump control job table can be examined.

Syntax and Description

```
MASTER_ONLY=[YES | NO]
```

Restrictions

- If the NETWORK_LINK parameter is also specified, then MASTER_ONLY=YES is not supported.

Example

```
> impdp hr SCHEMAS=hr DIRECTORY=dpump_dir1 LOGFILE=schemas.log
DUMPFILE=expdat.dmp MASTER_ONLY=YES
```

3.4.27 METRICS

The Oracle Data Pump Import command-line mode `METRICS` parameter indicates whether additional information about the job should be reported to the log file.

Default

`NO`

Purpose

Indicates whether additional information about the job should be reported to the Oracle Data Pump log file.

Syntax and Description

```
METRICS=[YES | NO]
```

When `METRICS=YES` is used, the number of objects and the elapsed time are recorded in the Oracle Data Pump log file.

Restrictions

- None

Example

```
> impdp hr SCHEMAS=hr DIRECTORY=dpump_dir1 LOGFILE=schemas.log
DUMPFILE=expdat.dmp METRICS=YES
```

3.4.28 NETWORK_LINK

The Oracle Data Pump Import command-line mode `NETWORK_LINK` parameter enables an import from a source database identified by a valid database link.

Default:

There is no default

Purpose

Enables an import from a source database identified by a valid database link. The data from the source database instance is written directly back to the connected database instance.

Syntax and Description

`NETWORK_LINK=`*`source_database_link`*

The `NETWORK_LINK` parameter initiates an import using a database link. This means that the system to which the `impdp` client is connected contacts the source database referenced by the *`source_database_link`*, retrieves data from it, and writes the data directly to the database on the connected instance. There are no dump files involved.

The *`source_database_link`* provided must be the name of a database link to an available database. If the database on that instance does not already have a database link, then you or your DBA must create one using the SQL `CREATE DATABASE LINK` statement.

When you perform a network import using the transportable method, you must copy the source data files to the target database before you start the import.

If the source database is read-only, then the connected user must have a locally managed tablespace assigned as the default temporary tablespace on the source database. Otherwise, the job will fail.

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

The following types of database links are supported for use with Oracle Data Pump Import:

- Public fixed user
- Public connected user
- Public shared user (only when used by link owner)
- Private shared user (only when used by link owner)
- Private fixed user (only when used by link owner)

> **Caution:**
>
> If an import operation is performed over an unencrypted network link, then all data is imported as clear text even if it is encrypted in the database. See *Oracle Database Security Guide* for more information about network security.

Restrictions

- The following types of database links are not supported for use with Oracle Data Pump Import:
  - Private connected user
  - Current user
- The Import `NETWORK_LINK` parameter is not supported for tables containing SecureFiles that have ContentType set, or that are currently stored outside of the SecureFiles segment through Oracle Database File System Links.
- Network imports do not support the use of evolved types.
- When operating across a network link, Data Pump requires that the source and target databases differ by no more than two versions. For example, if one database is Oracle Database 12c, then the other database must be 12c, 11g, or 10g. Note that Oracle Data Pump checks only the major version number (for example, 10g, 11g, 12c), not specific release numbers (for example, 12.1, 12.2, 11.1, 11.2, 10.1, or 10.2).
- If the `USERID` that is executing the import job has the `DATAPUMP_IMP_FULL_DATABASE` role on the target database, then that user must also have the `DATAPUMP_EXP_FULL_DATABASE` role on the source database.
- Network mode import does not use parallel query (PQ) child processes.
- Metadata cannot be imported in parallel when the `NETWORK_LINK` parameter is also used
- When transporting a database over the network using full transportable import, auditing cannot be enabled for tables stored in an administrative tablespace (such as `SYSTEM` and `SYSAUX`) if the audit trail information itself is stored in a user-defined tablespace.

Example

In the following syntax example, replace *source_database_link* with the name of a valid database link.

```
> impdp 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.

Related Topics

- PARALLEL

> **See Also:**
>
> - *Oracle Database Administrator's Guide* for more information about database links
> - *Oracle Database SQL Language Reference* for more information about the `CREATE DATABASE LINK` statement
> - *Oracle Database Administrator's Guide* for more information about locally managed tablespaces

3.4.29 NOLOGFILE

The Oracle Data Pump Import command-line mode `NOLOGFILE` parameter specifies whether to suppress the default behavior of creating a log file.

Default

`NO`

Purpose

Specifies whether to suppress the default behavior of creating a log file.

Syntax and Description

```
NOLOGFILE=[YES | NO]
```

If you specify `NOLOGFILE=YES` to suppress creation of a log file, then 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=YES`, then you run the risk of losing important progress and error information.

Example

The following is an example of using the `NOLOGFILE` parameter.

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

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 `YES`.

3.4.30 PARALLEL

The Oracle Data Pump Import command-line mode `PARALLEL` parameter sets the maximum number of worker processes that can load in parallel.

Default

`1`

Purpose

Specifies the maximum number of worker processes of active execution operating on behalf of the Data Pump control import job.

Syntax and Description

```
PARALLEL=integer
```

The value that you specify for *`integer`* specifies the maximum number of processes of active execution operating on behalf of the import job. This execution set consists of a combination of worker processes and parallel input/output (I/O) server processes. The Data Pump control process, idle worker processes, 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, then multiple processes can read from the same file, but performance can be limited by I/O contention.

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

Using PARALLEL During a Network Mode Import

During a network mode import, the `PARALLEL` parameter defines the maximum number of worker processes that can be assigned to the job. To understand the effect of the `PARALLEL` parameter during a network import mode, it is important to understand the concept of "`table_data` objects" as defined by Oracle Data Pump. When Oracle Data Pump moves data, it considers the following items to be individual "`table_data` objects:"

- a complete table (one that is not partitioned or subpartitioned)
- partitions, if the table is partitioned but not subpartitioned
- subpartitions, if the table is subpartitioned

For example:

- A nonpartitioned table, `scott.non_part_table`, has one `table_data` object:

  ```
  scott.non_part_table
  ```

- A partitioned table, `scott.part_table` (having partition `p1` and partition `p2`), has two `table_data` objects:

  ```
  scott.part_table:p1
  ```

  ```
  scott.part_table:p2
  ```

- A subpartitioned table, `scott.sub_part_table` (having partition `p1` and `p2`, and subpartitions `p1s1`, `p1s2`, `p2s1`, and `p2s2`) has four `table_data` objects:

  `scott.sub_part_table:p1s1`

  `scott.sub_part_table:p1s2`

  `scott.sub_part_table:p2s1`

  `scott.sub_part_table:p2s2`

During a network mode import, each `table_data` object is assigned its own worker process, up to the value specified for the `PARALLEL` parameter. No parallel query (PQ) worker processes are assigned because network mode import does not use parallel query (PQ) worker processes. Multiple `table_data` objects can be unloaded at the same time. However, each `table_data` object is unloaded using a single process.

Using PARALLEL During An Import In An Oracle RAC Environment

In an Oracle Real Application Clusters (Oracle RAC) environment, if an import operation has `PARALLEL=1`, then all Oracle Data Pump processes reside on the instance where the job is started. Therefore, the directory object can point to local storage for that instance.

If the import operation has `PARALLEL` set to a value greater than 1, then Oracle Data Pump processes can reside on instances other than the one where the job was started. Therefore, the directory object must point to shared storage that is accessible by all Oracle RAC cluster member nodes.

Restrictions

- This parameter is valid only in the Enterprise Edition of Oracle Database 11g or later.
- Transportable tablespace metadata cannot be imported in parallel.
- To import a table or table partition in parallel (using parallel query worker processes), you must have the `DATAPUMP_IMP_FULL_DATABASE` role.
- In addition, the following objects cannot be imported in parallel:
  - `TRIGGER`
  - `VIEW`
  - `OBJECT_GRANT`
  - `SEQUENCE`
  - `CONSTRAINT`
  - `REF_CONSTRAINT`

Example

The following is an example of using the `PARALLEL` parameter.

```
> impdp 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) The names of the dump files are `par_exp01.dmp`, `par_exp02.dmp`, and `par_exp03.dmp`.

Related Topics

- PARALLEL

3.4.31 PARALLEL_THRESHOLD

The Oracle Data Pump Import command-line utility `PARALLEL_THRESHOLD` parameter specifies the size of the divisor that Data Pump uses to calculate potential parallel DML based on table size.

Default

`250MB`

Purpose

`PARALLEL_THRESHOLD` should only be used with export or import jobs of a single unpartitioned table, or one partition of a partitioned table. When you specify `PARALLEL` in the job, you can specify `PARALLEL_THRESHOLD` to modify the size of the divisor that Oracle Data Pump uses to determine if a table should be exported or imported using parallel data manipulation statements (PDML) during imports and exports. If you specify a lower value than the default, then it enables a smaller table size to use the Oracle Data Pump parallel algorithm. For example, if you have a 100MB table and you want it to use PDML of 5, to break it into five units, then you specify `PARALLEL_THRESHOLD=20M`

Syntax and Description

The parameter value specifies the threshold size in bytes:

`PARALLEL_THRESHOLD=`*`size-in-bytes`*

For a single table export or import, if you want a higher degree of parallelism, then you may want to set `PARALLEL_THRESHOLD` to lower values, to take advantage of parallelism for a smaller table or table partition. However, the benefit of this resource allocation can be limited by the performance of the I/O of the file systems to which you are loading or unloading. Also, if the job involves more than one object, for both tables and metadata objects, then the PQ allocation request specified by `PARALLEL` with `PARALLEL_THRESHOLD` is of limited value. The actual amount of PQ processes allocated to a table is impacted by how many operations Oracle Data Pump is running concurrently, where the amount of parallelism has to be shared. The database, the optimizer, and the execution plan produced by the optimizer for the SQL determine the actual degree of parallelism used to load or unload the object specified in the job.

One use case for this parameter: Using Oracle Data Pump to load a large table from one database into a larger table in another database. For example: Uploading weekly sales data from an OLTP database into a reporting or business analytics data warehouse database.

Restrictions

`PARALLEL_THRESHOLD` is used only in conjunction when the `PARALLEL` parameter is specified with a value greater than 1.

Example

The following is an example of using the `PARALLEL_THRESHOLD` parameter to export the table `table_to_use_PDML`, where the size of the divisor for PQ processes is set to 1 KB, the

variables *user* and *user-password* are the user and password of the user running Import (impdp), and the job name is parathresh_example.

```
impdp user/user-password \
    directory=dpump_dir \
    dumpfile=parathresh_example.dmp
    tables=table_to_use_PDML \
    parallel=8 \
    parallel_threshold=1K \
    job_name=parathresh_example
```

3.4.32 PARFILE

The Oracle Data Pump Import command-line mode PARFILE parameter specifies the name of an import parameter file.

Default

There is no default

Purpose

Specifies the name of an import parameter file, also known as a **parfile**.

Syntax and Description

```
PARFILE=[directory_path]file_name
```

A parameter file allows you to specify Oracle Data Pump parameters within a file. Whe you create a parameter file, that file can be specified on the command line instead of entering all the individual commands. This option can be useful if you use the same parameter combination many times. The use of parameter files is also highly recommended if you are using parameters whose values require the use of quotation marks.

A directory object is not specified for the parameter file because 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 impdp client. The default location of the parameter file is the user's current directory.

Within a parameter file, a comma is implicit at every newline character so you do not have to enter commas at the end of each line. If you have a long line that wraps, such as a long table name, enter the backslash continuation character (\) at the end of the current line to continue onto the next line.

The contents of the parameter file are written to the Oracle Data Pump log file.

Restrictions

- The PARFILE parameter cannot be specified within a parameter file.

Example

Suppose the content of an example parameter file, hr_imp.par, are as follows:

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

```
DIRECTORY=dpump_dir1
PARALLEL=3
```

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

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

As a result of the command, the tables named `countries`, `locations`, and `regions` are imported from the dump file set that is created when you run the example for the Export `DUMPFILE` parameter. (See the Export `DUMPFILE` parameter.) 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 is also written to `dpump_dir1`.

Related Topics

- DUMPFILE
- About Import Command-Line Mode

3.4.33 PARTITION_OPTIONS

The Oracle Data Pump Import command-line mode `PARTITION_OPTIONS` parameter specifies how you want table partitions created during an import operation.

Default

The default is `departition` when partition names are specified on the `TABLES` parameter and `TRANPORTABLE=ALWAYS` is set (whether on the import operation or during the export). Otherwise, the default is `none`.

Purpose

Specifies how you want table partitions created during an import operation.

Syntax and Description

```
PARTITION_OPTIONS=[NONE | DEPARTITION | MERGE]
```

A value of `NONE` creates tables as they existed on the system from which the export operation was performed. If the export was performed with the transportable method, with a partition or subpartition filter, then you cannot use either the `NONE` option or the `MERGE` option. In that case, you must use the `DEPARTITION` option.

A value of `DEPARTITION` promotes each partition or subpartition to a new individual table. The default name of the new table is the concatenation of the table and partition name, or the table and subpartition name, as appropriate.

A value of `MERGE` combines all partitions and subpartitions into one table.

Parallel processing during import of partitioned tables is subject to the following:

- If a partitioned table is imported into an existing partitioned table, then Data Pump only processes one partition or subpartition at a time, regardless of any value specified with the `PARALLEL` parameter.

- If the table into which you are importing does not already exist, and Data Pump has to create it, then the import runs in parallel up to the parallelism specified on the `PARALLEL` parameter when the import is started.

Restrictions

- You use departitioning to create and populate tables that are based on the source tables partitions.

  To avoid naming conflicts, when the value for `PARTITION_OPTIONS` is set to `DEPARTITION`, then the dependent objects, such as constraints and indexes, are not created along with these tables. This error message is included in the log file if any tables are affected by this restriction: `ORA-39427: Dependent objects of partitioned tables will not be imported`. To suppress this message, you can use the `EXCLUDE` parameter to exclude dependent objects from the import.
- When the value for `PARTITION_OPTIONS` is set to `MERGE`, domain indexes are not created with these tables. If this event occurs, then the error is reported in the log file: `ORA-39426: Domain indexes of partitioned tables will not be imported`. To suppress this message, you can use the `EXCLUDE` parameter to exclude the indexes: `EXCLUDE=DOMAIN_INDEX`.
- If the export operation that created the dump file was performed with the transportable method, and if a partition or subpartition was specified, then the import operation must use the `DEPARTITION` option.
- If the export operation that created the dump file was performed with the transportable method, then the import operation cannot use `PARTITION_OPTIONS=MERGE`.
- If there are any grants on objects being departitioned, then an error message is generated, and the objects are not loaded.

Example

The following example assumes that the `sh.sales` table has been exported into a dump file named `sales.dmp`. It uses the `merge` option to merge all the partitions in `sh.sales` into one non-partitioned table in `scott` schema.

```
> impdp system TABLES=sh.sales PARTITION_OPTIONS=MERGE
DIRECTORY=dpump_dir1 DUMPFILE=sales.dmp REMAP_SCHEMA=sh:scott
```

Related Topics

- TRANSPORTABLE

> **See Also:**
>
> The Export `TRANSPORTABLE` parameter for an example of performing an import operation using `PARTITION_OPTIONS=DEPARTITION`

3.4.34 QUERY

The Oracle Data Pump Import command-line mode `QUERY` parameter enables you to specify a query clause that filters the data that is imported.

Default

There is no default

Purpose

Enables you to specify a query clause that filters the data that is imported.

Syntax and Description

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

The *query_clause* typically is a SQL `WHERE` clause for fine-grained row selection. However, it can be any SQL clause. For example, you can use an `ORDER BY` clause to speed up a migration from a heap-organized table to an index-organized table. If a schema and table name are not supplied, then the query is applied to (and must be valid for) all tables in the source dump file set or database. A table-specific query overrides a query applied to all tables.

When you want to apply the query to a specific table, you must separate the table name from the query cause with a colon (:). You can specify more than one table-specific query , but only one query can be specified per table.

If the `NETWORK_LINK` parameter is specified along with the `QUERY` parameter, then any objects specified in the *query_clause* that are on the remote (source) node must be explicitly qualified with the `NETWORK_LINK` value. Otherwise, Data Pump assumes that the object is on the local (target) node; if it is not, then an error is returned and the import of the table from the remote (source) system fails.

For example, if you specify `NETWORK_LINK=dblink1`, then the *query_clause* of the `QUERY` parameter must specify that link, as shown in the following example:

```
QUERY=(hr.employees:"WHERE last_name IN(SELECT last_name
FROM hr.employees@dblink1)")
```

Depending on your operating system, the use of quotation marks when you specify a value for this parameter may also require that you use escape characters. Oracle recommends that you place this parameter in a parameter file, which can reduce the number of escape characters that might otherwise be needed on the command line. See ”About Import Command-Line Mode."

If you use the `QUERY` parameter , then 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 must be granted access to that specific table.

Restrictions

- When trying to select a subset of rows stored in the export dump file, the `QUERY` parameter cannot contain references to virtual columns for import

The reason for this restriction is that virtual column values are only present in a table in the database. Such a table does not contain the virtual column data in an Oracle Data Pump export file, so having a reference to a virtual column in an import QUERY parameter does not match any known column in the source table in the dump file. However, you can include the virtual column in an import QUERY parameter if you use a network import link (NETWORK_LINK=*dblink to source db*) that imports directly from the source table in the remote database.

- You cannot use the QUERY parameter with the following parameters:
  - CONTENT=METADATA_ONLY
  - SQLFILE
  - TRANSPORT_DATAFILES
- When the QUERY parameter is specified for a table, Oracle Data Pump uses external tables to load the target table. External tables uses a SQL INSERT statement with a SELECT clause. The value of the QUERY parameter is included in the WHERE clause of the SELECT portion of the INSERT statement. If the QUERY parameter includes references to another table with columns whose names match the table being loaded, and if those columns are used in the query, then you must use a table alias to distinguish between columns in the table being loaded, and columns in the SELECT statement with the same name.

  For example, suppose you are importing a subset of the sh.sales table based on the credit limit for a customer in the sh.customers table. In the following example, the table alias used by Data Pump for the table being loaded is KU$. KU$ is used to qualify the cust_id field in the QUERY parameter for loading sh.sales. As a result, Data Pump imports only rows for customers whose credit limit is greater than $10,000.

  ```
  QUERY='sales:"WHERE EXISTS (SELECT cust_id FROM customers c
  WHERE cust_credit_limit > 10000 AND ku$.cust_id = c.cust_id)"'
  ```

  If KU$ is not used for a table alias, then all rows are loaded:

  ```
  QUERY='sales:"WHERE EXISTS (SELECT cust_id FROM customers c
  WHERE cust_credit_limit > 10000 AND cust_id = c.cust_id)"'
  ```

- The maximum length allowed for a QUERY string is 4000 bytes, including quotation marks, which means that the actual maximum length allowed is 3998 bytes.

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 the Export FULL parameter. Because the QUERY value uses quotation marks, Oracle recommends that you use a parameter file.

Suppose you have a parameter file, query_imp.par, that contains the following:

```
QUERY=departments:"WHERE department_id < 120"
```

You can then enter the following command:

```
> impdp hr DIRECTORY=dpump_dir1 DUMPFILE=expfull.dmp
  PARFILE=query_imp.par NOLOGFILE=YES
```

All tables in `expfull.dmp` are imported, but for the `departments` table, only data that meets the criteria specified in the `QUERY` parameter is imported.

Related Topics

- About Import Command-Line Mode
- FULL

3.4.35 REMAP_DATA

The Oracle Data Pump Import command-line mode `REMAP_DATA` parameter enables you to remap data as it is being inserted into a new database.

Default

There is no default

Purpose

The `REMAP_DATA` parameter enables you to remap data as it is being inserted into a new database. A common use is to regenerate primary keys to avoid conflict when importing a table into a pre-existing table on the target database.

You can specify a remap function that takes as a source the value of the designated column from either the dump file or a remote database. The remap function then returns a remapped value that replaces the original value in the target database.

The same function can be applied to multiple columns being dumped. This function is useful when you want to guarantee consistency in remapping both the child and parent column in a referential constraint.

Syntax and Description

```
REMAP_DATA=[schema.]tablename.column_name:[schema.]pkg.function
```

The following is a list of each syntax element, in the order in which they appear in the syntax:

schema: the schema containing the table that you want remapped. By default, this schema is the schema of the user doing the import.

tablename: the table whose column is remapped.

column_name: the column whose data is to be remapped.

schema: the schema containing the PL/SQL package you created that contains the remapping function. As a default, this is the schema of the user doing the import.

pkg: the name of the PL/SQL package you created that contains the remapping function.

function: the name of the function within the PL/SQL that is called to remap the column table in each row of the specified table.

Restrictions

- The data types and sizes of the source argument and the returned value must both match the data type and size of the designated column in the table.
- Remapping functions should not perform commits or rollbacks except in autonomous transactions.

- The use of synonyms as values for the REMAP_DATA parameter is not supported. For example, if the regions table in the hr schema had a synonym of regn, an error would be returned if you specified regn as part of the REMPA_DATA specification.
- Remapping LOB column data of a remote table is not supported.
- REMAP_DATA does not support columns of the following types: User-Defined Types, attributes of User-Defined Types, LONG, REF, VARRAY, Nested Tables, BFILE, and XMLtype.

Example

The following example assumes a package named remap has been created that contains a function named plusx that changes the values for first_name in the employees table.

```
> impdp hr DIRECTORY=dpump_dir1 DUMPFILE=expschema.dmp
TABLES=hr.employees REMAP_DATA=hr.employees.first_name:hr.remap.plusx
```

3.4.36 REMAP_DATAFILE

The Oracle Data Pump Import command-line mode REMAP_DATAFILE parameter changes the name of the source data file to the target data file name in all SQL statements where the source data file is referenced.

Default

There is no default

Purpose

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

Syntax and Description

```
REMAP_DATAFILE=source_datafile:target_datafile
```

Remapping data files 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 data file names in quotation marks to eliminate ambiguity on platforms for which a colon is a valid file specification character.

Depending on your operating system, escape characters can be required if you use quotation marks when you specify a value for this parameter. Oracle recommends that you place this parameter in a parameter file, which can reduce the number of escape characters that you otherwise would require on the command line.

You must have the DATAPUMP_IMP_FULL_DATABASE role to specify this parameter.

Example

Suppose you had a parameter file, payroll.par, with the following content:

```
DIRECTORY=dpump_dir1
FULL=YES
DUMPFILE=db_full.dmp
```

```
REMAP_DATAFILE="'DB1$:[HRDATA.PAYROLL]tbs6.dbf':'/db1/hrdata/payroll/
tbs6.dbf'"
```

You can then issue the following command:

```
> impdp hr PARFILE=payroll.par
```

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

Related Topics

- About Import Command-Line Mode

3.4.37 REMAP_DIRECTORY

The Oracle Data Pump Import command-line mode `REMAP_DIRECTORY` parameter lets you remap directories when you move databases between platforms.

Default

There is no default.

Purpose

The `REMAP_DIRECTORY` parameter changes the source directory string to the target directory string in all SQL statements where the source directory is the left-most portion of a full file or directory specification: `CREATE TABLESPACE`, `CREATE LIBRARY`, and `CREATE DIRECTORY`.

Syntax and Description

```
REMAP_DIRECTORY=source_directory_string:target_directory_string
```

Remapping a directory is useful when you move databases between platforms that have different directory file naming conventions. This provides an easy way to remap multiple data files in a directory when you only *want* to change the directory file specification while preserving the original data file names.

The *source_directory_string* and *target_directory_string* should be exactly as you want them to appear in the SQL statements where they are referenced. In addition, Oracle recommends that the directory be properly terminated with the directory file terminator for the respective source and target platform. Oracle recommends that you enclose the directory names in quotation marks to eliminate ambiguity on platforms for which a colon is a valid directory file specification character.

Depending on your operating system, escape characters can be required if you use quotation marks when you specify a value for this parameter. Oracle recommends that you place this parameter in a parameter file, which can reduce the number of escape characters that you otherwise would require on the command line.

You must have the `DATAPUMP_IMP_FULL_DATABASE` role to specify this parameter.

Restrictions

- The `REMAP_DIRECTORY` and `REMAP_DATAFILE` parameters are mutually exclusive.

Example

Suppose you want to remap the following data files:

```
DB1$:[HRDATA.PAYROLL]tbs5.dbf
DB1$:[HRDATA.PAYROLL]tbs6.dbf
```

In addition, you have a parameter file, `payroll.par`, with the following content:

```
DIRECTORY=dpump_dir1
FULL=YES
DUMPFILE=db_full.dmp
REMAP_DIRECTORY="'DB1$:[HRDATA.PAYROLL]':'/db1/hrdata/payroll/'"
```

You can issue the following command:

```
> impdp hr PARFILE=payroll.par
```

This example remaps the VMS file specifications (`DB1$:[HRDATA.PAYROLL]tbs5.dbf`, and `DB1$:[HRDATA.PAYROLL]tbs6.dbf`) to UNIX file specifications, (`/db1/hrdata/payroll/tbs5.dbf`, and `/db1/hrdata/payroll/tbs6.dbf`) for all SQL DDL statements during the import. The dump file, `db_full.dmp`, is located by the directory object, `dpump_dir1`.

3.4.38 REMAP_SCHEMA

The Oracle Data Pump Import command-line mode `REMAP_SCHEMA` parameter loads all objects from the source schema into a target schema.

Default

There is no default

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 can be incomplete; see the Restrictions section in this topic.

If the schema you are remapping to does not exist before the import, then the import operation can create it, except in the case of `REMAP_SCHEMA` for the `SYSTEM` user. The target schema of the `REMAP_SCHEMA` must exist before the import. To create the schema, the dump file set must contain the necessary `CREATE USER` metadata for the source schema, and you must be carrying out the import with enough privileges. For example, the following Export commands create dump file sets with the necessary metadata to create a schema, because the user `SYSTEM` has the necessary privileges:

```
> expdp system SCHEMAS=hr
Password: password
```

```
> expdp system FULL=YES
Password: password
```

If your dump file set does not contain the metadata necessary to create a schema, or if you do not have privileges, then the target schema must be created before the import operation is performed. You must have the target schema created before the import, because the unprivileged dump files do not contain the necessary information for the import to create the schema automatically.

For Oracle Database releases earlier than Oracle Database 11g, if the import operation does create the schema, then after the import is complete, you must assign it a valid password to connect to it. You can then use the following SQL statement to assign the password; note that you require privileges:

```
SQL> ALTER USER schema_name IDENTIFIED BY new_password
```

In Oracle Database releases after Oracle Database 11g Release 1 (11.1.0.1), it is no longer necessary to reset the schema password; the original password remains valid.

Restrictions

- Unprivileged users can perform schema remaps only if their schema is the target schema of the remap. (Privileged users can perform unrestricted schema remaps.) For example, `SCOTT` can remap his `BLAKE`'s objects to `SCOTT`, but `SCOTT` cannot remap `SCOTT`'s objects to `BLAKE`.
- The mapping can be incomplete, because there are certain schema references that Import is not capable of finding. For example, Import does not find schema references embedded within the body of definitions of types, views, procedures, and packages.
- For triggers, `REMAP_SCHEMA` affects only the trigger owner.
- If any table in the schema being remapped contains user-defined object types, and that table changes between the time it is exported and the time you attempt to import it, then the import of that table fails. However, the import operation itself continues.
- By default, if schema objects on the source database have object identifiers (OIDs), then they are imported to the target database with those same OIDs. If an object is imported back into the same database from which it was exported, but into a different schema, then the OID of the new (imported) object is the same as that of the existing object and the import fails. For the import to succeed, you must also specify the `TRANSFORM=OID:N` parameter on the import. The transform `OID:N` causes a new OID to be created for the new object, which allows the import to succeed.

Example

Suppose that, as user `SYSTEM`, you run the following Export and Import commands to remap the `hr` schema into the `scott` schema:

```
> expdp system SCHEMAS=hr DIRECTORY=dpump_dir1 DUMPFILE=hr.dmp

> impdp system 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 adds 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, then Import automatically creates it with an unusable password. This action 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.

3.4.39 REMAP_TABLE

The Oracle Data Pump Import command-line mode `REMAP_TABLE` parameter enables you to rename tables during an import operation.

Default

There is no default

Purpose

Enables you to rename tables during an import operation.

Syntax and Description

You can use either of the following syntaxes (see the Usage Notes):

```
REMAP_TABLE=[schema.]old_tablename[.partition]:new_tablename
```

OR

```
REMAP_TABLE=[schema.]old_tablename[:partition]:new_tablename
```

If the table is being departitioned, then you can use the `REMAP_TABLE` parameter to rename entire tables, or to rename table partitions (See `PARTITION_OPTIONS`).

You can also use `REMAP_TABLE` to override the automatic naming of exported table partitions.

Usage Notes

With the first syntax, if you specify `REMAP_TABLE=A.B:C`, then Import assumes that `A` is a schema name, `B` is the old table name, and `C` is the new table name. To use the first syntax to rename a partition that is being promoted to a nonpartitioned table, you must specify a schema name.

To use the second syntax to rename a partition being promoted to a nonpartitioned table, you qualify it with the old table name. No schema name is required.

Restrictions

- The `REMAP_TABLE` parameter only handles user-created tables. Data Pump does not have enough information for any dependent tables created internally. Therefore, the `REMAP_TABLE` parameter cannot remap internally created tables.
- Only objects created by the Import are remapped. In particular, pre-existing tables are not remapped.
- If the table being remapped has named constraints in the same schema, and the constraints must be created when the table is created, then `REMAP_TABLE` parameter does not work

Example

The following is an example of using the REMAP_TABLE parameter to rename the employees table to a new name of emps:

```
> impdp hr DIRECTORY=dpump_dir1 DUMPFILE=expschema.dmp
TABLES=hr.employees REMAP_TABLE=hr.employees:emps
```

Related Topics

- PARTITION_OPTIONS

3.4.40 REMAP_TABLESPACE

The Oracle Data Pump Import command-line mode REMAP_TABLESPACE parameter remaps all objects selected for import with persistent data in the source tablespace to be created in the target tablespace.

Default

There is no default

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.

The Data Pump Import method of using the REMAP_TABLESPACE parameter works for all objects, including the CREATE USER statement.

With Oracle Database 19c and later releases, the % wildcard can used in place of the source tablespaces for the REMAP_TABLESPACE parameter. When you specify the source database using the % wildcard, Oracle Data Pump combines the tablespaces from the source database export dumpfile into a target permanent tablespace. It is applied to the object types USER, TABLE, INDEX, MVIEW, MVIEW_LOG, MVIEW_ZONEMAP, and CLUSTERS.

A production database can have multiple tablespaces. You may want to consolidate those tablespaces during migration into a particular target tablespace. For example, you may want to consolidate tablespaces when migrating to Oracle Autonomous Database where only the USER tablespace is available for applications. Using this parameter with the % wildcard makes it easy to do so without specifying all of the source tablespaces.

The % wildcard can also replace the source tablespace specified in place of 'TBS_OLD' in this API example: DBMS_METADATA.SET_REMAP_PARAM(*handle*, 'REMAP_TABLESPACE','TBS_OLD','TBS_NEW','*object-type*');

Restrictions

- Oracle Data Pump Import can only remap tablespaces for transportable imports in databases where the compatibility level is set to 10.1 or later.

- Only objects created by the Import are remapped. In particular, if TABLE_EXISTS_ACTION is set to SKIP, TRUNCATE, or APPEND, then the tablespaces for pre-existing tables are not remapped.
- You cannot use REMAP_TABLESPACE with domain indexes to exclude the storage clause of the source metadata. If you customized the tablespace using storage clauses, then REMAP_TABLESPACE does not apply to those storage clauses. If you used a default tablespace without storage clauses, then REMAP_TABLESPACE should work for that tablespace.
- If the index preferences have customized tablespaces in the storage clauses at the source table, then you must recreate those customized tablespaces on the target before attempting to import those tablespaces. If you do not recreate the customized tablespaces on the target database, then the Text index rebuild will fail.
- The target tablespace must be a permanent tablespace, and it must exist before the import.
- The target tablespace cannot be a temporary tablespace.
- The % wildcard cannot be used with multiple REMAP_TABLESPACE parameters.
- The REMAP_TABLESPACE parameter and TRANSFORM=TABLESPACE:N transform parameter are mutually exclusive.

Example

The following is an example of using the REMAP_TABLESPACE parameter.

```
> impdp hr REMAP_TABLESPACE=tbs_1:tbs_6 DIRECTORY=dpump_dir1
  DUMPFILE=employees.dmp
```

3.4.41 SCHEMAS

The Oracle Data Pump Import command-line mode SCHEMAS parameter specifies that you want a schema-mode import to be performed.

Default

There is no default

Purpose

Specifies that you want a schema-mode import to be performed.

Syntax and Description

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

If you have the DATAPUMP_IMP_FULL_DATABASE role, then you can use this parameter to perform a schema-mode import by specifying a list of schemas to import. First, the user definitions are imported (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. Unprivileged users can specify only their own schemas, or schemas remapped to their own schemas. In that case, no information about the schema definition is imported, only the objects contained within it.

To restrict what is imported by using this import mode, you can use filtering.

Schema mode is the default mode when you are performing a network-based import.

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.

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

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

Related Topics

- Filtering During Import Operations
- SCHEMAS

3.4.42 SERVICE_NAME

The Oracle Data Pump Import command-line mode SERVICE_NAME parameter specifies a service name that you want to use in conjunction with the CLUSTER parameter.

Default

There is no default

Purpose

Used to specify a service name to be used with the CLUSTER parameter.

Syntax and Description

```
SERVICE_NAME=name
```

The SERVICE_NAME parameter can be used with the CLUSTER=YES parameter to specify an existing service associated with a resource group that defines a set of Oracle Real Application Clusters (Oracle RAC) instances belonging to that resource group, typically a subset of all the Oracle RAC instances.

The service name is only used to determine the resource group and instances defined for that resource group. The instance where the job is started is always used, regardless of whether it is part of the resource group.

The SERVICE_NAME parameter is ignored whenCLUSTER=NO is also specified.

Suppose you have an Oracle RAC configuration containing instances A, B, C, and D. Also suppose that a service named my_service exists with a resource group consisting of instances A, B, and C only. In such a scenario, the following would be true:

- If you start an Oracle Data Pump job on instance A, and specify CLUSTER=YES (or accept the default, which is YES), and you do not specify the SERVICE_NAME parameter, then Oracle Data Pump creates workers on all instances: A, B, C, and D, depending on the degree of parallelism specified.
- If you start an Oracle Data Pump job on instance A, and specify CLUSTER=YES and SERVICE_NAME=my_service, then workers can be started on instances A, B, and C only.

- If you start an Oracle Data Pump job on instance D, and specify `CLUSTER=YES` and `SERVICE_NAME=my_service`, then workers can be started on instances A, B, C, and D. Even though instance D is not in `my_service` it is included because it is the instance on which the job was started.
- If you start an Oracle Data Pump job on instance A, and specify `CLUSTER=NO`, then any `SERVICE_NAME` parameter that you specify is ignored, and all processes start on instance A.

Example

```
> impdp system DIRECTORY=dpump_dir1 SCHEMAS=hr
  SERVICE_NAME=sales NETWORK_LINK=dbs1
```

This example starts a schema-mode network import of the `hr` schema. Even though `CLUSTER=YES` is not specified on the command line, it is the default behavior, so the job uses all instances in the resource group associated with the service name `sales`. The `NETWORK_LINK` value of `dbs1` is replaced with the name of the source database from which you are importing data. (Note that there is no dump file generated with a network import.)

The `NETWORK_LINK` parameter is simply being used as part of the example. It is not required when using the `SERVICE_NAME` parameter.

Related Topics

- CLUSTER

3.4.43 SKIP_UNUSABLE_INDEXES

The Oracle Data Pump Import command-line mode `SKIP_UNUSABLE_INDEXES` parameter specifies whether Import skips loading tables that have indexes that were set to the Index Unusable state (by either the system or the user).

Default

The value of the Oracle Database configuration parameter, `SKIP_UNUSABLE_INDEXES`.

Purpose

Specifies whether 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=[YES | NO]
```

If `SKIP_UNUSABLE_INDEXES` is set to `YES`, and a table or partition with an index in the Unusable state is encountered, then the load of that table or partition proceeds anyway, as if the unusable index did not exist.

If `SKIP_UNUSABLE_INDEXES` is set to `NO`, and a table or partition with an index in the Unusable state is encountered, then that table or partition is not loaded. Other tables, with indexes not previously set Unusable, continue to be updated as rows are inserted.

If the `SKIP_UNUSABLE_INDEXES` parameter is not specified, then the setting of the Oracle Database configuration parameter, `SKIP_UNUSABLE_INDEXES` is used to determine how to handle unusable indexes. The default value for that parameter is `y`).

If indexes used to enforce constraints are marked unusable, then the data is not imported into that table.

> **Note:**
>
> `SKIP_UNUSABLE_INDEXES` is useful only when importing data into an existing table. It has no practical effect when a table is created as part of an import. In that case, the table and indexes are newly created, and are not marked unusable.

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.

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

Related Topics

- FULL

3.4.44 SOURCE_EDITION

The Oracle Data Pump Import command-line mode `SOURCE_EDITION` parameter specifies the database edition on the remote node from which objects are fetched.

Default

The default database edition on the remote node from which objects are fetched.

Purpose

Specifies the database edition on the remote node from which objects are e fetched.

Syntax and Description

`SOURCE_EDITION=`*`edition_name`*

If `SOURCE_EDITION=`*`edition_name`* is specified, then the objects from that edition are imported. Oracle Data Pump selects all inherited objects that have not changed, and all actual objects that have changed.

If this parameter is not specified, then the default edition is used. If the specified edition does not exist or is not usable, then an error message is returned.

Restrictions

- The `SOURCE_EDITION` parameter is valid on an import operation only when the `NETWORK_LINK` parameter is also specified.
- This parameter is only useful if there are two or more versions of the same versionable objects in the database.
- The job version must be set to 11.2 or later.

Example

The following is an example of using the import SOURCE_EDITION parameter:

```
> impdp hr DIRECTORY=dpump_dir1 SOURCE_EDITION=exp_edition
NETWORK_LINK=source_database_link EXCLUDE=USER
```

In this example, we assume the existence of an edition named exp_edition on the system from which objects are being imported. Because no import mode is specified, the default, which is schema mode, is used. Replace *source_database_link* with the name of the source database from which you are importing data. The EXCLUDE=USER parameter excludes only the definitions of users, not the objects contained within user schemas. No dump file is generated, because this is a network import.

Related Topics

- NETWORK_LINK
- VERSION

> **See Also:**
>
> - CREATE EDITION in *Oracle Database SQL Language Reference* for information about how editions are created
> - Editions in *Oracle Database Development Guide* for more information about the editions feature, including inherited and actual objects

3.4.45 SQLFILE

The Oracle Data Pump Import command-line mode SQLFILE parameter specifies a file into which all the SQL DDL that Import prepares to execute is written, based on other Import parameters selected.

Default

There is no default

Purpose

Specifies a file into which all the SQL DDL that Import prepares to execute is written, based on other Import parameters selected.

Syntax and Description

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

The *file_name* specifies where the import job writes the DDL that is prepared to run during the job. The SQL is not actually run, and the target system remains unchanged. The file is written to the directory object specified in the DIRECTORY parameter, unless you explicitly specify another directory object. Any existing file that has a name matching the one specified with this parameter is overwritten.

Note that passwords are not included in the SQL file. For example, if a CONNECT statement is part of the DDL that was run, then it is replaced by a comment with only the schema name

shown. In the following example, the dashes (--) indicate that a comment follows. The hr schema name is shown, but not the password.

```
-- CONNECT hr
```

Therefore, before you can run the SQL file, you must edit it by removing the dashes indicating a comment, and adding the password for the hr schema.

Oracle Data Pump places any ALTER SESSION statements at the top of the SQL file created by the Oracle Data Pump import. If the import operation has different connection statements, then you must manually copy each of the ALTER SESSION statements, and paste them after the appropriate CONNECT statements.

For some Oracle Database options, anonymous PL/SQL blocks can appear within the SQLFILE output. Do not run these PL/SQL blocks directly.

Restrictions

- If SQLFILE is specified, then the CONTENT parameter is ignored if it is set to either ALL or DATA_ONLY.
- To perform an Oracle Data Pump Import to a SQL file using Oracle Automatic Storage Management (Oracle ASM), the SQLFILE parameter that you specify must include a directory object that does not use the Oracle ASM + notation. That is, the SQL file must be written to a disk file, not into the Oracle ASM storage.
- You cannot use the SQLFILE parameter in conjunction with the QUERY parameter.
- When you specify the same filename, the SQLFILE filename you provide must have a file extension (SQL, sql, LOG, log, LST, lst). The file name you provide cannot have multiple dots in the filename (specifically, a compound suffix). Compound suffixes are not supported.

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.

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

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

Related Topics

- FULL

3.4.46 STATUS

The Oracle Data Pump Import command-line mode STATUS parameter specifies the frequency at which the job status is displayed.

Default

0

Purpose

Specifies the frequency at which the job status is displayed.

Syntax and Description

```
STATUS[=integer]
```

If you supply a value for *integer*, then 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, then no additional information is displayed beyond information about the 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..

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

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

Related Topics

- FULL

3.4.47 STREAMS_CONFIGURATION

The Oracle Data Pump Import command-line mode `STREAMS_CONFIGURATION` parameter specifies whether to import any GoldenGate Replication metadata that may be present in the export dump file.

Default

```
YES
```

Purpose

Specifies whether to import any GoldenGate Replication metadata that can be present in the export dump file.

Syntax and Description

```
STREAMS_CONFIGURATION=[YES | NO]
```

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.

```
> impdp hr DIRECTORY=dpump_dir1 DUMPFILE=expfull.dmp STREAMS_CONFIGURATION=NO
```

3.4.48 TABLE_EXISTS_ACTION

The Oracle Data Pump Import command-line mode TABLE_EXISTS_ACTION parameter specifies for Import what to do if the table it is trying to create already exists.

Default

SKIP

> **Note:**
>
> If CONTENT=DATA_ONLY is specified, then the default is APPEND, not SKIP.

Purpose

Specifies for 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 option is not valid when 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 option is not valid when the CONTENT parameter is set to DATA_ONLY.

When you are using these options, be aware of the following:

- When you use TRUNCATE or REPLACE, ensure 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 not modified. For REPLACE, the dependent objects are dropped and recreated from the source, if they are not explicitly or implicitly excluded (using EXCLUDE) and if they exist in the source dump file or system.
- When you use APPEND or TRUNCATE, Import checks that rows from the source are compatible with the existing table before performing any action.

  If the existing table has active constraints and triggers, then it is loaded using the external tables access method. If any row violates an active constraint, then the load fails and no data is loaded. You can override this behavior by specifying DATA_OPTIONS=SKIP_CONSTRAINT_ERRORS on the Import command line.

  If you have data that must be loaded, but that can cause constraint violations, then consider disabling the constraints, loading the data, and then deleting the problem rows before re-enabling 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 want to compress your data after the load.

- If you use parallel processing, then review the description of the Import `PARTITION_OPTIONS` parameter for information about how parallel processing of partitioned tables is affected, depending on whether the target table already exists or not.
- If you are importing into an existing table (`TABLE_EXISTS_ACTION=REPLACE` or `TRUNCATE`), then follow these guidelines, depending on the table partitioning scheme:
  - If the partitioning scheme matches between the source and target, then use `DATA_OPTIONS=TRUST_EXISTING_TABLE_PARTITIONS` on import.
  - If the partitioning scheme differs between source and target, then use `DATA_OPTIONS=GROUP_PARTITION_TABLE_DATA` on export.

> **Note:**
>
> When Oracle Data Pump detects that the source table and target table do not match (the two tables do not have the same number of columns or the target table has a column name that is not present in the source table), it then compares column names between the two tables. If the tables have at least one column in common, then the data for the common columns is imported into the table (assuming the data types are compatible). The following restrictions apply:
>
> - This behavior is not supported for network imports.
> - The following types of columns cannot be dropped: object columns, object attributes, nested table columns, and ref columns based on a primary key.

Restrictions

- `TRUNCATE` cannot be used on clustered tables.

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.

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

Related Topics

- PARTITION_OPTIONS
- FULL

3.4.49 REUSE_DATAFILES

The Oracle Data Pump Import command-line mode `REUSE_DATAFILES` parameter specifies whether you want the import job to reuse existing data files for tablespace creation.

Default

`NO`

Purpose

Specifies whether you want the import job to reuse existing data files for tablespace creation.

Syntax and Description

```
REUSE_DATAFILES=[YES | NO]
```

If you use the default (`n`), and the data files specified in `CREATE TABLESPACE` statements already exist, then an error message from the failing `CREATE TABLESPACE` statement is issued, but the import job continues.

If this parameter is specified as `y`, then the existing data files are reinitialized.

> **Caution:**
>
> Specifying `REUSE_DATAFILES=YES` 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.

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

This example reinitializes data files referenced by `CREATE TABLESPACE` statements in the `expfull.dmp` file.

Related Topics

- FULL

3.4.50 TABLES

The Oracle Data Pump Import command-line mode `TABLES` parameter specifies that you want to perform a table-mode import.

Default

There is no default.

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*, then it defaults to that of the current user. To specify a schema other than your own, you must either have the DATAPUMP_IMP_FULL_DATABASE role or remap the schema to the current user.

If you want to restrict what is imported, you can use filtering with this import mode.

If you specify *partition_name*, then it must be the name of a partition or subpartition in the associated table.

You can specify table names and partition names by using the wildcard character %.

The following restrictions apply to table names:

- By default, table names in a database are stored as uppercase characters. If you have a table name in mixed-case or lowercase characters, and you want to preserve case sensitivity for the table name, then 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 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 (#), then unless the table name is enclosed in quotation marks, the Import utility interprets the rest of the line as a comment.

  For example, if the parameter file contains the following line, then 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, if the parameter file contains the following line, then 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 operating system 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 use these special characters in the names so that the operating system shell ignores them, and they can be used with Import.

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`, then it would not be valid to use `TABLES=regn`. An error would be returned.
- You can only specify partitions from one table if `PARTITION_OPTIONS=DEPARTITION` is also specified on the import.
- If you specify `TRANSPORTABLE=ALWAYS`, then all partitions specified on the `TABLES` parameter must be in the same table.
- The length of the table name list specified for the `TABLES` parameter is limited to a maximum of 4 MB, unless you are using the `NETWORK_LINK` parameter to an Oracle Database release 10.2.0.3 or earlier or to a read-only database. In such cases, the limit is 4 KB.

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.

```
> impdp hr DIRECTORY=dpump_dir1 DUMPFILE=expfull.dmp TABLES=employees,jobs
```

The following example is a command to import partitions using the `TABLES`:

```
> impdp hr DIRECTORY=dpump_dir1 DUMPFILE=expdat.dmp
TABLES=sh.sales:sales_Q1_2012,sh.sales:sales_Q2_2012
```

This example imports the partitions `sales_Q1_2012` and `sales_Q2_2012` for the table `sales` in the schema `sh`.

Related Topics

- Filtering During Import Operations
- FULL

3.4.51 TABLESPACES

The Oracle Data Pump Import command-line mode `TABLESPACES` parameter specifies that you want to perform a tablespace-mode import.

Default

There is no default

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).

During the following import situations, Data Pump automatically creates the tablespaces into which the data will be imported:

- The import is being done in FULL or TRANSPORT_TABLESPACES mode
- The import is being done in table mode with TRANSPORTABLE=ALWAYS

In all other cases, the tablespaces for the selected objects must already exist on the import database. You could also use the Import REMAP_TABLESPACE parameter to map the tablespace name to an existing tablespace on the import database.

If you want to restrict what is imported, you can use filtering with this import mode.

Restrictions

- The length of the list of tablespace names specified for the TABLESPACES parameter is limited to a maximum of 4 MB, unless you are using the NETWORK_LINK parameter to a 10.2.0.3 or earlier database or to a read-only database. In such cases, the limit is 4 KB.

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.

```
> impdp hr DIRECTORY=dpump_dir1 DUMPFILE=expfull.dmp
TABLESPACES=tbs_1,tbs_2,tbs_3,tbs_4
```

This example imports all tables that have data in tablespaces tbs_1, tbs_2, tbs_3, and tbs_4.

Related Topics

- Filtering During Import Operations
- FULL

3.4.52 TARGET_EDITION

The Oracle Data Pump Import command-line mode TARGET_EDITION parameter specifies the database edition into which you want objects imported.

Default

The default database edition on the system.

Purpose

Specifies the database edition into which you want objects imported.

Syntax and Description

```
TARGET_EDITION=name
```

If you specify TARGET_EDITION=*name*, then Data Pump Import creates all of the objects found in the dump file. Objects that are not editionable are created in all editions.

For example, tables are not editionable, so if there is a table in the dump file, then the table is created, and all editions see it. Objects in the dump file that are editionable, such as procedures, are created only in the specified target edition.

If this parameter is not specified, then Import uses the default edition on the target database, even if an edition was specified in the export job. If the specified edition does not exist, or is not usable, then an error message is returned.

Restrictions

- This parameter is only useful if there are two or more versions of the same versionable objects in the database.
- The job version must be 11.2 or later.

Example

The following is an example of using the `TARGET_EDITION` parameter:

```
> impdp hr DIRECTORY=dpump_dir1 DUMPFILE=exp_dat.dmp
TARGET_EDITION=exp_edition
```

This example assumes the existence of an edition named `exp_edition` on the system to which objects are being imported. Because no import mode is specified, the default of schema mode will be used.

See *Oracle Database SQL Language Reference* for information about how editions are created. See *Oracle Database Development Guide* for more information about the editions features.

Related Topics

- VERSION
- CREATE EDITION in *Oracle Database SQL Language Reference*
- Editions in *Oracle Database Development Guide*

3.4.53 TRANSFORM

The Oracle Data Pump Import command-line mode `TRANSFORM` parameter enables you to alter object creation DDL for objects being imported.

Default

There is no default.

Purpose

Enables you to alter object creation DDL for objects being imported.

Syntax and Description

```
TRANSFORM = transform_name:value[:object_type]
```

The *`transform_name`* specifies the name of the transform.

Specifying *`object_type`* is optional. If supplied, this parameter designates the object type to which the transform is applied. If no object type is specified, then the transform applies to all valid object types.

The available transforms are as follows, in alphabetical order:

- `CONSTRAINT_NAME_FROM_INDEX: [Y | N ]`

  This transform is valid for the following object types:`TABLE` and `CONSTRAINT` object types.

  This transform parameter affects the generation of the `pk` or `fk` constraint which reference user created indexes. If set to `Y`, it forces the name of the constraint to match the name of the index.

  If set to `N` (the default), then the constraint is created as named on the source database.

- `CONSTRAINT_NOVALIDATE:[Y|N]`

  The default value for this parameter is `N`. If the parameter is set to `Y`, then constraints are not validated during import. Validating constraints during import that were valid on the source can be unnecessary and slow the migration process. Validation can be done after import.

  You cannot choose `CONSTRAINT_NOVALIDATE = Y` for tables with the following properties because these constraints must be in the `VALIDATE` state to complete the import:

  – Reference partitioned table
  – Reference partitioned child table
  – Table with Primary key OID
  – Table is clustered

- `CONSTRAINT_USE_DEFAULT_INDEX: [Y | N ]`

  This transform is valid for the following object types:`TABLE` and `CONSTRAINT` object types.

  This transform parameter affects the generation of index relating to the `pk` or `fk` constraint. If the transform parameter is set to `Y`, then the transform forces the name of an index automatically created to enforce the constraint to be identical to the constraint name. In addition, the index is created and defined using the default constraint definition for the target database, and will not use any special characteristics that might have been defined in the source database.

  Default Indexes are not allowed unless they use standard schema integrity constraints, such as `UNIQUE`, `PRIMARY KEY`, or `FOREIGN KEY`. Accordingly, if you run an Oracle Data Pump import from a system where no restrictions exist, and you have additional constraints in the source index (for example, user generated constraints, such as a hash-partitioned index), then these additional constraints are removed during the import.

  If set to `N` (the default), then the index is created as named and defined on the source database.

- `DISABLE_ARCHIVE_LOGGING:[Y | N]`

  This transform is valid for the following object types: `INDEX` and `TABLE`.

  If set to `Y`, then the logging attributes for the specified object types (`TABLE` and/or `INDEX`) are disabled before the data is imported. If set to `N` (the default), then archive logging is not disabled during import. After the data has been loaded, the logging attributes for the objects are restored to their original settings. If no object type is specified, then the `DISABLE_ARCHIVE_LOGGING` behavior is applied to both `TABLE` and `INDEX` object types. This transform works for both file mode imports and network mode imports. It does not apply to transportable tablespace imports.

> **Note:**
>
> If the database is in `FORCE LOGGING` mode, then the `DISABLE_ARCHIVE_LOGGING` option does not disable logging when indexes and tables are created.

- `DWCS_CVT_IOTS: [Y | N ]`

  This transform is valid for `TABLE` object types.

  If set to `Y`, it directs Oracle Data Pump to transform Index Organized tables to heap organized tables by suppressing the `ORGANIZATION INDEX` clause when creating the table.

  If set to `N` (the default), the generated DDL retains the table characteristics of the source object.

- `DWCS_CVT_CONSTRAINTS: [Y | N ]`

  This transform is valid for the following object types:`TABLE` and `CONSTRAINT` object types.

  If set to `Y`, it directs Oracle Data Pump to create `pk`, `fk`, or `uk` constraints as disabled.

  If set to `N` (the default), it directs Oracle Data Pump to create`pk`, `fk`, or `uk` constraints based on the source database status.

- `INMEMORY:[Y | N]`

  This transform is valid for the following object types: `TABLE` and `TABLESPACE`

  The `INMEMORY` transform is related to the In-Memory Column Store (IM column store). The IM column store is an optional portion of the system global area (SGA) that stores copies of tables, table partitions, and other database objects. In the IM column store, data is populated by column rather than row as it is in other parts of the SGA, and data is optimized for rapid scans. The IM column store does not replace the buffer cache, but acts as a supplement so that both memory areas can store the same data in different formats. The IM column store is included with the Oracle Database In-Memory option.

  If `Y` (the default value) is specified on import, then Data Pump keeps the IM column store clause for all objects that have one. When those objects are recreated at import time, Data Pump generates the IM column store clause that matches the setting for those objects at export time.

  If `N` is specified on import, then Data Pump drops the IM column store clause from all objects that have one. If there is no IM column store clause for an object that is stored in a tablespace, then the object inherits the IM column store clause from the tablespace. So if you are migrating a database, and you want the new database to use IM column store features, then you can pre-create the tablespaces with the appropriate IM column store clause and then use `TRANSFORM=INMEMORY:N` on the import command. The object then inherits the IM column store clause from the new pre-created tablespace.

  If you do not use the `INMEMORY` transform, then you must individually alter every object to add the appropriate IM column store clause.

> **Note:**
>
> The INMEMORY transform is available only in Oracle Database 12c Release 1 (12.1.0.2) or later.
>
> See *Oracle Database Administrator's Guide* for information about using the In-Memory Column Store (IM column store).

- INMEMORY_CLAUSE:*"string with a valid in-memory parameter"*

  This transform is valid for the following object types: TABLE and TABLESPACE.

  The INMEMORY_CLAUSE transform is related to the In-Memory Column Store (IM column store). The IM column store is an optional portion of the system global area (SGA) that stores copies of tables, table partitions, and other database objects. In the IM column store, data is populated by column rather than row as it is in other parts of the SGA, and data is optimized for rapid scans. The IM column store does not replace the buffer cache, but acts as a supplement so that both memory areas can store the same data in different formats. The IM column store is included with the Oracle Database In-Memory option.

  When you specify this transform, Data Pump uses the contents of the string as the INMEMORY_CLAUSE for all objects being imported that have an IM column store clause in their DDL. This transform is useful when you want to override the IM column store clause for an object in the dump file.

  The string that you supply must be enclosed in double quotation marks. If you are entering the command on the command line, be aware that some operating systems can strip out the quotation marks during parsing of the command, which causes an error. You can avoid this error by using backslash escape characters (\). For example:

  ```
  transform=inmemory_clause:\"INMEMORY MEMCOMPRESS FOR DML PRIORITY
  CRITICAL\"
  ```

  Alternatively you can put parameters in a parameter file. Quotation marks in the parameter file are maintained during processing.

  > **Note:**
  >
  > The INMEMORY_CLAUSE transform is available only in Oracle Database 12c Release 1 (12.1.0.2) or later.
  >
  > See *Oracle Database Administrator's Guide* for information about using the In-Memory Column Store (IM column store). See *Oracle Database Reference* for a listing and description of parameters that can be specified in an IM column store clause

- LOB_STORAGE:[SECUREFILE | BASICFILE | DEFAULT | NO_CHANGE]

  This transform is valid for the object type TABLE.

  LOB segments are created with the specified storage, either SECUREFILE or BASICFILE. If the value is NO_CHANGE (the default), then the LOB segments are created with the same storage that they had in the source database. If the value is DEFAULT, then the keyword (SECUREFILE or BASICFILE) is omitted, and the LOB segment is created with the default storage.

Specifying this transform changes LOB storage for all tables in the job, including tables that provide storage for materialized views.

The LOB_STORAGE transform is not valid in transportable import jobs.

- OID:[Y | N]

  This transform is valid for the following object types: INC_TYPE, TABLE, and TYPE.

  If Y (the default value) is specified on import, then the exported OIDs are assigned to new object tables and types. Oracle Data Pump also performs OID checking when looking for an existing matching type on the target database.

  If N is specified on import, then:

  – The assignment of the exported OID during the creation of new object tables and types is inhibited. Instead, a new OID is assigned. Inhibiting assignment of exported OIDs can be useful for cloning schemas, but does not affect referenced objects.

  – Before loading data for a table associated with a type, Oracle Data Pump skips normal type OID checking when looking for an existing matching type on the target database. Other checks using a hash code for a type, version number, and type name are still performed.

- OMIT_ENCRYPTION_CLAUSE: [Y | N]

  This transform is valid for TABLE object types.

  If set to Y, it directs Oracle Data Pump to suppress column encryption clauses. Columns which were encrypted in the source database are not encrypted in imported tables.

  If set to N (the default), it directs Oracle Data Pump to create column encryption clauses, as in the source database.

- PCTSPACE:*some_number_greater_than_zero*

  This transform is valid for the following object types: CLUSTER, CONSTRAINT, INDEX, ROLLBACK_SEGMENT, TABLE, and TABLESPACE.

  The value supplied for this transform must be a number greater than zero. It represents the percentage multiplier used to alter extent allocations and the size of data files.

  You can use the PCTSPACE transform with the Data Pump Export SAMPLE parameter so that the size of storage allocations matches the sampled data subset. (See the SAMPLE export parameter.)

- SEGMENT_ATTRIBUTES:[Y | N]

  This transform is valid for the following object types: CLUSTER, CONSTRAINT, INDEX, ROLLBACK_SEGMENT, TABLE, and TABLESPACE.

  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.

- SEGMENT_CREATION:[Y | N]

  This transform is valid for the object type TABLE.

  If set to Y (the default), then this transform causes the SQL SEGMENT CREATION clause to be added to the CREATE TABLE statement. That is, the CREATE TABLE statement explicitly says either SEGMENT CREATION DEFERRED or SEGMENT CREATION IMMEDIATE. If the value is N, then the SEGMENT CREATION clause is omitted from the CREATE TABLE statement. Set this parameter to N to use the default segment creation attributes for the tables being loaded. (This functionality is available with Oracle Database 11g release 2 (11.2.0.2) and later releases.

- `STORAGE:[Y | N]`

  This transform is valid for the following object types: `CLUSTER`, `CONSTRAINT`, `INDEX`, `ROLLBACK_SEGMENT`, and `TABLE`.

  If the value is specified as `Y`, then the storage clauses are included, with appropriate DDL. The default is `Y`. This parameter is ignored if `SEGMENT_ATTRIBUTES=N`.

- `TABLE_COMPRESSION_CLAUSE:[NONE | compression_clause]`

  This transform is valid for the object type `TABLE`.

  If `NONE` is specified, then the table compression clause is omitted (and the table is given the default compression for the tablespace). Otherwise, the value is a valid table compression clause (for example, `NOCOMPRESS`, `COMPRESS BASIC`, and so on). Tables are created with the specified compression. See *Oracle Database SQL Language Reference* for information about valid table compression syntax.

  If the table compression clause is more than one word, then it must be contained in single or double quotation marks. Also, your operating system can require you to enclose the clause in escape characters, such as the backslash character. For example:

  ```
  TRANSFORM=TABLE_COMPRESSION_CLAUSE:\"COLUMN STORE COMPRESS FOR QUERY HIGH\"
  ```

  Specifying this transform changes the type of compression for all tables in the job, including tables that provide storage for materialized views.

- `XMLTYPE_STORAGE_CLAUSE:[TRANSPORTABLE BINARY XML | BINARY XML]`

  There is no default. If the transform is not used, then he source datatype in the dumpfile is the datatype defined on the target, and the `NOT TRANSPORTABLE` clauses remain as they are.

  Oracle recommends that you use the `TRANSPORTABLE BINARY XML` XMLType with Oracle Database 23ai to store data in a self-contained binary format. This format supports sharding and greater scalability. It does not store the metadata used to encode or decode XML data in a central table (central token tables and schema registries), which simplifies the XML data storage and makes it easier to transport. If `TRANSPORTABLE BINARY XML` is set, then it forces the `TRANSPORTABLE` clause to be present in table creation DDLs for Binary XML data. This data type is available for Oracle Database 21c and Oracle Database 19c in Oracle Cloud Infrastructure.

  Use the `BINARY XML` storage XMLType (which is non-transportable) to store the data in a post-parse, binary format designed specifically for XML data. Binary XML is compact, post-parse, XML schema-aware XML data. The metadata used to encode or decode XML data is stored efficiently in a central table. When `BINARY XML` is set, it forces the `NOT TRANSPORTABLE` clause to be present in table creation DDLs for Binary XML data. When tables with Binary XML data have neither `TRANSPORTABLE` nor `NOT TRANSPORTABLE` clauses, the default is `NOT TRANSPORTABLE`, and the XMLType column remains stored as Binary XML.

  You can export and import data of type XMLType regardless of the source database XMLType storage format (object-relational, binary XML or CLOB). However, Oracle Data Pump exports and imports XML data as binary XML data only. The underlying tables and columns used for object-relational storage of XMLType are consequently not exported. Instead, they are converted to binary form and exported as self-describing binary XML data with a token map preamble.

  Because XMLType data is exported and imported as XML data, the source and target databases can use different XMLType storage models for that data. You can export data

from a database that stores XMLType data one way and import it into a database that stores XMLType data a different way. For details, see *Oracle XML DB Developer's Guide*

Restrictions

For the `XMLTYPE_STORAGE_CLAUSE` data type, the following restrictions apply:

- Do not use the option `table_exists_action=append` to import more than once from the same dump file into an XMLType table, regardless of the XMLType storage model used. Doing so raises a unique-constraint violation error, because rows in XMLType tables are always exported and imported using a unique object identifier.
- Transportable Binary XML can only be stored using SecureFile `LOB`. If a `BasicFile` clause is specified for TBX, then an error is raised. The only exceptions are for the `SYS` and XDB users, which are permitted to use the BasicFile clause.
- Binary XML defaults to SecureFiles storage option. However, if either of the following is true, it is not possible to use SecureFiles LOB storage. In that case, BasicFile is the default option for binary XML data:
  - The tablespace for the XMLType table does not use automatic segment space management.
  - A setting in file init.ora prevents SecureFiles LOB storage. For example, see parameter `DB_SECUREFILE`.

Examples

TRANSFORM use case example

The following is a common example of using `TRANSFORM`. 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, then 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 TABLES=hr.employees DIRECTORY=dpump_dir1 DUMPFILE=hr_emp.dmp
  TRANSFORM=SEGMENT_ATTRIBUTES:N:table
```

The resulting CREATE TABLE statement for the employees table then looks similar to the following. It does not contain a STORAGE or TABLESPACE clause; the attributes for the default tablespace for the HR schema are 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 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 DIRECTORY=dpump_dir1 DUMPFILE=hr.dmp SCHEMAS=hr TRANSFORM=SEGMENT_ATTRIBUTES:N
```

Related Topics

- XMLTYPE_STORAGE_CLAUSE: Export/Import Limitations for Oracle XML DB Repository in *Oracle XML DB Developer's Guide*
- XMLTYPE_STORAGE_CLAUSE: Oracle XML DB Features in *Oracle XML DB Developer's Guide*
- CREATE INDEX in *Oracle Database Administrator's Guide*
- Improved Analytics Using the In-Memory Column Store in *Oracle Database Data Warehousing Guide*
- SAMPLE
  The Oracle Data Pump Export command-line utility SAMPLE parameter specifies a percentage of the data rows that you want to be sampled and unloaded from the source database.
- CREATE TABLE in *Oracle Database SQL Language Reference*

3.4.54 TRANSPORT_DATAFILES

The Oracle Data Pump Import command-line mode `TRANSPORT_DATAFILES` parameter specifies a list of data files that are imported into the target database when `TRANSPORTABLE=ALWAYS` is set during the export.

Default

There is no default.

Purpose

Specifies a list of data files that are imported into the target database by a transportable-tablespace mode import, or by a table-mode or full-mode import, when `TRANSPORTABLE=ALWAYS` is set during the export. The data files must already exist on the target 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.

The *`datafile_name`* can also use wildcards in the file name portion of an absolute path specification. An Asterisk (`*`) matches 0 to *N* characters. A question mark (`?`) matches exactly one character. You cannot use wildcards in the directory portions of the absolute path specification. If a wildcard is used, then all matching files must be part of the transport set. If any files are found that are not part of the transport set, then an error is displayed, and the import job terminates.

At some point before the import operation, you must copy the data files from the source system to the target system. You can copy the data files by using any copy method supported by your operating system. If desired, you can rename the files when you copy them to the target system. See Example 2.

If you already have a dump file set generated by any transportable mode export, then you can perform a transportable-mode import of that dump file by specifying the dump file (which contains the metadata) and the `TRANSPORT_DATAFILES` parameter. The presence of the `TRANSPORT_DATAFILES` parameter tells import that it is a transportable-mode import and where to get the actual data.

Depending on your operating system, the use of quotation marks when you specify a value for this parameter can also require that you use escape characters. Oracle recommends that you place this parameter in a parameter file, which can reduce the number of escape characters that you would otherwise be required to use on the command line.

Restrictions

- You cannot use the `TRANSPORT_DATAFILES` parameter in conjunction with the `QUERY` parameter.
- You cannot restart transportable import jobs.
- The `TRANSPORT_DATAFILES` directory portion of the absolute file path cannot contain wildcards. However, the file name portion of the absolute file path can contain wildcards

Example 1

The following is an example of using the TRANSPORT_DATAFILES parameter. Assume you have a parameter file, trans_datafiles.par, with the following content:

```
DIRECTORY=dpump_dir1
DUMPFILE=tts.dmp
TRANSPORT_DATAFILES='/user01/data/tbs1.dbf'
```

You can then issue the following command:

```
> impdp hr PARFILE=trans_datafiles.par
```

Example 2

This example illustrates the renaming of data files as part of a transportable tablespace export and import operation. Assume that you have a data file named employees.dat on your source system.

1. Using a method supported by your operating system, manually copy the data file named employees.dat from your source system to the system where your target database resides. As part of the copy operation, rename it to workers.dat.
2. Perform a transportable tablespace export of tablespace tbs_1.

   ```
   > expdp hr DIRECTORY=dpump_dir1 DUMPFILE=tts.dmp
   TRANSPORT_TABLESPACES=tbs_1
   ```

   The metadata only (no data) for tbs_1 is exported to a dump file named tts.dmp. The actual data was copied over to the target database in step 1.
3. Perform a transportable tablespace import, specifying an absolute directory path for the data file named workers.dat:

   ```
   > impdp hr DIRECTORY=dpump_dir1 DUMPFILE=tts.dmp
   TRANSPORT_DATAFILES='/user01/data/workers.dat'
   ```

   The metadata contained in tts.dmp is imported, and Oracle Data Pump then assigns the information in the workers.dat file to the correct place in the database.

Example 3

This example illustrates use of the asterisk (*) wildcard character in the file name when used with the TRANSPORT_DATAFILES parameter.

```
TRANSPORT_DATAFILES='/db1/hrdata/payroll/emp*.dbf'
```

This parameter use results in Oracle Data Pump validating that all files in the directory /db1/hrdata/payroll/ of type .dbf whose names begin with emp are part of the transport set.

Example 4

This example illustrates use of the question mark (?) wildcard character in the file name when used with the TRANSPORT_DATAFILES parameter.

```
TRANSPORT_DATAFILES='/db1/hrdata/payroll/m?emp.dbf'
```

This parameter use results in Oracle Data Pump validating that all files in the directory /db1/hrdata/payroll/ of type .dbf whose name begins with m, followed by any other single character, and ending in emp are part of the transport set. For example, a file named myemp.dbf is included, but memp.dbf is not included.

Related Topics

- About Import Command-Line Mode

3.4.55 TRANSPORT_FULL_CHECK

The Oracle Data Pump Import command-line mode TRANSPORT_FULL_CHECK parameter specifies whether to verify that the specified transportable tablespace set is being referenced by objects in other tablespaces.

Default

NO

Purpose

Specifies whether to verify that the specified transportable tablespace set is being referenced by objects in other tablespaces.

Syntax and Description

```
TRANSPORT_FULL_CHECK=[YES | NO]
```

If TRANSPORT_FULL_CHECK=YES, 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, then 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=NO, 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, then 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 (or table mode or full mode when TRANSPORTABLE=ALWAYS was specified on the export) 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 data file named tbs6.dbf already exists.

Assume you have a parameter file, full_check.par, with the following content:

```
DIRECTORY=dpump_dir1
TRANSPORT_TABLESPACES=tbs_6
NETWORK_LINK=source_database_link
TRANSPORT_FULL_CHECK=YES
TRANSPORT_DATAFILES='/wkdir/data/tbs6.dbf'
```

You can then issue the following command:

```
> impdp hr PARFILE=full_check.par
```

3.4.56 TRANSPORT_TABLESPACES

The Oracle Data Pump Import command-line mode TRANSPORT_TABLESPACES parameter specifies that you want to perform an import in transportable-tablespace mode over a database link.

Default

There is no default.

Purpose

Specifies that you want to perform an import in transportable-tablespace mode over a database link (as specified with the NETWORK_LINK parameter.)

Syntax and Description

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

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

Because this import is a transportable-mode import, the tablespaces into which the data is imported are automatically created by Data Pump.You do not need to pre-create them. However, copy the data files to the target database before starting the import.

When you specify TRANSPORT_TABLESPACES on the import command line, you must also use the NETWORK_LINK parameter to specify a database link. A database link is a connection between two physical database servers that allows a client to access them as one logical database. Therefore, the NETWORK_LINK parameter is required, because the object metadata is exported from the source (the database being pointed to by NETWORK_LINK) and then imported directly into the target (database from which the impdp command is issued), using that database link.

There are no dump files involved in this situation. If you copied the actual data to the target in a separate operation using some other means, then specify the TRANSPORT_DATAFILES parameter and indicate where the data is located.

> **Note:**
>
> If you already have a dump file set generated by a transportable-tablespace mode export, then you can perform a transportable-mode import of that dump file, but in this case you do not specify TRANSPORT_TABLESPACES or NETWORK_LINK. Doing so would result in an error. Rather, you specify the dump file (which contains the metadata) and the TRANSPORT_DATAFILES parameter. The presence of the TRANSPORT_DATAFILES parameter tells import that it's a transportable-mode import and where to get the actual data.

When transportable jobs are performed, it is best practice to keep a copy of the data files on the source system until the import job has successfully completed on the target system. If the import job fails, then you still have uncorrupted copies of the data files.

Restrictions

- You cannot export transportable tablespaces and then import them into a database at a lower release level. The target database into which you are importing must be at the same or later release level as the source database.
- The TRANSPORT_TABLESPACES parameter is valid only when the NETWORK_LINK parameter is also specified.
- To use the TRANSPORT_TABLESPACES parameter to perform a transportable tablespace import, the COMPATIBLE initialization parameter must be set to at least 11.0.0.
- Depending on your operating system, the use of quotation marks when you specify a value for this parameter can also require that you use escape characters. Oracle recommends that you place this parameter in a parameter file. If you use a parameter file, then that can reduce the number of escape characters that you have to use on a command line.
- Transportable tablespace jobs do not support the ACCESS_METHOD parameter for Data Pump Import.

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 data file named tbs6.dbf has already been copied from the source database to the local system. Suppose you have a parameter file, tablespaces.par, with the following content:

```
DIRECTORY=dpump_dir1
NETWORK_LINK=source_database_link
TRANSPORT_TABLESPACES=tbs_6
TRANSPORT_FULL_CHECK=NO
TRANSPORT_DATAFILES='user01/data/tbs6.dbf'
```

You can then issue the following command:

```
> impdp hr PARFILE=tablespaces.par
```

Related Topics

- Database Links in *Oracle Database Administrator's Guide*
- Using Data File Copying to Move Data
- How Does Oracle Data Pump Handle Timestamp Data?
- About Import Command-Line Mode

3.4.57 TRANSPORTABLE

The optional Oracle Data Pump Import command-line mode `TRANSPORTABLE` parameter specifies either that transportable tables are imported with `KEEP_READ_ONLY`, or `NO_BITMAP_REBUILD`.

Default

None.

Purpose

This optional parameter enables you to specify two values to control how transportable table imports are managed: `KEEP_READ_ONLY` and `NO_BITMAP_REBUILD`.There is no default value for the `TRANSPORTABLE` parameter.

Syntax and Description

```
TRANSPORTABLE = [ALWAYS|NEVER|KEEP_READ_ONLY|NO_BITMAP_REBUILD]
```

The definitions of the allowed values are as follows:

- `ALWAYS` (valid for Full and Table Export) indicates a transportable export. If specified, then only the metadata is exported, and data files are plugged into the target database during the import.
- `NEVER` indicates that only a traditional data export is enabled.
- `KEEP_READ_ONLY`: Valid with transportable mode imports (table, tablespace, full). If specified, then tablespaces and data files remain in read-only mode. Keeping tablespaces and data files in read-only mode enables the transportable data file set to be available to be plugged in to multiple target databases. When data files are in read-only mode, this disables updating tables containing `TSTZ` column data, if that data needs to be updated, to avoid issues with different `TSTZ` versions. For this reason, tables with `TSTZ` columns are dropped from the transportable import. Placing data files in read-only mode also disables rebuilding of tablespace storage bitmaps to reclaim segments.
- `NO_BITMAP_REBUILD`: Indicates that you do not want Oracle Data Pump to reclaim storage segments by rebuilding tablespace storage bitmaps during the transportable import. Not rebuilding the bitmaps can speed up the import. You can reclaim segments at a later time by using the `DBMS_SPACE_ADMIN.TABLESPACE_REBUILD_BITMAPS()` procedure.

APIs or Classes

You can set the `TRANSPORTABLE` parameter value by using the existing procedure `DBMS_DATAPUMP.SET_PARAMETER`.

Restrictions

- The Import TRANSPORTABLE parameter is valid only if the NETWORK_LINK parameter is also specified.
- The TRANSPORTABLE parameter is only valid in table mode imports and full mode imports.
- The user performing a transportable import requires both the DATAPUMP_EXP_FULL_DATABASE role on the source database, and the DATAPUMP_IMP_FULL_DATABASE role on the target database.
- All objects with storage that are selected for network import must have all of their storage segments on the source system either entirely within administrative, non-transportable tablespaces (SYSTEM / SYSAUX), or entirely within user-defined, transportable tablespaces. Storage for a single object cannot straddle the two kinds of tablespaces.
- To use the TRANSPORTABLE parameter to perform a network-based full transportable import, the Data Pump VERSION parameter must be set to at least 12.0 if the source database is release 11.2.0.3. If the source database is release 12.1 or later, then the VERSION parameter is not required, but the COMPATIBLE database initialization parameter must be set to 12.0.0 or later.

Example of a Network Link Import

The following example shows the use of the TRANSPORTABLE parameter during a network link import, where *datafile_name* is the data file that you want to import.

```
> impdp system TABLES=hr.sales TRANSPORTABLE=ALWAYS
  DIRECTORY=dpump_dir1 NETWORK_LINK=dbs1 PARTITION_OPTIONS=DEPARTITION
  TRANSPORT_DATAFILES=datafile_name
```

Example of a Full Transportable Import

The following example shows the use of the TRANSPORTABLE parameter when performing a full transportable import over the database link dbs1. The import specifies a password for the tables with encrypted columns.

```
> impdp import_admin FULL=Y TRANSPORTABLE=ALWAYS VERSION=12 NETWORK_LINK=dbs1
  ENCRYPTION_PASSWORD=password TRANSPORT_DATAFILES=datafile_name
  LOGFILE=dpump_dir1:fullnet.log
```

Example of Setting NEVER or ALWAYS

Setting the TRANSPORTABLE parameter with string values is limited to NEVER or ALWAYS values:

```
SYS.DBMS_DATAPUMP.SET_PARAMETER(jobhdl, 'TRANSPORTABLE','ALWAYS');
SYS.DBMS_DATAPUMP.SET_PARAMETER(jobhdl, 'TRANSPORTABLE','NEVER');
```

The new TRANSPORTABLE parameter options are set using the new numeric bitmask values:

DBMS_DATAPUMP.KU$_TTS_NEVER is the value 1

DBMS_DATAPUMP.KU$_TTS_ALWAYS is the value 2

DBMS_DATAPUMP.KU$_TTS_KEEP_READ_ONLY is the value 4

DBMS_DATAPUMP.KU$_TTS_NO_BITMAP_REBUILD is the value 8

```
SYS.DBMS_DATAPUMP.SET_PARAMETER(jobhdl, 'TRANSPORTABLE',
DBMS_DATAPUMP.KU$_TTS_ALWAYS+DBMS_DATAPUMP.KU$_TTS_KEEP_READ_ONLY);
```

Example of a File-Based Transportable Tablespace Import

The following example shows the use of the TRANSPORTABLE parameter during a file-based transportable tablespace import. The specified KEEP_READ_ONLY option indicates that the data file remains in read–only access throughout the import operation. The required data files are reported by the transportable tablespace export.

```
impdp system DIRECTORY=dpump_dir DUMPFILE=dumpfile_name
TRANSPORT_DATAFILES=datafile_name TRANSPORTABLE=KEEP_READ_ONLY
```

Related Topics

- About Import Command-Line Mode
- Using Data File Copying to Move Data

3.4.58 VERSION

The Oracle Data Pump Import command-line mode VERSION parameter specifies the version of database objects that you want to import.

Default

You should rarely have to specify the VERSION parameter on an import operation. Oracle Data Pump uses whichever of the following is earlier:

- The version associated with the dump file, or source database in the case of network imports
- The version specified by the COMPATIBLE initialization parameter on the target database

Purpose

Specifies the version of database objects that you want to be imported (that is, only database objects and attributes that are compatible with the specified release will be imported). Note that this does not mean that Oracle Data Pump Import can be used with releases of Oracle Database earlier than 10.1. Oracle Data Pump Import only works with Oracle Database 10g release 1 (10.1) or later. The VERSION parameter simply allows you to identify the version of the objects being 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 compatibility release than that of the source system. When the VERSION parameter is set, database objects or attributes on the source system that are incompatible with the specified release are not moved to the target. For example, tables containing new data types that are not supported in the specified release are not 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 later.
- `LATEST` - The version of the metadata corresponds to the database release. Specifying `VERSION=LATEST` on an import job has no effect when the target database's actual version is later than the version specified in its `COMPATIBLE` initialization parameter.
- *`version_string`* - A specific database release (for example, `12.2.0`).

Restrictions

- If the Oracle Data Pump `VERSION` parameter is specified as any value earlier than 12.1, then the Oracle Data Pump dump file excludes any tables that contain `VARCHAR2` or `NVARCHAR2` columns longer than 4000 bytes and any `RAW` columns longer than 2000 bytes.
- Full imports performed over a network link require that you set `VERSION=12` if the target is Oracle Database 12c Release 1 (12.1.0.1) or later and the source is Oracle Database 11g Release 2 (11.2.0.3) or later.
- Dump files created on Oracle Database 11g releases with the Oracle Data Pump parameter `VERSION=12` can only be imported on Oracle Database 12c Release 1 (12.1) and later.
- The value of the `VERSION` parameter affects the import differently depending on whether data-bound collation (DBC) is enabled.

Example

In the following example, assume that the target is an Oracle Database 12c Release 1 (12.1.0.1) database and the source is an Oracle Database 11g Release 2 (11.2.0.3) database. In that situation, you must set `VERSION=12` for network-based imports. Also note that even though full is the default import mode, you must specify it on the command line when the `NETWORK_LINK` parameter is being used.

```
> impdp hr FULL=Y DIRECTORY=dpump_dir1
  NETWORK_LINK=source_database_link VERSION=12
```

Related Topics

- Oracle Data Pump Behavior with Data-Bound Collation
- Exporting and Importing Between Different Oracle Database Releases

3.4.59 VIEWS_AS_TABLES (Network Import)

The Oracle Data Pump Import command-line mode `VIEWS_AS_TABLES` (Network Import) parameter specifies that you want one or more views to be imported as tables.

Default

There is no default.

> **Note:**
>
> This description of `VIEWS_AS_TABLES` is applicable during network imports, meaning that you supply a value for the Data Pump Import `NETWORK_LINK` parameter.

Purpose

Specifies that you want one or more views to be imported as tables.

Syntax and Description

```
VIEWS_AS_TABLES=[schema_name.]view_name[:table_name], ...
```

Oracle Data Pump imports a table with the same columns as the view and with row data fetched from the view. Oracle Data Pump also imports objects dependent on the view, such as grants and constraints. Dependent objects that do not apply to tables (for example, grants of the `UNDER` object privilege) are not imported. You can use the `VIEWS_AS_TABLES` parameter by itself, or along with the `TABLES` parameter. If either is used, then Oracle Data Pump performs a table-mode import.

The syntax elements are defined as follows:

`schema_name`: The name of the schema in which the view resides. If a schema name is not supplied, it defaults to the user performing the import.

`view_name`: The name of the view to be imported as a table. The view must exist and it must be a relational view with only scalar, non-LOB columns. If you specify an invalid or non-existent view, the view is skipped and an error message is returned.

`table_name`: The name of a table that you want to serve as the source of the metadata for the imported view. By default, Oracle Data Pump automatically creates a temporary "template table" with the same columns and data types as the view, but no rows. If the database is read-only, then this default creation of a template table fails. In such a case, you can specify a table name. The table must be in the same schema as the view. It must be a non-partitioned relational table with heap organization. It cannot be a nested table.

If the import job contains multiple views with explicitly specified template tables, then the template tables must all be different. For example, in the following job (in which two views use the same template table), one of the views is skipped:

```
impdp hr DIRECTORY=dpump_dir NETWORK_LINK=dblink1
VIEWS_AS_TABLES=v1:employees,v2:employees
```

An error message is returned reporting the omitted object.

Template tables are automatically dropped after the import operation is completed. While they exist, you can perform the following query to view their names (which all begin with `KU$VAT`):

```
SQL> SELECT * FROM user_tab_comments WHERE table_name LIKE 'KU$VAT%';
TABLE_NAME                     TABLE_TYPE
------------------------------ -----------
COMMENTS
--------------------------------------------------------
KU$VAT_63629                   TABLE
Data Pump metadata template table for view HR.EMPLOYEESV
```

Restrictions

- The `VIEWS_AS_TABLES` parameter cannot be used with the `TRANSPORTABLE=ALWAYS` parameter.

- Tables created using the `VIEWS_AS_TABLES` parameter do not contain any hidden columns that were part of the specified view.
- The `VIEWS_AS_TABLES` parameter does not support tables that have columns with a data type of `LONG`.

Example

The following example performs a network import to import the contents of the view `hr.v1` from a read-only database. The `hr` schema on the source database must contain a template table with the same geometry as the view `view1` (call this table `view1_tab`). The `VIEWS_AS_TABLES` parameter lists the view name and the table name separated by a colon:

```
> impdp hr VIEWS_AS_TABLES=view1:view1_tab NETWORK_LINK=dblink1
```

The view is imported as a table named `view1` with rows fetched from the view. The metadata for the table is copied from the template table `view1_tab`.

3.5 Commands Available in Oracle Data Pump Import Interactive-Command Mode

In interactive-command mode, the current job continues running, but logging to the terminal is suspended, and the Import prompt (`Import>`) is displayed.

- About Oracle Data Pump Import Interactive Command Mode
  Learn how to run Oracle Data Pump commands from an attached client, or from a terminal other than the one on which the job is running.
- CONTINUE_CLIENT
  The Oracle Data Pump Import interactive command mode `CONTINUE_CLIENT` parameter changes the mode from interactive-command mode to logging mode.
- EXIT_CLIENT
  The Oracle Data Pump Import interactive command mode `EXIT_CLIENT` parameter stops the import client session, exits Import, and discontinues logging to the terminal, but leaves the current job running.
- HELP
  The Oracle Data Pump Import interactive command mode `HELP` parameter provides information about Import commands available in interactive-command mode.
- KILL_JOB
  The Oracle Data Pump Import interactive command mode `KILL_JOB` parameter detaches all currently attached client sessions and then terminates the current job. It exits Import and returns to the terminal prompt.
- PARALLEL
  The Oracle Data Pump Import interactive command mode `PARALLEL` parameter enables you to increase or decrease the number of active child processes, PQ child processes, or both, for the current job.
- START_JOB
  The Oracle Data Pump Import interactive command mode `START_JOB` parameter starts the current job to which you are attached.
- STATUS
  The Oracle Data Pump Import interactive command `STATUS` parameter displays job status, and enables update of the display intervals for logging mode status.

- STOP_JOB
  The Oracle Data Pump Import interactive command mode `STOP_JOB` parameter stops the current job, either immediately or after an orderly shutdown, and exits Import.

3.5.1 About Oracle Data Pump Import Interactive Command Mode

Learn how to run Oracle Data Pump commands from an attached client, or from a terminal other than the one on which the job is running.

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 feature is useful in situations in which you start a job at one location, and must check it at a later time from a different location.

Commands for Oracle Data Pump Interactive Mode

The following table lists the activities that you can perform for the current job from the Oracle Data Pump Import prompt in interactive-command mode.

Table 3-1 Supported Activities in Oracle Data Pump Import's Interactive-Command Mode

Activity	Command Used
Exit interactive-command mode.	`CONTINUE_CLIENT`
Stop the import client session, but leave the current job running.	`EXIT_CLIENT`
Display a summary of available commands.	`HELP`
Detach all currently attached client sessions and terminate the current job.	`KILL_JOB`
Increase or decrease the number of active worker processes for the current job. This command is valid only in Oracle Database Enterprise Edition.	`PARALLEL`
Restart a stopped job to which you are attached.	`START_JOB`
Display detailed status for the current job.	`STATUS`
Stop the current job.	`STOP_JOB`

3.5.2 CONTINUE_CLIENT

The Oracle Data Pump Import interactive command mode `CONTINUE_CLIENT` parameter changes the mode from interactive-command mode to logging mode.

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` also causes the client to attempt to start the job.

Example

```
Import> CONTINUE_CLIENT
```

3.5.3 EXIT_CLIENT

The Oracle Data Pump Import interactive command mode `EXIT_CLIENT` parameter stops the import client session, exits Import, and discontinues logging to the terminal, but leaves the current job running.

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` leaves the job running, you can attach to the job at a later time if the job is still running, or if the job is 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

```
Import> EXIT_CLIENT
```

3.5.4 HELP

The Oracle Data Pump Import interactive command mode `HELP` parameter provides information about Import commands available in interactive-command mode.

Purpose

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

Syntax and Description

```
HELP
```

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

Example

```
Import> HELP
```

3.5.5 KILL_JOB

The Oracle Data Pump Import interactive command mode `KILL_JOB` parameter detaches all currently attached client sessions and then terminates the current job. It exits Import and returns to the terminal prompt.

Purpose

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

Syntax and Description

```
KILL_JOB
```

A job that is terminated 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 terminated by the current user, and are then detached. After all clients are detached, the job process structure is immediately run down, and the Data Pump control job table is deleted. Log files are not deleted.

Example

```
Import> KILL_JOB
```

3.5.6 PARALLEL

The Oracle Data Pump Import interactive command mode `PARALLEL` parameter enables you to increase or decrease the number of active child processes, PQ child processes, or both, for the current job.

Purpose

Enables you to increase or decrease the number of active child processes, parallel query (PQ) child processes, or both, 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 enough resources, and if there is enough work requiring parallelization. A decrease does not take effect until an existing process finishes its current task. If the integer value is decreased, then child processes are idled but not deleted until the job exits.

Restrictions

- This parameter is valid only in the Enterprise Edition of Oracle Database 11g or later releases.
- Transportable tablespace metadata cannot be imported in parallel.
- Metadata cannot be imported in parallel when the `NETWORK_LINK` parameter is also used
- The following objects cannot be imported in parallel:

- TRIGGER
- VIEW
- OBJECT_GRANT
- SEQUENCE
- CONSTRAINT
- REF_CONSTRAINT

Example

```
Import> PARALLEL=10
```

3.5.7 START_JOB

The Oracle Data Pump Import interactive command mode START_JOB parameter starts the current job to which you are attached.

Purpose

Starts the current job to which you are attached.

Syntax and Description

```
START_JOB[=SKIP_CURRENT=YES]
```

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 parent job table remain undisturbed..

The SKIP_CURRENT option allows you to restart a job that previously failed, or that is hung or performing slowly on a particular object. The failing statement or current object being processed is skipped and the job is restarted from the next work item. For parallel jobs, this option causes each worker to skip whatever it is currently working on and to move on to the next item at restart.

You cannot restart either SQLFILE jobs or imports done in transportable-tablespace mode.

Example

```
Import> START_JOB
```

3.5.8 STATUS

The Oracle Data Pump Import interactive command STATUS parameter displays job status, and enables update of the display intervals for logging mode status.

Purpose

Displays cumulative status of the job, a description of the current operation, and an estimated completion percentage. It also allows you to reset the display interval for logging mode status.

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, then 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 displays the current job status, and changes the logging mode display interval to two minutes (120 seconds).

```
Import> STATUS=120
```

3.5.9 STOP_JOB

The Oracle Data Pump Import interactive command mode `STOP_JOB` parameter stops the current job, either immediately or after an orderly shutdown, and exits Import.

Purpose

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

Syntax and Description

```
STOP_JOB[=IMMEDIATE]
```

After you run `STOP_JOB`, you can attach and restart jobs later with `START_JOB`. To attach and restart jobs, the master table and dump file set must not be disturbed, either when you issue the command, or after you issue the command.

To perform an orderly shutdown, use `STOP_JOB` (without any associated value). A warning requiring confirmation is then 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 is then issued. All attached clients, including the one issuing the `STOP_JOB` command, receive a warning that the current user is stopping the job. They are then detached. After all clients are detached, the process structure of the job is immediately run down. That is, the Data Pump control job process does not wait for the worker processes to finish their current tasks. When you specify `STOP_JOB=IMMEDIATE`, there is no risk of corruption or data loss. However, you can be required to redo some tasks that were incomplete at the time of shutdown at restart time.

Example

```
Import> STOP_JOB=IMMEDIATE
```

3.6 Examples of Using Oracle Data Pump Import

You can use these common scenario examples to learn how you can use Oracle Data Pump Import to move your data.

- Performing a Data-Only Table-Mode Import
  See how to use Oracle Data Pump to perform a data-only table-mode import.
- Performing a Schema-Mode Import
  See how to use Oracle Data Pump to perform a schema-mode import.
- Performing a Network-Mode Import
  See how to use Oracle Data Pump to perform a network-mode import.
- Using Wildcards in URL-Based Dumpfile Names
  Oracle Data Pump simplifies importing multiple dump files into Oracle Autonomous Database from the Oracle Object Store Service by allowing wildcards for URL-based dumpfile names.

3.6.1 Performing a Data-Only Table-Mode Import

See how to use Oracle Data Pump to perform a data-only table-mode import.

In the example, the table is named `employees`. It uses the dump file created in "Performing a Table-Mode Export.".

The `CONTENT=DATA_ONLY` parameter filters out any database object definitions (metadata). Only table row data is loaded.

Example 3-1 Performing a Data-Only Table-Mode Import

```
> impdp hr TABLES=employees CONTENT=DATA_ONLY DUMPFILE=dpump_dir1:table.dmp
NOLOGFILE=YES
```

Related Topics

- Performing a Table-Mode Export

3.6.2 Performing a Schema-Mode Import

See how to use Oracle Data Pump to perform a schema-mode import.

The example is a schema-mode import of the dump file set created in "Performing a Schema-Mode Export".

Example 3-2 Performing a Schema-Mode Import

```
> impdp 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, then 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.

Related Topics

- Performing a Schema-Mode Export

3.6.3 Performing a Network-Mode Import

See how to use Oracle Data Pump to perform a network-mode import.

The network-mode import uses as its source the database specified by the `NETWORK_LINK` parameter.

Example 3-3 Network-Mode Import of Schemas

```
> impdp 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.

To remap the schema, user `hr` must have the `DATAPUMP_IMP_FULL_DATABASE` role on the local database and the `DATAPUMP_EXP_FULL_DATABASE` role on the source database.

`REMAP_SCHEMA` loads all the objects from the source schema into the target schema.

Related Topics

- NETWORK_LINK
  The Oracle Data Pump Import command-line mode `NETWORK_LINK` parameter enables an import from a source database identified by a valid database link.

3.6.4 Using Wildcards in URL-Based Dumpfile Names

Oracle Data Pump simplifies importing multiple dump files into Oracle Autonomous Database from the Oracle Object Store Service by allowing wildcards for URL-based dumpfile names.

Example 3-4 Wildcards Used in a URL-based Filename

This example shows how to use wildcards in the file name for importing multiple dump files into Oracle Autonomous Database from the Oracle Object Store Service.

```
> impdp admin/password@ATPC1_high
    directory=data_pump_dir credential=my_cred_name
    dumpfile= https://objectstorage.example.com/v1/atpc/atpc_user/exp%u.dmp"
```

> **Note:**
>
> You cannot use wildcard characters in the bucket-name component of the URL.

3.7 Syntax Diagrams for Oracle Data Pump Import

You can use syntax diagrams to understand the valid SQL syntax for Oracle Data Pump Import.

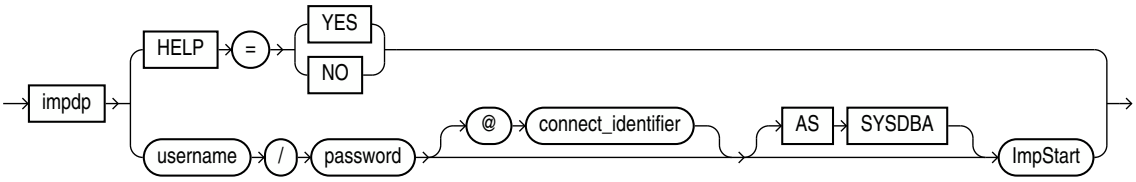
How to Read Graphic Syntax Diagrams

Syntax diagrams are drawings that illustrate valid SQL syntax. To read a diagram, trace it from left to right, in the direction shown by the arrows.

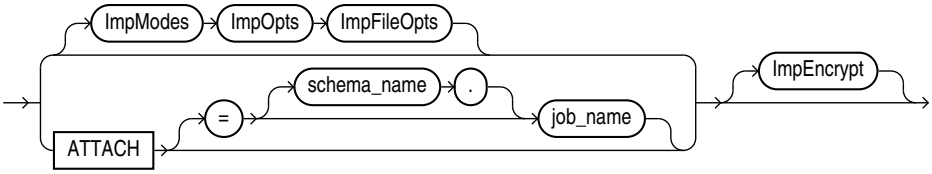
For more information about standard SQL syntax notation, see:

How to Read Syntax Diagrams in *Oracle Database SQL Language Reference*

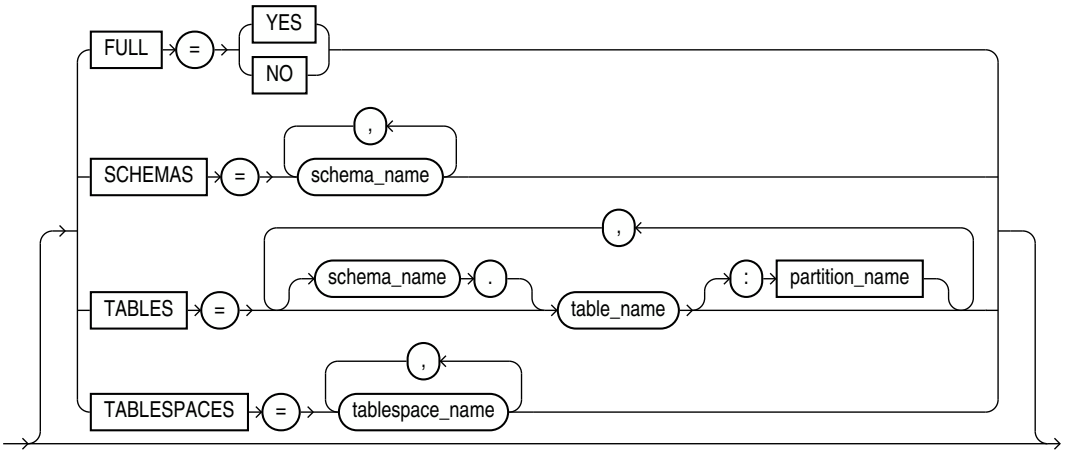
ImpInit



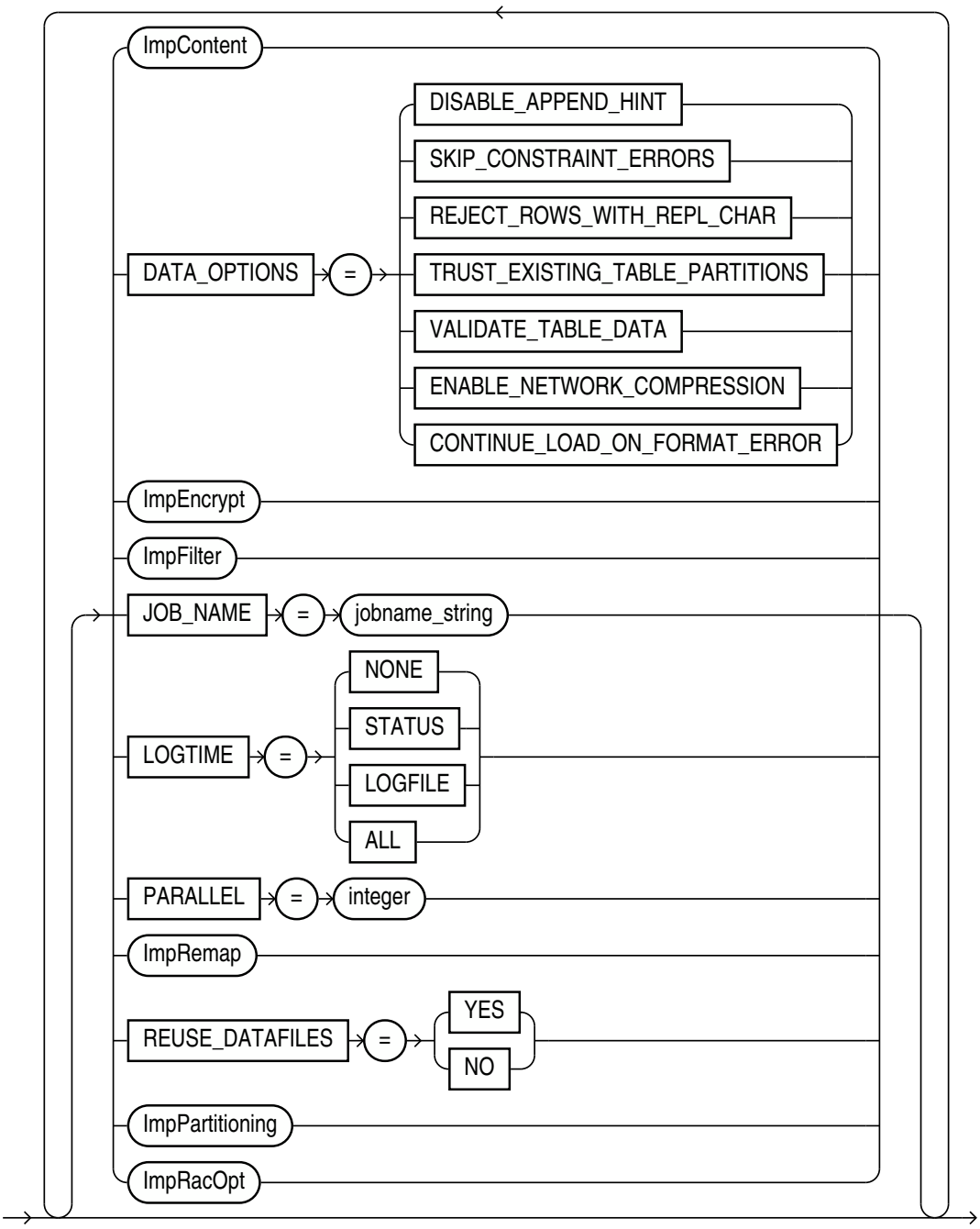
ImpStart



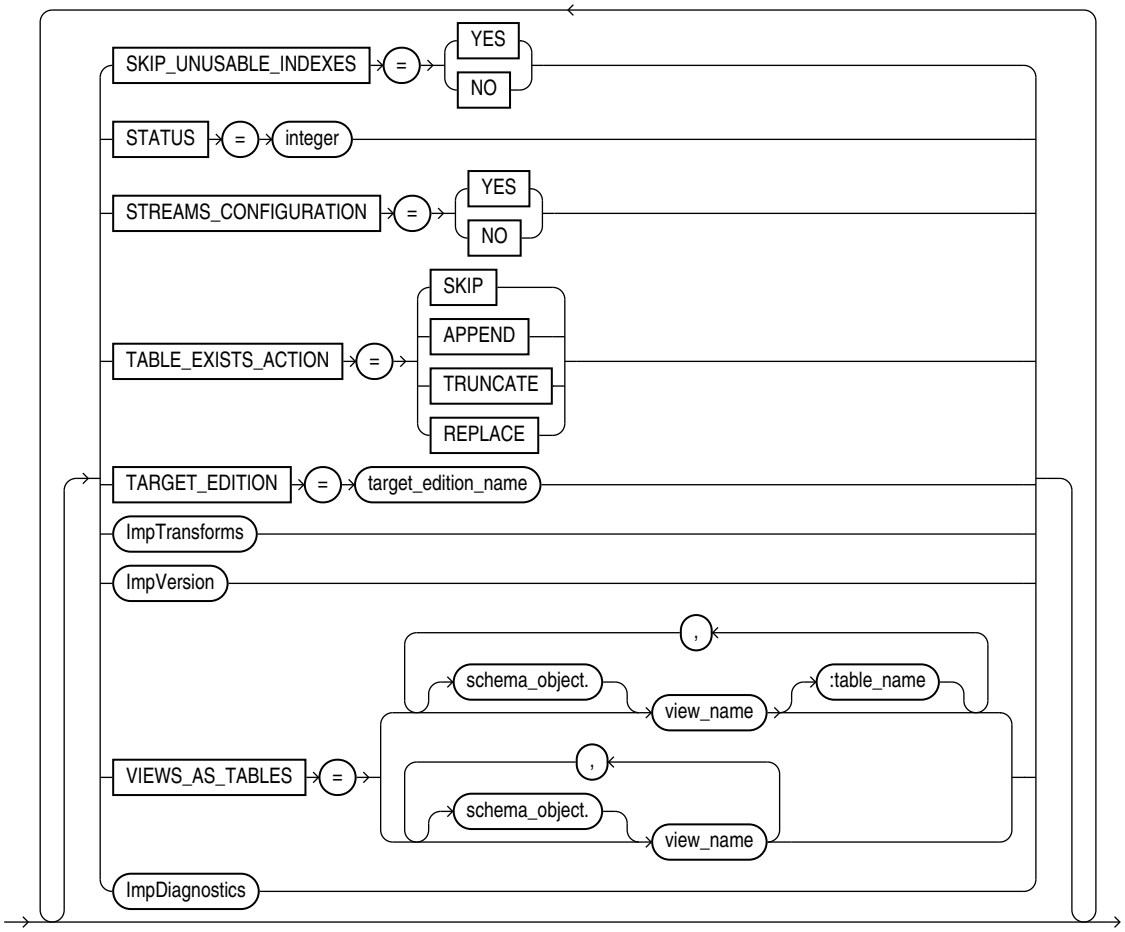
ImpModes



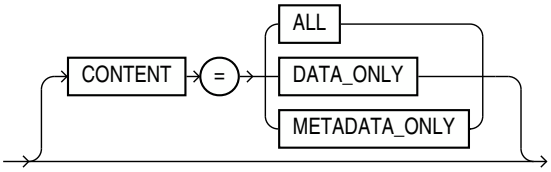
ImpOpts


ImpContent
DISABLE_APPEND_HINT
SKIP_CONSTRAINT_ERRORS
REJECT_ROWS_WITH_REPL_CHAR
DATA_OPTIONS
=
TRUST_EXISTING_TABLE_PARTITIONS
VALIDATE_TABLE_DATA
ENABLE_NETWORK_COMPRESSION
CONTINUE_LOAD_ON_FORMAT_ERROR
ImpEncrypt
ImpFilter
JOB_NAME
=
jobname_string
NONE
LOGTIME
=
STATUS
LOGFILE
ALL
PARALLEL
=
integer
ImpRemap
REUSE_DATAFILES
=
YES
NO
ImpPartitioning
ImpRacOpt

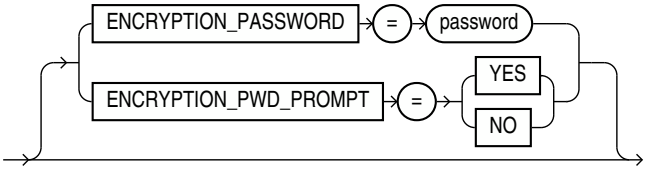
ImpOpts_Cont



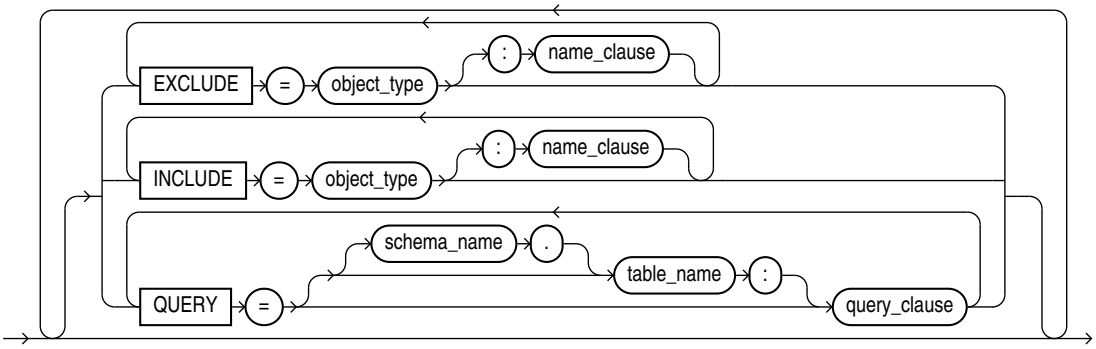
ImpContent



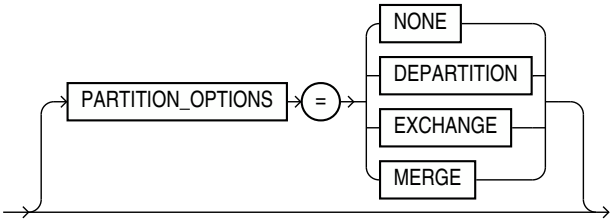
ImpEncrypt



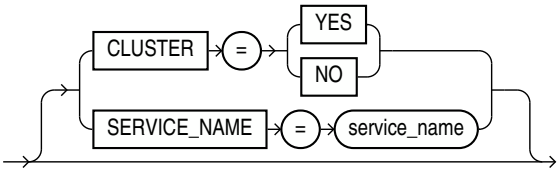
ImpFilter



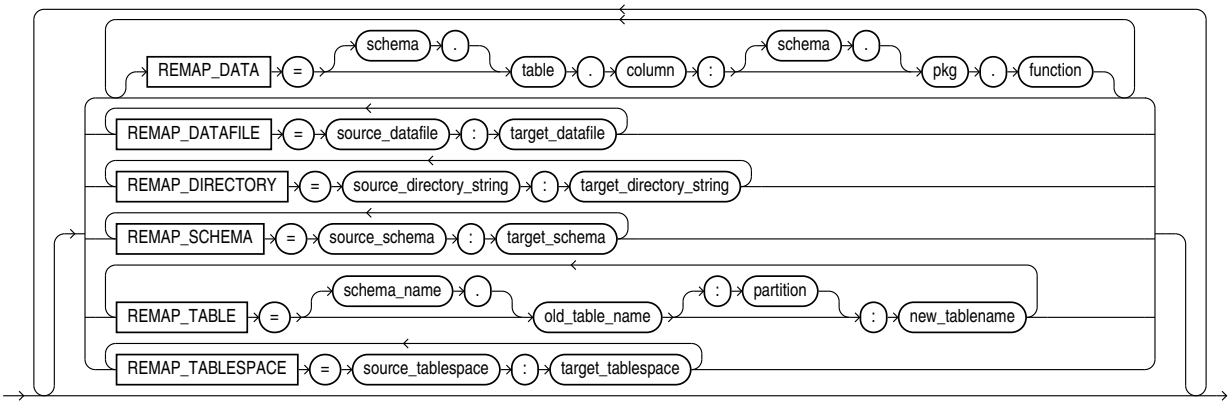
ImpPartitioning



ImpRacOpt

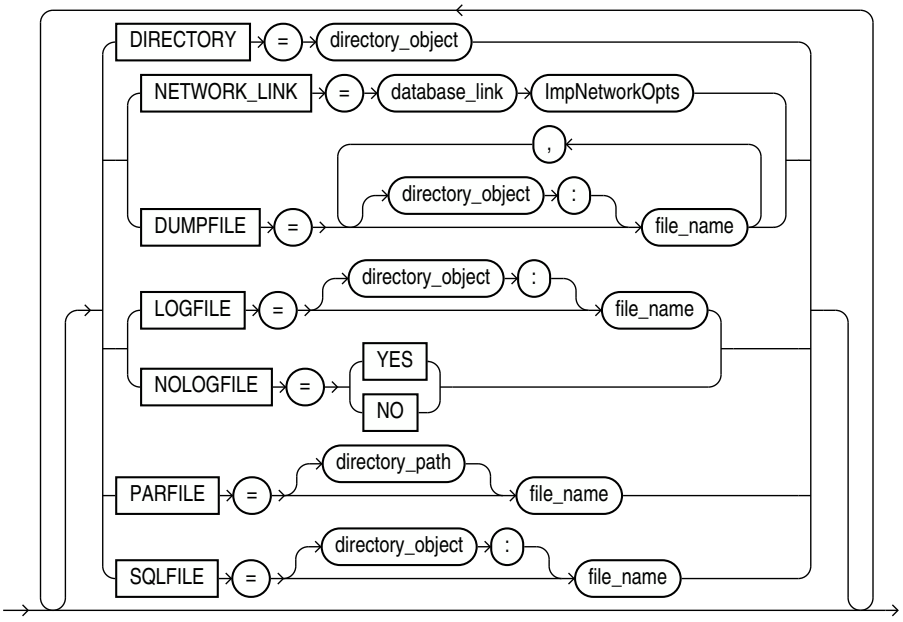


ImpRemap

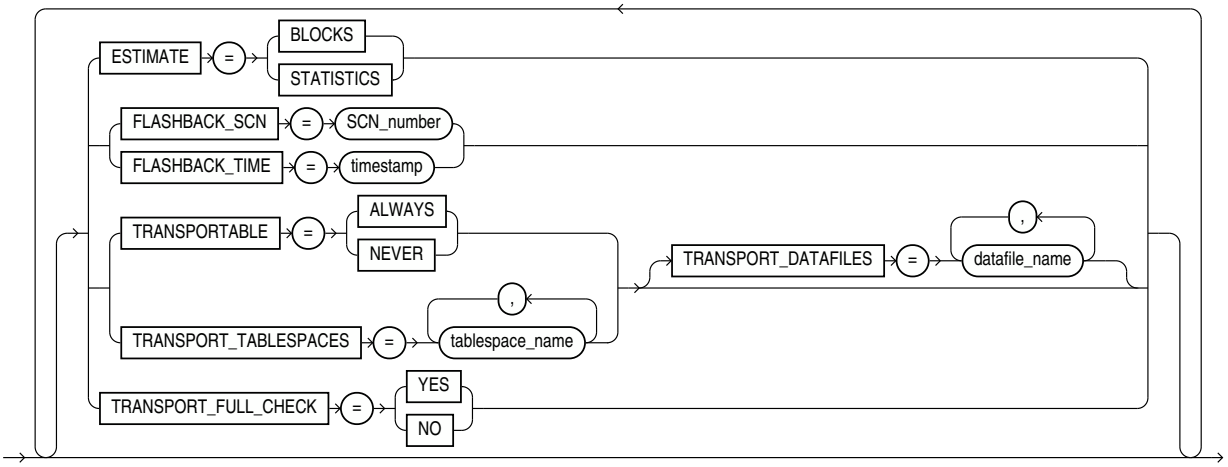


Note: The `REMAP_DATAFILE` and `REMAP_DIRECTORY` parameters are mutually exclusive.

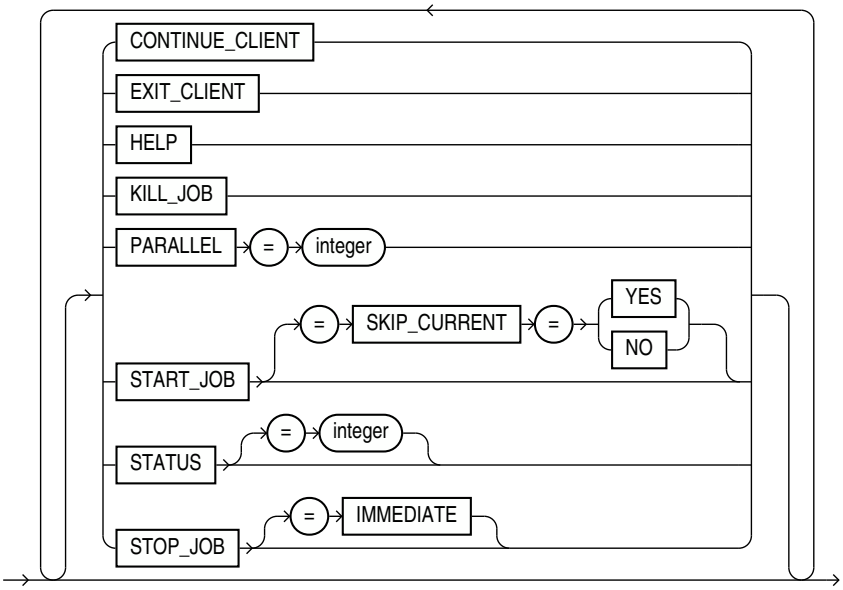
ImpFileOpts



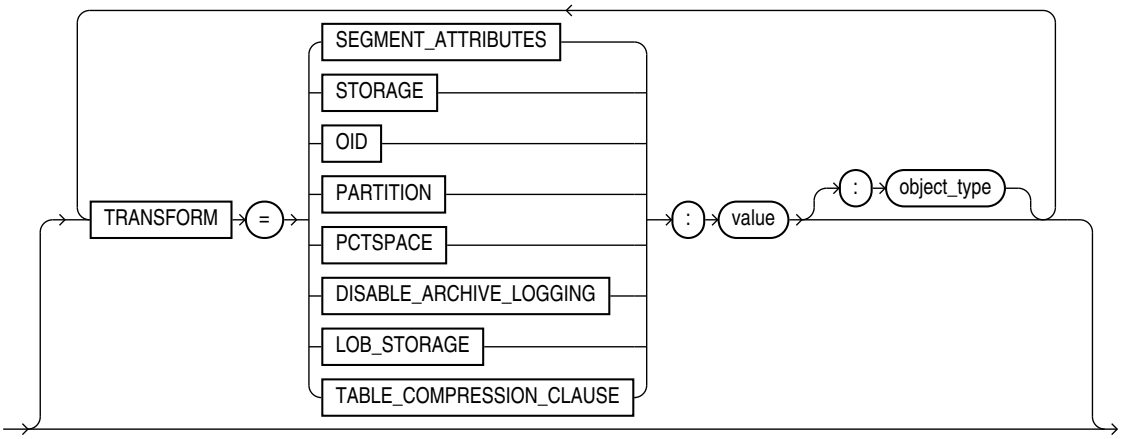
ImpNetworkOpts



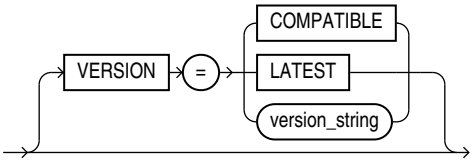
ImpDynOpts



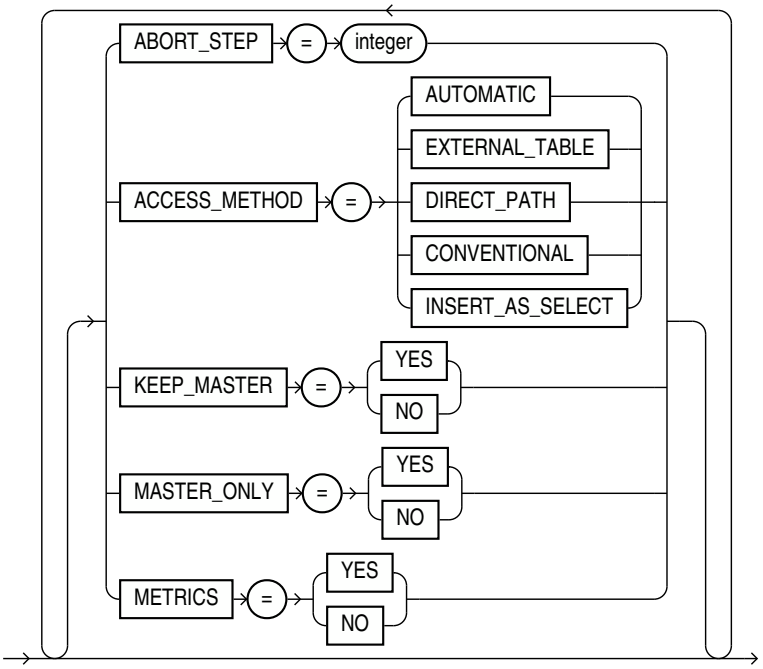
ImpTransforms



ImpVersion



ImpDiagnostics



4
Oracle Data Pump Legacy Mode

With Oracle Data Pump legacy mode, you can use original Export and Import parameters on the Oracle Data Pump Export and Data Pump Import command lines.

- Oracle Data Pump Legacy Mode Use Cases
  Oracle Data Pump enters legacy mode when it encounters legacy export or import parameters, so that you can continue using existing scripts.
- Parameter Mappings
  You can use original Export and Import parameters when they map to Oracle Data Pump Export and Import parameters that supply similar functionality.
- Management of File Locations in Data Pump Legacy Mode
  Original Export and Import and Data Pump Export and Import differ on where dump files and log files can be written to and read from because the original version is client-based and Data Pump is server-based.
- Adjusting Existing Scripts for Oracle Data Pump Log Files and Errors
  When you use Oracle Data Pump in legacy mode, you must review and update your existing scripts written for original Export and Import

4.1 Oracle Data Pump Legacy Mode Use Cases

Oracle Data Pump enters legacy mode when it encounters legacy export or import parameters, so that you can continue using existing scripts.

If you use original Export (`exp`) and Import (`imp`), then you probably have scripts you have been using for many years. Data Pump provides a legacy mode, which allows you to continue to use your existing scripts with Oracle Data Pump.

Oracle Data Pump enters legacy mode when it determines that a parameter unique to original Export or Import is present, either on the command line, or in a script. As Data Pump processes the parameter, the analogous Oracle Data Pump Export or Oracle Data Pump Import parameter is displayed. Oracle strongly recommends that you view the new syntax and make script changes as time permits.

> **Note:**
>
> The Oracle Data Pump Export and Import utilities create and read dump files and log files in Oracle Data Pump format only. They never create or read dump files compatible with original Export or Import. If you have a dump file created with original Export, then you must use original Import to import the data into the database.

4.2 Parameter Mappings

You can use original Export and Import parameters when they map to Oracle Data Pump Export and Import parameters that supply similar functionality.

- Using Original Export Parameters with Data Pump
  Data Pump Export accepts original Export parameters when they map to a corresponding Data Pump parameter.
- Using Original Import Parameters with Data Pump
  Data Pump Import accepts original Import parameters when they map to a corresponding Data Pump parameter.

Related Topics

- Oracle Data Pump Export
  The Oracle Data Pump Export utility is used to unload data and metadata into a set of operating system files, which are called a dump file set.
- Oracle Data Pump Import
  With Oracle Data Pump Import, you can load an export dump file set into a target database, or load a target database directly from a source database with no intervening files.
- Original Export
  The original Export utility (`exp`) was used to write data from Oracle Database into an operating system file in binary format.
- Original Import
  The original Import utility (`imp`) imports dump files that were created using the original Export utility (`exp`).

4.2.1 Using Original Export Parameters with Data Pump

Data Pump Export accepts original Export parameters when they map to a corresponding Data Pump parameter.

This table describes how Data Pump Export interprets original Export parameters. Parameters that have the same name and functionality in both original Export and Data Pump Export are not included in this table.

Table 4-1 How Data Pump Export Handles Original Export Parameters

Original Export Parameter	Action Taken by Data Pump Export Parameter
`BUFFER`	This parameter is ignored.
`COMPRESS`	This parameter is ignored. In original Export, the `COMPRESS` parameter affected how the initial extent was managed. Setting `COMPRESS=n` caused original Export to use current storage parameters for the initial and next extent. The Data Pump Export `COMPRESSION` parameter is used to specify how data is compressed in the dump file, and is not related to the original Export `COMPRESS` parameter.
`CONSISTENT`	Data Pump Export determines the current time and uses `FLASHBACK_TIME`.
`CONSTRAINTS`	If original Export used `CONSTRAINTS=n`, then Data Pump Export uses `EXCLUDE=CONSTRAINTS`. The default behavior is to include constraints as part of the export.

Table 4-1 (Cont.) How Data Pump Export Handles Original Export Parameters

Original Export Parameter	Action Taken by Data Pump Export Parameter
DIRECT	This parameter is ignored. Data Pump Export automatically chooses the best export method.
FEEDBACK	The Data Pump Export STATUS=30 command is used. Note that this is not a direct mapping because the STATUS command returns the status of the export job, as well as the rows being processed. In original Export, feedback was given after a certain number of rows, as specified with the FEEDBACK command. In Data Pump Export, the status is given every so many seconds, as specified by STATUS.
FILE	Data Pump Export attempts to determine the path that was specified or defaulted to for the FILE parameter, and also to determine whether a directory object exists to which the schema has read and write access. See Management of File Locations in Data Pump Legacy Mode for more information about how Data Pump handles the original Export FILE parameter.
GRANTS	If original Export used GRANTS=n, then Data Pump Export uses EXCLUDE=GRANT. If original Export used GRANTS=y, then the parameter is ignored and does not need to be remapped because that is the Data Pump Export default behavior.
INDEXES	If original Export used INDEXES=n, then Data Pump Export uses the EXCLUDE=INDEX parameter. If original Export used INDEXES=y, then the parameter is ignored and does not need to be remapped because that is the Data Pump Export default behavior.
LOG	Data Pump Export attempts to determine the path that was specified or defaulted to for the LOG parameter, and also to determine whether a directory object exists to which the schema has read and write access. See Management of File Locations in Data Pump Legacy Mode for more information about how Data Pump handles the original Export LOG parameter. The contents of the log file will be those of a Data Pump Export operation. See Log Files for information about log file location and content.
OBJECT_CONSISTENT	This parameter is ignored because Data Pump Export processing ensures that each object is in a consistent state when being exported.
OWNER	The Data Pump SCHEMAS parameter is used.
RECORDLENGTH	This parameter is ignored because Data Pump Export automatically takes care of buffer sizing.

Table 4-1 (Cont.) How Data Pump Export Handles Original Export Parameters

Original Export Parameter	Action Taken by Data Pump Export Parameter
RESUMABLE	This parameter is ignored because Data Pump Export automatically provides this functionality to users who have been granted the EXP_FULL_DATABASE role.
RESUMABLE_NAME	This parameter is ignored because Data Pump Export automatically provides this functionality to users who have been granted the EXP_FULL_DATABASE role.
RESUMABLE_TIMEOUT	This parameter is ignored because Data Pump Export automatically provides this functionality to users who have been granted the EXP_FULL_DATABASE role.
ROWS	If original Export used ROWS=y, then Data Pump Export uses the CONTENT=ALL parameter. If original Export used ROWS=n, then Data Pump Export uses the CONTENT=METADATA_ONLY parameter.
STATISTICS	This parameter is ignored because statistics are always saved for tables as part of a Data Pump export operation.
TABLESPACES	If original Export also specified TRANSPORT_TABLESPACE=n, then Data Pump Export ignores the TABLESPACES parameter. If original Export also specified TRANSPORT_TABLESPACE=y, then Data Pump Export takes the names listed for the TABLESPACES parameter and uses them on the Data Pump Export TRANSPORT_TABLESPACES parameter.
TRANSPORT_TABLESPACE	If original Export used TRANSPORT_TABLESPACE=n (the default), then Data Pump Export uses the TABLESPACES parameter. If original Export used TRANSPORT_TABLESPACE=y, then Data Pump Export uses the TRANSPORT_TABLESPACES parameter and only the metadata is exported.
TRIGGERS	If original Export used TRIGGERS=n, then Data Pump Export uses the EXCLUDE=TRIGGER parameter. If original Export used TRIGGERS=y, then the parameter is ignored and does not need to be remapped because that is the Data Pump Export default behavior.
TTS_FULL_CHECK	If original Export used TTS_FULL_CHECK=y, then Data Pump Export uses the TRANSPORT_FULL_CHECK parameter. If original Export used TTS_FULL_CHECK=y, then the parameter is ignored and does not need to be remapped because that is the Data Pump Export default behavior.

Table 4-1 (Cont.) How Data Pump Export Handles Original Export Parameters

Original Export Parameter	Action Taken by Data Pump Export Parameter
VOLSIZE	When the original Export VOLSIZE parameter is used, it means the location specified for the dump file is a tape device. The Data Pump Export dump file format does not support tape devices. Therefore, this operation terminates with an error.

4.2.2 Using Original Import Parameters with Data Pump

Data Pump Import accepts original Import parameters when they map to a corresponding Data Pump parameter.

This table describes how Data Pump Import interprets original Import parameters. Parameters that have the same name and functionality in both original Import and Data Pump Import are not included in this table.

Table 4-2 How Data Pump Import Handles Original Import Parameters

Original Import Parameter	Action Taken by Data Pump Import Parameter
BUFFER	This parameter is ignored.
CHARSET	This parameter was desupported several releases ago and should no longer be used. It will cause the Data Pump Import operation to abort.
COMMIT	This parameter is ignored. Data Pump Import automatically performs a commit after each table is processed.
COMPILE	This parameter is ignored. Data Pump Import compiles procedures after they are created. A recompile can be executed if necessary for dependency reasons.
CONSTRAINTS	If original Import used CONSTRAINTS=n, then Data Pump Import uses the EXCLUDE=CONSTRAINT parameter. If original Import used CONSTRAINTS=y, then the parameter is ignored and does not need to be remapped because that is the Data Pump Import default behavior.
DATAFILES	The Data Pump Import TRANSPORT_DATAFILES parameter is used.
DESTROY	If original Import used DESTROY=y, then Data Pump Import uses the REUSE_DATAFILES=y parameter. If original Import used DESTROY=n, then the parameter is ignored and does not need to be remapped because that is the Data Pump Import default behavior.

Table 4-2 (Cont.) How Data Pump Import Handles Original Import Parameters

Original Import Parameter	Action Taken by Data Pump Import Parameter
FEEDBACK	The Data Pump Import STATUS=30 command is used. Note that this is not a direct mapping because the STATUS command returns the status of the import job, as well as the rows being processed. In original Import, feedback was given after a certain number of rows, as specified with the FEEDBACK command. In Data Pump Import, the status is given every so many seconds, as specified by STATUS.
FILE	Data Pump Import attempts to determine the path that was specified or defaulted to for the FILE parameter, and also to determine whether a directory object exists to which the schema has read and write access. See Management of File Locations in Data Pump Legacy Mode for more information about how Data Pump handles the original Import FILE parameter.
FILESIZE	This parameter is ignored because the information is already contained in the Data Pump dump file set.
FROMUSER	The Data Pump Import SCHEMAS parameter is used. If FROMUSER was used without TOUSER also being used, then import schemas that have the IMP_FULL_DATABASE role cause Data Pump Import to attempt to create the schema and then import that schema's objects. Import schemas that do not have the IMP_FULL_DATABASE role can only import their own schema from the dump file set.
GRANTS	If original Import used GRANTS=n, then Data Pump Import uses the EXCLUDE=OBJECT_GRANT parameter. If original Import used GRANTS=y, then the parameter is ignored and does not need to be remapped because that is the Data Pump Import default behavior.
IGNORE	If original Import used IGNORE=y, then Data Pump Import uses the TABLE_EXISTS_ACTION=APPEND parameter. This causes the processing of table data to continue. If original Import used IGNORE=n, then the parameter is ignored and does not need to be remapped because that is the Data Pump Import default behavior.
INDEXES	If original Import used INDEXES=n, then Data Pump Import uses the EXCLUDE=INDEX parameter. If original Import used INDEXES=y, then the parameter is ignored and does not need to be remapped because that is the Data Pump Import default behavior.

Table 4-2 (Cont.) How Data Pump Import Handles Original Import Parameters

Original Import Parameter	Action Taken by Data Pump Import Parameter
INDEXFILE	The Data Pump Import SQLFILE={directory-object:}filename and INCLUDE=INDEX parameters are used. The same method and attempts made when looking for a directory object described for the FILE parameter also take place for the INDEXFILE parameter. If no directory object was specified on the original Import, then Data Pump Import uses the directory object specified with the DIRECTORY parameter.
LOG	Data Pump Import attempts to determine the path that was specified or defaulted to for the LOG parameter, and also to determine whether a directory object exists to which the schema has read and write access. See Management of File Locations in Data Pump Legacy Mode for more information about how Data Pump handles the original Import LOG parameter. The contents of the log file will be those of a Data Pump Import operation. See Log Files for information about log file location and content.
RECORDLENGTH	This parameter is ignored because Data Pump handles issues about record length internally.
RESUMABLE	This parameter is ignored because this functionality is automatically provided for users who have been granted the IMP_FULL_DATABASE role.
RESUMABLE_NAME	This parameter is ignored because this functionality is automatically provided for users who have been granted the IMP_FULL_DATABASE role.
RESUMABLE_TIMEOUT	This parameter is ignored because this functionality is automatically provided for users who have been granted the IMP_FULL_DATABASE role.
ROWS=N	If original Import used ROWS=n, then Data Pump Import uses the CONTENT=METADATA_ONLY parameter. If original Import used ROWS=y, then Data Pump Import uses the CONTENT=ALL parameter.

Table 4-2 (Cont.) How Data Pump Import Handles Original Import Parameters

Original Import Parameter	Action Taken by Data Pump Import Parameter
SHOW	If SHOW=y is specified, then the Data Pump Import SQLFILE=[directory_object:]file_name parameter is used to write the DDL for the import operation to a file. Only the DDL (not the entire contents of the dump file) is written to the specified file. (Note that the output is not shown on the screen as it was in original Import.) The name of the file will be the file name specified on the DUMPFILE parameter (or on the original Import FILE parameter, which is remapped to DUMPFILE). If multiple dump file names are listed, then the first file name in the list is used. The file will be located in the directory object location specified on the DIRECTORY parameter or the directory object included on the DUMPFILE parameter. (Directory objects specified on the DUMPFILE parameter take precedence.)
STATISTICS	This parameter is ignored because statistics are always saved for tables as part of a Data Pump Import operation.
STREAMS_CONFIGURATION	This parameter is ignored because Data Pump Import automatically determines it; it does not need to be specified.
STREAMS_INSTANTIATION	This parameter is ignored because Data Pump Import automatically determines it; it does not need to be specified
TABLESPACES	If original Import also specified TRANSPORT_TABLESPACE=n (the default), then Data Pump Import ignores the TABLESPACES parameter. If original Import also specified TRANSPORT_TABLESPACE=y, then Data Pump Import takes the names supplied for this TABLESPACES parameter and applies them to the Data Pump Import TRANSPORT_TABLESPACES parameter.
TOID_NOVALIDATE	This parameter is ignored. OIDs are no longer used for type validation.
TOUSER	The Data Pump Import REMAP_SCHEMA parameter is used. There may be more objects imported than with original Import. Also, Data Pump Import may create the target schema if it does not already exist. The FROMUSER parameter must also have been specified in original Import or the operation will fail.
TRANSPORT_TABLESPACE	The TRANSPORT_TABLESPACE parameter is ignored, but if you also specified the DATAFILES parameter, then the import job continues to load the metadata. If the DATAFILES parameter is not specified, then an ORA-39002:invalid operation error message is returned.

Table 4-2 (Cont.) How Data Pump Import Handles Original Import Parameters

Original Import Parameter	Action Taken by Data Pump Import Parameter
TTS_OWNERS	This parameter is ignored because this information is automatically stored in the Data Pump dump file set.
VOLSIZE	When the original Import VOLSIZE parameter is used, it means the location specified for the dump file is a tape device. The Data Pump Import dump file format does not support tape devices. Therefore, this operation terminates with an error.

4.3 Management of File Locations in Data Pump Legacy Mode

Original Export and Import and Data Pump Export and Import differ on where dump files and log files can be written to and read from because the original version is client-based and Data Pump is server-based.

Original Export and Import use the FILE and LOG parameters to specify dump file and log file names, respectively. These file names always refer to files local to the client system and they may also contain a path specification.

Data Pump Export and Import use the DUMPFILE and LOGFILE parameters to specify dump file and log file names, respectively. These file names always refer to files local to the server system and cannot contain any path information. Instead, a directory object is used to indirectly specify path information. The path value defined by the directory object must be accessible to the server. The directory object is specified for a Data Pump job through the DIRECTORY parameter. It is also possible to prepend a directory object to the file names passed to the DUMPFILE and LOGFILE parameters. For privileged users, Data Pump supports the use of a default directory object if one is not specified on the command line. This default directory object, DATA_PUMP_DIR, is set up at installation time.

If Data Pump legacy mode is enabled and the original Export FILE=*filespec* parameter and/or LOG=*filespec* parameter are present on the command line, then the following rules of precedence are used to determine a file's location:

> **Note:**
>
> If the FILE parameter and LOG parameter are both present on the command line, then the rules of precedence are applied separately to each parameter.
>
> Also, when a mix of original Export/Import and Data Pump Export/Import parameters are used, separate rules apply to them. For example, suppose you have the following command:
>
> ```
> expdp system FILE=/user/disk/foo.dmp LOGFILE=foo.log DIRECTORY=dpump_dir
> ```
>
> The Data Pump legacy mode file management rules, as explained in this section, would apply to the FILE parameter. The normal (that is, non-legacy mode) Data Pump file management rules, as described in Default Locations for Dump_ Log_ and SQL Files, would apply to the LOGFILE parameter.

1. If a path location is specified as part of the file specification, then Data Pump attempts to look for a directory object accessible to the schema executing the export job whose path location matches the path location of the file specification. If such a directory object cannot be found, then an error is returned. For example, assume that a server-based directory object named `USER_DUMP_FILES` has been defined with a path value of `'/disk1/user1/dumpfiles/'` and that read and write access to this directory object has been granted to the `hr` schema. The following command causes Data Pump to look for a server-based directory object whose path value contains `'/disk1/user1/dumpfiles/'` and to which the `hr` schema has been granted read and write access:

   ```
   expdp hr FILE=/disk1/user1/dumpfiles/hrdata.dmp
   ```

   In this case, Data Pump uses the directory object `USER_DUMP_FILES`. The path value, in this example `'/disk1/user1/dumpfiles/'`, must refer to a path on the server system that is accessible to the Oracle Database.

   If a path location is specified as part of the file specification, then any directory object provided using the `DIRECTORY` parameter is ignored. For example, if the following command is issued, then Data Pump does not use the `DPUMP_DIR` directory object for the file parameter, but instead looks for a server-based directory object whose path value contains `'/disk1/user1/dumpfiles/'` and to which the `hr` schema has been granted read and write access:

   ```
   expdp hr FILE=/disk1/user1/dumpfiles/hrdata.dmp DIRECTORY=dpump_dir
   ```

2. If no path location is specified as part of the file specification, then the directory object named by the `DIRECTORY` parameter is used. For example, if the following command is issued, then Data Pump applies the path location defined for the `DPUMP_DIR` directory object to the `hrdata.dmp` file:

   ```
   expdp hr FILE=hrdata.dmp DIRECTORY=dpump_dir
   ```

3. If no path location is specified as part of the file specification and no directory object is named by the `DIRECTORY` parameter, then Data Pump does the following, in the order shown:

   a. Data Pump looks for the existence of a directory object of the form `DATA_PUMP_DIR_`*`schema_name`*, where *`schema_name`* is the schema that is executing the Data Pump job. For example, the following command would cause Data Pump to look for the existence of a server-based directory object named `DATA_PUMP_DIR_HR`:

      ```
      expdp hr FILE=hrdata.dmp
      ```

      The `hr` schema also must have been granted read and write access to this directory object. If such a directory object does not exist, then the process moves to step b.

   b. Data Pump looks for the existence of the client-based environment variable `DATA_PUMP_DIR`. For instance, assume that a server-based directory object named `DUMP_FILES1` has been defined and the `hr` schema has been granted read and write access to it. Then on the client system, the environment variable `DATA_PUMP_DIR` can be set to point to `DUMP_FILES1` as follows:

      ```
      setenv DATA_PUMP_DIR DUMP_FILES1
      expdp hr FILE=hrdata.dmp
      ```

      Data Pump then uses the served-based directory object `DUMP_FILES1` for the `hrdata.dmp` file.

      If a client-based environment variable `DATA_PUMP_DIR` does not exist, then the process moves to step c.

c. If the schema that is executing the Data Pump job has DBA privileges, then the default Data Pump directory object, `DATA_PUMP_DIR`, is used. This default directory object is established at installation time. For example, the following command causes Data Pump to attempt to use the default `DATA_PUMP_DIR` directory object, assuming that system has DBA privileges:

```
expdp system FILE=hrdata.dmp
```

> **See Also:**
>
> Default Locations for Dump_ Log_ and SQL Files for information about Data Pump file management rules of precedence under normal Data Pump conditions (that is, non-legacy mode)

4.4 Adjusting Existing Scripts for Oracle Data Pump Log Files and Errors

When you use Oracle Data Pump in legacy mode, you must review and update your existing scripts written for original Export and Import

Oracle Data Pump legacy mode requires that you make adjustments to existing scripts, because of differences in file format and error reporting.

- Log Files
  Oracle Data Pump Export and Import do not generate log files in the same format as those created by original Export and Import.
- Error Cases
  The errors that Oracle Data Pump Export and Import generate can be different from the errors generated by original Export and Import.
- Exit Status
  Oracle Data Pump Export and Import have enhanced exit status values to enable scripts to better determine the success or failure of export and import jobs.

4.4.1 Log Files

Oracle Data Pump Export and Import do not generate log files in the same format as those created by original Export and Import.

You must update any scripts you have that parse the output of original Export and Import, so that they handle the log file format used by Oracle Data Pump Export and Import. For example, the message `Successfully Terminated` does not appear in Oracle Data Pump log files.

4.4.2 Error Cases

The errors that Oracle Data Pump Export and Import generate can be different from the errors generated by original Export and Import.

For example, suppose that a parameter that is ignored by Oracle Data Pump Export would have generated an out-of-range value in original Export. In that case, an informational message is written to the log file stating that the parameter is being ignored. However, no value checking is performed, so no error message is generated.

4.4.3 Exit Status

Oracle Data Pump Export and Import have enhanced exit status values to enable scripts to better determine the success or failure of export and import jobs.

Because Oracle Data Pump Export and Import can have different exit status values, Oracle recommends that you review, and if necessary, update, any scripts that look at the exit status.

5

Oracle Data Pump Performance

Learn how Oracle Data Pump Export and Import is better than that of original Export and Import, and how to enhance performance of export and import operations.

The Oracle Data Pump Export and Import utilities are designed especially for very large databases. If you have large quantities of data versus metadata, then you should experience increased data performance compared to the original Export and Import utilities. (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 Oracle Data Pump Export and Import
  Oracle Data Pump Export (`expdp`) and Import (`impdp`) contain many features that improve performance compared to legacy Export (`exp`) and Import (`imp`).
- Tuning Performance
  Oracle Data Pump is designed to fully use all available resources to maximize throughput, and minimize elapsed job time.
- Initialization Parameters That Affect Data Pump Performance
  The settings for certain Oracle Database initialization parameters can affect the performance of Data Pump Export and Import.

5.1 Data Performance Improvements for Oracle Data Pump Export and Import

Oracle Data Pump Export (`expdp`) and Import (`impdp`) contain many features that improve performance compared to legacy Export (`exp`) and Import (`imp`).

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.
- Oracle Data Pump uses parallelism to build indexes and load package bodies.
- Because Dump files are read and written directly by the server, they 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 data files 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 Oracle Database.

- Metadata and data extraction can be overlapped during export.

5.2 Tuning Performance

Oracle Data Pump is designed to fully use all available resources to maximize throughput, and minimize elapsed job time.

To maximize available resources, a system must be well-balanced across CPU, memory, and I/O. In addition, standard performance tuning principles apply. For example, for maximum performance, ensure that the files that are members of a dump file set reside on separate disks, because the dump files are 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 let you dynamically increase and decrease resource consumption for each job.
- Effect of Compression and Encryption on Performance
  You can improve performance by using Oracle Data Pump parameters related to compression and encryption, particularly in the case of jobs performed in network mode.
- Memory Considerations When Exporting and Importing Statistics
  When you use Oracle Data Pump Export dump files created with a release prior to 12.1, and that contain large amounts of statistics data, this can cause large memory demands during an import operation.

5.2.1 Controlling Resource Consumption

The Data Pump Export and Import utilities let you dynamically increase and decrease resource consumption for each job.

This is done using the Data Pump `PARALLEL` parameter to specify a degree of parallelism for the job. For maximum throughput, do not set `PARALLEL` to much more than twice the number of CPUs (two workers for each CPU).

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.

To maximize parallelism, you must supply at least one file for each degree of parallelism. The simplest way of doing this is to use substitution variables in your file names (for example, `file%u.dmp`). However, depending upon your disk set up (for example, simple, non-striped disks), you might not want to put all dump files on one device. In this case, it is best to specify multiple file names using substitution variables, with each in a separate directory resolving to a separate disk. Even with fast CPUs and fast disks, the path between the CPU and the disk may be the constraining factor in the degree of parallelism that can be sustained.

The Data Pump `PARALLEL` parameter is valid only in the Enterprise Edition of Oracle Database 11*g* or later.

5.2.2 Effect of Compression and Encryption on Performance

You can improve performance by using Oracle Data Pump parameters related to compression and encryption, particularly in the case of jobs performed in network mode.

When you attempt to tune performance, keep in mind your resource availability. Performance can be affected negatively with compression and encryption, because of the additional CPU resources required to perform transformations on the raw data. There are trade-offs on both sides.

5.2.3 Memory Considerations When Exporting and Importing Statistics

When you use Oracle Data Pump Export dump files created with a release prior to 12.1, and that contain large amounts of statistics data, this can cause large memory demands during an import operation.

To avoid running out of memory during the import operation, be sure to allocate enough memory before beginning the import. The exact amount of memory needed depends on how much data you are importing, the platform you are using, and other variables unique to your configuration.

One way to avoid this problem altogether is to set the Data Pump `EXCLUDE=STATISTICS` parameter on either the export or import operation. To regenerate the statistics on the target database, you can use the `DBMS_STATS` PL/SQL package after the import has completed.

Related Topics

- EXCLUDE
- EXCLUDE
- *Oracle Database SQL Tuning Guide*

5.3 Initialization Parameters That Affect Data Pump Performance

The settings for certain Oracle Database 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`

The following initialization parameters must have values set high enough to allow for maximum parallelism:

- `PROCESSES`
- `SESSIONS`
- `PARALLEL_MAX_SERVERS`

Additionally, the `SHARED_POOL_SIZE` and `UNDO_TABLESPACE` initialization parameters should be generously sized. The exact values depend upon the size of your database.

- Setting the Size Of the Buffer Cache In a GoldenGate Replication Environment
  Oracle Data Pump uses GoldenGate Replication functionality to communicate between processes.
- Managing Resource Usage for Multiple User Data Pump Jobs
  When you have multiple users performing data pump jobs in the same database environment, you can use the `MAX_DATAPUMP_JOBS_PER_PDB` and

MAX_DATAPUMP_PARALLEL_PER_JOB initialization parameters to obtain more control over resource utilization.

5.3.1 Setting the Size Of the Buffer Cache In a GoldenGate Replication Environment

Oracle Data Pump uses GoldenGate Replication functionality to communicate between processes.

If the SGA_TARGET initialization parameter is set, then the STREAMS_POOL_SIZE initialization parameter is automatically set to a reasonable value.

If the SGA_TARGET initialization parameter is not set and the STREAMS_POOL_SIZE initialization parameter is not defined, then the size of the streams pool automatically defaults to 10% of the size of the shared pool.

When the streams pool is created, the required SGA memory is taken from memory allocated to the buffer cache, reducing the size of the cache to less than what was specified by the DB_CACHE_SIZE initialization parameter. This means that if the buffer cache was configured with only the minimal required SGA, then Data Pump operations may not work properly. A minimum size of 10 MB is recommended for STREAMS_POOL_SIZE to ensure successful Data Pump operations.

5.3.2 Managing Resource Usage for Multiple User Data Pump Jobs

When you have multiple users performing data pump jobs in the same database environment, you can use the MAX_DATAPUMP_JOBS_PER_PDB and MAX_DATAPUMP_PARALLEL_PER_JOB initialization parameters to obtain more control over resource utilization.

The initialization parameter MAX_DATAPUMP_JOBS_PER_PDB determines the maximum number of concurrent Oracle Data Pump jobs for each pluggable database (PDB). With Oracle Database 19c and later releases, you can set the parameter to AUTO. This setting means that Oracle Data Pump derives the actual value of MAX_DATAPUMP_JOBS_PER_PDB to be 50 percent (50%) of the value of the SESSIONS initialization parameter. If you do not set the value to AUTO, then the default value is 100. You can set the value from 0 to 250.

Oracle Database Release 19c and later releases contain the initialization parameter MAX_DATAPUMP_PARALLEL_PER_JOB. When you have multiple users performing data pump jobs at the same time in a given database environment, you can use this parameter to obtain more control over resource utilization. The parameter MAX_DATAPUMP_PARALLEL_PER_JOB specifies the maximum number of parallel processes that are made available for each Oracle Data Pump job. You can specify a specific maximum number of processes, or you can select AUTO. If you choose to specify a set value, then this maximum number can be from1 to 1024 (the default is 1024). If you choose to specify AUTO, then Oracle Data Pump derives the actual value of the parameter MAX_DATAPUMP_PARALLEL_PER_JOB to be 25 percent (25%) of the value of the SESSIONS initialization parameter.

Related Topics

- MAX_DATAPUMP_JOBS_PER_PDB *Oracle Database Reference*
- MAX_DATAPUMP_PARALLEL_PER_JOB *Oracle Database Reference*

6

Using the Oracle Data Pump API

You can automate data movement operations by using the Oracle Data Pump PL/SQL API `DBMS_DATAPUMP`.

The Oracle Data Pump API `DBMS_DATAPUMP` provides a high-speed mechanism that you can use to move all or part of the data and metadata for a site from one Oracle Database to another. The Oracle Data Pump Export and Oracle Data Pump Import utilities are based on the Oracle Data Pump API.

Oracle Database PL/SQL Packages and Types Reference

- How Does the Oracle Data Pump Client Interface API Work?
  The main structure used in the client interface is a job handle, which appears to the caller as an integer.
- What Are the Basic Steps in Using the Oracle Data Pump API?
  To use the Oracle Data Pump API, you use the procedures provided in the `DBMS_DATAPUMP` package.
- Examples of Using the Data Pump API
  Using the Data Pump API.

Related Topics

- *Oracle Database PL/SQL Packages and Types Reference*

6.1 How Does the Oracle Data Pump Client Interface 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. Handles are session specific. The same job can create different handles in different sessions. As a DBA, the benefit of this feature is that you can start up a job before departing from work, and then watch the progress of the job from home.

- DBMS_DATAPUMP Job States
  Use Oracle Data Pump `DBMS_DATAPUMP` job states show to know which stage your data movement job is performing, and what options are available at each stage.

6.1.1 DBMS_DATAPUMP Job States

Use Oracle Data Pump `DBMS_DATAPUMP` job states show to know which stage your data movement job is performing, and what options are available at each stage.

Job State Definitions

Each phase of a job is associated with a state:

- **Undefined** — before a handle is created
- **Defining** — when the handle is first created

- **Executing** — when the `DBMS_DATAPUMP.START_JOB` procedure is running
- **Completing** — when the job has finished its work and the Oracle Data Pump processes are ending
- **Completed** — when the job is completed
- **Stop Pending** — when an orderly job shutdown has been requested
- **Stopping** — when the job is stopping
- **Idling** — the period between the time that a `DBMS_DATAPUMP.ATTACH` is run to attach to a stopped job, and the time that a `DBMS_DATAPUMP.START_JOB` is run to restart that job
- **Not Running** — when a Data Pump control job table exists for a job that is not running (has no Oracle Data Pump processes associated with it)

Usage Notes

Performing `DBMS_DATAPUMP.START_JOB` on a job in an **Idling** state returns that job to an **Executing** state.

If all users run `DBMS_DATAPUMP.DETACH` to detach from a job in the **Defining** state, then the job is totally removed from the database.

If a job terminates unexpectedly, or if an instance running the job is shut down, and the job was previously in an **Executing** or **Idling** state, then the job is placed in the **Not Running** state. You can then restart the job.

The Oracle Data Pump control job 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 Data Pump control table for the job exists in all states except the **Undefined** state. Child processes are only active in the **Executing** and **Stop Pending** states, and briefly in the **Defining** state for import jobs.

Detaching while a job is in the **Executing** state does not halt the job. You can reattach to a running job at any time to resume obtaining status information about the job.

A Detach can occur explicitly, when the `DBMS_DATAPUMP.DETACH` procedure is run, or it can occur implicitly when an Oracle Data Pump API session is run down, when the Oracle Data Pump API is unable to communicate with an Oracle Data Pump job, or when the `DBMS_DATAPUMP.STOP_JOB` procedure is run.

The **Not Running** state indicates that a Data Pump control job table exists outside the context of a running job. This state occurs if a job is stopped (and likely can restart later), or if a job has terminated in an unusual way. You can also see this state momentarily during job state transitions at the beginning of a job, and at the end of a job before the Oracle Data Pump control job table is dropped. Note that the **Not Running** state is shown only in the views `DBA_DATAPUMP_JOBS` and `USER_DATAPUMP_JOBS`. It is never returned by the `GET_STATUS` procedure.

The following table shows the valid job states in which `DBMS_DATAPUMP` procedures can be run. The states listed are valid for both export and import jobs, unless otherwise noted.

Table 6-1 Valid Job States in Which DBMS_DATAPUMP Procedures Can Be Run

Procedure Name	Valid States	Description
ADD_FILE	**Defining** (valid for both export and import jobs) **Executing** and **Idling** (valid only for specifying dump files for export jobs)	Specifies a file for the dump file set, the log file, or the SQLFILE output.
ATTACH	**Defining**, **Executing**, **Idling**, **Stopped**, **Completed**, **Completing**, **Not Running**	Enables a user session to monitor a job, or to restart a stopped job. If the dump file set or Data Pump control job table for the job have been deleted or altered in any way, then the attach fails.
DATA_FILTER	**Defining**	Restricts data processed by a job.
DETACH	All	Disconnects a user session from a job.
GET_DUMPFILE_INFO	All	Retrieves dump file header information.
GET_STATUS	All, except **Completed**, **Not Running**, **Stopped**, and **Undefined**	Obtains the status of a job.
LOG_ENTRY	**Defining**, **Executing**, **Idling**, Stop **Pending**, **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** Note: You can enter the ENCRYPTION_PASSWORD parameter during the **Defining** and **Idling** states.	Alters default processing by a job.
START_JOB	**Defining**, **Idling**	Begins or resumes execution of a job.
STOP_JOB	**Defining**, **Executing**, **Idling**, Stop **Pending**	Initiates shutdown of a job.
WAIT_FOR_JOB	All, except **Completed**, **Not Running**, **Stopped**, and **Undefined**	Waits for a job to end.

6.2 What Are the Basic Steps in Using the Oracle Data Pump API?

To use the Oracle 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, including the point in running an Oracle Data Pump job in which you can perform optional steps. The steps are presented in the order in which you would generally perform the activities.

1. To create an Oracle Data Pump job and its infrastructure, run the `DBMS_DATAPUMP.OPEN` procedure.

   When you run the procedure, the Oracle Data Pump job is started.
2. Define any parameters for the job.
3. Start the job.
4. (Optional) Monitor the job until it completes.
5. (Optional) Detach from the job, and reattach at a later time.
6. (Optional) Stop the job.
7. (Optional) Restart the job, if desired.

Related Topics

- *Oracle Database PL/SQL Packages and Types Reference*

6.3 Examples of Using the Data Pump API

Using the Data Pump API.

This section provides the following examples to help you get started using the Data Pump API:

- Performing a Simple Schema Export (Example 6-1)
- Importing a Dump File and Remapping All Schema Objects (Example 6-2)
- Using Exception Handling During a Simple Schema Export (Example 6-3)
- Displaying Dump File Information (Example 6-4)

The examples are in the form of PL/SQL scripts. If you choose to copy these scripts and run them, then 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, then 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, then you will not see any output to your screen. You must do this in the same session in which you run the example. If you exit SQL*Plus, then this setting is lost and must be reset when you begin a new session. (It must also be reset if you connect to a different user name.)

Example 6-1 Performing a Simple Schema Export

The PL/SQL script 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.

Connect as user `SYSTEM` to use this script.

```
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 6-2 Importing a Dump File and Remapping All Schema Objects

The script in this example imports the dump file created in Example 6-1 (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.

Connect as user `SYSTEM` to use this script.

```
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 6-3 Using Exception Handling During a Simple Schema Export

The script in this example shows a simple schema export using the Data Pump API. It extends Example 6-1 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. To get exception information about a `DBMS_DATAPUMP.OPEN` or `DBMS_DATAPUMP.ATTACH` failure, call `DBMS_DATAPUMP.GET_STATUS` with a `DBMS_DATAPUMP.KU$_STATUS_JOB_ERROR` information mask and a NULL job handle to retrieve the error details.

Connect as user `SYSTEM` to use this example.

```
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,
-- 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;

-- 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;
/
```

Example 6-4 Displaying Dump File Information

The PL/SQL script in this example shows how to use the Data Pump API procedure `DBMS_DATAPUMP.GET_DUMPFILE_INFO` to display information about a Data Pump dump file outside the context of any Data Pump job. This example displays information contained in the `example1.dmp` dump file created by the sample PL/SQL script in Example 6-1.

This PL/SQL script can also be used to display information for dump files created by original Export (the `exp` utility) as well as by the `ORACLE_DATAPUMP` external tables access driver.

Connect as user `SYSTEM` to use this script.

```
SET VERIFY OFF
SET FEEDBACK OFF

DECLARE
  ind        NUMBER;
  fileType   NUMBER;
  value      VARCHAR2(2048);
  infoTab    KU$_DUMPFILE_INFO := KU$_DUMPFILE_INFO();

BEGIN
  --
  -- Get the information about the dump file into the infoTab.
  --
  BEGIN
    DBMS_DATAPUMP.GET_DUMPFILE_INFO('example1.dmp','DMPDIR',infoTab,fileType);
    DBMS_OUTPUT.PUT_LINE('----------------------------------------------');
    DBMS_OUTPUT.PUT_LINE('Information for file: example1.dmp');

    --
    -- Determine what type of file is being looked at.
    --
    CASE fileType
      WHEN 1 THEN
        DBMS_OUTPUT.PUT_LINE('example1.dmp is a Data Pump dump file');
      WHEN 2 THEN
        DBMS_OUTPUT.PUT_LINE('example1.dmp is an Original Export dump file');
      WHEN 3 THEN
        DBMS_OUTPUT.PUT_LINE('example1.dmp is an External Table dump file');
      ELSE
        DBMS_OUTPUT.PUT_LINE('example1.dmp is not a dump file');
        DBMS_OUTPUT.PUT_LINE('----------------------------------------------');
    END CASE;

  EXCEPTION
    WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('----------------------------------------------');
      DBMS_OUTPUT.PUT_LINE('Error retrieving information for file: ' ||
                           'example1.dmp');
      DBMS_OUTPUT.PUT_LINE(SQLERRM);
      DBMS_OUTPUT.PUT_LINE('----------------------------------------------');
      fileType := 0;
  END;

  --
  -- If a valid file type was returned, then loop through the infoTab and
  -- display each item code and value returned.
  --
  IF fileType > 0
  THEN
    DBMS_OUTPUT.PUT_LINE('The information table has ' ||
                         TO_CHAR(infoTab.COUNT) || ' entries');
    DBMS_OUTPUT.PUT_LINE('----------------------------------------------');

    ind := infoTab.FIRST;
    WHILE ind IS NOT NULL
    LOOP
      --
      -- The following item codes return boolean values in the form
      -- of a '1' or a '0'. Display them as 'Yes' or 'No'.
      --
```

```
value := NVL(infoTab(ind).value, 'NULL');
IF infoTab(ind).item_code IN
   (DBMS_DATAPUMP.KU$_DFHDR_MASTER_PRESENT,
    DBMS_DATAPUMP.KU$_DFHDR_DIRPATH,
    DBMS_DATAPUMP.KU$_DFHDR_METADATA_COMPRESSED,
    DBMS_DATAPUMP.KU$_DFHDR_DATA_COMPRESSED,
    DBMS_DATAPUMP.KU$_DFHDR_METADATA_ENCRYPTED,
    DBMS_DATAPUMP.KU$_DFHDR_DATA_ENCRYPTED,
    DBMS_DATAPUMP.KU$_DFHDR_COLUMNS_ENCRYPTED)
THEN
  CASE value
    WHEN '1' THEN value := 'Yes';
    WHEN '0' THEN value := 'No';
  END CASE;
END IF;

--
-- Display each item code with an appropriate name followed by
-- its value.
--
CASE infoTab(ind).item_code
  --
  -- The following item codes have been available since Oracle
  -- Database 10g, Release 10.2.
  --
  WHEN DBMS_DATAPUMP.KU$_DFHDR_FILE_VERSION   THEN
    DBMS_OUTPUT.PUT_LINE('Dump File Version:         ' || value);
  WHEN DBMS_DATAPUMP.KU$_DFHDR_MASTER_PRESENT THEN
    DBMS_OUTPUT.PUT_LINE('Master Table Present:      ' || value);
  WHEN DBMS_DATAPUMP.KU$_DFHDR_GUID THEN
    DBMS_OUTPUT.PUT_LINE('Job Guid:                  ' || value);
  WHEN DBMS_DATAPUMP.KU$_DFHDR_FILE_NUMBER THEN
    DBMS_OUTPUT.PUT_LINE('Dump File Number:          ' || value);
  WHEN DBMS_DATAPUMP.KU$_DFHDR_CHARSET_ID  THEN
    DBMS_OUTPUT.PUT_LINE('Character Set ID:          ' || value);
  WHEN DBMS_DATAPUMP.KU$_DFHDR_CREATION_DATE THEN
    DBMS_OUTPUT.PUT_LINE('Creation Date:             ' || value);
  WHEN DBMS_DATAPUMP.KU$_DFHDR_FLAGS THEN
    DBMS_OUTPUT.PUT_LINE('Internal Dump Flags:       ' || value);
  WHEN DBMS_DATAPUMP.KU$_DFHDR_JOB_NAME THEN
    DBMS_OUTPUT.PUT_LINE('Job Name:                  ' || value);
  WHEN DBMS_DATAPUMP.KU$_DFHDR_PLATFORM THEN
    DBMS_OUTPUT.PUT_LINE('Platform Name:             ' || value);
  WHEN DBMS_DATAPUMP.KU$_DFHDR_INSTANCE THEN
    DBMS_OUTPUT.PUT_LINE('Instance Name:             ' || value);
  WHEN DBMS_DATAPUMP.KU$_DFHDR_LANGUAGE THEN
    DBMS_OUTPUT.PUT_LINE('Language Name:             ' || value);
  WHEN DBMS_DATAPUMP.KU$_DFHDR_BLOCKSIZE THEN
    DBMS_OUTPUT.PUT_LINE('Dump File Block Size:      ' || value);
  WHEN DBMS_DATAPUMP.KU$_DFHDR_DIRPATH THEN
    DBMS_OUTPUT.PUT_LINE('Direct Path Mode:          ' || value);
  WHEN DBMS_DATAPUMP.KU$_DFHDR_METADATA_COMPRESSED THEN
    DBMS_OUTPUT.PUT_LINE('Metadata Compressed:       ' || value);
  WHEN DBMS_DATAPUMP.KU$_DFHDR_DB_VERSION THEN
    DBMS_OUTPUT.PUT_LINE('Database Version:          ' || value);

  --
  -- The following item codes were introduced in Oracle Database 11g
  -- Release 11.1
  --

  WHEN DBMS_DATAPUMP.KU$_DFHDR_MASTER_PIECE_COUNT THEN
```

```
        DBMS_OUTPUT.PUT_LINE('Master Table Piece Count:  ' || value);
      WHEN DBMS_DATAPUMP.KU$_DFHDR_MASTER_PIECE_NUMBER THEN
        DBMS_OUTPUT.PUT_LINE('Master Table Piece Number: ' || value);
      WHEN DBMS_DATAPUMP.KU$_DFHDR_DATA_COMPRESSED THEN
        DBMS_OUTPUT.PUT_LINE('Table Data Compressed:     ' || value);
      WHEN DBMS_DATAPUMP.KU$_DFHDR_METADATA_ENCRYPTED THEN
        DBMS_OUTPUT.PUT_LINE('Metadata Encrypted:        ' || value);
      WHEN DBMS_DATAPUMP.KU$_DFHDR_DATA_ENCRYPTED THEN
        DBMS_OUTPUT.PUT_LINE('Table Data Encrypted:      ' || value);
      WHEN DBMS_DATAPUMP.KU$_DFHDR_COLUMNS_ENCRYPTED THEN
        DBMS_OUTPUT.PUT_LINE('TDE Columns Encrypted:     ' || value);

      --
      -- For the DBMS_DATAPUMP.KU$_DFHDR_ENCRYPTION_MODE item code a
      -- numeric value is returned. So examine that numeric value
      -- and display an appropriate name value for it.
      --
      WHEN DBMS_DATAPUMP.KU$_DFHDR_ENCRYPTION_MODE THEN
        CASE TO_NUMBER(value)
          WHEN DBMS_DATAPUMP.KU$_DFHDR_ENCMODE_NONE THEN
            DBMS_OUTPUT.PUT_LINE('Encryption Mode:           None');
          WHEN DBMS_DATAPUMP.KU$_DFHDR_ENCMODE_PASSWORD THEN
            DBMS_OUTPUT.PUT_LINE('Encryption Mode:           Password');
          WHEN DBMS_DATAPUMP.KU$_DFHDR_ENCMODE_DUAL THEN
            DBMS_OUTPUT.PUT_LINE('Encryption Mode:           Dual');
          WHEN DBMS_DATAPUMP.KU$_DFHDR_ENCMODE_TRANS THEN
            DBMS_OUTPUT.PUT_LINE('Encryption Mode:           Transparent');
        END CASE;

      --
      -- The following item codes were introduced in Oracle Database 12c
      -- Release 12.1
      --

      --
      -- For the DBMS_DATAPUMP.KU$_DFHDR_COMPRESSION_ALG item code a
      -- numeric value is returned. So examine that numeric value and
      -- display an appropriate name value for it.
      --
      WHEN DBMS_DATAPUMP.KU$_DFHDR_COMPRESSION_ALG THEN
        CASE TO_NUMBER(value)
          WHEN DBMS_DATAPUMP.KU$_DFHDR_CMPALG_NONE THEN
            DBMS_OUTPUT.PUT_LINE('Compression Algorithm:     None');
          WHEN DBMS_DATAPUMP.KU$_DFHDR_CMPALG_BASIC THEN
            DBMS_OUTPUT.PUT_LINE('Compression Algorithm:     Basic');
          WHEN DBMS_DATAPUMP.KU$_DFHDR_CMPALG_LOW THEN
            DBMS_OUTPUT.PUT_LINE('Compression Algorithm:     Low');
          WHEN DBMS_DATAPUMP.KU$_DFHDR_CMPALG_MEDIUM THEN
            DBMS_OUTPUT.PUT_LINE('Compression Algorithm:     Medium');
          WHEN DBMS_DATAPUMP.KU$_DFHDR_CMPALG_HIGH THEN
            DBMS_OUTPUT.PUT_LINE('Compression Algorithm:     High');
        END CASE;
      ELSE NULL;  -- Ignore other, unrecognized dump file attributes.
    END CASE;
    ind := infoTab.NEXT(ind);
  END LOOP;
 END IF;
END;
/
```

Part II
SQL*Loader

Learn about SQL*Loader and its features, as well as data loading concepts, including object support.

- Understanding How to Use SQL*Loader
  Learn about the basic concepts you should understand before loading data into an Oracle Database using SQL*Loader.
- SQL*Loader Command-Line Reference
  To start regular SQL*Loader, use the command-line parameters.
- SQL*Loader Control File Reference
  The SQL*Loader control file is a text file that contains data definition language (DDL) instructions for a SQL*Loader job.
- SQL*Loader Field List Reference
  The field-list portion of a SQL*Loader control file provides information about fields being loaded, such as position, data type, conditions, and delimiters.
- Loading Objects, LOBs, and Collections with SQL*Loader
  You can use SQL*Loader to load column objects in various formats and to load object tables, REF columns, LOBs, and collections.
- Conventional and Direct Path Loads
  SQL*Loader provides the option to load data using a conventional path load method, and a direct path load method.
- SQL*Loader Express
  SQL*Loader express mode allows you to quickly and easily use SQL*Loader to load simple data types.

7

Understanding How to Use SQL*Loader

Learn about the basic concepts you should understand before loading data into an Oracle Database using SQL*Loader.

- SQL*Loader Features
  SQL*Loader loads data from external files into tables of an Oracle database.
- SQL*Loader Parameters
  SQL*Loader is started either when you specify the `sqlldr` command, or when you specify parameters that establish various characteristics of the load operation.
- SQL*Loader Control File
  The control file is a text file written in a language that SQL*Loader understands.
- How SQL*Loader Reads Input Data and Data Files
  SQL*Loader reads data from one or more data files (or operating system equivalents of files) specified in the control file.
- LOBFILEs and Secondary Data Files (SDFs)
  Large Object (LOB) data can be lengthy enough that it makes sense to load it from a LOBFILE.
- Data Conversion and Data Type Specification
  During a conventional path load, *data fields* in the data file are converted into *columns* in the database (direct path loads are conceptually similar, but the implementation is different).
- SQL*Loader Discarded and Rejected Records
  SQL*Loader can reject or discard some records read from the input file, either because of issues with the files, or because you have selected to filter the records out of the load.
- Log File and Logging Information
  When SQL*Loader begins processing, it creates a **log file.**
- Conventional Path Loads, Direct Path Loads, and External Table Loads
  SQL*Loader provides several methods to load data.
- Loading Objects, Collections, and LOBs with SQL*Loader
  You can bulk-load the column, row, LOB, and JSON database objects that you need to model real-world entities, such as customers and purchase orders.
- Partitioned Object Support in SQL*Loader
  Partitioned database objects enable you to manage sections of data, either collectively or individually. SQL*Loader supports loading partitioned objects.
- Application Development: Direct Path Load API
  Direct path loads enable you to load data from external files into tables and partitions.Oracle provides a direct path load API for application developers.
- SQL*Loader Case Studies
  To learn how you can use SQL*Loader features, you can run a variety of case studies that Oracle provides.

7.1 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 data file. You can use SQL*Loader to do the following:

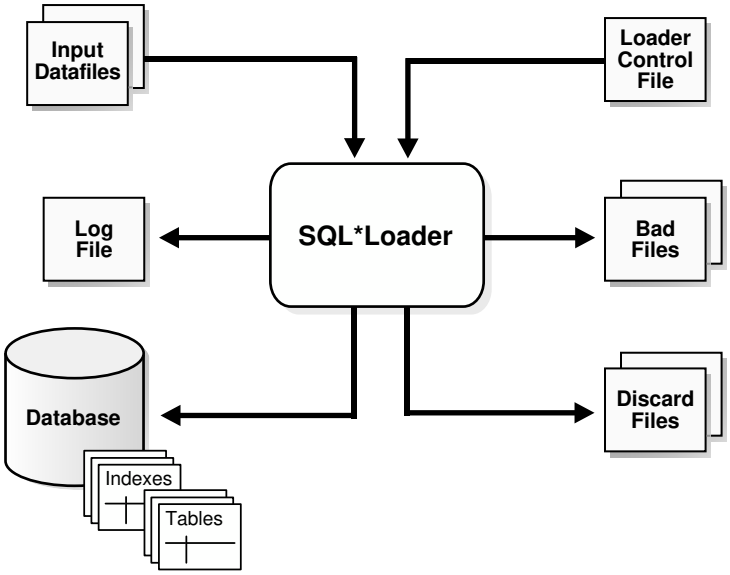
- Load data across a network if your data files are on a different system than the database.
- Load data from multiple data files during the same load session.
- Load data into multiple tables during the same load session.
- Specify the character set of the data.
- Selectively load data (you can load 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 data files.
- Load data from disk, tape, or named pipe.
- Generate sophisticated error reports, which greatly aid troubleshooting.
- Load arbitrarily complex object-relational data.
- Use secondary data files for loading LOBs and collections.
- Use conventional, direct path, or external table loads. See Conventional Path Loads_ Direct Path Loads_ and External Table Loads.

You can use SQL*Loader in two ways: with or without a control file. A control file controls the behavior of SQL*Loader and one or more data files used in the load. Using a control file gives you more control over the load operation, which might be desirable for more complicated load situations. But for simple loads, you can use SQL*Loader without specifying a control file; this is referred to as SQL*Loader express mode. See SQL*Loader Express .

The output of SQL*Loader is an Oracle database (where the data is loaded), a log file, a bad file if there are rejected records, and potentially, a discard file.

The following figure shows an example of the flow of a typical SQL*Loader session that uses a control file.

Figure 7-1 SQL*Loader Overview



7.2 SQL*Loader Parameters

SQL*Loader is started either when you specify the `sqlldr` command, or when you specify parameters that establish various characteristics of the load operation.

In situations where you always use the same parameters for which the values seldom change, it can be more efficient to specify parameters by using the following methods, rather than on the command line:

- You can group parameters together in a parameter file. You can then specify the name of the parameter file on the command line by using the `PARFILE` parameter.
- You can specify some parameters 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.

Related Topics

- SQL*Loader Command-Line Reference
  To start regular SQL*Loader, use the command-line parameters.
- PARFILE
  The `PARFILE` SQL*Loader command-line parameter specifies the name of a file that contains commonly used command-line parameters.
- OPTIONS Clause
  The following command-line parameters can be specified using the `OPTIONS` clause.

7.3 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.

In general, the control file has three main sections, in the following order:

- Session-wide information
- Table and field-list information
- Input data (optional section)

Some control file syntax considerations to keep in mind are:

- The syntax is free-format (statements can extend over multiple lines).
- The syntax 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.

Related Topics

- SQL*Loader Control File Reference
  The SQL*Loader control file is a text file that contains data definition language (DDL) instructions for a SQL*Loader job.

7.4 How SQL*Loader Reads Input Data and Data Files

SQL*Loader reads data from one or more data files (or operating system equivalents of files) specified in the control file.

From SQL*Loader's perspective, the data in the data file is organized as *records*. A particular data file 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, then 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 data file are the same byte length.
- Variable Record Format and SQL*Loader
  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 data file.
- Stream Record Format and SQL*Loader
  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**.

- Logical Records and SQL*Loader
  SQL*Loader organizes input data into physical records, according to the specified record format. By default, a physical record is a logical record.
- Data Field Setting and SQL*Loader
  Learn how SQL*Loader determines the field setting on the logical record after a logical record is formed.

7.4.1 Fixed Record Format

A file is in fixed record format when all records in a data file are the same byte length.

Although the fixed record format is the least flexible format, using 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 data file as being in fixed record format where every record is `n` bytes long.

The following example shows a control file that specifies a data file (`example1.dat`) to be interpreted in the fixed record format. The data file in the example contains five physical records; each record has fields that contain the number and name of an employee. Each of the five records is 11 bytes long, including spaces. For the purposes of explaining this example, periods are used to represent spaces in the records, but in the actual records there would be no periods. With that in mind, the first physical record is `396,...ty,.` which is exactly eleven bytes (assuming a single-byte character set). The second record is `4922,beth,` followed by the newline character (`\n`) which is the eleventh byte, and so on. (Newline characters are not required with the fixed record format; it is simply used here to illustrate that if used, it counts as a byte in the record length.)

Example 7-1 Loading Data in Fixed Record Format

Loading data:

```
load data
infile 'example1.dat'  "fix 11"
into table example
fields terminated by ',' optionally enclosed by '"'
(col1, col2)
```

Contents of `example1.dat`:

```
396,...ty,.4922,beth,\n
68773,ben,.
1,..."dave",
5455,mike,.
```

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 can contain a mix of fields. Some are processed with character-length semantics, and others are processed with byte-length semantics.

Related Topics

- Character-Length Semantics
  Byte-length semantics are the default for all data files except those that use the UTF16 character set (which uses character-length semantics by default).

7.4.2 Variable Record Format and SQL*Loader

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 data file.

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 data file 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, then SQL*Loader assumes a length of 5 bytes. Specifying *n* larger than 40 results in an error.

The following example shows a control file specification that tells SQL*Loader to look for data in the data file `example2.dat` and to expect variable record format where the record's first three bytes indicate the length of the field. The `example2.dat` data file consists of three physical records. The first is specified to be 009 (9) bytes long, the second is 010 (10) bytes long (plus a 1-byte newline), and the third is 012 (12) bytes long (plus 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 data file. For the purposes of this example, periods in `example2.dat` represent spaces; the fields do not contain actual periods.

Example 7-2 Loading Data in Variable Record Format

Loading data:

```
load data
infile 'example2.dat'  "var 3"
into table example
fields terminated by ',' optionally enclosed by '"'
(col1 char(5),
 col2 char(7))
```

Contents of `example2.dat`:

```
009.396,.ty,0104922,beth,01268773,benji,
```

Note that the lengths are always interpreted in bytes, even if character-length semantics are in effect for the file. This is necessary because the file can contain a mix of fields, some processed with character-length semantics and others processed with byte-length semantics.

Related Topics

- Character-Length Semantics
  Byte-length semantics are the default for all data files except those that use the UTF16 character set (which uses character-length semantics by default).

7.4.3 Stream Record Format and SQL*Loader

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 using it can result in a negative effect on performance. The specification of a data file to be interpreted as being in stream record format looks similar to the following:

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

In the preceding example, `str` indicates that the file is in stream record format. 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*' byte string. However, you can specify some nonprintable characters 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` initialization parameter for your session is different from the character set of the data file, then character strings are converted to the character set of the data file. This is done before SQL*Loader checks for the default record terminator.

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

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

On Windows-based platforms, 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 data file. This means that if you know that one or more records in your data file has `\n` embedded in a field, but you want `\r\n` to be used as the record terminator, then you must specify it.

The following example 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.

> **See Also:**
>
> - *Oracle Database Globalization Support Guide* for information about using the Language and Character Set File Scanner (LCSSCAN) utility to determine the language and character set for unknown file text

Example 7-3 Loading Data in Stream Record Format

Loading data:

```
load data
infile 'example3.dat'  "str '|\n'"
into table example
fields terminated by ',' optionally enclosed by '"'
(col1 char(5),
 col2 char(7))


example3.dat

396,ty,|
4922,beth,|
```

7.4.4 Logical Records and SQL*Loader

SQL*Loader organizes input data into physical records, according to the specified record format. By default, a physical record is a logical record.

For added flexibility, SQL*Loader can be instructed to combine several 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.

Related Topics

- Assembling Logical Records from Physical Records
  This section describes assembling logical records from physical records.
- SQL*Loader Case Studies
  To learn how you can use SQL*Loader features, you can run a variety of case studies that Oracle provides.

7.4.5 Data Field Setting and SQL*Loader

Learn how SQL*Loader determines the field setting on the logical record after a logical record is formed.

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, terminating, or both) 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.
- You can specify the byte offset, the length of the data field, or both. This way each field starts a specified number of bytes from where the last one ended and continues for a specified length.
- Length-value data types 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.

Related Topics

- Specifying Delimiters
  The boundaries of `CHAR`, datetime, interval, or numeric `EXTERNAL` fields can also be marked by delimiter characters contained in the input data record.

7.5 LOBFILEs and Secondary Data Files (SDFs)

Large Object (LOB) data can be lengthy enough that it makes sense to load it from a LOBFILE.

With LOBFILEs, LOB data instances are still considered to be in fields (predetermined size, delimited, length-value). However, 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, suppose you have a table that stores employee names, IDs, and their resumes. When loading this table, you can read the employee names and IDs from the main data files and you can read the resumes, which can be quite lengthy, from LOBFILEs.

You can 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 data files (SDFs) are similar in concept to primary data files. As with primary data files, SDFs are a collection of records, and each record is made up of fields. The SDFs are specified as needed for a control file field. Only a `collection_fld_spec` can name an SDF as its data source.

You specify SDFs by using the `SDF` parameter. You can enter a value for the `SDF` parameter either by using the file specification string, or by using a `FILLER` field that is mapped to a data field containing one or more file specification strings.

Related Topics

- 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 data file.
- Secondary Data Files (SDFs)
  When you need to load large nested tables and `VARRAY`s, you can use secondary data files (SDFs). They are similar in concept to primary data files.

7.6 Data Conversion and Data Type Specification

During a conventional path load, *data fields* in the data file 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 data file, parse the input data, and populate the bind arrays that correspond to a SQL `INSERT` statement using that data. A bind array is an area in memory where SQL*Loader stores data that is to be loaded. When the bind array is full, the data is transmitted to the database. The bind array size is controlled by the SQL*Loader `BINDSIZE` and `READSIZE` parameters.
2. The database accepts the data and executes the `INSERT` statement to store the data in the database.

Oracle Database uses the data type of the column to convert the data into its final, stored form. Keep in mind the distinction between a *field* in a data file and a *column* in the database. Remember also that the field data types defined in a SQL*Loader control file are *not* the same as the column data types.

See Also:

- BINDSIZE
- READSIZE

7.7 SQL*Loader Discarded and Rejected Records

SQL*Loader can reject or discard some records read from the input file, either because of issues with the files, or because you have selected to filter the records out of the load.

Rejected records are placed in a bad file, and discarded records are placed in a discard file.

- The SQL*Loader Bad File
  The bad file contains records that were rejected, either by SQL*Loader or by Oracle Database.
- The SQL*Loader Discard File
  As SQL*Loader runs, it can filter some records out of the load, and create a file called the discard file.

7.7.1 The SQL*Loader Bad File

The bad file contains records that were rejected, either by SQL*Loader or by Oracle Database.

If you do not specify a bad file, and there are rejected records, then SQL*Loader automatically creates one. A rejected record has the same name as the data file, with a `.bad` extension. There can be several causes for rejections.

- Records Rejected by SQL*Loader
  Data file records are rejected by SQL*Loader when the input format is invalid.

- Records Rejected by Oracle Database During a SQL*Loader Operation
  After a data file record is accepted for processing by SQL*Loader, it is sent to the database for insertion into a table as a row.

7.7.1.1 Records Rejected by SQL*Loader

Data file 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, then SQL*Loader rejects the record. Rejected records are placed in the bad file.

7.7.1.2 Records Rejected by Oracle Database During a SQL*Loader Operation

After a data file record is accepted for processing by SQL*Loader, it is sent to the database for insertion into a table as a row.

If the 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 data type.

7.7.2 The SQL*Loader Discard File

As SQL*Loader runs, it can filter some records out of the load, and create a file called the discard file.

A discard 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.

Because the discard file contains record filtered out of the load, the contents of the discard file are 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.

7.8 Log File and Logging Information

When SQL*Loader begins processing, it creates a **log file.**

If SQL*Loader cannot create a log file, then processing terminates. The log file contains a detailed summary of the load, including a description of any errors that occurred during the load.

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

SQL*Loader provides several methods to load data.

- 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 (an area in memory where SQL*Loader stores data to be loaded).

- Direct Path Loads
  A direct path load parses the input records according to the field specifications, converts the input field data to the column data type, and builds a column array.
- Parallel Direct Path
  A parallel direct path load allows multiple direct path load sessions to concurrently load the same data segments (allows intrasegment parallelism).
- External Table Loads
  External tables are defined as tables that do not reside in the database, and can be in any format for which an access driver is provided.
- Choosing External Tables Versus SQL*Loader
  Learn which method can provide the best load performance for your data load situations.
- Behavior Differences Between SQL*Loader and External Tables
  Oracle recommends that you review the differences between loading data with external tables, using the `ORACLE_LOADER` access driver, and loading data with SQL*Loader conventional and direct path loads.
- Loading Tables Using Data Stored into Object Storage
  Learn how to load your data from Object Storage into standard Oracle Database tables using SQL*Loader.

7.9.1 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 (an area in memory where SQL*Loader stores data to be loaded).

When the bind array is full (or no more data is left to read), an array insert operation is performed.

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), then 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 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.

See Also:

- Data Loading Methods
- Bind Arrays and Conventional Path Loads

7.9.2 Direct Path Loads

A direct path load parses the input records according to the field specifications, converts the input field data to the column data type, 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.

7.9.3 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
>
> Direct Path Load

7.9.4 External Table Loads

External tables are defined as tables that do not reside in the database, and can be in any format for which an access driver is provided.

Oracle Database provides two access drivers: `ORACLE_LOADER`, and `ORACLE_DATAPUMP`. By providing the database with metadata describing an external table, the database is able to expose the data in the external table as if it were data residing in a regular database table.

An external table load creates an external table for data that is contained in an external data file. The load runs `INSERT` statements to insert the data from the data file into the target table.

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

- If a data file is big enough, then an external table load attempts 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.

> **Note:**
>
> An external table load is not supported using a named pipe on Windows operating systems.

Related Topics

- The ORACLE_LOADER Access Driver
  Learn how to control the way external tables are accessed by using the ORACLE_LOADER access driver parameters to modify the default behavior of the access driver.

- The ORACLE_DATAPUMP Access Driver
  The `ORACLE_DATAPUMP` access driver provides a set of access parameters that are unique to external tables of the type `ORACLE_DATAPUMP`.
- Managing External Tables in *Oracle Database Administrator's Guide*

7.9.5 Choosing External Tables Versus SQL*Loader

Learn which method can provide the best load performance for your data load situations.

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 may be more appropriate than the other.

Use external tables for the best load performance in the following situations:

- 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

Use SQL*Loader for the best load performance in the following situations:

- You want to load data remotely
- Transformations are not required on the data, and the data does not need to be loaded in parallel
- You want to load data, and additional indexing of the staging table is required

7.9.6 Behavior Differences Between SQL*Loader and External Tables

Oracle recommends that you review the differences between loading data with external tables, using the `ORACLE_LOADER` access driver, and loading data with SQL*Loader conventional and direct path loads.

The information in this section does not apply to the `ORACLE_DATAPUMP` access driver.

- Multiple Primary Input Data Files
  If there are multiple primary input data files with SQL*Loader loads, then a bad file and a discard file are created for each input data file.
- Syntax and Data Types
  This section provides a description of unsupported syntax and data types with external table loads.
- Byte-Order Marks
  With SQL*Loader, whether the byte-order mark is written depends on the character set or on the table load.
- Default Character Sets, Date Masks, and Decimal Separator
  The display of NLS character sets are controlled by different settings for SQL*Loader and external tables.
- Use of the Backslash Escape Character
  SQL*Loader and external tables use different conventions to identify single quotation marks as an enclosure character.

7.9.6.1 Multiple Primary Input Data Files

If there are multiple primary input data files with SQL*Loader loads, then a bad file and a discard file are created for each input data file.

With external table loads, there is only one bad file and one discard file for all input data files. If parallel access drivers are used for the external table load, then each access driver has its own bad file and discard file.

7.9.6.2 Syntax and Data Types

This section provides a description of unsupported syntax and data types 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 data types: `GRAPHIC`, `GRAPHIC EXTERNAL`, and `VARGRAPHIC`
- Use of the following database column types: `LONG`, nested table, `VARRAY`, `REF`, primary key `REF`, and `SID`

7.9.6.3 Byte-Order Marks

With SQL*Loader, whether the byte-order mark is written depends on the character set or on the table load.

If a primary data file 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.

7.9.6.4 Default Character Sets, Date Masks, and Decimal Separator

The display of NLS character sets are controlled by different settings for SQL*Loader and external tables.

With SQL*Loader, the default character set, date mask, and decimal separator are determined by the settings of NLS environment variables on the client.

For fields in external tables, the database settings of the NLS parameters determine the default character set, date masks, and decimal separator.

7.9.6.5 Use of the Backslash Escape Character

SQL*Loader and external tables use different conventions to identify single quotation marks as an enclosure character.

With SQL*Loader, to identify a single quotation mark as the enclosure character, you can use the backslash (`\`) escape character. For example

```
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '\''
```

In external tables, the use of the backslash escape character within a string raises an error. The workaround is to use double quotation marks to identify a single quotation mark as the enclosure character. For example:

```
TERMINATED BY ',' ENCLOSED BY "'"
```

7.9.7 Loading Tables Using Data Stored into Object Storage

Learn how to load your data from Object Storage into standard Oracle Database tables using SQL*Loader.

In the following example, you have a table (`T`) into which you are loading data:

```
SQL> create table t (x int, y int);
```

You have a data file that you want to load to this table, named `file1.txt`. The contents are as follows:

```
X,Y
1,2
4,5
```

To load this table into an object store, complete the following procedure:

1. Install the libraries required to enable object store input/output (I/O):

   ```
   % cd $ORACLE_HOME/rdbms/lib
   % make -f ins_rdbms.mk opc_on
   ```

2. Upload the file `file1.txt` to the bucket in Object Storage.

   The easiest way to upload file to object storage is to upload the file from the Oracle Cloud console:

   a. Open the Oracle Cloud console.
   b. Select the Object Storage tile.
   c. If not already created, create a bucket.
   d. Click **Upload**, and select the file `file1.txt` to upload it into the bucket.

3. In Oracle Database, create the wallet and the credentials.

   For example:

   ```
   $ orapki wallet create -wallet /home/oracle/wallets  -pwd mypassword-
   auto_login
   $ mkstore -wrl /home/oracle/wallets -createEntry
   oracle.sqlldr.credential.myfedcredential.username
   oracleidentitycloudservice/myuseracct@example.com
   $ mkstore -wrl /home/oracle/wallets -createEntry
   oracle.sqlldr.credential.myfedcredential.password "MhAVCDfW+-ReskK4:Ho-
   zH"
   ```

   This example shows the use of a federated user account (*myfedcredential*). The password is automatically generated, as described in Oracle Cloud Infrastructure Documentation. "Managing Credentials," in the section "To create an auth token."

4. After creating the wallet, add the location in the `sqlnet.ora` file in the directory `$ORACLE_HOME/network/admin` directory.
   For example:

```
vi test.ctl
LOAD DATA
INFILE  'https://objectstorage.eu-frankfurt-1.oraclecloud.com/n/
dbcloudoci/b/myobjectstore/o/file1.txt'
truncate
INTO TABLE T
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
(X,Y)
```

5. Run SQL*Loader to load the data into the object store.

   For example:

```
sqlldr test/mypassword@pdb1 /home/oracle/test.ctl
credential=myfedcredentiallog=test.log  external_table=not_used
```

Related Topics

- "Managing Credentials: To create an auth token," Oracle Cloud Infrastructure Documentation
- Using the Console, Oracle Cloud Infrastructure Documentation

7.10 Loading Objects, Collections, and LOBs with SQL*Loader

You can bulk-load the column, row, LOB, and JSON database objects that you need to model real-world entities, such as customers and purchase orders.

- Supported Object Types
  SQL*Loader supports loading of the column and row object types.
- Supported Collection Types
  SQL*Loader supports loading of nested tables and `VARRAY` collection types.
- Supported LOB Data Types
  A LOB is a large object type.

7.10.1 Supported Object Types

SQL*Loader supports loading of the column and row 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.
- row objects
  These objects are stored in tables, known as object tables, that have columns corresponding to the attributes of the object.

7.10.1.1 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.

7.10.1.2 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
> - Loading Object Tables

7.10.2 Supported Collection Types

SQL*Loader supports loading of nested tables and `VARRAY` collection types.

- Nested Tables
  A nested table is a table that appears as a column in another table.
- VARRAYs
  A `VARRAY` is a variable sized arrays.

7.10.2.1 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.

7.10.2.2 VARRAYs

A `VARRAY` is a 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 you create a `VARRAY` type, you must specify the maximum size. Once you have declared a `VARRAY` type, it can be used as the data type 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) for details on using SQL*Loader control file data definition language to load these collection types

7.10.3 Supported LOB Data Types

A LOB is a large object type.

This release of SQL*Loader supports loading of four LOB data 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 data types, and except for `NCLOB`, they can be an object's attribute data types. LOBs can have an actual value, they can be `null`, or they can be "empty."

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

7.11 Partitioned Object Support in SQL*Loader

Partitioned database objects enable you to manage sections of data, either collectively or individually. SQL*Loader supports loading partitioned objects.

A **partitioned object** in Oracle Database instances is a table or index consisting of partitions (pieces) that have been grouped, typically by common logical attributes. For example, sales data for a particular year 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

7.12 Application Development: Direct Path Load API

Direct path loads enable you to load data from external files into tables and partitions.Oracle provides a direct path load API for application developers.

Related Topics

- *Oracle Call Interface Programmer's Guide*

7.13 SQL*Loader Case Studies

To learn how you can use SQL*Loader features, you can run a variety of case studies that Oracle provides.

- Case Study Files
  Each of the SQL*Loader case study files has a set of files required to use that case study
- Running the Case Studies
  The typical steps for running SQL*Loader case studies is similar for all of the cases.
- Case Study Log Files
  Log files for the case studies are not provided in the `$ORACLE_HOME/rdbms/demo` directory.
- Checking the Results of a Case Study
  To check the results of running a case study, start SQL*Plus and perform a select operation from the table that was loaded in the case study.

7.13.1 Case Study Files

Each of the SQL*Loader case study files has a set of files required to use that case study

Usage Notes

Generally, each case study is comprised of the following types of files:

- Control files (for example, `ulcase5.ctl`)
- Data files (for example, `ulcase5.dat`)
- Setup files (for example, `ulcase5.sql`)

These files are installed when you install the Oracle Database Examples (formerly Companion) media. They are installed in the directory `$ORACLE_HOME/rdbms/demo`.

If the example data for the case study is contained within the control file, then there is no `.dat` file for that case.

Case study 2 does not require any special set up, so there is no `.sql` script for that case. Case study 7 requires that you run both a starting (setup) script and an ending (cleanup) script.

The following table lists the files associated with each case:

Table 7-1 Case Studies and Their Related Files

Case	.ctl	.dat	.sql
1	ulcase1.ctl	N/A	ulcase1.sql
2	ulcase2.ctl	ulcase2.dat	N/A
3	ulcase3.ctl	N/A	ulcase3.sql
4	ulcase4.ctl	ulcase4.dat	ulcase4.sql
5	ulcase5.ctl	ulcase5.dat	ulcase5.sql
6	ulcase6.ctl	ulcase6.dat	ulcase6.sql

Table 7-1 (Cont.) Case Studies and Their Related Files

Case	.ctl	.dat	.sql
7	ulcase7.ctl	ulcase7.dat	ulcase7s.sql ulcase7e.sql
8	ulcase8.ctl	ulcase8.dat	ulcase8.sql
9	ulcase9.ctl	ulcase9.dat	ulcase9.sql
10	ulcase10.ctl	N/A	ulcase10.sql
11	ulcase11.ctl	ulcase11.dat	ulcase11.sql

7.13.2 Running the Case Studies

The typical steps for running SQL*Loader case studies is similar for all of the cases.

Be sure you are in the `$ORACLE_HOME/rdbms/demo` directory, which is where the case study files are located.

Also, be sure to read the control file for each case study before you run it. The beginning of the control file contains information about what is being demonstrated in the case study, and any other special information you need to know. For example, case study 6 requires that you add `DIRECT=TRUE` to the SQL*Loader command line.

1. At the system prompt, type `sqlplus` and press Enter to start SQL*Plus. At the user-name prompt, enter `scott`. At the password prompt, enter `tiger`.

   The SQL prompt is displayed.

2. At the SQL prompt, execute the SQL script for the case study. :

   For example, to execute the SQL script for case study 1, enter the following command:

   ```
   SQL> @ulcase1
   ```

   This command prepares and populates tables for the case study and then returns you to the system prompt.

3. At the system prompt, start SQL*Loader and run the case study.

   For example, to run case 1, enter the following command:

   ```
   sqlldr USERID=scott CONTROL=ulcase1.ctl LOG=ulcase1.log
   ```

   Substitute the appropriate control file name and log file name for the `CONTROL` and `LOG` parameters, and press **Enter**. When you are prompted for a password, type `tiger` and then press **Enter**.

7.13.3 Case Study Log Files

Log files for the case studies are not provided in the `$ORACLE_HOME/rdbms/demo` directory.

This is because the log file for each case study is produced when you execute the case study, provided that you use the `LOG` parameter. If you do not want to produce a log file, then omit the `LOG` parameter from the command line.

7.13.4 Checking the Results of a Case Study

To check the results of running a case study, start SQL*Plus and perform a select operation from the table that was loaded in the case study.

1. At the system prompt, type `sqlplus` and press Enter to start SQL*Plus. At the user-name prompt, enter `scott`. At the password prompt, enter `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 you load the table `emp`, then enter the following statement:

   ```
   SQL> SELECT * FROM emp;
   ```

   The contents of each row in the `emp` table are displayed.

8

SQL*Loader Command-Line Reference

To start regular SQL*Loader, use the command-line parameters.

> **Note:**
>
> Regular SQL*Loader and SQL*Loader Express mode share some of the same parameters, but the behavior of these parameters can be different for each utility. The parameter descriptions described here are for regular SQL*Loader. For SQL*Loader Express options, refer to the SQL*Loader Express parameters.

- Starting SQL*Loader
  Learn how to start SQL*Loader, and how to specify parameters that manage how the load is run.
- Command-Line Parameters for SQL*Loader
  Manage SQL*Loader by using the command-line parameters.
- Exit Codes for Inspection and Display
  Oracle SQL*Loader provides the results of a SQL*Loader run immediately upon completion.

8.1 Starting SQL*Loader

Learn how to start SQL*Loader, and how to specify parameters that manage how the load is run.

To display a help screen that lists all SQL*Loader parameters, enter `sqlldr` at the prompt. and press **Enter**. The output shows each parameter, including default values for parameters, and a brief description of each parameter.

- Specifying Parameters on the Command Line
  When you start SQL*Loader, you specify parameters to establish various characteristics of the load operation.
- Alternative Ways to Specify SQL*Loader Parameters
  Learn how you can move some command-line parameters into the control file, or place commonly used parameters in a parameter file.
- Using SQL*Loader to Load Data Across a Network
  To use SQL*Loader to load data across a network connection, you can specify a connect identifier in the connect string when you start the SQL*Loader utility.

8.1.1 Specifying Parameters on the Command Line

When you start SQL*Loader, you specify parameters to establish various characteristics of the load operation.

To see how to specify SQL*Loader parameters, refer to the following examples:

You can separate the parameters by commas. However, it is not required to delimit parameters by commas:

```
> sqlldr CONTROL=ulcase1.ctl LOG=ulcase1.log
Username: scott
Password: password
```

Specifying by position means that you enter a value, but not the parameter name. In the following example, the username `scott` is provided, and then the name of the control file, `ulcase1.ctl`. You are prompted for the password:

```
> sqlldr scott ulcase1.ctl
Password: password
```

After a parameter name is used, you must supply parameter names for all subsequent specifications. No further positional specification is allowed. For example, in the following command, the `CONTROL` parameter is used to specify the control file name, but then the log file name is supplied without the `LOG` parameter, even though the LOG parameter was previously specified. Submitting this command now results in an error, even though the position of `ulcase1.log` is correct:

```
> sqlldr scott CONTROL=ulcase1.ctl ulcase1.log
```

For the command to run, you must enter the command with the `LOG` parameter specifically specified:

```
> sqlldr scott CONTROL=ulcase1.ctl LOG=ulcase1.log
```

8.1.2 Alternative Ways to Specify SQL*Loader Parameters

Learn how you can move some command-line parameters into the control file, or place commonly used parameters in a parameter file.

If the length of the command line exceeds the maximum line size for your system, then 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 start 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.

Related Topics

- OPTIONS Clause
  The following command-line parameters can be specified using the `OPTIONS` clause.
- PARFILE
  The `PARFILE` SQL*Loader command-line parameter specifies the name of a file that contains commonly used command-line parameters.

8.1.3 Using SQL*Loader to Load Data Across a Network

To use SQL*Loader to load data across a network connection, you can specify a connect identifier in the connect string when you start the SQL*Loader utility.

This identifier can specify a database instance that is different from the current instance identified by the setting of the `ORACLE_SID` environment variable for the current user. The connect identifier can be an Oracle Net connect descriptor or a net service name (usually defined in the `tnsnames.ora` file) that maps to a connect descriptor. Use of a connect identifier requires that you have Oracle Net Listener running (to start the default listener, enter `lsnrctl start`). The following example starts SQL*Loader for user `scott` using the connect identifier `inst1`:

```
> sqlldr CONTROL=ulcase1.ctl
Username: scott@inst1
Password: password
```

The local SQL*Loader client connects to the database instance defined by the connect identifier `inst1` (a net service name), and loads the data, as specified in the `ulcase1.ctl` control file.

> **Note:**
>
> To load data into a pluggable database (PDB), simply specify its connect identifier on the connect string when you start SQL*Loader.

> **See Also:**
>
> - *Oracle Database Net Services Administrator's Guide* for more information about connect identifiers and Oracle Net Listener
> - *Oracle Database Concepts* for more information about PDBs

8.2 Command-Line Parameters for SQL*Loader

Manage SQL*Loader by using the command-line parameters.

The defaults and maximum values listed for these parameters are for Linux and Unix-based systems. They can be different on your operating system. Refer to your operating system documentation for more information.

- BAD
  The `BAD` command-line parameter for SQL*Loader specifies the name or location, or both, of the bad file associated with the first data file specification.
- BINDSIZE
  The `BINDSIZE` command-line parameter for SQL*Loader specifies the maximum size (in bytes) of the bind array.

- COLUMNARRAYROWS
  The `COLUMNARRAYROWS` command-line parameter for SQL*Loader specifies the number of rows to allocate for direct path column arrays.
- CONTROL
  The `CONTROL` command-line parameter for SQL*Loader specifies the name of the SQL*Loader control file that describes how to load the data.
- DATA
  The `DATA` command-line parameter for SQL*Loader specifies the names of the data files containing the data that you want to load.
- DATE_CACHE
  The `DATE_CACHE` command-line parameter for SQL*Loader specifies the date cache size (in entries).
- DEFAULTS
  The `DEFAULTS` command-line parameter for SQL*Loader controls evaluation and loading of default expressions.
- DEGREE_OF_PARALLELISM
  The `DEGREE_OF_PARALLELISM` command-line parameter for SQL*Loader specifies the degree of parallelism to use during the load operation.
- DIRECT
  The `DIRECT` command-line parameter for SQL*Loader specifies the load method to use, either conventional path or direct path.
- DIRECT_PATH_LOCK_WAIT
  The DIRECT_PATH_LOCK_WAIT command-line parameter for SQL*Loader controls direct path load behavior when waiting for table locks.
- DISCARD
  The `DISCARD` command-line parameter for SQL*Loader lets you optionally specify a discard file to store records that are neither inserted into a table nor rejected.
- DISCARDMAX
  The `DISCARDMAX` command-line parameter for SQL*Loader specifies the number of discard records to allow before data loading is terminated.
- DNFS_ENABLE
  The `DNFS_ENABLE` SQL*Loader command-line parameter lets you enable and disable use of the Direct NFS Client on input data files during a SQL*Loader operation.
- DNFS_READBUFFERS
  The `DNFS_READBUFFERS` SQL*Loader command-line parameter lets you control the number of read buffers used by the Direct NFS Client.
- EMPTY_LOBS_ARE_NULL
  The `EMPTY_LOBS_ARE_NULL` SQL*Loader command-line parameter specifies that any LOB column for which there is no data available is set to NULL, rather than to an empty LOB.
- ERRORS
  The ERRORS SQL*Loader command line parameter specifies the maximum number of allowed insert errors.
- EXTERNAL_TABLE
  The `EXTERNAL_TABLE` parameter instructs SQL*Loader whether to load data using the external tables option.
- FILE
  The `FILE` SQL*Loader command-line parameter specifies the database file from which the extents are allocated.

- HELP
  The `HELP` SQL*Loader command-line parameter displays online help for the SQL*Loader utility.
- LOAD
  The `LOAD` SQL*Loader command-line parameter specifies the maximum number of records to load.
- LOG
  The `LOG` SQL*Loader command-line parameter specifies a directory path, or file name, or both for the log file where SQL*Loader stores logging information about the loading process.
- MULTITHREADING
  The `MULTITHREADING` SQL*Loader command-line parameter enables stream building on the client system to be done in parallel with stream loading on the server system.
- NO_INDEX_ERRORS
  The `NO_INDEX_ERRORS` SQL*Loader command-line parameter specifies whether indexing errors are tolerated during a direct path load.
- PARALLEL
  The SQL*Loader `PARALLEL` parameter specifies whether loads that use direct path or external tables can operate in multiple concurrent sessions to load data into the same table.
- PARFILE
  The `PARFILE` SQL*Loader command-line parameter specifies the name of a file that contains commonly used command-line parameters.
- PARTITION_MEMORY
  The PARFILE SQL*Loader command-line parameter specifies the amount of memory that you want to have used when you are loading many partitions.
- READSIZE
  The `READSIZE` SQL*Loader command-line parameter specifies (in bytes) the size of the read buffer, if you choose not to use the default.
- RESUMABLE
  The `RESUMABLE` SQL*Loader command-line parameter enables and disables resumable space allocation.
- RESUMABLE_NAME
  The `RESUMABLE_NAME` SQL*Loader command-line parameter identifies a statement that is resumable.
- RESUMABLE_TIMEOUT
  The `RESUMABLE_TIMEOUT` SQL*Loader command-line parameter specifies the time period, in seconds, during which an error must be fixed.
- ROWS
  For conventional path loads, the `ROWS` SQL*Loader command-line parameter specifies the number of rows in the bind array, and in direct path loads, the number of rows to read from data files before a save.
- SDF_PREFIX
  The `SDF_PREFIX` SQL*Loader command-line parameter specifies a directory prefix, which is added to file names of LOBFILEs and secondary data files (SDFs) that are opened as part of a load operation.

- SILENT
  The `SILENT` SQL*Loader command-line parameter suppresses some of the content that is written to the screen during a SQL*Loader operation.
- SKIP
  The `SKIP` SQL*Loader command-line parameter specifies the number of logical records from the beginning of the file that should not be loaded.
- SKIP_INDEX_MAINTENANCE
  The `SKIP_INDEX_MAINTENANCE` SQL*Loader command-line parameter specifies whether to stop index maintenance for direct path loads.
- SKIP_UNUSABLE_INDEXES
  The `SKIP_UNUSABLE_INDEXES` SQL*Loader command-line parameter specifies whether to skip an index encountered in an Index Unusable state and continue the load operation.
- STREAMSIZE
  The `STREAMSIZE` SQL*Loader command-line parameter specifies the size (in bytes) of the data stream sent from the client to the server.
- TRIM
  The `TRIM` SQL*Loader command-line parameter specifies whether you want spaces trimmed from the beginning of a text field, the end of a text field, both, or neither.
- USERID
  The `USERID` SQL*Loader command-line parameter provides your Oracle username and password for SQL*Loader.

8.2.1 BAD

The `BAD` command-line parameter for SQL*Loader specifies the name or location, or both, of the bad file associated with the first data file specification.

Default

The name of the data file, with an extension of `.bad`.

Purpose

Specifies the name or location, or both, of the bad file associated with the first data file specification.

Syntax and Description

```
BAD=[directory/][filename]
```

The bad file stores records that cause errors during insert, or that are improperly formatted. If you specify the `BAD` parameter, then you must supply either a directory, or file name, or both. If there are rejected records, and you have not specified a name for the bad file, then the name defaults to the name of the data file with an extension or file type of `.bad`.

The value you provide for *`directory`* specifies the directory where you want the bad file to be written. The specification can include the name of a device or network node. The value of *`directory`* is determined as follows:

- If the `BAD` parameter is not specified at all, and a bad file is needed, then the default directory is the one in which the SQL*Loader control file resides.

- If the BAD parameter is specified with a file name, but without a directory, then the directory defaults to the current directory.
- If the BAD parameter is specified with a directory, but without a file name, then the specified directory is used, and the name defaults to the name of the data file, with an extension or file type of .bad.

The value you provide for *filename* specifies a file name that is recognized as valid on your platform. You must specify only a name (and extension, if you want to use one other than .bad). Any spaces or punctuation marks in the file name must be enclosed within single quotation marks.

A bad file specified on the command line becomes the bad file associated with the first INFILE statement (if there is one) in the control file. You can also specify the of the bad file in the SQL*Loader control file by using the BADFILE clause. If the bad file is specified in both the control file and by command line, then the command-line value is used. If a bad file with that name already exists, then it is either overwritten, or a new version is created, depending on your operating system.

Example

The following specification creates a bad file named emp1.bad in the current directory:

```
BAD=emp1
```

Related Topics

- Understanding and 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.

8.2.2 BINDSIZE

The BINDSIZE command-line parameter for SQL*Loader specifies the maximum size (in bytes) of the bind array.

Default

```
256000
```

Purpose

Specifies the maximum size (in bytes) of the bind array.

Syntax and Description

```
BINDSIZE=n
```

A **bind array** is an area in memory where SQL*Loader stores data that is to be loaded. When the bind array is full, the data is transmitted to the database. The bind array size is controlled by the parameters BINDSIZE and READSIZE.

The size of the bind array given by BINDSIZE overrides the default size (which is system dependent) and any size determined by ROWS.

Restrictions

- The `BINDSIZE` parameter is used only for conventional path loads.

Example

The following `BINDSIZE` specification limits the maximum size of the bind array to 356,000 bytes.

```
BINDSIZE=356000
```

Related Topics

- Bind Arrays and Conventional Path Loads
  Multiple rows are read at one time and stored in the bind array.
- READSIZE
  The `READSIZE` SQL*Loader command-line parameter specifies (in bytes) the size of the read buffer, if you choose not to use the default.
- ROWS
  For conventional path loads, the `ROWS` SQL*Loader command-line parameter specifies the number of rows in the bind array, and in direct path loads, the number of rows to read from data files before a save.

8.2.3 COLUMNARRAYROWS

The `COLUMNARRAYROWS` command-line parameter for SQL*Loader specifies the number of rows to allocate for direct path column arrays.

Default

```
5000
```

Purpose

Specifies the number of rows that you want to allocate for direct path column arrays.

Syntax and Description

```
COLUMNARRARYROWS=n
```

The value for this parameter is not calculated by SQL*Loader. You must either specify it or accept the default.

Example

The following example specifies that you want to allocate 1000 rows for direct path column arrays.

```
COLUMNARRAYROWS=1000
```

Related Topics

- 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.
- 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. T

8.2.4 CONTROL

The `CONTROL` command-line parameter for SQL*Loader specifies the name of the SQL*Loader control file that describes how to load the data.

Default

There is no default.

Purpose

Specifies the name of the SQL*Loader control file that describes how to load the data.

Syntax and Description

```
CONTROL=control_file_name
```

If you do not specify a file extension or file type, then it defaults to `.ctl`. If the `CONTROL` parameter is not specified, then SQL*Loader prompts you for it.

If the name of your SQL*Loader control file contains special characters, then your operating system can require that you enter the control file name preceded by an escape character. Also, if your operating system uses backslashes in its file system paths, then you can be required to use multiple escape characters, or you can be required to enclose the path in quotation marks. Refer to your operating system documentation for more information about how to use special characters.

Example

The following example specifies a control file named `emp1`. It is automatically given the default extension of `.ctl`.

```
CONTROL=emp1
```

Related Topics

- SQL*Loader Control File Reference
  The SQL*Loader control file is a text file that contains data definition language (DDL) instructions for a SQL*Loader job.

8.2.5 DATA

The `DATA` command-line parameter for SQL*Loader specifies the names of the data files containing the data that you want to load.

Default

The same name as the control file, but with an extension of `.dat`.

Purpose

The `DATA` parameter specifies the name of the data file containing the data that you want to load.

Syntax and Description

```
DATA=data_file_name
```

If you do not specify a file extension, then the default is `.dat`.

The file specification can contain wildcards (only in the file name and file extension, not in a device or directory name). An asterisk (*) represents multiple characters and a question mark (?) represents a single character. For example:

```
DATA='emp*.dat'

DATA='m?emp.dat'
```

To list multiple data file specifications (each of which can contain wild cards), the file names must be separated by commas.

If the file name contains any special characters (for example, spaces, *, ?,), then the entire name must be enclosed within single quotation marks.

The following are three examples of possible valid uses of the `DATA` parameter (the single quotation marks would only be necessary if the file name contained special characters):

```
DATA='file1','file2','file3','file4','file5','file6'
DATA='file1','file2'
DATA='file3,'file4','file5'
DATA='file6'
```

> **Caution:**
>
> If multiple data files are being loaded and you are also specifying the `BAD` parameter, it is recommended that you specify only a directory for the bad file, not a file name. If you specify a file name, and a file with that name already exists, then it is either overwritten or a new version is created, depending on your operating system.

If you specify data files on the command line with the `DATA` parameter and also specify data files in the control file with the `INFILE` clause, then the first `INFILE` specification in the control

file is ignored. All other data files specified on the command line and in the control file are processed.

If you specify a file processing option along with the `DATA` parameter when loading data from the control file, then a warning message is issued.

Example

The following example specifies that a data file named `employees.dat` is to be loaded. The `.dat` extension is assumed as the default because no extension is provided.

```
DATA=employees
```

8.2.6 DATE_CACHE

The `DATE_CACHE` command-line parameter for SQL*Loader specifies the date cache size (in entries).

Default

Enabled (for `1000` elements). To completely disable the date cache feature, set it to `0` (zero).

Purpose

Specifies the date cache size (in entries).

The date cache is used to store the results of conversions from text strings to internal date format. The cache is useful, because the cost of looking up dates is much less than converting from text format to date format. If the same dates occur repeatedly in the date file, then using the date cache can improve the speed of a direct path load.

Syntax and Description

```
DATE_CACHE=n
```

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 data type conversion before it can 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 if 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 you specify a nonzero date cache size, and that size is exceeded, then the cache is not disabled.

To tune the size of the cache for future similar loads, use the date cache statistics (entries, hits, and misses) contained in the log file.

Restrictions

- The date cache feature is only available for direct path and external tables loads.

Example

The following specification completely disables the date cache feature.

```
DATE_CACHE=0
```

Related Topics

- Specifying a Value for DATE_CACHE
  To improve performance where the same date or timestamp is used many times during a direct path load, you can use the SQL*Loader date cache.

8.2.7 DEFAULTS

The `DEFAULTS` command-line parameter for SQL*Loader controls evaluation and loading of default expressions.

Default

`EVALUATE_ONCE`, unless a sequence is involved. If a sequence is involved, then the default is `EVALUATE_EVERY_ROW`.

Purpose

Controls evaluation and loading of default expressions.

The `DEFAULTS` parameter is only applicable to direct path loads.

Syntax and Description

```
DEFAULTS={IGNORE | IGNORE_UNSUPPORTED_EVALUATE_ONCE |
IGNORE_UNSUPPORTED_EVALUATE_EVERY_ROW |
EVALUATE_ONCE | EVALUATE_EVERY_ROW}
```

The behavior of each of the options is as follows:

- `IGNORE`: Default clauses on columns are ignored.
- `IGNORE_UNSUPPORTED_EVALUATE_ONCE`: Evaluate default expressions once at the start of the load. Unsupported default expressions are ignored. If the `DEFAULTS` parameter is not used, then default expressions are evaluated once, unless the default expression references a sequence, in which case every row is evaluated.
- `IGNORE_UNSUPPORTED_EVALUATE_EVERY_ROW`: Evaluate default expressions in every row, ignoring unsupported default clauses.
- `EVALUATE_ONCE`: Evaluate default expressions once at the start of the load. If the `DEFAULTS` parameter is not used, then default expressions are evaluated once, unless the default references a sequence, in which case every row is evaluated. An error is issued for unsupported default expression clauses. (This is the default option for this parameter.)
- `EVALUATE_EVERY_ROW`: Evaluate default expressions in every row, and issue an error for unsupported defaults.

Example

This example shows that a table is created with the name `test`, and a SQL*Loader control file named `test.ctl`:

```
create table test
(
   c0 varchar2(10),
   c1 number default '100'
)
;

test.ctl:

load data
infile *
truncate
into table test
fields terminated by ','
trailing nullcols
(
  c0 char
)
begindata
1,
```

To then load a NULL into `c1`, issue the following statement:

```
sqlldr scott/password t.ctl direct=true defaults=ignore
```

To load the default value of 100 into `c1`, issue the following statement:

```
sqlldr scott/password t.ctl direct=true
```

8.2.8 DEGREE_OF_PARALLELISM

The `DEGREE_OF_PARALLELISM` command-line parameter for SQL*Loader specifies the degree of parallelism to use during the load operation.

Default

```
NONE
```

Purpose

The `DEGREE_OF_PARALLELISM` parameter specifies the degree of parallelism to use during the load operation.

Syntax and Description

```
DEGREE_OF_PARALLELISM=[degree-num|DEFAULT|AUTO|NONE]
```

If a *degree-num* is specified, then it must be a whole number value from 1 to *n*.

If DEFAULT is specified, then the default parallelism *of the database* (not the default parameter value of AUTO) is used.

If AUTO is used, then the Oracle database automatically sets the degree of parallelism for the load.

If NONE is specified, then the load is not performed in parallel.

> **See Also:**
>
> - *Oracle Database VLDB and Partitioning Guide* for more information about parallel execution

Restrictions

- The DEGREE_OF_PARALLELISM parameter is valid only when the external table load method is used.

Example

The following example sets the degree of parallelism for the load to 3.

```
DEGREE_OF_PARALLELISM=3
```

8.2.9 DIRECT

The DIRECT command-line parameter for SQL*Loader specifies the load method to use, either conventional path or direct path.

Default

```
FALSE
```

Purpose

The DIRECT parameter specifies the load method to use, either conventional path or direct path.

Syntax and Description

```
DIRECT=[TRUE | FALSE]
```

A value of TRUE specifies a direct path load. A value of FALSE specifies a conventional path load.

> **See Also:**
>
> Conventional and Direct Path Loads

Example

The following example specifies that the load be performed using conventional path mode.

```
DIRECT=FALSE
```

8.2.10 DIRECT_PATH_LOCK_WAIT

The DIRECT_PATH_LOCK_WAIT command-line parameter for SQL*Loader controls direct path load behavior when waiting for table locks.

Default

```
FALSE
```

Purpose

Controls direct path load behavior when waiting for table locks. Direct path loads must lock the table before the load can proceed. The `DIRECT_PATH_LOCK_WAIT` command controls the direct path API behavior while waiting for a lock.

Syntax and Description

```
DIRECT_PATH_LOCK_WAIT = {TRUE | FALSE}
```

- `TRUE`: Direct path waits until it can get a lock on the table before proceeding with the load.
- `FALSE`: (Default). When set to `FALSE`, the direct path API tries to lock the table multiple times and waits one second between attempts. The maximum number of attempts made is 30. If the table cannot be locked after 30 attempts, then the direct path API returns the error that was generated when trying to lock the table.

8.2.11 DISCARD

The `DISCARD` command-line parameter for SQL*Loader lets you optionally specify a discard file to store records that are neither inserted into a table nor rejected.

Default

The same file name as the data file, but with an extension of `.dsc`.

Purpose

The `DISCARD` parameter lets you optionally specify a discard file to store records that are neither inserted into a table nor rejected. They are not bad records, they simply did not match any record-selection criteria specified in the control file, such as a `WHEN` clause for example.

Syntax and Description

```
DISCARD=[directory/][filename]
```

If you specify the `DISCARD` parameter, then you must supply either a directory or file name, or both.

The *directory* parameter specifies a directory to which the discard file will be written. The specification can include the name of a device or network node. The value of directory is determined as follows:

- If the DISCARD parameter is not specified at all, but the DISCARDMAX parameter is, then the default directory is the one in which the SQL*Loader control file resides.
- If the DISCARD parameter is specified with a file name but no directory, then the directory defaults to the current directory.
- If the DISCARD parameter is specified with a directory but no file name, then the specified directory is used and the default is used for the name and the extension.

The *filename* parameter specifies a file name recognized as valid on your platform. You must specify only a name (and extension, if one other than .dsc is desired). Any spaces or punctuation marks in the file name must be enclosed in single quotation marks.

If neither the DISCARD parameter nor the DISCARDMAX parameter is specified, then a discard file is not created even if there are discarded records.

If the DISCARD parameter is not specified, but the DISCARDMAX parameter is, and there are discarded records, then the discard file is created using the default name and the file is written to the same directory in which the SQL*Loader control file resides.

> **Caution:**
>
> If multiple data files are being loaded and you are also specifying the DISCARD parameter, it is recommended that you specify only a directory for the discard file, not a file name. If you specify a file name, and a file with that name already exists, then it is either overwritten or a new version is created, depending on your operating system.

A discard file specified on the command line becomes the discard file associated with the first INFILE statement (if there is one) in the control file. If the discard file is also specified in the control file, then the command-line value overrides it. If a discard file with that name already exists, then it is either overwritten or a new version is created, depending on your operating system.

> **See Also:**
>
> Discarded and Rejected Records for information about the format of discard files

Example

Assume that you are loading a data file named employees.dat. The following example supplies only a directory name so the name of the discard file will be employees.dsc and it will be created in the mydir directory.

```
DISCARD=mydir/
```

8.2.12 DISCARDMAX

The `DISCARDMAX` command-line parameter for SQL*Loader specifies the number of discard records to allow before data loading is terminated.

Default

`ALL`

Purpose

The `DISCARDMAX` parameter specifies the number of discard records to allow before data loading is terminated.

Syntax and Description

`DISCARDMAX=`*`n`*

To stop on the first discarded record, specify a value of 0.

If `DISCARDMAX` is specified, but the `DISCARD` parameter is not, then the name of the discard file is the name of the data file with an extension of `.dsc`.

Example

The following example allows 25 records to be discarded during the load before it is terminated.

```
DISCARDMAX=25
```

8.2.13 DNFS_ENABLE

The `DNFS_ENABLE` SQL*Loader command-line parameter lets you enable and disable use of the Direct NFS Client on input data files during a SQL*Loader operation.

Default

`TRUE`

Purpose

The `DNFS_ENABLE` parameter lets you enable and disable use of the Direct NFS Client on input data files during a SQL*Loader operation.

Syntax and Description

```
DNFS_ENABLE=[TRUE|FALSE]
```

The Direct NFS Client is an API that can be implemented by file servers to allow improved performance when an Oracle database accesses files on those servers.

SQL*Loader uses the Direct NFS Client interfaces by default when it reads data files over 1 GB. For smaller files, the operating system's I/O interfaces are used. To use the Direct NFS Client on *all* input data files, use `DNFS_ENABLE=TRUE`.

To disable use of the Direct NFS Client for all data files, specify `DNFS_ENABLE=FALSE`.

The `DNFS_READBUFFERS` parameter can be used to specify the number of read buffers used by the Direct NFS Client; the default is 4.

> **See Also:**
>
> - Oracle Grid Infrastructure Installation Guide for your platform for more information about enabling the Direct NFS Client

Example

The following example disables use of the Direct NFS Client on input data files during the load.

```
DNFS_ENABLE=FALSE
```

8.2.14 DNFS_READBUFFERS

The `DNFS_READBUFFERS` SQL*Loader command-line parameter lets you control the number of read buffers used by the Direct NFS Client.

Default

4

Purpose

The `DNFS_READBUFFERS` parameter lets you control the number of read buffers used by the Direct NFS Client. The Direct NFS Client is an API that can be implemented by file servers to allow improved performance when an Oracle database accesses files on those servers.

Syntax and Description

```
DNFS_READBUFFERS=n
```

The value for *n* is the number of read buffers you specify. It is possible that you can compensate for inconsistent input/output (I/O) from the Direct NFS Client file server by increasing the number of read buffers. However, using larger values can result in increased memory usage.

Restrictions

- To use this parameter without also specifying the `DNFS_ENABLE` parameter, the input file must be larger than 1 GB.

Example

The following example specifies 10 read buffers for use by the Direct NFS Client.

```
DNFS_READBUFFERS=10
```

Related Topics

- *Oracle Grid Infrastructure Installation Guide* for your platform

8.2.15 EMPTY_LOBS_ARE_NULL

The `EMPTY_LOBS_ARE_NULL` SQL*Loader command-line parameter specifies that any LOB column for which there is no data available is set to NULL, rather than to an empty LOB.

Default

`FALSE`

Purpose

If the SQL*Loader `EMPTY_LOBS_ARE_NULL` parameter is specified, then any Large Object (LOB) columns for which there is no data available are set to NULL, rather than to an empty LOB. Setting LOB columns for which there is no data available to NULL negates the need to make that change through post-processing after the data is loaded.

Syntax and Description

```
EMPTY_LOBS_ARE_NULL = {TRUE | FALSE}
```

You can specify the `EMPTY_LOBS_ARE_NULL` parameter on the SQL*Loader command line, and also on the `OPTIONS` clause in a SQL*Loader control file.

Restrictions

None.

Example

In the following example, as a result of setting `empty_lobs_are_null=true`, the LOB columns in `c1` are set to `NULL` instead of to an empty LOB.

```
create table t
(
   c0 varchar2(10),
   c1 clob
)
;

sqlldr control file:

options (empty_lobs_are_null=true)
load data
infile *
truncate
into table t
fields terminated by ','
trailing nullcols
(
  c0 char,
  c1 char
)
```

```
begindata
1,,
```

8.2.16 ERRORS

The ERRORS SQL*Loader command line parameter specifies the maximum number of allowed insert errors.

Default

50

Purpose

The `ERRORS` parameter specifies the maximum number of insert errors to allow.

Syntax and Description

```
ERRORS=n
```

If the number of errors exceeds the value specified for `ERRORS`, then SQL*Loader terminates the load. Any data inserted up to that point is committed.

To permit no errors at all, set `ERRORS=0`. To specify that all errors be allowed, use a very high number.

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.

Example

The following example specifies a maximum of 25 insert errors for the load. After that, the load is terminated.

```
ERRORS=25
```

8.2.17 EXTERNAL_TABLE

The `EXTERNAL_TABLE` parameter instructs SQL*Loader whether to load data using the external tables option.

Default

```
NOT_USED
```

Syntax and Description

```
EXTERNAL_TABLE=[NOT_USED | GENERATE_ONLY | EXECUTE]
```

The possible values are as follows:

- `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 log 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 deletes 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 Language Reference.)*

> **Note:**
>
> When the `EXTERNAL_TABLE` parameter is specified, any datetime data types (for example, `TIMESTAMP`) in a SQL*Loader control file are automatically converted to a `CHAR` data type and use the external tables `date_format_spec` clause. See date_format_spec.

Note that the external table option uses directory objects in the database to indicate where all input data files 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 data files, 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 data file or output file, then SQL*Loader will generate the SQL statement to create one. Therefore, when the `EXECUTE` option is specified, you must have the `CREATE ANY DIRECTORY` privilege. If you want the directory object to be deleted at the end of the load, then you must also have the `DROP ANY DIRECCTORY` privilege.

> **Note:**
>
> The `EXTERNAL_TABLE=EXECUTE` qualifier tells SQL*Loader to create an external table that can be used to load data and then executes the `INSERT` statement to load the data. All files in the external table must be identified as being in a directory object. SQL*Loader attempts to use directory objects that already exist and that you have privileges to access. However, if SQL*Loader does not find the matching directory object, then it 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 multi-table load, SQL*Loader does the following:

1. Creates a table in the database that describes all fields in the input data file 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, 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:**
>
> - "SQL*Loader Case Studies" for information on how to access case studies
> - External Tables Concepts
> - The ORACLE_LOADER Access Driver

Restrictions

- 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.

Example

```
EXTERNAL_TABLE=EXECUTE
```

8.2.18 FILE

The `FILE` SQL*Loader command-line parameter specifies the database file from which the extents are allocated.

Default

There is no default.

Purpose

The `FILE` parameter specifies the database file from which the extents are allocated.

> **See Also:**
>
> Parallel Data Loading Models

Syntax and Description

```
FILE=tablespace_file
```

By varying the value of the `FILE` parameter for different SQL*Loader processes, data can be loaded onto a system with minimal disk contention.

Restrictions

- The `FILE` parameter is used only for direct path parallel loads.

8.2.19 HELP

The `HELP` SQL*Loader command-line parameter displays online help for the SQL*Loader utility.

Default

```
FALSE
```

Syntax and Description

```
HELP = [TRUE | FALSE]
```

If `HELP=TRUE` is specified, then SQL*Loader displays a summary of all SQL*Loader command-line parameters.

You can also display a summary of all SQL*Loader command-line parameters by entering `sqlldr -help` on the command line.

8.2.20 LOAD

The `LOAD` SQL*Loader command-line parameter specifies the maximum number of records to load.

Default

All records are loaded.

Purpose

Specifies the maximum number of records to load.

Syntax and Description

```
LOAD=n
```

To test that all parameters you have specified for the load are set correctly, use the `LOAD` parameter to specify a limited number of records rather than loading all records. No error occurs if fewer than the maximum number of records are found.

Example

The following example specifies that a maximum of 10 records be loaded.

```
LOAD=10
```

For external tables method loads, only successfully loaded records are counted toward the total. So if there are 15 records in the input data file and records 2 and 4 are bad, then the following records are loaded into the table, for a total of 10 records: 1, 3, 5, 6, 7, 8, 9, 10, 11, and 12.

For conventional and direct path loads, both successful and unsuccessful load attempts are counted toward the total. So if there are 15 records in the input data file, and records 2 and 4 are bad, then only the following 8 records are actually loaded into the table: 1, 3, 5, 6, 7, 8, 9, and 10.

8.2.21 LOG

The `LOG` SQL*Loader command-line parameter specifies a directory path, or file name, or both for the log file where SQL*Loader stores logging information about the loading process.

Default

The current directory, if no value is specified.

Purpose

Specifies a directory path, or file name, or both for the log file that SQL*Loader uses to store logging information about the loading process.

Syntax and Description

```
LOG=[[directory/][log_file_name]]
```

If you specify the `LOG` parameter, then you must supply a directory name, or a file name, or both.

If no directory name is specified, it defaults to the current directory.

If a directory name is specified without a file name, then the default log file name is used.

Example

The following example creates a log file named `emp1.log` in the current directory. The extension `.log` is used even though it is not specified, because it is the default.

```
LOG=emp1
```

8.2.22 MULTITHREADING

The `MULTITHREADING` SQL*Loader command-line parameter enables stream building on the client system to be done in parallel with stream loading on the server system.

Default

`TRUE` on multiple-CPU systems, `FALSE` on single-CPU systems

Syntax and Description

```
MULTITHREADING=[TRUE | FALSE]
```

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.

Restrictions

- The `MULTITHREADING` parameter is available only for direct path loads.
- Multithreading functionality is operating system-dependent. Not all operating systems support multithreading.

Example

The following example enables multithreading on a single-CPU system. On a multiple-CPU system it is enabled by default.

```
MULTITHREADING=TRUE
```

Related Topics

- Optimizing Direct Path Loads on Multiple-CPU Systems
  If you are performing direct path loads on a multiple-CPU system, then 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.

8.2.23 NO_INDEX_ERRORS

The `NO_INDEX_ERRORS` SQL*Loader command-line parameter specifies whether indexing errors are tolerated during a direct path load.

Default

```
FALSE
```

Syntax and Description

```
NO_INDEX_ERRORS=[TRUE | FALSE]
```

A setting of `NO_INDEX_ERRORS=FALSE` means that if a direct path load results in an index becoming unusable then the rows are loaded and the index is left in an unusable state. This is the default behavior.

A setting of `NO_INDEX_ERRORS=TRUE` means that if a direct path load results in any indexing errors, then the load is aborted. No rows are loaded and the indexes are left as they were.

Restrictions

- The `NO_INDEX_ERRORS` parameter is valid only for direct path loads. If it is specified for conventional path loads, then it is ignored.

Example

```
NO_INDEX_ERRORS=TRUE
```

8.2.24 PARALLEL

The SQL*Loader `PARALLEL` parameter specifies whether loads that use direct path or external tables can operate in multiple concurrent sessions to load data into the same table.

the current directory
`FALSE`

Purpose

Specifies whether loads that use direct path or external tables can operate in multiple concurrent sessions to load data into the same table.

Syntax and Description

```
PARALLEL=[TRUE | FALSE]
```

Restrictions

- The `PARALLEL` parameter is not valid in conventional path loads.

Example

The following example specifies that the load will be performed in parallel.

```
PARALLEL=TRUE
```

Related Topics

- About SQL*Loader Parallel Data Loading Models
  There are three basic models of concurrency that you can use to minimize the elapsed time required for data loading.

8.2.25 PARFILE

The `PARFILE` SQL*Loader command-line parameter specifies the name of a file that contains commonly used command-line parameters.

Default

There is no default.

Syntax and Description

```
PARFILE=file_name
```

Instead of specifying each parameter on the command line, you can simply specify the name of the parameter file. For example, a parameter file named `daily_report.par` might have the following contents:

```
USERID=scott
CONTROL=daily_report.ctl
ERRORS=9999
LOG=daily_report.log
```

For security reasons, do not include your `USERID` password in a parameter file. After you specify the parameter file at the command line, SQL*Loader prompts you for the password. For example:

```
sqlldr PARFILE=daily_report.par
Password: password
```

Restrictions

- On some systems it can be necessary to have no spaces around the equal sign (=) in the parameter specifications.

Example

See the example in the Syntax and Description section.

8.2.26 PARTITION_MEMORY

The PARFILE SQL*Loader command-line parameter specifies the amount of memory that you want to have used when you are loading many partitions.

Default

`0` (zero) This setting limits memory use based on the value of the `PGA_AGGREGATE_TARGET` initialization parameter. When memory use approaches that value, loading of some partitions is delayed.

Purpose

Specifies the amount of memory that you want to have used when you are loading many partitions. This parameter is helpful in situations in which the number of partitions you are loading use up large amounts of memory, perhaps even exceeding available memory. (This scenario can occur, especially when the data is compressed).

After the specified limit is reached, loading of some partition rows is delayed until memory use falls below the limit.

Syntax and Description

```
PARTITION_MEMORY=n
```

The parameter value *n* is in kilobytes.

If *n* is set to `0` (the default), then SQL*Loader uses a value that is a function of the `PGA_AGGREGATE_TARGET` initialization parameter.

If *n* is set to -1 (minus 1), then SQL*Loader makes no attempt to use less memory when loading many partitions.

Restrictions

- This parameter is only valid for direct path loads.
- This parameter is available only in Oracle Database 12c Release 1 (12.1.0.2) and later releases.

Example

The following example limits memory use to 1 GB.

```
> sqlldr hr CONTROL=t.ctl DIRECT=true PARTITION_MEMORY=1000000
```

8.2.27 READSIZE

The READSIZE SQL*Loader command-line parameter specifies (in bytes) the size of the read buffer, if you choose not to use the default.

Default

```
1048576
```

Syntax and Description

```
READSIZE=n
```

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, setting READSIZE to 1000000 enables SQL*Loader to perform reads from the data file in chunks of 1,000,000 bytes before a commit is required.

> **Note:**
>
> If the READSIZE value specified is smaller than the BINDSIZE value, then the READSIZE value is increased.

Restrictions

- The READSIZE parameter is used *only* when reading data from data files. When reading records from a control file, a value of 64 kilobytes (KB) is *always* used as the READSIZE.
- The READSIZE parameter has no effect on Large Objects (LOBs). The size of the LOB read buffer is fixed at 64 kilobytes (KB).
- The maximum size allowed is platform-dependent.

Example

The following example sets the size of the read buffer to 500,000 bytes, which means that commit operations will be required more often than if the default or a value larger than the default were used.

```
READSIZE=500000
```

Related Topics

- BINDSIZE

8.2.28 RESUMABLE

The `RESUMABLE` SQL*Loader command-line parameter enables and disables resumable space allocation.

Default

```
FALSE
```

Purpose

Enables and disables resumable space allocation.

Syntax and Description

```
RESUMABLE=[TRUE | FALSE]
```

> **See Also:**
>
> *Oracle Database Administrator's Guide* for more information about resumable space allocation.

Restrictions

- Because this parameter is disabled by default, you must set `RESUMABLE=TRUE` to use its associated parameters, `RESUMABLE_NAME` and `RESUMABLE_TIMEOUT`.

Example

The following example enables resumable space allocation:

```
RESUMABLE=TRUE
```

8.2.29 RESUMABLE_NAME

The `RESUMABLE_NAME` SQL*Loader command-line parameter identifies a statement that is resumable.

Default

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

Syntax and Description

```
RESUMABLE_NAME='text_string'
```

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.

Restrictions

- This parameter is ignored unless the `RESUMABLE` parameter is set to `TRUE` to enable resumable space allocation.

Example

```
RESUMABLE_NAME='my resumable sql'
```

8.2.30 RESUMABLE_TIMEOUT

The `RESUMABLE_TIMEOUT` SQL*Loader command-line parameter specifies the time period, in seconds, during which an error must be fixed.

Default

`7200` seconds (2 hours)

Syntax and Description

```
RESUMABLE_TIMEOUT=n
```

If the error is not fixed within the timeout period, then execution of the statement is terminated, without finishing.

Restrictions

- This parameter is ignored unless the `RESUMABLE` parameter is set to `TRUE` to enable resumable space allocation.

Example

The following example specifies that errors must be fixed within ten minutes (600 seconds).

```
RESUMABLE_TIMEOUT=600
```

8.2.31 ROWS

For conventional path loads, the ROWS SQL*Loader command-line parameter specifies the number of rows in the bind array, and in direct path loads, the number of rows to read from data files before a save.

Default

Specifies the number of rows in the bind array. The Conventional path default is 64. Direct path default is all rows.

Purpose

For conventional path loads the ROWS parameter specifies the number of rows in the bind array.For direct path loads, the ROWS parameter specifies the number of rows that SQL*Loader reads from the data files before a data save.

Syntax

```
ROWS=n
```

Conventional Path Loads Description

In conventional path loads only, the ROWS parameter specifies the number of rows in the bind array. The maximum number of rows is 65534.

Direct Path Loads Description

In direct path loads only, the ROWS parameter identifies the number of rows that you want to read from the data file before a data save. The default is to read all rows and save data once at the end of the load. 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:**
>
> If you specify a low value for ROWS, and then attempt to compress data using table compression, then the compression ratio probably will be degraded. When compressing the data, Oracle recommends that you either specify a high value, or accept the default value.

Restrictions

- The ROWS parameter is ignored for direct path loads when data is loaded into an Index Organized Table (IOT), or into a table containing VARRAY types, XML columns, or Large Objects (LOBs). This means that the load still takes place, but no save points are done.
- For direct path loads, because LONG VARCHAR data type data are stored as LOBs, you cannot use the ROWS parameter. If you attempt to use the ROWS parameter with LONG VARCHAR data in direct path loads, then you receive an ORA-39777 error (Data saves are not allowed when loading LOB columns).

Example

In a conventional path load, the following example would result in an error because the specified value exceeds the allowable maximum of 65534 rows.

```
ROWS=65900
```

Related Topics

- Using Data Saves to Protect Against Data Loss
  When you have a savepoint, if you encounter an instance failure during a SQL*Loader load, then use the `SKIP` parameter to continue the load.

8.2.32 SDF_PREFIX

The `SDF_PREFIX` SQL*Loader command-line parameter specifies a directory prefix, which is added to file names of LOBFILEs and secondary data files (SDFs) that are opened as part of a load operation.

Default

There is no default.

Purpose

Specifies a directory prefix, which is added to file names of LOBFILEs and secondary data files (SDFs) that are opened as part of a load operation.

> **Note:**
>
> The `SDF_PREFIX` parameter can also be specified in the `OPTIONS` clause in the SQL Loader control file.

Syntax and Description

```
SDF_PREFIX=string
```

If `SDF_PREFIX` is specified, then the string value must be specified as well. There is no validation or verification of the string. The value of `SDF_PREFIX` is prepended to the filenames used for all LOBFILEs and SDFs opened during the load. If the resulting string is not the name of as valid file, then the attempt to open that file fails and an error is reported.

If `SDF_PREFIX` is not specified, then file names for LOBFILEs and SDFs are assumed to be relative to the current working directory. Using `SDF_PREFIX` allows those files names to be relative to a different directory.

Quotation marks are only required around the string if it contains characters that would confuse the command line parser (for example, a space).

The file names that are built by prepending `SDF_PREFIX` to the file names found in the record are passed to the operating system to open the file. The prefix can be relative to the current working directory from which SQL*Loader is being executed or it can be the start of an absolute path.

Restrictions

- The `SDF_PREFIX` parameter should not be used if the file specifications for the LOBFILEs or SDFs contain full file names.

Example

The following SQL*Loader command looks for LOB files in the `lobdir` subdirectory of the current directory

```
sqlldr control=picts.ctl log=picts.log sdf_prefix=lobdir/
```

8.2.33 SILENT

The `SILENT` SQL*Loader command-line parameter suppresses some of the content that is written to the screen during a SQL*Loader operation.

Default

There is no default.

Syntax and Description

```
SILENT=[HEADER | FEEDBACK | ERRORS | DISCARDS | PARTITIONS | ALL]
```

Use the appropriate values to suppress one or more of the following (if more than one option is specified, they must be separated by commas):

- `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" messages and the status messages for the load that normally appear on the screen. But `"XX Rows successfully loaded."` even prints 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`. But `"XX Rows successfully loaded."` even prints on the screen.

Example

You can suppress the header and feedback messages that normally appear on the screen with the following command-line argument:

```
SILENT=HEADER, FEEDBACK
```

But `"XX Rows successfully loaded."` even prints on the screen.

8.2.34 SKIP

The `SKIP` SQL*Loader command-line parameter specifies the number of logical records from the beginning of the file that should not be loaded.

Default

`0` (No records are skipped.)

Purpose

Specifies the number of logical records from the beginning of the file that should not be loaded. Using this specification enables you to continue loads that have been interrupted for some reason, without loading records that have already been processed.

Syntax and Description

```
SKIP=n
```

You can use the `SKIP` parameter for all conventional loads, for single-table direct path loads, and for multiple-table direct path loads when the same number of records was loaded into each table. You cannot use `SKIP` for multiple-table direct path 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, then the secondary data is skipped only if the `WHEN` clause succeeds for the record in the primary data file.

Restrictions

- The `SKIP` parameter cannot be used for external table loads.

Example

The following example skips the first 500 logical records in the data files before proceeding with the load:

```
SKIP=500
```

Related Topics

- Interrupted Loads
  Loads are interrupted and discontinued for several reasons.

8.2.35 SKIP_INDEX_MAINTENANCE

The `SKIP_INDEX_MAINTENANCE` SQL*Loader command-line parameter specifies whether to stop index maintenance for direct path loads.

Default

`FALSE`

Purpose

Specifies whether to stop index maintenance for direct path loads.

Syntax and Description

```
SKIP_INDEX_MAINTENANCE=[TRUE | FALSE]
```

If set to `TRUE`, this parameter causes the index partitions that would have had index keys added to them to instead be marked Index Unusable because the index segment is inconsistent with respect to the data it indexes. Index segments that are unaffected by the load retain the state they had before the load.

The `SKIP_INDEX_MAINTENANCE` parameter:

- Applies to both local and global indexes
- Can be used (with the `PARALLEL` parameter) to perform 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
- Records a list (in the SQL*Loader log file) of the indexes and index partitions that the load set to an Index Unusable state

Restrictions

- The `SKIP_INDEX_MAINTENANCE` parameter does not apply to conventional path loads.
- 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.

Example

The following example stops index maintenance from taking place during a direct path load operation:

```
SKIP_INDEX_MAINTENANCE=TRUE
```

8.2.36 SKIP_UNUSABLE_INDEXES

The `SKIP_UNUSABLE_INDEXES` SQL*Loader command-line parameter specifies whether to skip an index encountered in an Index Unusable state and continue the load operation.

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`.

Purpose

Specifies whether to skip an index encountered in an Index Unusable state and continue the load operation.

Syntax and Description

```
SKIP_UNUSABLE_INDEXES=[TRUE | FALSE]
```

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 before 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 instead will remain in an Unusable state at load completion.

Both SQL*Loader and 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, then 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 Oracle 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 setting for SKIP_UNUSABLE_INDEXES, then the default setting is TRUE.

The SKIP_UNUSABLE_INDEXES parameter applies to both conventional and direct path loads.

Restrictions

- 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.

Example

If the Oracle Database initialization parameter has a value of SKIP_UNUSABLE_INDEXES=FALSE, then setting SKIP_UNUSABLE_INDEXES=TRUE on the SQL*Loader command line overrides it. Therefore, if an index in an Index Unusable state is encountered after this parameter is set, then it is skipped, and the load operation continues.

```
SKIP_UNUSABLE_INDEXES=TRUE
```

8.2.37 STREAMSIZE

The STREAMSIZE SQL*Loader command-line parameter specifies the size (in bytes) of the data stream sent from the client to the server.

Default

256000

Purpose

Specifies the size (in bytes) of the data stream sent from the client to the server.

Syntax and Description

```
STREAMSIZE=n
```

The STREAMSIZE parameter specifies the size of the direct path stream buffer. The number of column array rows (specified with the COLUMNARRAYROWS parameter) determines the number of rows loaded before the stream buffer is built. The optimal values for these parameters vary, depending on the system, input data types, and Oracle column data types used. When you are using optimal values for your particular configuration, the elapsed time in the SQL*Loader log file should go down.

Restrictions

- The STREAMSIZE parameter applies only to direct path loads.
- The minimum value for STREAMSIZE is 65536. If a value lower than 65536 is specified, then 65536 is used instead.

Example

The following example specifies a direct path stream buffer size of 300,000 bytes.

```
STREAMSIZE=300000
```

Related Topics

- 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. T

8.2.38 TRIM

The TRIM SQL*Loader command-line parameter specifies whether you want spaces trimmed from the beginning of a text field, the end of a text field, both, or neither.

Default

```
LDRTRIM
```

Purpose

Specifies that spaces should be trimmed from the beginning of a text field, the end of a text field, both, or neither. Spaces include blanks and other nonprinting characters, such as tabs, line feeds, and carriage returns.

Syntax and Description

```
TRIM=[LRTRIM | NOTRIM | LTRIM | RTRIM | LDRTRIM]
```

The valid values for the TRIM parameter are as follows:

- NOTRIM indicates that you want no characters trimmed from the field. This setting generally yields the fastest performance.
- LRTRIM indicates that you want both leading and trailing spaces trimmed from the field.
- LTRIM indicates that you want leading spaces trimmed from the field
- RTRIM indicates that you want trailing spaces trimmed from the field.
- LDRTRIM is the same as NOTRIM except in the following cases:

- If the field is not a delimited field, then spaces are 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 are trimmed from the left.

> **Note:**
>
> If trimming is specified for a field that consists only of spaces, then the field is set to `NULL`.

Restrictions

- The `TRIM` parameter is valid only when the external table load method is used.

Example

The following example specifies a load operation for which no characters are trimmed from any fields:

```
TRIM=NOTRIM
```

8.2.39 USERID

The `USERID` SQL*Loader command-line parameter provides your Oracle username and password for SQL*Loader.

Default

There is no default.

Purpose

Provides your Oracle user name and password for SQL*Loader, so that you are not prompted to provide them. If it is omitted, then you are prompted for them. If you provide as the value a slash (`/`), then `USERID` defaults to your operating system login.

Syntax and Description

```
USERID=[username | / | SYS]
```

Specify a user name. For security reasons, Oracle recommends that you specify only the user name on the command line. SQL*Loader then prompts you for a password.

If you do not specify the `USERID` parameter, then you are prompted for it. If you use a forward slash (virgule), then `USERID` defaults to your operating system login.

If you connect as user `SYS`, then you must also specify `AS SYSDBA` in the connect string.

Restrictions

- Because the string `AS SYSDBA`, contains a blank, some operating systems can require that you place the entire connect string inside quotation marks, or marked as a literal by some other method. Some operating systems also require that quotation marks on the command line are preceded by an escape character, such as backslashes.

Refer to your operating system-specific documentation for information about special and reserved characters on your system.

Example

The following example specifies a user name of `hr`. SQL*Loader then prompts for a password. Because it is the first and only parameter specified, you do not need to include the parameter name `USERID`:

```
> sqlldr hr
Password:
```

Related Topics

- Specifying Parameters on the Command Line
  When you start SQL*Loader, you specify parameters to establish various characteristics of the load operation.

8.3 Exit Codes for Inspection and Display

Oracle SQL*Loader provides the results of a SQL*Loader run immediately upon completion.

Usage Notes

In addition to recording the results in a log file, SQL*Loader may also report the outcome in a process exit code. This Oracle SQL*Loader functionality allows for checking the outcome of a SQL*Loader invocation from the command line or a script. The following table shows the exit codes for various results:

Table 8-1 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_FTL`

Examples

For Linux and Unix operating systems, the exit codes are as follows:

```
EX_SUCC 0
EX_FAIL 1
EX_WARN 2
EX_FTL  3
```

For Windows operating systems, the exit codes are as follows:

```
EX_SUCC 0
EX_FAIL 1
EX_WARN 2
EX_FTL  4
```

If SQL*Loader returns any exit code other than zero, then consult your system log files and SQL*Loader log files for more detailed diagnostic information.

On Unix platforms, you can check the exit code from the shell to determine the outcome of a load.

9

SQL*Loader Control File Reference

The SQL*Loader control file is a text file that contains data definition language (DDL) instructions for a SQL*Loader job.

> **Note:**
>
> You can also use SQL*Loader without a control file; this is known as SQL*Loader express mode. See SQL*Loader Express for more information.

- Control File Contents
  The SQL*Loader control file is a text file that contains data definition language (DDL) instructions.
- Comments in the Control File
  Comments can appear anywhere in the parameter section of the file, but they should not appear within the data.
- Specifying Command-Line Parameters in the Control File
  You can specify command-line parameters in the SQL*Loader control file using the `OPTIONS` clause.
- Specifying File Names and Object Names
  In general, SQL*Loader follows the SQL standard for specifying object names (for example, table and column names).
- Identifying XMLType Tables
  You can identify and select XML type tables to load by using the `XMLTYPE` clause in a SQL*Loader control file.
- Specifying Field Order
  You can use the `FIELD NAMES` clause in the SQL*Loader control file to specify field order.
- Specifying Data Files
  Learn how you can use the SQL*Loader control file to specify how data files are loaded.
- Specifying CSV Format Files
  To direct SQL*Loader to access the data files as comma-separated-values format files, use the `CSV` clause.
- Identifying Data in the Control File with BEGINDATA
  Specify the `BEGINDATA` statement before the first data record.
- Specifying Data File Format and Buffering
  You can specify an operating system-dependent file processing specifications string option using `os_file_proc_clause`.
- Specifying the Bad File
  Learn what SQL*Loader bad files are, and how to specify them.
- Specifying the Discard File
  Learn what SQL*Loader discard files are, what they contain, and how to specify them.

- Specifying a NULLIF Clause At the Table Level
  To load a table character field as NULL when it contains certain character strings or hex strings, you can use a `NULLIF` clause at the table level with SQL*Loader.
- Specifying Datetime Formats At the Table Level
  You can specify certain datetime formats in a SQL*Loader control file at the table level, or override a table level format by specifying a mask at the field level.
- Handling Different Character Encoding Schemes
  SQL*Loader supports different character encoding schemes (called character sets, or code pages).
- Interrupted Loads
  Loads are interrupted and discontinued for several reasons.
- Assembling Logical Records from Physical Records
  This section describes assembling logical records from physical records.
- Loading Logical Records into Tables
  This section describes loading logical records into tables.
- Index Options
  This section describes index options.
- Benefits of Using Multiple INTO TABLE Clauses
  These sections describe the benefits of using multiple INTO TABLE clauses.
- Bind Arrays and Conventional Path Loads
  Multiple rows are read at one time and stored in the bind array.

Related Topics

- SQL*Loader Express
  SQL*Loader express mode allows you to quickly and easily use SQL*Loader to load simple data types.

9.1 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 SQL*Loader Syntax Diagrams for syntax diagrams of the SQL*Loader DDL.

To create the SQL*Loader control file, use a text editor, such as `vi` or `xemacs`.

In general, the control file has three main sections, in the following order:

- Session-wide information
- Table and field-list information
- Input data (optional section)

The following is an example of a control file.

Example 9-1 Control File

```
1    -- This is an example 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"
   )
```

The numbers that appear to the left in this In this control file example would not appear in a real control file. They are keyed in this sample to the explanatory notes in the following list:

1. This comment prefacing the entries in the control file is an example of how to enter comments in a control file. See Comments in the Control File.
2. The `LOAD DATA` statement tells SQL*Loader that this is the beginning of a new data load. See SQL*Loader Syntax Diagrams for syntax information.
3. The `INFILE` clause specifies the name of a data file containing the data you want to load. See Specifying Data Files.
4. The `BADFILE` clause specifies the name of a file into which rejected records are placed. See Specifying the Bad File.
5. The `DISCARDFILE` clause specifies the name of a file into which discarded records are placed. See Specifying the Discard File.
6. The `APPEND` clause is one of the options that you can use when loading data into a table that is not empty. See Loading Data into Nonempty Tables.

   To load data into a table that is empty, use the `INSERT` clause. See Loading Data into Empty Tables.
7. The `INTO TABLE` clause enables you to identify tables, fields, and data types. It defines the relationship between records in the data file, and tables in the database. See Specifying Table Names.
8. The `WHEN` clause specifies one or more field conditions. SQL*Loader decides whether to load the data based on these field conditions. See Loading Records Based on a Condition.
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.

10. The remainder of the control file contains the field list, which provides information about column formats in the table being loaded. See SQL*Loader Field List Reference for information about that section of the control file.

9.2 Comments in the Control File

Comments can appear anywhere in the parameter 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.

9.3 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 if you often use 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.
- Specifying the Number of Default Expressions to Be Evaluated At One Time
  Use the SQL*Loader `DEFAULT EXPRESSION CACHE n` clause to specify how many default expressions are evaluated at a time by the direct path load. The default value is 100.

9.3.1 OPTIONS Clause

The following command-line parameters can be specified using the `OPTIONS` clause.

These parameters are described in greater detail in SQL*Loader Command-Line Reference.

```
BINDSIZE = n
COLUMNARRAYROWS = n
DATE_CACHE = n
DEGREE_OF_PARALLELISM= {degree-num|DEFAULT|AUTO|NONE}
DIRECT = {TRUE | FALSE}
EMPTY_LOBS_ARE_NULL = {TRUE | FALSE}
ERRORS = n
EXTERNAL_TABLE = {NOT_USED | GENERATE_ONLY | EXECUTE}
FILE = tablespace file
LOAD = n
MULTITHREADING = {TRUE | FALSE}
PARALLEL = {TRUE | FALSE}
READSIZE = n
RESUMABLE = {TRUE | FALSE}
RESUMABLE_NAME = 'text string'
RESUMABLE_TIMEOUT = n
ROWS = n
SDF_PREFIX = string
SILENT = {HEADER | FEEDBACK | ERRORS | DISCARDS | PARTITIONS | ALL}
SKIP = n
SKIP_INDEX_MAINTENANCE = {TRUE | FALSE}
SKIP_UNUSABLE_INDEXES = {TRUE | FALSE}
```

```
STREAMSIZE = n
TRIM= {LRTRIM|NOTRIM|LTRIM|RTRIM|LDRTRIM}
```

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.

9.3.2 Specifying the Number of Default Expressions to Be Evaluated At One Time

Use the SQL*Loader `DEFAULT EXPRESSION CACHE n` clause to specify how many default expressions are evaluated at a time by the direct path load. The default value is 100.

Using the `DEFAULT EXPRESSION CACHE` clause can significantly improve performance when default column expressions that include sequences are evaluated.

At the end of the load there may be sequence numbers left in the cache that never get used. This can happen when the number of rows to load is not a multiple of *n*. If you require no loss of sequence numbers, then specify a value of 1 for this clause.

9.4 Specifying File Names and Object Names

In general, SQL*Loader follows the SQL standard for specifying object names (for example, table and column names).

- File Names That Conflict with SQL and SQL*Loader Reserved Words
  SQL and SQL*Loader reserved words, and words with special characters or case-sensitivity, must be enclosed in quotation marks.
- Specifying SQL Strings in the SQL*Loader Control File
  When you apply SQL operators to field data with the SQL string, you must specify SQL strings within double quotation marks.
- Operating Systems and SQL Loader Control File Characters
  The characters that you use in control files are affected by operating system reserved characters, escape characters, and special characters.

9.4.1 File Names That Conflict with SQL and SQL*Loader Reserved Words

SQL and SQL*Loader reserved words, and words with special characters or case-sensitivity, must be enclosed in quotation marks.

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.

Related Topics

- Oracle SQL Reserved Words and Keywords in *Oracle Database SQL Language Reference*

9.4.2 Specifying SQL Strings in the SQL*Loader Control File

When you apply SQL operators to field data with the SQL string, you must specify SQL strings within double quotation marks.

> **See Also:**
>
> Applying SQL Operators to Fields

9.4.3 Operating Systems and SQL Loader Control File Characters

The characters that you use in control files are affected by operating system reserved characters, escape characters, and special characters.

Learn how the the operating system that you are using affects the characters you can use in your SQL*Loader Control file.

- Specifying a Complete Path
  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 backslash escape character (\), if the escape character is allowed on your operating system.
- 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.
- Using the Backslash as an Escape Character
  To separate directories in a path name, use the backslash character if both your operating system and database implements the backslash escape character.
- Escape Character Is Sometimes Disallowed
  Your operating system can disallow the use of escape characters for nonportable strings in Oracle Database.

9.4.3.1 Specifying a Complete Path

Specifying the path name within single quotation marks prevents errors.

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.

9.4.3.2 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 backslash 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.

9.4.3.3 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.

9.4.3.4 Using the Backslash as an Escape Character

To separate directories in a path name, use the backslash character if both your operating system and database implements the backslash escape character.

If your operating system uses the backslash character to separate directories in a path name, *and* if the Oracle Database release running on your operating system implements the backslash escape character for file names and other nonportable strings, then you must specify double backslashes in your path names, and use single quotation marks.

9.4.3.5 Escape Character Is Sometimes Disallowed

Your operating system can disallow the use of escape characters for nonportable strings in Oracle Database.

When the operating sytem disallows the use of the backslash character (\) as an escape character, a backslash is treated as a normal character, rather than as an escape character.

The backslash character is still usable in all other strings. As a result of this operating system restriction, 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 applies to the use of double quotation marks. A string within double quotation marks cannot be embedded inside another string delimited by double quotation marks.

9.5 Identifying XMLType Tables

You can identify and select XML type tables to load by using the `XMLTYPE` clause in a SQL*Loader control file.

As of Oracle Database 10g, the `XMLTYPE` clause is available for use in a SQL*Loader control file. This clause is of the format `XMLTYPE(field name)`. You can use this clause to identify `XMLType` tables, so that the correct SQL statement can be constructed. You can use the `XMLTYPE` clause in a SQL*Loader control file to load data into a schema-based `XMLType` table.

Example 9-2 Identifying XMLType Tables in the SQL*Loader Control File

The XML schema definition is as follows. It registers the XML schema, `xdb_user.xsd`, in the Oracle XML DB, and then creates the table, `xdb_tab5`.

```
begin dbms_xmlschema.registerSchema('xdb_user.xsd',
'<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
            xmlns:xdb="http://xmlns.oracle.com/xdb">
 <xs:element name = "Employee"
        xdb:defaultTable="EMP31B_TAB">
   <xs:complexType>
    <xs:sequence>
      <xs:element name = "EmployeeId" type = "xs:positiveInteger"/>
      <xs:element name = "Name" type = "xs:string"/>
      <xs:element name = "Salary" type = "xs:positiveInteger"/>
      <xs:element name = "DeptId" type = "xs:positiveInteger"
             xdb:SQLName="DEPTID"/>
    </xs:sequence>
   </xs:complexType>
 </xs:element>
</xs:schema>',
TRUE, TRUE, FALSE); end;
/
```

The table is defined as follows:

```
CREATE TABLE xdb_tab5 OF XMLTYPE XMLSCHEMA "xdb_user.xsd" ELEMENT "Employee";
```

In this next example, the control file used to load data into the table, `xdb_tab5`, loads `XMLType` data by using the registered XML schema, `xdb_user.xsd`. The `XMLTYPE` clause is used to

identify this table as an XMLType table. To load the data into the table, you can use either direct path mode, or conventional mode.

```
LOAD DATA
INFILE *
INTO TABLE xdb_tab5 TRUNCATE
xmltype(xmldata)
(
  xmldata   char(4000)
)
BEGINDATA
<Employee>  <EmployeeId>111</EmployeeId>  <Name>Ravi</Name>
<Salary>100000</Sal
ary>  <DeptId>12</DeptId></Employee>
<Employee>  <EmployeeId>112</EmployeeId>  <Name>John</Name>
<Salary>150000</Sal
ary>  <DeptId>12</DeptId></Employee>
<Employee>  <EmployeeId>113</EmployeeId>  <Name>Michael</Name>
<Salary>75000</S
alary>  <DeptId>12</DeptId></Employee>
<Employee>  <EmployeeId>114</EmployeeId>  <Name>Mark</Name>
<Salary>125000</Sal
ary>  <DeptId>16</DeptId></Employee>
<Employee>  <EmployeeId>115</EmployeeId>  <Name>Aaron</Name>
<Salary>600000</Sa
lary>  <DeptId>16</DeptId></Employee>
```

Related Topics

- Identifying XMLType Tables
  You can identify and select XML type tables to load by using the XMLTYPE clause in a SQL*Loader control file.

9.6 Specifying Field Order

You can use the FIELD NAMES clause in the SQL*Loader control file to specify field order.

The syntax is as follows:

```
FIELD NAMES {FIRST FILE|FIRST FILE IGNORE|ALL FILES|ALL FILES IGNORE|NONE}
```

The FIELD NAMES options are:

- FIRST FILE: Indicates that the first data file contains a list of field names for the data in the first record. This list uses the same delimiter as the data in the data file. The record is read for setting up the mapping between the fields in the data file and the columns in the target table. The record is skipped when the data is processed. This can be useful if the order of the fields in the data file is different from the order of the columns in the table, or if the number of fields in the data file is different from the number of columns in the target table
- FIRST FILE IGNORE: Indicates that the first data file contains a list of field names for the data in the first record, but that the information should be ignored. The record will be skipped when the data is processed, but it will not be used for setting up the fields.
- ALL FILES: Indicates that all data files contain a list of field names for the data in the first record. The first record is skipped in each data file when the data is processed. The fields

can be in a different order in each data file. SQL*Loader sets up the load based on the order of the fields in each data file.

- `ALL FILES IGNORE`: Indicates that all data files contain a list of field names for the data in the first record, but that the information should be ignored. The record is skipped when the data is processed in every data file, but it will not be used for setting up the fields.
- `NONE`: Indicates that the data file contains normal data in the first record. This is the default.

The `FIELD NAMES` clause cannot be used for complex column types such as column objects, nested tables, or VARRAYs.

9.7 Specifying Data Files

Learn how you can use the SQL*Loader control file to specify how data files are loaded.

- Understanding How to Specify Data Files
  To load data files with SQL*Loader, you can specify data files in the control file using the `INFILE` keyword.
- Examples of INFILE Syntax
  The following list shows different ways you can specify `INFILE` syntax.
- Specifying Multiple Data Files
  To load data from multiple data files in one SQL*Loader run, use an `INFILE` clause for each data file.

9.7.1 Understanding How to Specify Data Files

To load data files with SQL*Loader, you can specify data files in the control file using the `INFILE` keyword.

To specify a data file that contains the data that you want to load, use the `INFILE` keyword, followed by the file name, and the optional file processing options string.

You can specify multiple single files by using multiple `INFILE` keywords. You can also use wildcards in the file names (an asterisk (*) for multiple characters and a question mark (?) for a single character).

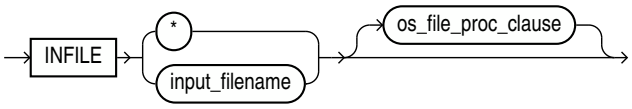
> **Note:**
>
> You can also specify the data file from the command line by using the `DATA` parameter. Refer to the available command-line parameters for SQL*Loader. A file name specified on the command line overrides the first `INFILE` clause in the control file.

If no file name is specified, then the file name defaults to the control file name with an extension or file type of `.dat`.

If the control file itself contains the data that you want loaded, then specify an asterisk (*). This specification is described in the topic "Identifying Data in the Control File with BEGINDATA. .

> **Note:**
>
> The information in this section applies only to primary data files. It does not apply to LOBFILEs or SDFs.

The syntax for `INFILE` is as follows:



The following table describes the parameters for the `INFILE` keyword.

Table 9-1 Parameters for the INFILE Keyword

Parameter	Description
`INFILE`	Specifies that a data file specification follows.

`input_filename`	Name of the file containing the data. The file name can contain wildcards. An asterisk (*) represents multiple characters, and a question mark (?) represents a single character. For example:
	
	`INFILE 'emp*.dat'`
	`INFILE 'm?emp.dat'`
	
	Any spaces or punctuation marks in the file name must be enclosed within single quotation marks.

*	If your data is in the control file itself, then use an asterisk instead of the file name. If you have data in the control file and in data files, then for the data to be read, you must specify the asterisk first.
`os_file_proc_clause`	This is the file-processing options string. It specifies the data file format. It also optimizes data file reads. The syntax used for this string is specific to your operating system.

Related Topics

- Identifying Data in the Control File with BEGINDATA
  Specify the `BEGINDATA` statement before the first data record.
- Specifying File Names and Object Names
  In general, SQL*Loader follows the SQL standard for specifying object names (for example, table and column names).
- Specifying Data File Format and Buffering
  You can specify an operating system-dependent file processing specifications string option using `os_file_proc_clause`.

9.7.2 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:**
  >
  > File names that include spaces or punctuation marks must be enclosed in single quotation marks.

- Data contained in any file of type `.dat` whose name begins with `emp`:

  ```
  INFILE 'emp*.dat'
  ```

- Data contained in any file of type .dat whose name begins with m, followed by any other single character, and ending in emp. For example, a file named myemp.dat would be included in the following:

  ```
  INFILE 'm?emp.dat'
  ```

9.7.3 Specifying Multiple Data Files

To load data from multiple data files in one SQL*Loader run, use an `INFILE` clause for each data file.

Data files 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 data file. In such a case, the separate bad files and discard files must be declared immediately after each data file name. For example, the following excerpt from a control file specifies four data files 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 file name 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`.

9.8 Specifying CSV Format Files

To direct SQL*Loader to access the data files as comma-separated-values format files, use the `CSV` clause.

This assumes that the file is a stream record format file with the normal carriage return string (for example, `\n` on UNIX or Linux operating systems and either `\n` or `\r\n` on Windows operating systems). Record terminators can be included (embedded) in data values. The syntax for the `CSV` clause is as follows:

```
FIELDS CSV [WITH EMBEDDED|WITHOUT EMBEDDED] [FIELDS TERMINATED BY ','] [OPTIONALLY
ENCLOSED BY '""']
```

The following are key points regarding the `FIELDS CSV` clause:

- The SQL*Loader default is to not use the `FIELDS CSV` clause.
- The `WITH EMBEDDED` and `WITHOUT EMBEDDED` options specify whether record terminators are included (embedded) within any fields in the data.
- If `WITH EMBEDDED` is used, then embedded record terminators must be enclosed, and intra-datafile parallelism is disabled for external table loads.
- The `TERMINATED BY ','` and `OPTIONALLY ENCLOSED BY '""'` options are the defaults and do not have to be specified. You can override them with different termination and enclosure characters.
- When the `CSV` clause is used, only delimitable data types are allowed as control file fields. Delimitable data types include `CHAR`, datetime, interval, and numeric `EXTERNAL`.
- The `TERMINATED BY` and `ENCLOSED BY` clauses cannot be used at the field level when the CSV clause is specified.
- When the `CSV` clause is specified, normal SQL*Loader blank trimming is done by default. You can specify `PRESERVE BLANKS` to avoid trimming of spaces. Or, you can use the SQL functions `LTRIM` and `RTRIM` in the field specification to remove left and/or right spaces.
- When the `CSV` clause is specified, the `INFILE *` clause in not allowed. This means that there cannot be any data included in the SQL*Loader control file.

The following sample SQL*Loader control file uses the `FIELDS CSV` clause with the default delimiters:

```
LOAD DATA
INFILE "mydata.dat"
TRUNCATE
INTO TABLE mytable
FIELDS CSV WITH EMBEDDED
TRAILING NULLCOLS
(
  c0 char,
  c1 char,
  c2 char,
)
```

9.9 Identifying Data in the Control File with BEGINDATA

Specify the `BEGINDATA` statement before the first data record.

If the data is included in the control file itself, then the `INFILE` clause is followed by an asterisk rather than a file name. The actual data is placed in the control file after the load configuration specifications.

The syntax is:

```
BEGINDATA
first_data_record
```

Keep the following points in mind when using the `BEGINDATA` statement:

- If you omit the `BEGINDATA` statement but include data in the control file, then SQL*Loader tries to interpret your data as control information and issues an error message. If your data is in a separate file, then 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.

Related Topics

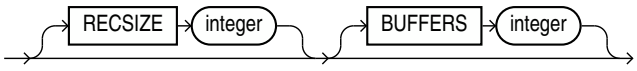
- Examples of INFILE Syntax
  The following list shows different ways you can specify `INFILE` syntax.
- SQL*Loader Case Studies
  To learn how you can use SQL*Loader features, you can run a variety of case studies that Oracle provides.

9.10 Specifying Data File Format and Buffering

You can specify an operating system-dependent file processing specifications string option using `os_file_proc_clause`.

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 file names, and double quotation marks for everything else.

Related Topics

- Windows Processing Options in *Oracle Database Administrator's Reference for Microsoft Windows*

9.11 Specifying the Bad File

Learn what SQL*Loader bad files are, and how to specify them.

- Understanding and 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.
- Examples of Specifying a Bad File Name
  See how you can specify a bad file in a SQL*Loader control file by file name, file name and extension, or by directory.
- How Bad Files Are Handled with LOBFILEs and SDFs
  SQL*Loader manages errors differently for LOBFILE and SDF data.
- Criteria for Rejected Records
  Learn about the criteria SQL*Loader applies for rejecting records in conventional path loads and direct path loads.

9.11.1 Understanding and 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 you want a bad file to be created, then the following processes occur:

- If one or more records are rejected, then the bad file is created and the rejected records are logged.
- If no records are rejected, then the bad file is not created.
- If the bad file is created, then it overwrites any existing file with the same name; ensure that you do not overwrite a file you want to retain.

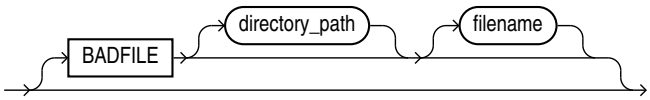
> **Note:**
>
> On some systems, a new version of the file can be created if a file with the same name already exists.

To specify the name of the bad file, use the `BADFILE` clause. You can also specify the bad file from the command line by using the `BAD` parameter.

A file name specified on the command line is associated with the first `INFILE` clause in the control file. If present, then this association overrides any bad file previously specified as part of that clause.

The bad file is created in the same record and file format as the data file, so that you can reload the data after you correct it. For data files in stream record format, the record terminator that is found in the data file is also used in the bad file.

The syntax for the `BADFILE` clause is as follows:



The `BADFILE` clause specifies that a directory path or file name, or both, for the bad file follows. If you specify `BADFILE`, then you must supply either a directory path or a file name, or both.

The *`directory`* parameter specifies a directory path to which the bad file will be written.

The *`filename`* parameter specifies a valid file name specification for your platform. Any spaces or punctuation marks in the file name must be enclosed in single quotation marks. If you do not specify a name for the bad file, then the name defaults to the name of the data file with an extension or file type of `.bad`.

Related Topics

- Command-Line Parameters for SQL*Loader
  Manage SQL*Loader by using the command-line parameters.

9.11.2 Examples of Specifying a Bad File Name

See how you can specify a bad file in a SQL*Loader control file by file name, file name and extension, or by directory.

To specify a bad file with file name `sample` and default file extension or file type of `.bad`, enter the following in the control file:

```
BADFILE sample
```

To specify only a directory name, enter the following in the control file:

```
BADFILE '/mydisk/bad_dir/'
```

To specify a bad file with file name `bad0001` and file extension or file type of `.rej`, enter either of the following lines in the control file:

```
BADFILE bad0001.rej
BADFILE '/REJECT_DIR/bad0001.rej'
```

9.11.3 How Bad Files Are Handled with LOBFILEs and SDFs

SQL*Loader manages errors differently for LOBFILE and SDF data.

When there are rejected rows, SQL*Loader does not write LOBFILE and SDF data to a bad file.

If SQL*Loader encounters an error loading a large object (LOB), then the row is *not* rejected. Instead, 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.

9.11.4 Criteria for Rejected Records

Learn about the criteria SQL*Loader applies for rejecting records in conventional path loads and direct path loads.

SQL*Loader can reject a record for the following reasons:

1. Upon insertion, the record causes an Oracle error (such as invalid data for a given data type).
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 list of reasons.

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.

In a conventional path load, if the data file has a record that is being loaded into multiple tables and that record is rejected from at least one of the tables, then that record is not loaded into any of the tables.

The log file indicates the Oracle error for each rejected record. Case study 4 in "SQL*Loader Case Studies" demonstrates rejected records.

Related Topics

- SQL*Loader Case Studies
  To learn how you can use SQL*Loader features, you can run a variety of case studies that Oracle provides.

9.12 Specifying the Discard File

Learn what SQL*Loader discard files are, what they contain, and how to specify them.

- Understanding and Specifying the Discard File
  During processing of records, SQL*Loader can create a discard file for records that do not meet any of the loading criteria.
- Specifying the Discard File in the Control File
  To specify the name of the file, use the DISCARDFILE clause, followed by a directory path and/or file name.
- Limiting the Number of Discard Records
  To limit the number of records that are discarded for each data file, specify an integer value for either the DISCARDS or DISCARDMAX parameter.
- Examples of Specifying a Discard File Name
  The list shows different ways that you can specify a name for the discard file from within the control file.

- Criteria for Discarded Records
  If there is no `INTO TABLE` clause specified for a record, then the record is discarded.
- How Discard Files Are Handled with LOBFILEs and SDFs
  When there are discarded rows, SQL*Loader does not write data from large objects (LOB) data LOBFILEs and Secondary Data File (SDF) files to a discard file.
- Specifying the Discard File from the Command Line
  To specify a discard file at the time you run SQL*Loader from the command line, use the `DISCARD` command-line parameter for SQL*Loader

9.12.1 Understanding and Specifying the Discard File

During processing of records, SQL*Loader can create a discard file for records that do not meet any of the loading criteria.

The records that are contained in the discard 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 file name and one or more records fail to satisfy all of the `WHEN` clauses specified in the control file. (Be aware that if the discard file is created, then it overwrites any existing file with the same name.)
- If no records are discarded, then a discard file is not created.

You can specify the discard file from within the control file either by specifying its directory, or name, or both, or by specifying the maximum number of discards. Any of the following clauses result in a discard file being created, if necessary:

- `DISCARDFILE=[[directory/][filename]]`
- `DISCARDS`
- `DISCARDMAX`

The discard file is created in the same record and file format as the data file. For data files in stream record format, the same record terminator that is found in the data file is also used in the discard file.

You can also create a discard file from the command line by specifying either the `DISCARD` or `DISCARDMAX` parameter.

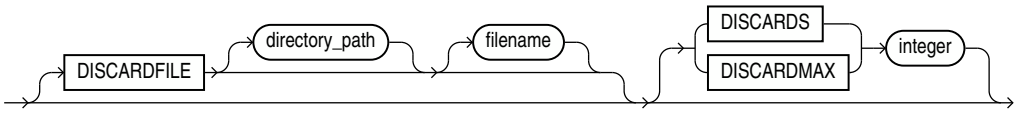
If no discard clauses are included in the control file or on the command line, then a discard file is not created even if there are discarded records (that is, records that fail to satisfy all of the `WHEN` clauses specified in the control file).

Related Topics

- SQL*Loader Command-Line Reference
  To start regular SQL*Loader, use the command-line parameters.

9.12.2 Specifying the Discard File in the Control File

To specify the name of the file, use the `DISCARDFILE` clause, followed by a directory path and/or file name.



The `DISCARDFILE` clause specifies that a discard directory path and/or file name follows. Neither the `directory_path` nor the `filename` is required. However, you must specify at least one.

The *`directory`* parameter specifies a directory to which the discard file will be written.

The *`filename`* parameter specifies a valid file name specification for your platform. Any spaces or punctuation marks in the file name must be enclosed in single quotation marks.

The default file name is the name of the data file, and the default file extension or file type is `.dsc`. A discard file name specified on the command line overrides one specified in the control file. If a discard file with that name already exists, then it is either overwritten or a new version is created, depending on your operating system.

9.12.3 Limiting the Number of Discard Records

To limit the number of records that are discarded for each data file, specify an integer value for either the `DISCARDS` or `DISCARDMAX` parameter.

The integer that you specify for either the `DISCARDS` or `DISCARDMAX` keyword is the numerical maximum number of discard records. If you do not specify a maximum number discard records, then SQL*Loader will continue to discard records. Otherwise, when the discard limit is reached, processing of the data file terminates, and continues with the next data file, if one exists.
You can choose to specify a different number of discards for each data file. 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 file name, then SQL*Loader creates a discard file with the default file name (named after the process that creates it), and the default file extension or file type (`dsc`). For example, The file is named after the process that creates it. For example: `finance.dsc`.

The following example allows 25 records to be discarded during the load before it is terminated.

```
DISCARDMAX=25
```

9.12.4 Examples of Specifying a Discard File Name

The list shows different ways that you can specify a name for the discard file from within the control file.

- To specify a discard file with file name `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'
```

9.12.5 Criteria for Discarded Records

If there is no `INTO TABLE` clause specified for a record, then 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, provides an example of using a discard file. (See SQL*Loader Case Studies for information on how to access case studies.)

9.12.6 How Discard Files Are Handled with LOBFILEs and SDFs

When there are discarded rows, SQL*Loader does not write data from large objects (LOB) data LOBFILEs and Secondary Data File (SDF) files to a discard file.

9.12.7 Specifying the Discard File from the Command Line

To specify a discard file at the time you run SQL*Loader from the command line, use the `DISCARD` command-line parameter for SQL*Loader

The `DISCARD` parameter gives you the option to provide a specification at the command line to identify a discard file where you can store records that are neither inserted into a table nor rejected.

When you specify a file name on the command line, this specification overrides any discard file name that you may have specified in the control file.

Related Topics

- DISCARD
  The `DISCARD` command-line parameter for SQL*Loader lets you optionally specify a discard file to store records that are neither inserted into a table nor rejected.

9.13 Specifying a NULLIF Clause At the Table Level

To load a table character field as NULL when it contains certain character strings or hex strings, you can use a `NULLIF` clause at the table level with SQL*Loader.

The `NULLIF` syntax in the SQL*Loader control file is as follows:

```
NULLIF {=|!=}{"char_string"|x'hex_string'|BLANKS}
```

The `char_string` and `hex_string` values must be enclosed in either single quotation marks or double quotation marks.

This specification is used for each mapped character field unless a `NULLIF` clause is specified at the field level. A `NULLIF` clause specified at the field level overrides a `NULLIF` clause specified at the table level.

SQL*Loader checks the specified value against the value of the field in the record. If there is a match using the equal or not equal specification, then the field is set to `NULL` for that row. Any field that has a length of 0 after blank trimming is also set to `NULL`.

If you do not want the default `NULLIF` or any other `NULLIF` clause applied to a field, then you can specify `NO NULLIF` at the field level.

Related Topics

- Using the WHEN, NULLIF, and DEFAULTIF Clauses
  Learn how SQL*Loader processes the `WHEN`, `NULLIF`, and`DEFAULTIF` clauses with scalar fields.

9.14 Specifying Datetime Formats At the Table Level

You can specify certain datetime formats in a SQL*Loader control file at the table level, or override a table level format by specifying a mask at the field level.

You can specify certain datetime data type (**datetime**) formats at the table level in a SQL*Loader control file.

The syntax for each datetime format that you can specify at the table level is as follows:

```
DATE FORMAT mask
TIMESTAMP FORMAT mask
TIMESTAMP WITH TIME ZONE mask
TIMESTAMP WITH LOCAL TIME ZONE mask
```

This datetime specification is used for every date or timestamp field, unless a different mask is specified at the field level. A mask specified at the field level overrides a mask specified at the table level.

The following is an example of using the `DATE FORMAT` clause in a SQL*Loader control file. The `DATE FORMAT` clause is overridden by `DATE` at the field level for the `hiredate` and `entrydate` fields:

```
LOAD DATA
     INFILE myfile.dat
     APPEND
     INTO TABLE EMP
     FIELDS TERMINATED BY ","
     DATE FORMAT "DD-Month-YYYY"
     (empno,
      ename,
      job,
      mgr,
      hiredate DATE,
      sal,
      comm,
      deptno,
      entrydate DATE)
```

Related Topics

- Categories of Datetime and Interval Data Types
  The SQL*Loader portable value datetime records date and time fields, and the interval data types record time intervals.

9.15 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*

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.
- Unicode Character Sets
  SQL*Loader supports loading data that is in a Unicode character set.
- Database Character Sets
  The Oracle database uses the database character set for data stored in SQL `CHAR` data types (`CHAR`, `VARCHAR2`, `CLOB`, and `LONG`), for identifiers such as table names, and for SQL statements and PL/SQL source code.
- Data File Character Sets
  By default, the data file is in the character set defined by the `NLS_LANG` parameter.
- Input Character Conversion
  The default character set for all data files, if the `CHARACTERSET` parameter is not specified, is the session character set defined by the `NLS_LANG` parameter.
- Shift-sensitive Character Data
  In general, loading shift-sensitive character data can be much slower than loading simple ASCII or EBCDIC data.

9.15.1 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.

9.15.2 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.
- Oracle recommends using AL32UTF8 as the database character set. AL32UTF8 is the proper implementation of the Unicode encoding UTF-8. Starting with Oracle Database 12*c* Release 2, AL32UTF8 is used as the default database character set while creating a database using Oracle Universal Installer (OUI) as well as Oracle Database Configuration Assistant (DBCA).
- Do not use UTF8 as the database character set as it is not a proper implementation of the Unicode encoding UTF-8. If the UTF8 character set is used where UTF-8 processing is expected, then data loss and security issues may occur. This is especially true for Web related data, such as XML and URL addresses.
- AL32UTF8 and UTF8 character sets are not compatible with each other as they have different maximum character widths (four versus three bytes per character).

See Also:

- Case study 11, Loading Data in the Unicode Character Set (see SQL*Loader Case Studies for information on how to access case studies)
- *Oracle Database Globalization Support Guide* for more information about Unicode encoding

9.15.3 Database Character Sets

The Oracle database uses the database character set for data stored in SQL `CHAR` data types (`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` data types (`NCHAR`, `NVARCHAR2`, 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.

9.15.4 Data File Character Sets

By default, the data file is in the character set defined by the `NLS_LANG` parameter.

The data file 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 data file (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 data file. 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 data file can be set up by using the `NLS_LANG` parameter or by specifying a SQL*Loader `CHARACTERSET` parameter.

9.15.5 Input Character Conversion

The default character set for all data files, if the `CHARACTERSET` parameter is not specified, is the session character set defined by the `NLS_LANG` parameter.

The character set used in input data files can be specified with the `CHARACTERSET` parameter.

SQL*Loader can automatically convert data from the data file 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 data file 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, then 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.

- Considerations When Loading Data into VARRAYs or Primary-Key-Based REFs
  If you load data into `VARRAY` or into a primary-key-based `REF`, then issues can occur when the data uses a different character set than the database.
- CHARACTERSET Parameter
  Specifying the `CHARACTERSET` parameter tells SQL*Loader the character set of the input data file.
- 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.
- Character-Length Semantics
  Byte-length semantics are the default for all data files except those that use the UTF16 character set (which uses character-length semantics by default).

See Also:

- Character-Length Semantics
- *Oracle Database Globalization Support Guide*

9.15.5.1 Considerations When Loading Data into VARRAYs or Primary-Key-Based REFs

If you load data into `VARRAY` or into a primary-key-based `REF`, then issues can occur when the data uses a different character set than the database.

If you use SQL*Loader conventional path or the Oracle Call Interface (OCI) to load data into `VARRAY`s or into primary-key-based `REF`s, and the data being loaded is in a different character set than the database character set, then problems such as the following might occur:

- Rows can be rejected because a field is too large for the database column, but in reality the field is not too large.
- A load can be terminated atypically, without any rows being loaded, when only the field that really was too large should have been rejected.
- Rows can 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.

9.15.5.2 CHARACTERSET Parameter

Specifying the `CHARACTERSET` parameter tells SQL*Loader the character set of the input data file.

The default character set for all data files, 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 data types `CHAR`, `VARCHAR`, `VARCHARC`, numeric `EXTERNAL`, and the datetime and interval data types) is affected by the character set of the data file.

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 data file, the data in the data file 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 data file that has little-endian byte order, then SQL*Loader issues a warning message and processes the data file as little-endian.

The CHARACTERSET parameter can be specified for primary data files and also for LOBFILEs and SDFs. All primary data files are assumed to be in the same character set. A CHARACTERSET parameter specified before the INFILE parameter applies to the entire list of primary data files. If the CHARACTERSET parameter is specified for primary data files, then 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 specified with the INFILE clause in the control file. The control file is always processed using the character set specified for your session by the NLS_LANG parameter. Therefore, 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 data file.

See Also:

- Byte Ordering
- *Oracle Database Globalization Support Guide* for more information about the names of the supported character sets
- Control File Character Set
- Case study 11, Loading Data in the Unicode Character Set, for an example of loading a data file that contains little-endian UTF-16 encoded data. (See SQL*Loader Case Studies for information on how to access case studies.)

9.15.5.3 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 data file character set, then 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 data file 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 data file in the UTF-16 Unicode encoding, then 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 data files 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.

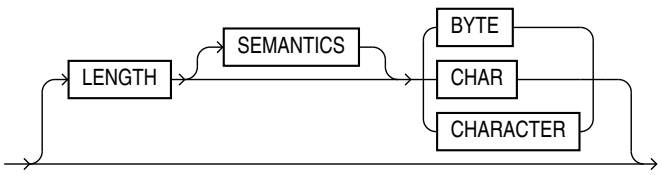
For the SQL*Loader data types (CHAR, VARCHAR, VARCHARC, DATE, and EXTERNAL numerics), SQL*Loader supports lengths of character fields that are specified in either bytes (byte-length semantics) or characters (character-length semantics). For example, the specification CHAR(10) in the control file can mean 10 bytes or 10 characters. These are equivalent if the data file uses a single-byte character set. However, they are often different if the data file 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 data file and the target database columns.

9.15.5.4 Character-Length Semantics

Byte-length semantics are the default for all data files 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 data files and also to LOBFILEs and secondary data files (SDFs). A LENGTH specification before the INFILE parameters applies to the entire list of primary data files. The LENGTH specification specified for the primary data file 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 data file, then the following SQL*Loader data types will use character-length semantics:

- CHAR

- VARCHAR
- VARCHARC
- DATE
- EXTERNAL numerics (INTEGER, FLOAT, DECIMAL, and ZONED)

For the VARCHAR data type, the length subfield is still a binary SMALLINT length subfield, but its value indicates the length of the character string in characters.

The following data types use byte-length semantics even if character-length semantics are being used for the data file, 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 data file. This is necessary to handle data files that have a mix of data of different data types, 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 data type, 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 data file can be used independent of whether character-length semantics are used for the database columns. Therefore, the data file and the database columns can use either the same or different length semantics.

9.15.6 Shift-sensitive Character Data

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, then 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.

9.16 Interrupted Loads

Loads are interrupted and discontinued for several 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 data file, 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.

- Discontinued Conventional Path Loads
  In a conventional path load, data is committed after all data in the bind array is loaded into all tables.
- Discontinued Direct Path Loads
  In a direct path load, the behavior of a discontinued load varies depending on the reason the load was discontinued.
- 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.
- 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 data file.
- 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.

> **See Also:**
>
> SKIP

9.16.1 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, then only the rows that were processed up to the time of the last commit operation are loaded. There is no partial commit of data.

9.16.2 Discontinued Direct Path Loads

In a direct path load, the behavior of a discontinued load varies depending on the reason the load was discontinued.

These sections describe the reasons why a load was discontinued:

- Load Discontinued Because of Space Errors
  If a load is discontinued because of space errors, then the behavior of SQL*Loader depends on whether you are loading data into multiple subpartitions.
- Load Discontinued Because Maximum Number of Errors Exceeded
  If the maximum number of errors is exceeded, then SQL*Loader stops loading records into any table and the work done to that point is committed.
- Load Discontinued Because of Fatal Errors
  If a fatal error is encountered, then the load is stopped and no data is saved unless `ROWS` was specified at the beginning of the load.
- Load Discontinued Because a Ctrl+C Was Issued
  If SQL*Loader is in the middle of saving data when a Ctrl+C is issued, then it continues to do the save and then stops the load after the save completes.

9.16.2.1 Load Discontinued Because of Space Errors

If a load is discontinued because of space errors, then the behavior of SQL*Loader depends on whether you are loading data into multiple subpartitions.

- **Space errors when loading data into multiple subpartitions (that is, loading into a partitioned table, a composite partitioned table, or one partition of a composite partitioned table):**

  If space errors occur when loading into multiple subpartitions, then the load is discontinued and no data is saved unless `ROWS` has been specified (in which case, all data that was previously committed will be saved). The reason for this behavior is that it is possible rows might be loaded out of order. This is because each row is assigned (not necessarily in order) to a partition and each partition is loaded separately. If the load discontinues before all rows assigned to partitions are loaded, then the row for record "n" may have been loaded, but not the row for record "n-1". Therefore, the load cannot be continued by simply using `SKIP=N`.

- **Space errors when loading data into an unpartitioned table, one partition of a partitioned table, or one subpartition of a composite partitioned table:**

  If there is one `INTO TABLE` statement in the control file, then SQL*Loader commits as many rows as were loaded before the error occurred.

  If there are multiple `INTO TABLE` statements in the control file, then SQL*Loader loads data already read from the data file into other tables and then commits the data.

  In either case, this behavior is independent of whether the `ROWS` parameter was specified. When you continue the load, you can use the `SKIP` parameter to skip rows that have already been loaded. In the case of multiple `INTO TABLE` statements, a different number of rows could have been loaded into each table, so 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.

9.16.2.2 Load Discontinued Because Maximum Number of Errors Exceeded

If the maximum number of errors is exceeded, then 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.

9.16.2.3 Load Discontinued Because of Fatal Errors

If a fatal error is encountered, then 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.

9.16.2.4 Load Discontinued Because a Ctrl+C Was Issued

If SQL*Loader is in the middle of saving data when a Ctrl+C is issued, then 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.

9.16.3 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, then all indexes are left in a valid state.

If the direct path load method is used, then any indexes on the table are left in an unusable state. You can either rebuild or re-create the indexes before continuing, or after the load is restarted and completes.

Other indexes are valid if no other errors occurred. See Indexes Left in an Unusable State for other reasons why an index might be left in an unusable state.

9.16.4 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 data file.

Use this information to resume the load where it left off.

9.16.5 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

9.17 Assembling Logical Records from Physical Records

This section describes assembling logical records from physical records.

To combine multiple physical records 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.
- Using CONTINUEIF to Assemble Logical Records
  Use CONTINUEIF if the number of physical records to be combined varies.

9.17.1 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. In direct path loads, the default value for COLUMNARRAYROWS is large, so 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
> - Specifying the Number of Column Array Rows and Size of Stream Buffers

9.17.2 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, then the second record would not be added to the first.

The full syntax for `CONTINUEIF` adds even more flexibility:

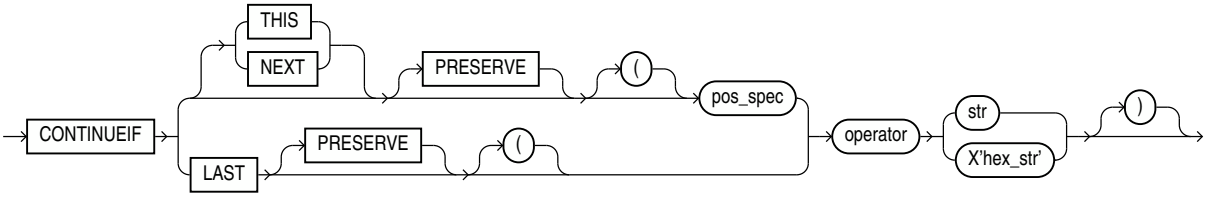


Table 9-2 describes the parameters for the `CONTINUEIF` clause.

Table 9-2 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 can 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`, then 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, then the shorter one is padded. Character strings are padded with blanks, hexadecimal strings with zeros.

Table 9-2 (Cont.) Parameters for the CONTINUEIF Clause

Parameter	Description
str	A string of characters to be compared to the continuation field defined by start and end, according to the operator. The string must be enclosed in double or single quotation marks. The comparison is made character by character, blank padding on the right if necessary.
X'hex-str'	A string of bytes in hexadecimal format used in the same way as `str`. `X'1FB033'` would represent the three bytes with values 1F, B0, and 33 (hexadecimal).
`PRESERVE`	Includes '*char_string*' or *X'hex_string*' in the logical record. The default is to exclude them.

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, then 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, then 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 9-3 through Example 9-6 show the use of `CONTINUEIF THIS` and `CONTINUEIF NEXT`, with and without the `PRESERVE` parameter.

Example 9-3 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....
ddddddddd..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 9-4 CONTINUEIF THIS with the PRESERVE Parameter

Assume that you have the same physical records as in Example 9-3.

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....fffffffff..
```

Note that columns 1 and 2 are not removed from the physical records when the logical records are assembled.

Example 9-5 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..
%%fffffffffff..
```

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 9-3).

```
aaaaaaaa....bbbbbbbb....cccccccc....
ddddddddddd..eeeeeeeeee..fffffffff..
```

Example 9-6 CONTINUEIF NEXT with the PRESERVE Parameter

Assume that you have the same physical records as in Example 9-5.

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..%%fffffffffff..
```

> **See Also:**
>
> Case study 4, Loading Combined Physical Records, for an example of the `CONTINUEIF` clause. (See SQL*Loader Case Studies for information on how to access case studies.)

9.18 Loading Logical Records into Tables

This section describes 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 data types.
- 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.
- 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.)
- 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.
- Specifying Default Data Delimiters
  If all data fields are terminated similarly in the data file, then you can use the `FIELDS` clause to indicate the default termination and enclosure delimiters.
- 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.

9.18.1 Specifying Table Names

The `INTO TABLE` clause of the `LOAD DATA` statement enables you to identify tables, fields, and data types.

It defines the relationship between records in the data file and tables in the database. The specification of fields and data types 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.

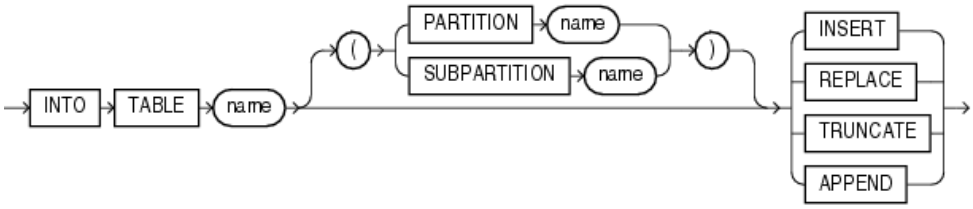
9.18.1.1 INTO TABLE Clause

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 want 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.

9.18.2 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 with INSERT
  To load data into empty tables, use the `INSERT` option.
- Loading Data into Nonempty Tables
  This section describes loading data into nonempty tables.

9.18.2.1 Loading Data into Empty Tables with INSERT

To load data into empty tables, use the `INSERT` option.

If the tables you are loading into are empty, then use the `INSERT` option. The `INSERT` option is the default method for SQL*Loader. To use `INSERT`, the table into which you want to load data

must be empty before you load it. If the table into which you attempt to load data contains rows, then SQL*Loader terminates with an error. Case study 1, Loading Variable-Length Data, provides an example. (See SQL*Loader Case Studies for information on how to access case studies.)

SQL*Loader checks the table into which you insert data to ensure that it is empty. For this reason, the user with which you run `INSERT` must be granted both the `SELECT` and the `INSERT` privilege.

Related Topics

- SQL*Loader Case Studies
  To learn how you can use SQL*Loader features, you can run a variety of case studies that Oracle provides.

9.18.2.2 Loading Data into Nonempty Tables

This section describes loading data into nonempty tables.

If the tables you are loading into already contain data, then you have three options:

- `APPEND`
- `REPLACE`
- `TRUNCATE`

> **Note:**
>
> 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
  You use the `APPEND` clause of `INTO TABLE` to append rows to tables with SQL*Loader.
- REPLACE
- Updating Existing Rows
- TRUNCATE

9.18.2.2.1 APPEND

You use the `APPEND` clause of `INTO TABLE` to append rows to tables with SQL*Loader.

If data already exists in the table, then SQL*Loader appends the new rows to it. If data does not already exist, then 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" provides an example. (See "SQL*Loader Case Studies" for information about how to access case studies.)

Related Topics

- SQL*Loader Case Studies

9.18.2.2.2 REPLACE

The `REPLACE` option executes a SQL `DELETE FROM TABLE` statement. 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, provides an example. (See SQL*Loader Case Studies for information on how to access case studies.)

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 *Oracle Database Concepts.*

9.18.2.2.3 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 `UPDATE` statement with correlated subqueries.
3. Drop the work table.

9.18.2.2.4 TRUNCATE

The `TRUNCATE` option executes a SQL `TRUNCATE TABLE` *`table_name`* `REUSE STORAGE` statement, which means that the table's extents will be reused. The `TRUNCATE` option 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, then SQL*Loader returns an error.

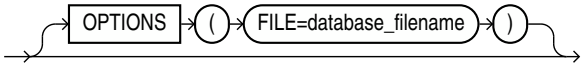
Once the integrity constraints have been disabled, `DELETE CASCADE` is no longer defined for the table. If the `DELETE CASCADE` functionality is needed, then the contents of the table must be manually deleted before the load begins.

The table must be in your schema, or you must have the `DROP ANY TABLE` privilege.

9.18.3 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:

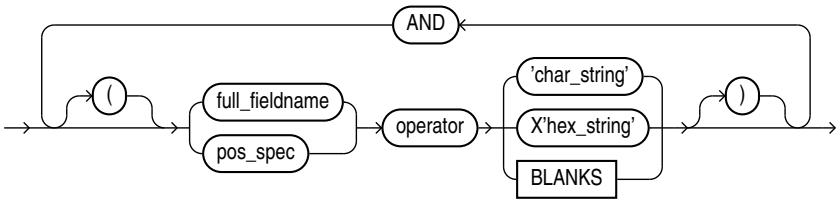


> **See Also:**
>
> Parameters for Parallel Direct Path Loads

9.18.4 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')
```

- Using the WHEN Clause with LOBFILEs and SDFs
  See how to use the `WHEN` clause with `LOBFILE`s and `SDF`s.

> **See Also:**
>
> - Using the WHEN_ NULLIF_ and DEFAULTIF Clauses for information about how SQL*Loader evaluates `WHEN` clauses, as opposed to `NULLIF` and `DEFAULTIF` clauses
> - Case study 5, Loading Data into Multiple Tables, for an example of using the `WHEN` clause (see "SQL*Loader Case Studies" for information on how to access case studies)

9.18.4.1 Using the WHEN Clause with LOBFILEs and SDFs

See how to use the `WHEN` clause with `LOBFILE`s and `SDF`s.

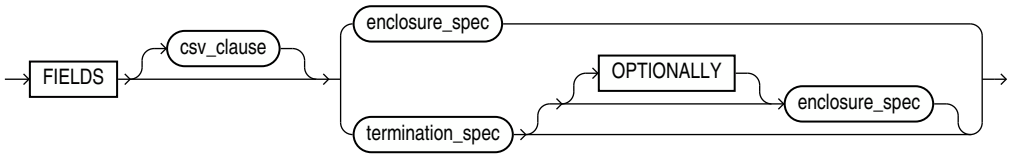
If a record with a `LOBFILE` or `SDF` is discarded, then SQL*Loader does not skip the corresponding data in that `LOBFILE` or `SDF`.

9.18.5 Specifying Default Data Delimiters

If all data fields are terminated similarly in the data file, then you can use the `FIELDS` clause to indicate the default termination and enclosure delimiters.

- fields_spec
- termination_spec
  The syntax for termination specifications is described here.
- enclosure_spec

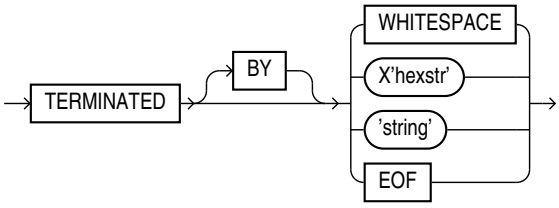
9.18.5.1 fields_spec



> **See Also:**
>
> - Specifying CSV Format Files for information about the `csv_clause`

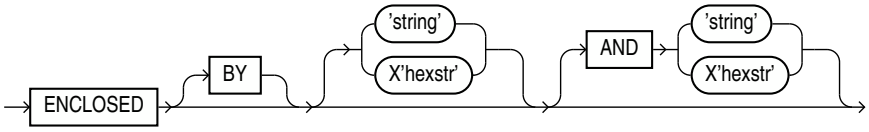
9.18.5.2 termination_spec

The syntax for termination specifications is described here.



> **Note:**
>
> Terminator strings can contain one or more characters. Also, `TERMINATED BY EOF` applies only to loading LOBs from a LOBFILE.

9.18.5.3 enclosure_spec



> **Note:**
>
> Enclosure strings can contain one or more characters.

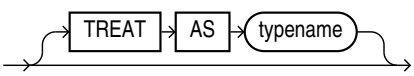
You can override the delimiter for any given column by specifying it after the column name. Case study 3, Loading a Delimited Free-Format File, provides an example. (See SQL*Loader Case Studies for information on how to access case studies.)

> **See Also:**
>
> - Specifying Delimiters for a complete description of the syntax
> - Loading LOB Data from LOBFILEs

9.18.6 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

9.18.6.1 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, for an example of using `TRAILING NULLCOLS` (see SQL*Loader Case Studies for information on how to access case studies)

9.19 Index Options

This section describes index options.

This section describes the following SQL*Loader options that control how index entries are created:

- `SORTED INDEXES`
- `SINGLEROW`
- SORTED INDEXES Clause
- SINGLEROW Option

9.19.1 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

9.19.2 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).

9.20 Benefits of Using Multiple INTO TABLE Clauses

These sections describe the 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.

9.20.1 Extracting Multiple Logical Records

When the data records are short, more than one can be stored in a single, physical record to use the storage space efficiently.

Some data storage and transfer media have fixed-length physical records.

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)
```


9.20.1.1 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 and in Loading Data into Multiple Tables.

9.20.2 Distinguishing Different Input Record Formats

A single data file 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
  Records can be loaded as delimited data.

9.20.2.1 Relative Positioning Based on the POSITION Parameter

Records can be loaded as delimited data.

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.

9.20.3 Distinguishing Different Input Row Object Subtypes

A single data file 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 data file 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 for more information about loading object types

9.20.4 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, for an example. (See SQL*Loader Case Studies for information about how to access case studies.).

9.20.5 Summary of Using Multiple INTO TABLE Clauses

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.

9.21 Bind Arrays and Conventional Path Loads

Multiple rows are read at one time and stored in the bind array.

SQL*Loader uses the SQL array-interface option to transfer data to the database. 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. In general, it does not apply to the direct path load method because a direct path load uses the direct path API. However, the bind array might be used for special cases of direct path load where data conversion is necessary.

- Size Requirements for Bind Arrays
  The bind array must be large enough to contain a single row.
- Performance Implications of Bind Arrays
  Large bind arrays minimize the number of calls to the Oracle database and maximize performance.
- Specifying Number of Rows Versus Size of Bind Array
  When you specify a bind array size using the command-line parameter BINDSIZE or the OPTIONS clause in the control file, you impose an upper limit on the bind array.
- 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.
- 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.
- 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.

> **See Also:**
>
> *Oracle Call Interface Programmer's Guide* for more information about the concepts of direct path loading

9.21.1 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, then 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 maximum value for `ROWS` in a conventional path load is 65534.)

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, then SQL*Loader generates an error.

See Also:

- BINDSIZE
- ROWS

9.21.2 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.

9.21.3 Specifying Number of Rows Versus Size of Bind Array

When you specify a bind array size using the command-line parameter `BINDSIZE` or the `OPTIONS` clause in the control file, 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, then 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` or the `OPTIONS` clause in the control file.

If that size fits within the bind array maximum, then 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, then 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, then SQL*Loader uses a smaller number of rows that fits within the maximum.

9.21.4 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 equals 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 Data Types. 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 data types is described in SQL*Loader Data Types. 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 data types (`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 data types 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
  Use the control file to determine the size of the length indicator.
- Calculating the Size of Field Buffers
  This section describes how to calculate the size of the field buffers.

9.21.4.1 Determining the Size of the Length Indicator

Use the control file to determine 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.

9.21.4.2 Calculating the Size of Field Buffers

This section describes how to calculate the size of the field buffers.

Table 9-3 through Table 9-6 summarize the memory requirements for each data type. "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 Data Types.

Table 9-3 Fixed-Length Fields

Data Type	Size in Bytes (Operating System-Dependent)
INTEGER	The size of the INT data type, in C
INTEGER(*N*)	*N* bytes
SMALLINT	The size of SHORT INT data type, in C
FLOAT	The size of the FLOAT data type, in C
DOUBLE	The size of the DOUBLE data type, 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 data type 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 9-4 Nongraphic Fields

Data Type	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 9-5 Graphic Fields

Data Type	Default Size	Length Specified with POSITION	Length Specified with DATA TYPE
GRAPHIC	None	L	2*L
GRAPHIC EXTERNAL	None	L - 2	2*(L-2)
VARGRAPHIC	4KB*2	L+S	(2*L)+S

Table 9-6 Variable-Length Fields

Data Type	Default Size	Maximum Length Specified (L)
VARCHAR	4 KB	L+S
CHAR (delimited)	255	L+S

Table 9-6 (Cont.) Variable-Length Fields

Data Type	Default Size	Maximum Length Specified (L)
datetime and interval (delimited)	255	L+S
numeric `EXTERNAL` (delimited)	255	L+S

9.21.5 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 data file 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.

9.21.6 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, then 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.

10

SQL*Loader Field List Reference

The field-list portion of a SQL*Loader control file provides information about fields being loaded, such as position, data type, conditions, and delimiters.

- 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 data file.

10.1 Field List Contents

The field-list portion of a SQL*Loader control file provides information about fields being loaded.

The fields are position, data type, conditions, and delimiters.

Example 10-1 shows the field list section of the sample control file that was introduced in SQL*Loader Control File Reference.

Example 10-1 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 .
2. `POSITION` specifies the position of a data field. See Specifying the Position of a Data Field.

   `INTEGER EXTERNAL` is the data type for the field. See Specifying the Data Type of a Data Field and Numeric EXTERNAL.

   The `NULLIF` clause is one of the clauses that can be used to specify field conditions. See Using the WHEN_ NULLIF_ and DEFAULTIF Clauses.

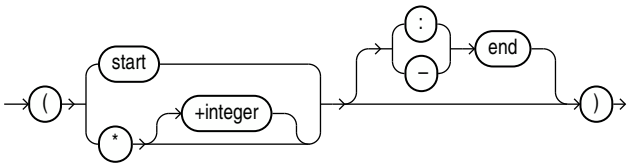
   In this sample, the field is being compared to blanks, using the `BLANKS` parameter. See Comparing Fields to BLANKS.
3. The `TERMINATED BY WHITESPACE` clause is one of the delimiters it is possible to specify for a field. See Specifying Delimiters.
4. The `ENCLOSED BY` clause is another possible field delimiter. See Specifying Delimiters.

10.2 Specifying the Position of a Data Field

To load data from the data file, 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 data file.

The syntax for the position specification (pos_spec) clause is as follows:



The following table describes the parameters for the position specification clause.

Table 10-1 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`, then the length of the field is derived from the data type in the data file. 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 data type, then 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, then 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 data type.
+i*nteger*	You can use an offset, specified as +i*nteger*, 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, then 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 data file.
- Using POSITION with Multiple Table Loads
  This section describes using POSITION with multiple table loads.
- Examples of Using POSITION
  This section shows examples using POSITION.

10.2.1 Using POSITION with Data Containing Tabs

When you are determining field positions, be alert for tabs in the data file.

Suppose you use the SQL*Loader advanced SQL string capabilities to load data from a formatted report. You would probably first look at a printed copy of the report, carefully measure all character positions, and then create your control file. In such a situation, it is highly likely that when you attempt to load the data, the load will fail 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 data file, however, each tab is still only one character. As a result, when SQL*Loader reads the data file, the `POSITION` specifications are wrong.

To fix the problem, inspect the data file for tabs and adjust the `POSITION` specifications, or else use delimited fields.

> **See Also:**
>
> Specifying Delimiters

10.2.2 Using POSITION with Multiple Table Loads

This section describes 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.

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*`)`.

10.2.3 Examples of Using POSITION

This section shows examples using POSITION.

```
siteid   POSITION (*) SMALLINT
siteloc POSITION (*) INTEGER
```

If these were the first two column specifications, then `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.

10.3 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 (unless it is marked `FILLER`) 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, then the specification includes the `RECNUM`, `SEQUENCE`, or `CONSTANT` parameter. See Using SQL*Loader to Generate Data for Input.

If the column's value is read from the data file, then 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, data type, 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 `BOUNDFILLER` or `FILLER` is a data file mapped field that does not correspond to a database column.
- Specifying the Data Type of a Data Field
  The data type specification of a field tells SQL*Loader how to interpret the data in the field.

10.3.1 Specifying Filler Fields

A filler field, specified by `BOUNDFILLER` or `FILLER` is a data file 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 regarding 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, then an error is generated.
- Filler fields can occur anyplace in the data file, 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.

Example 10-2 Filler Field Specification

A 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)
```

10.3.2 Specifying the Data Type of a Data Field

The data type specification of a field tells SQL*Loader how to interpret the data in the field.

For example, a data type 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* data type can be specified for each field; if a data type is not specified, then `CHAR` is assumed.

Before you specify the data type, you must specify the position of the field.

To find out how SQL*Loader data types are converted into Oracle data types, and obtain detailed information about each SQL*Loader data type, refer to "SQL*Loader Data Types."

Related Topics

- SQL*Loader Data Types
  SQL*Loader data types can be grouped into portable and nonportable data types.

10.4 SQL*Loader Data Types

SQL*Loader data types can be grouped into portable and nonportable data types.

- Portable and Nonportable Data Type Differences
  In SQL*Loader, portable data types are platform-independent. Nonportable data types can have several different dependencies that affect portability.
- Nonportable Data Types
  Use this reference to understand how to use the nonportable data types with SQL*Loader.
- Portable Data Types
  Use this reference to understand how to use the portable data types with SQL*Loader.
- Data Type Conversions
  SQL*Loader can perform most data type conversions automatically, but to avoid errors, you need to understand conversion rules.
- Data Type Conversions for Datetime and Interval Data Types
  Learn which conversions between Oracle Database data types and SQL*Loader control file datetime and interval data types are supported, and which are not.
- Specifying Delimiters
  The boundaries of `CHAR`, datetime, interval, or numeric `EXTERNAL` fields can also be marked by delimiter characters contained in the input data record.
- How Delimited Data Is Processed
  To specify delimiters, field definitions can use various combinations of the `TERMINATED BY`, `ENCLOSED BY`, and `OPTIONALLY ENCLOSED BY` clauses.
- Conflicting Field Lengths for Character Data Types
  A control file can specify multiple lengths for the character-data fields `CHAR`, `DATE`, and numeric `EXTERNAL`.

10.4.1 Portable and Nonportable Data Type Differences

In SQL*Loader, portable data types are platform-independent. Nonportable data types can have several different dependencies that affect portability.

For each SQL*Loader data tupe, the data types are subgrouped into value data types and length-value data types.

The terms **portable data type** and **nonportable data type** refer to whether the data type is platform-dependent. Platform dependency can exist for several 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 data types.

Both portable and nonportable data types can be values or length-values. Value data types assume that a data field has a single part. Length-value data types require that the data field consist of two subfields where the length subfield specifies how long the value subfield can be.

> **Note:**
>
> With Oracle Database 12c Release 1 (12.1) and later releases, the maximum size of the Oracle Database `VARCHAR2`, `NVARCHAR2`, and `RAW` data types is 32 KB. To obtain this size, the `COMPATIBLE` initialization parameter must be set to 12.0 or later, and the `MAX_STRING_SIZE` initialization parameter must be set to `EXTENDED`. SQL*Loader supports this maximum size.

10.4.2 Nonportable Data Types

Use this reference to understand how to use the nonportable data types with SQL*Loader.

- Categories of Nonportable Data Types
  Nonportable data types are grouped into two categories: **value data types**, and **length-value data types**.
- INTEGER(n)
  The SQL*Loader nonportable value data type `INTEGER(n)` is a length-specific integer field.
- SMALLINT
  The SQL*Loader nonportable value data type `SMALLINT` is a half-word binary integer.
- FLOAT
  The SQL*Loader nonportable value data type `FLOAT` is a single-precision, floating-point, binary number
- DOUBLE
  The SQL*Loader nonportable value data type `DOUBLE` is a double-precision floating-point binary number.
- BYTEINT
  The SQL*Loader nonportable value data type `BYTEINT` loads the decimal value of the binary representation of the byte.
- ZONED
  The SQL*Loader nonportable value data type ZONED is in zoned decimal format.
- DECIMAL

- VARGRAPHIC
  The SQL*Loader nonportable length-value data type VARGRAPHIC is a varying-length, double-byte character set (DBCS).
- VARCHAR
  The SQL*Loader nonportable length-value data type `VARCHAR` is a binary length subfield followed by a character string of the specified length.
- VARRAW
  The SQL*Loader nonportable length-value data type `VARROW` is a 2-byte binary length subfield, and a `RAW` string value subfield.
- LONG VARRAW
  The SQL*Loader nonportable length-value data type `LONG VARRAW` is a `VARRAW` with a 4-byte length subfield.

10.4.2.1 Categories of Nonportable Data Types

Nonportable data types are grouped into two categories: **value data types**, and **length-value data types**.

The nonportable value data types are:

- `INTEGER(n)`
- `SMALLINT`
- `FLOAT`
- `DOUBLE`
- `BYTEINT`
- `ZONED`
- (packed) `DECIMAL`

The nonportable length-value data types are:

- `VARGRAPHIC`
- `VARCHAR`
- `VARRAW`
- `LONG VARRAW`

To better understand the syntax for nonportable data types, refer to the syntax diagram for `datatype_spec`.

Related Topics

- SQL*Loader Syntax Diagrams
  This appendix describes SQL*Loader syntax in graphic form (sometimes called railroad diagrams or DDL diagrams).

10.4.2.2 INTEGER(*n*)

The SQL*Loader nonportable value data type `INTEGER(n)` is a length-specific integer field.

Definition

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.

Usage Notes

`INTEGER`s are not portable because their byte size, their byte order, and the representation of signed values can be different between systems. However, if the representation of signed values is the same between systems, then it is possible that SQL*Loader can 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 indicate 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, then specify `UNSIGNED`. To return to the default behavior, specify `SIGNED`.

Related Topics

- Loading Data Across Different Platforms
  When a data file 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.

10.4.2.3 SMALLINT

The SQL*Loader nonportable value data type `SMALLINT` is a half-word binary integer.

Definition

The length of a `SMALLINT` field is the length of a half-word integer on your system.

Usage Notes

By default, `SMALLINT` data is treated as a `SIGNED` quantity. If you want SQL*Loader to treat it as an unsigned quantity, then specify `UNSIGNED`. To return to the default behavior, specify `SIGNED`.

You can load `SMALLINT` data 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, then use the appropriate technique to indicate the byte order of the data.

> **Note:**
>
> This is the `SHORT INT` data type 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.

Related Topics

- Understanding how SQL*Loader Manages Byte Ordering
  SQL*Loader can load data from a data file 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 data file contains certain nonportable data types.

10.4.2.4 FLOAT

The SQL*Loader nonportable value data type `FLOAT` is a single-precision, floating-point, binary number

Definition

With FLOAT data, the length of the field is the length of a single-precision, floating-point binary number on your system. (The data type is `FLOAT` in C.) This length cannot be overridden in the control file. If you specify *`end`* in the `POSITION` clause, then *`end`* is ignored.

Usage Notes

You can load `FLOAT` 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, then use the appropriate technique to indicate the byte order of the data.

Related Topics

- Understanding how SQL*Loader Manages Byte Ordering
  SQL*Loader can load data from a data file 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 data file contains certain nonportable data types.

10.4.2.5 DOUBLE

The SQL*Loader nonportable value data type `DOUBLE` is a double-precision floating-point binary number.

Definition

The length of the `DOUBLE` field is the length of a double-precision, floating-point binary number on your system. (The data type is `DOUBLE` or `LONG FLOAT` in C.) This length cannot be overridden in the control file. If you specify *`end`* in the `POSITION` clause, then *`end`* is ignored.

Usage Notes

You can load `DOUBLE` 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, then use the appropriate technique to indicate the byte order of the data.

Related Topics

- Understanding how SQL*Loader Manages Byte Ordering
  SQL*Loader can load data from a data file 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 data file contains certain nonportable data types.

10.4.2.6 BYTEINT

The SQL*Loader nonportable value data type `BYTEINT` loads the decimal value of the binary representation of the byte.

Definition

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 you specify `POSITION (start:end)` then *end* is ignored. (The data type is `UNSIGNED CHAR` in C.)

Example

An example of the syntax for this data type is:

```
(column1 position(1) BYTEINT,
column2 BYTEINT,
...
)
```

10.4.2.7 ZONED

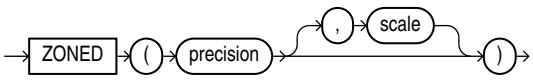
The SQL*Loader nonportable value data type ZONED is in zoned decimal format.

Definition

`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 equals the precision (number of digits) that you specify.

Syntax

The syntax for the `ZONED` data type is as follows:



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.

Example

The following example specifies an 8-digit integer starting at position 32:

```
sal    POSITION(32)    ZONED(8),
```

When the zoned data is generated on an ASCII-based platform, Oracle Database uses the VAX/VMS zoned decimal format. It is also possible to load zoned decimal data that is generated on an EBCDIC-based platform. In this case, Oracle Database uses the IBM format,as specified in the manual *ESA/390 Principles of Operations*, version 8.1. The format that is used depends on the character set encoding of the input data file.

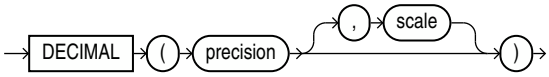
Related Topics

- CHARACTERSET Parameter
  Specifying the `CHARACTERSET` parameter tells SQL*Loader the character set of the input data file.

10.4.2.8 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` data type is as follows:



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.)

10.4.2.9 VARGRAPHIC

The SQL*Loader nonportable length-value data type VARGRAPHIC is a varying-length, double-byte character set (DBCS).

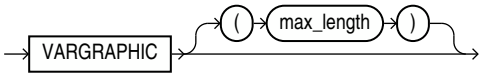
Definition

VARGRAPHIC data consists of a length subfield followed by a string of double-byte characters. Oracle Database does not support double-byte character sets; however, SQL*Loader reads them as single bytes, and loads them as `RAW` data. As with `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` data type on your system (C type `SHORT INT`).

Syntax

The syntax for the VARGRAPHIC data type is:



Usage Notes

You can load VARGRAPHIC data 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, then use the appropriate technique to indicate the byte order of the length subfield.

The length of the current field is given in the first 2 bytes. A maximum length specified for the VARGRAPHIC data type 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 times 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, then it provides the location of the length subfield, not of the first graphic character. If you specify pos_spec*(start:end)*, then 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, then 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, then 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.

Related Topics

- Understanding how SQL*Loader Manages Byte Ordering
  SQL*Loader can load data from a data file 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 data file contains certain nonportable data types.

10.4.2.10 VARCHAR

The SQL*Loader nonportable length-value data type VARCHAR is a binary length subfield followed by a character string of the specified length.

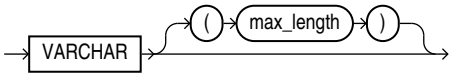
Definition

A VARCHAR field is a length-value data type. 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 data file. In that case, the length is in characters.

> **Note:**
>
> The size of the length subfield is the size of the SQL*Loader `SMALLINT` data type on your system (C type `SHORT INT`).

Syntax

The syntax for the `VARCHAR` data type is:



Usage Notes

`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, then 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.

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` data type, then a buffer of that size, in bytes, is allocated for these fields. However, if character-length semantics are used for the data file, then the buffer size in bytes is the `max_length` times the size in bytes of the largest possible character in the character set.

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)`, then 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, then 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, then 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.

Related Topics

- Character-Length Semantics
  Byte-length semantics are the default for all data files except those that use the UTF16 character set (which uses character-length semantics by default).
- Understanding how SQL*Loader Manages Byte Ordering
  SQL*Loader can load data from a data file 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 data file contains certain nonportable data types.

10.4.2.11 VARRAW

The SQL*Loader nonportable length-value data type `VARROW` is a 2-byte binary length subfield, and a `RAW` string value subfield.

Definition

`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.

Usage Notes

You can load `VARRAW` fields between systems with different byte orders if the appropriate technique is used to indicate the byte order of the length subfield.

Related Topics

- Understanding how SQL*Loader Manages Byte Ordering
  SQL*Loader can load data from a data file 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 data file contains certain nonportable data types.

10.4.2.12 LONG VARRAW

The SQL*Loader nonportable length-value data type `LONG VARRAW` is a `VARRAW` with a 4-byte length subfield.

Definition

`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.

Usage Notes

`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.

Related Topics

- Understanding how SQL*Loader Manages Byte Ordering
  SQL*Loader can load data from a data file 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 data file contains certain nonportable data types.

10.4.3 Portable Data Types

Use this reference to understand how to use the portable data types with SQL*Loader.

The portable data types are grouped into value data types and length-value data types. The portable value data types are `CHAR`, Datetime and Interval, `GRAPHIC`, `GRAPHIC EXTERNAL`, Numeric `EXTERNAL (INTEGER, FLOAT, DECIMAL, ZONE)`, and `RAW`.

The portable length-value data types are VARCHARC and VARRAWC.

The syntax for these data types is shown in the diagram for datatype_spec.

- Categories of Portable Data Types
  Portable data types are grouped into **value data types**, **length-value data types**, and **character data types**.
- CHAR
- Datetime and Interval
  The SQL*Loader portable value datatime data types (**datetime**) and interval data types (intervals) are fields that record dates and time intervals.
- GRAPHIC
  The SQL*Loader portable value data type GRAPHIC has the data in the form of a double-byte character set (DBCS).
- GRAPHIC EXTERNAL
  The SQL*Loader portable value data type GRAPHIC EXTERNAL specifies graphic data loaded from external tables.
- Numeric EXTERNAL
  The SQL*Loader portable value numeric EXTERNAL data types are human-readable, character form data.
- RAW
  The SQL*Loader portable value RAW specifies a load of raw binary data.
- VARCHARC
  The portable length-value data type VARCHARC specifies character string lengths and sizes
- VARRAWC
  The portable length-value data type VARRAWC consists of a RAW string value subfield.
- Conflicting Native Data Type Field Lengths
  If you are loading different data types, then learn what rules SQL*Loader follows to manage conflicts in field length specifications.
- Field Lengths for Length-Value Data Types
  The field lengths for length-value SQL*Loader portable data types such as VARCHAR, VARCHARC, VARGRAPHIC, VARRAW, and VARRAWC is in bytes or characters.

10.4.3.1 Categories of Portable Data Types

Portable data types are grouped into **value data types**, **length-value data types**, and **character data types**.

The portable value data types are:

- CHAR
- Datetime and Interval
- GRAPHIC
- GRAPHIC EXTERNAL
- Numeric EXTERNAL (INTEGER, FLOAT, DECIMAL, ZONE)
- RAW

The portable length-value data types are:

- VARCHARC

- VARRAWC

The character data types are:

- CHAR
- DATE
- numeric EXTERNAL

These fields can be delimited, and can have lengths (or maximum lengths) specified in the control file.

To better understand the syntax for nonportable data types, refer to the syntax diagram for datatype_spec.

Related Topics

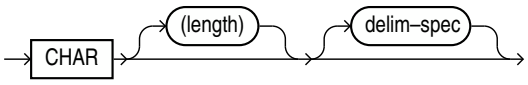
- SQL*Loader Syntax Diagrams
  This appendix describes SQL*Loader syntax in graphic form (sometimes called railroad diagrams or DDL diagrams).

10.4.3.2 CHAR

The data field contains character data. The length, which is optional, is a maximum length. Note the following regarding length:

- If a length is not specified, then it is derived from the POSITION specification.
- If a length *is* specified, then it overrides the length in the POSITION specification.
- If no length is given and there is no POSITION specification, then 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, then 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 data file exceeds maximum length.

The syntax for the CHAR data type is:



> **See Also:**
>
> Specifying Delimiters

10.4.3.3 Datetime and Interval

The SQL*Loader portable value datatime data types (**datetime**) and interval data types (intervals) are fields that record dates and time intervals.

- Categories of Datetime and Interval Data Types
  The SQL*Loader portable value datetime records date and time fields, and the interval data types record time intervals.
- DATE
  The SQL*Loader datetime data type `DATE` field contains character data defining a specified date.
- TIME
  The SQL*Loader datetime data type `TIME` stores hour, minute, and second values.
- TIME WITH TIME ZONE
  The SQL*Loader datetime data type `TIME WITH TIME ZONE` is a variant of `TIME` that includes a time zone displacement in its value.
- TIMESTAMP
  The SQL*Loader datetime data type `TIMESTAMP` is an extension of the `DATE` data type.
- TIMESTAMP WITH TIME ZONE
  The SQL*Loader datetime data type `TIMESTAMP WITH TIME ZONE` is a variant of `TIMESTAMP` that includes a time zone displacement in its value.
- TIMESTAMP WITH LOCAL TIME ZONE
  The SQL*Loader datetime data type `TIMESTAMP WITH LOCAL TIME ZONE`is another variant of `TIMESTAMP` that includes a time zone offset in its value.
- INTERVAL YEAR TO MONTH
  The SQL*Loader interval data type `INTERVAL YEAR TO MONTH` stores a period of time.
- INTERVAL DAY TO SECOND
  The SQL*Loader interval data type `INTERVAL DAY TO SECOND` stores a period of time using the `DAY` and `SECOND` datetime fields.

10.4.3.3.1 Categories of Datetime and Interval Data Types

The SQL*Loader portable value datetime records date and time fields, and the interval data types record time intervals.

Definition

Both datetimes and intervals are made up of fields. The values of these fields determine the value of the data type.

The datetime data types are:

- `DATE`
- `TIME`
- `TIME WITH TIME ZONE`
- `TIMESTAMP`
- `TIMESTAMP WITH TIME ZONE`
- `TIMESTAMP WITH LOCAL TIME ZONE`

The interval data types are:

- `INTERVAL YEAR TO MONTH`
- `INTERVAL DAY TO SECOND`

Usage Notes

Values of datetime data types are sometimes called **datetimes**. 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 data type, the value can be a number in the range 0 to 9. The default is 6.

Values of interval data types are sometimes called **intervals**. The `INTERVAL YEAR TO MONTH` data type gives you the option to 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` data type gives you the option to 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 data type, the value can be a number in the range 0 to 9. The default is 6.

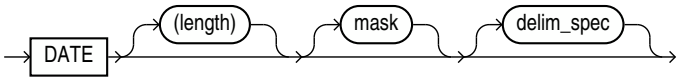
Related Topics

- Specifying Datetime Formats At the Table Level
  You can specify certain datetime formats in a SQL*Loader control file at the table level, or override a table level format by specifying a mask at the field level.
- Numeric Precedence

10.4.3.3.2 DATE

The SQL*Loader datetime data type `DATE` field contains character data defining a specified date.

Syntax



Usage Notes

The `DATE` field contains character data that should be converted to an Oracle date using the specified date mask.

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 data file. In that case, the length is in characters.

If an explicit length is not specified, then it can be derived from the `POSITION` clause. Oracle recommends that you 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 specifications overrides the length derived from the mask. The mask can be any valid Oracle date mask. If you omit the mask, then the default Oracle date mask of "dd-mon-yy" is used.

The length must be enclosed in parentheses, and the mask in quotation marks.

You can also specify a field of data type `DATE` using delimiters.

Example

```
LOAD DATA
INTO TABLE dates (col_a POSITION (1:15) DATE "DD-Mon-YYYY")
BEGINDATA
1-Jan-2012
1-Apr-2012 28-Feb-2012
```

Unless delimiters are present, whitespace is ignored and dates are parsed from left to right. (A `DATE` field that consists entirely of whitespace is loaded as a `NULL` field.)

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, 2012"`, then 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).

Related Topics

- Character-Length Semantics
  Byte-length semantics are the default for all data files except those that use the UTF16 character set (which uses character-length semantics by default).
- Specifying Delimiters
  The boundaries of `CHAR`, datetime, interval, or numeric `EXTERNAL` fields can also be marked by delimiter characters contained in the input data record.

10.4.3.3.3 TIME

The SQL*Loader datetime data type `TIME` stores hour, minute, and second values.

Syntax

```
TIME [(fractional_second_precision)]
```

10.4.3.3.4 TIME WITH TIME ZONE

The SQL*Loader datetime data type `TIME WITH TIME ZONE` is a variant of `TIME` that includes a time zone displacement in its value.

Definition

The time zone displacement is the difference (in hours and minutes) between local time and UTC (coordinated universal time, formerly Greenwich mean time).

Syntax

```
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.

10.4.3.3.5 TIMESTAMP

The SQL*Loader datetime data type `TIMESTAMP` is an extension of the `DATE` data type.

Definition

It stores the year, month, and day of the `DATE` data type, plus the hour, minute, and second values of the `TIME` data type.

Syntax

```
TIMESTAMP [(fractional_second_precision)]
```

If you specify a date value without a time component, then the default time is 12:00:00 a.m. (midnight).

10.4.3.3.6 TIMESTAMP WITH TIME ZONE

The SQL*Loader datetime data type `TIMESTAMP WITH TIME ZONE` is a variant of `TIMESTAMP` that includes a time zone displacement in its value.

Definition

The time zone displacement is the difference (in hours and minutes) between local time and UTC (coordinated universal time, formerly Greenwich mean time).

Syntax

```
TIMESTAMP [(fractional_second_precision)] WITH TIME ZONE
```

10.4.3.3.7 TIMESTAMP WITH LOCAL TIME ZONE

The SQL*Loader datetime data type `TIMESTAMP WITH LOCAL TIME ZONE`is another variant of `TIMESTAMP` that includes a time zone offset in its value.

Definition

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.

Syntax

It is specified as follows:

```
TIMESTAMP [(fractional_second_precision)] WITH LOCAL TIME ZONE
```

10.4.3.3.8 INTERVAL YEAR TO MONTH

The SQL*Loader interval data type `INTERVAL YEAR TO MONTH` stores a period of time.

Definintion

`INTERVAL YEAR TO MONTH` stores a period of time by using the `YEAR` and `MONTH` datetime fields.

Syntax

```
INTERVAL YEAR [(year_precision)] TO MONTH
```

10.4.3.3.9 INTERVAL DAY TO SECOND

The SQL*Loader interval data type `INTERVAL DAY TO SECOND` stores a period of time using the `DAY` and `SECOND` datetime fields.

Definition

The `INTERVAL DAY TO SECOND` data type stores a period of time using the `DAY` and `SECOND` datetime fields.

Syntax

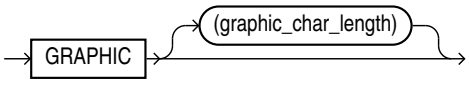
```
INTERVAL DAY [(day_precision)] TO SECOND [(fractional_second_precision)]
```

10.4.3.4 GRAPHIC

The SQL*Loader portable value data type `GRAPHIC` has the data in the form of a double-byte character set (DBCS).

Definition

the `GRAPHIC` data type specifies graphic data:



Usage Notes

The data in `GRAPHIC` is in the form of a double-byte character set (DBCS). Oracle Database does not support double-byte character sets; however, SQL*Loader reads them as single bytes. As with `RAW` data, `GRAPHIC` fields are stored without modification in whichever column you specify.

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) data type, 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 data type specification.

10.4.3.5 GRAPHIC EXTERNAL

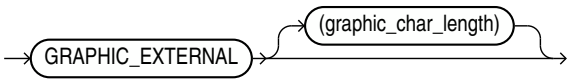
The SQL*Loader portable value data type GRAPHIC EXTERNAL specifies graphic data loaded from external tables.

Description

GRAPHIC indicates that the data is double-byte characters (DBCA). EXTERNAL indicates that the first and last characters are ignored.

If the DBCS field is surrounded by shift-in and shift-out characters, then use GRAPHIC EXTERNAL. This is identical to GRAPHIC, except that the first and last characters (the shift-in and shift-out) are not loaded.

Syntax



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.

Example

To see how GRAPHIC EXTERNAL works, 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).

Related Topics

- GRAPHIC
  The SQL*Loader portable value data type GRAPHIC has the data in the form of a double-byte character set (DBCS).

10.4.3.6 Numeric EXTERNAL

The SQL*Loader portable value numeric EXTERNAL data types are human-readable, character form data.

Definition

The numeric EXTERNAL data types are the numeric data types (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 data file. In that case, the length is in characters.

These data types are the human-readable, character form of numeric data. The same rules that apply to `CHAR` data regarding length, position, and delimiters apply to numeric `EXTERNAL` data. Refer to `CHAR` for a complete description of these rules.

The syntax for the numeric `EXTERNAL` data types is shown as part of the `datatype_spec` SQL*Loader data syntax.

`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.

> **Note:**
>
> The data is a number in character form, not binary representation. Therefore, these data types are identical to `CHAR` and are treated identically, except for the use of `DEFAULTIF`. If you want the default to be null, then use `CHAR`; if you want it to be zero, then use `EXTERNAL`.

Related Topics

- Using the WHEN, NULLIF, and DEFAULTIF Clauses
  Learn how SQL*Loader processes the `WHEN`, `NULLIF`, and`DEFAULTIF` clauses with scalar fields.
- Character-Length Semantics
  Byte-length semantics are the default for all data files except those that use the UTF16 character set (which uses character-length semantics by default).
- CHAR
- SQL*Loader Syntax Diagrams
  This appendix describes SQL*Loader syntax in graphic form (sometimes called railroad diagrams or DDL diagrams).

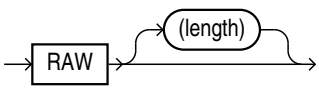
10.4.3.7 RAW

The SQL*Loader portable value `RAW` specifies a load of raw binary data.

Description

When raw, binary data is loaded "as is" into a `RAW` database column, it is not converted when it is place into Oracle Database files.
If the data is loaded into a `CHAR` column, then Oracle Database converts it to hexadecimal. It cannot be loaded into a `DATE` or number column.

Syntax



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 data file. `RAW` data fields cannot be delimited.

10.4.3.8 VARCHARC

The portable length-value data type `VARCHARC` specifies character string lengths and sizes

Description

The SQL*Loader data type `VARCHARC` consists of a character length subfield followed by a character string value-subfield.

Syntax

```
VARCHARC(character_length,character_string)
```

Usage Notes

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 data file, then the length and the maximum size are both in bytes. If character-length semantics are in use for the data file, then the length and maximum size are in characters. If a maximum size is not specified, then 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 data file. If character-length semantics are used, then 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 data file. If character-length semantics are used, then it results in a `VARCHARC` with a length subfield that is 3 characters long and a maximum size of 500 characters.

Example

```
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
```

Related Topics

- VARCHARC and VARRAWC
  The datatype_spec clause VARCHARC data type defines character data, and the VARRAWC data type defines binary data.
- Character-Length Semantics
  Byte-length semantics are the default for all data files except those that use the UTF16 character set (which uses character-length semantics by default).

10.4.3.9 VARRAWC

The portable length-value data type `VARRAWC` consists of a `RAW` string value subfield.

Description

The `VARRAWC` data type has a character count field, followed by binary data.

Syntax

```
VARRAWC(character_length,binary_data)
```

Usage Notes

- `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.

Example

In the following example, `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
```

Related Topics

- VARCHARC and VARRAWC
  The datatype_spec clause VARCHARC data type defines character data, and the VARRAWC data type defines binary data.

10.4.3.10 Conflicting Native Data Type Field Lengths

If you are loading different data types, then learn what rules SQL*Loader follows to manage conflicts in field length specifications.

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 (or precision) specified for 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 data type `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 Data Type
------------------------ --------- ----- ---- ---- ---------
COLUMN1                        1:6     4              INTEGER
```

10.4.3.11 Field Lengths for Length-Value Data Types

The field lengths for length-value SQL*Loader portable data types such as `VARCHAR`, `VARCHARC`, `VARGRAPHIC`, `VARRAW`, and `VARRAWC` is in bytes or characters.

A control file can specify a maximum length for the following length-value data types: `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, then the maximum length defaults to 4096 bytes. If the length of the field exceeds the maximum length, then the record is rejected with the following error:

```
Variable length field exceed maximum length
```

10.4.4 Data Type Conversions

SQL*Loader can perform most data type conversions automatically, but to avoid errors, you need to understand conversion rules.

The data type specifications in the control file tell SQL*Loader how to interpret the information in the data file. The server defines the data types 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 data type 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 data file character set to the database character set when they differ.

> **Note:**
>
> When you use SQL*Loader conventional path to load character data from the data file into a `LONG RAW` column, the character data is interpreted has a HEX string. SQL converts the HEX string into its binary representation. Be aware that any string longer than 4000 bytes exceeds the byte limit for the SQL `HEXTORAW` conversion operator. If a string is longer than the byte limit, then an error is returned. SQL*Loader rejects the row with an error, and continues loading.

The data type of the data in the file does not need to be the same as the data type of the column in the Oracle Database table. Oracle Database automatically performs conversions. However, you need to ensure that the conversion makes sense, and does not generate errors. For instance, when a data file field with data type `CHAR` is loaded into a database column with data type `NUMBER`, you must ensure that the contents of the character field represent a valid number.

> **Note:**
>
> SQL*Loader does *not* contain data type specifications for Oracle internal data types, such as `NUMBER` or `VARCHAR2`. The SQL*Loader data types describe data that can be produced with text editors (*character* data types) and with standard programming languages (*native* data types). However, although SQL*Loader does not recognize data types such as `NUMBER` and `VARCHAR2`, any data that Oracle Database can convert can be loaded into these or other database columns.

10.4.5 Data Type Conversions for Datetime and Interval Data Types

Learn which conversions between Oracle Database data types and SQL*Loader control file datetime and interval data types are supported, and which are not.

How to Read the Data Type Conversions for Datetime and Interval Data Types

In the table, the abbreviations for the Oracle Database data types 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 data types, the definitions for the abbreviations in the table are the same for D, T, TS, YM, and DS. SQL*Loader does *not* contain data type specifications for Oracle Database internal data types, such as NUMBER, CHAR, and VARCHAR2. However, any data that Oracle database can convert can be loaded into these or into other database columns.

For an example of how to read this table, look at the row for the SQL*Loader data type DATE (abbreviated as D). Reading across the row, you can see that data type conversion is supported for the Oracle database data types of CHAR, VARCHAR2, DATE, TIMESTAMP, and TIMESTAMP WITH TIME ZONE data types. However, conversion is not supported for the Oracle Database data types NUMBER, TIME, TIME WITH TIME ZONE, INTERVAL YEAR TO MONTH, or INTERVAL DAY TO SECOND data types.

Table 10-2 Data Type Conversions for Datetime and Interval Data Types

SQL*Loader Data Type	Oracle Database Data Type (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)

10.4.6 Specifying Delimiters

The boundaries of CHAR, datetime, interval, or numeric EXTERNAL fields can also be marked by delimiter characters contained in the input data record.

The delimiter characters are specified using various combinations of the TERMINATED BY, ENCLOSED BY, and OPTIONALLY ENCLOSED BY clauses (the TERMINATED BY clause, if used, must come first). The delimiter specification comes after the data type specification.

For a description of how data is processed when various combinations of delimiter clauses are used, see How Delimited Data Is Processed.

> **Note:**
>
> The RAW data type can 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).

- Syntax for Termination and Enclosure Specification
  The syntax for termination and enclosure specifications is described here.
- Delimiter Marks in the Data
  Sometimes the punctuation mark that is a delimiter must also be included in the data.

- Maximum Length of Delimited Data
  Delimited fields can require significant amounts of storage for the bind array.
- Loading Trailing Blanks with Delimiters
  You can load trailing blanks by specifying `PRESERVE BLANKS`, or you can declare data fields with delimiters, and add delimiters to the data files.

10.4.6.1 Syntax for Termination and Enclosure Specification

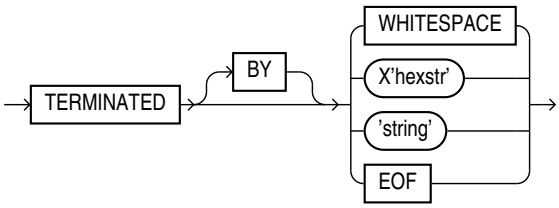
The syntax for termination and enclosure specifications is described here.

Purpose

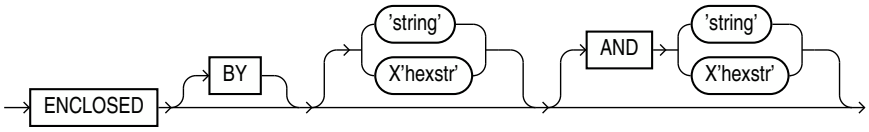
Specifying delimiter characters in the input data record.

Syntax

The following diagram shows the syntax for `termination_spec`.



The following diagram shows the syntax for `enclosure_spec`.



The following table describes the syntax for the termination and enclosure specifications used to specify delimiters.

Parameters

Table 10-3 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`.)

Table 10-3 (Cont.) Parameters Used for Specifying Delimiters

Parameter	Description
OPTIONALLY	Data can be enclosed by the specified character. If SQL*Loader finds a first occurrence of the character, then it reads the data value until it finds the second occurrence. If the data is not enclosed, then the data is read as a terminated field. If you specify an optional enclosure, then you must specify a TERMINATED BY clause (either locally in the field definition or globally in the FIELDS clause).
ENCLOSED	The data is enclosed between two delimiters.
string	The delimiter is a string.
X'hexstr'	The delimiter is a string that has the value specified by *X'hexstr'* in the character encoding scheme, such as X'1F' (equivalent to 31 decimal). "X" can be either lowercase or uppercase.
AND	Specifies a trailing enclosure delimiter that may be different from the initial enclosure delimiter. If AND is not present, then the initial and trailing delimiters are assumed to be the same.
EOF	Indicates that the entire file has been loaded into the LOB. This is valid only when data is loaded from a LOB file. Fields terminated by EOF cannot be enclosed.

Examples

The following is a set of examples of terminations and enclosures, with examples of the data that 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)
```

10.4.6.2 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`.

10.4.6.3 Maximum Length of Delimited Data

Delimited fields can require significant amounts of storage for the bind array.

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.

For example, if you have a string literal that is longer than 255 bytes, then in addition to using `SUBSTR()`, use `CHAR()` to specify the longest string in any record for the field. An example of how this would look is as follows, assuming that 600 bytes is the longest string in any record for `field1`:

```
field1 CHAR(600) SUBSTR(:field, 1, 240)
```

10.4.6.4 Loading Trailing Blanks with Delimiters

You can load trailing blanks by specifying `PRESERVE BLANKS`, or you can declare data fields with delimiters, and add delimiters to the data files.

By default, trailing blanks in nondelimited data types are not loaded unless you specify `PRESERVE BLANKS` in the control file.

If a data field is 9 characters long, and contains the value `DANIEL`*`bbb`*, where *bbb* is three blanks, then it is loaded into Oracle Database as `"DANIEL"` if declared as `CHAR(9)`, without a delimiter.

To include the trailing blanks with a delimiter, declare the data field as `CHAR(9) TERMINATED BY ':'`, and add a colon to the data file, so that the field is `DANIEL`*`bbb`*`:`. As a result of this change, the field is loaded as `"DANIEL "`, with the trailing blanks included. The same results are possible if you specify `PRESERVE BLANKS` without the `TERMINATED BY` clause.

Related Topics

- Trimming Whitespace
  Blanks, tabs, and other nonprinting characters (such as carriage returns and line feeds) constitute 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.

10.4.7 How Delimited Data Is Processed

To specify delimiters, field definitions can use various combinations of the `TERMINATED BY`, `ENCLOSED BY`, and `OPTIONALLY ENCLOSED BY` clauses.

Review these topics to understand how SQL*Loader processes each case of these field definitions.

- Fields Using Only TERMINATED BY
  Data fields that use only `TERMINATED BY` are affected by the location of the delimiter.
- Fields Using ENCLOSED BY Without TERMINATED BY
  When data fields use `ENCLOSED BY` without `TERMINATED BY`, there is a sequence of processing that SQL*Loader uses for those fields.
- Fields Using ENCLOSED BY With TERMINATED BY
  When data fields use `ENCLOSED BY` with `TERMINATED BY`, there is a sequence of processing that SQL*Loader uses for those fields.
- Fields Using OPTIONALLY ENCLOSED BY With TERMINATED BY
  When data fields use `OPTIONALLY ENCLOSED BY` with `TERMINATED BY`, there is a sequence of processing that SQL*Loader uses for those fields.

10.4.7.1 Fields Using Only TERMINATED BY

Data fields that use only `TERMINATED BY` are affected by the location of the delimiter.

If `TERMINATED BY` is specified for a field without `ENCLOSED BY`, then the data for the field is read from the starting position of the field up to, but not including, the first occurrence of the `TERMINATED BY` delimiter. If the terminator delimiter is found in the first column position of a field, then the field is null. If the end of the record is found before the `TERMINATED BY` delimiter, then all data up to the end of the record is considered part of the field.

If `TERMINATED BY WHITESPACE` is specified, then 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 processing behavior enables field values to be delimited by varying amounts of whitespace.

However, unlike non-whitespace terminators, if the first column position of a field is known, and a whitespace terminator is found there, then the field is *not* treated as null. This processing can result in record rejection, or in fields loaded into incorrect columns.

10.4.7.2 Fields Using ENCLOSED BY Without TERMINATED BY

When data fields use `ENCLOSED BY` without `TERMINATED BY`, there is a sequence of processing that SQL*Loader uses for those fields.

The following steps take place when a field uses an `ENCLOSED BY` clause without also using a `TERMINATED BY` clause.

1. Any whitespace at the beginning of the field is skipped.
2. The first non-whitespace character found must be the start of a string that matches the first `ENCLOSED BY` delimiter. If it is not, then the row is rejected.
3. If the first `ENCLOSED BY` delimiter is found, then the search for the second `ENCLOSED BY` delimiter begins.
4. If two of the second `ENCLOSED BY` delimiters are found adjacent to each other, then they are interpreted as a single occurrence of the delimiter, and included as part of the data for the field. The search then continues for another instance of the second `ENCLOSED BY` delimiter.
5. If the end of the record is found before the second `ENCLOSED BY` delimiter is found, then the row is rejected.

10.4.7.3 Fields Using ENCLOSED BY With TERMINATED BY

When data fields use `ENCLOSED BY` with `TERMINATED BY`, there is a sequence of processing that SQL*Loader uses for those fields.

The following steps take place when a field uses an `ENCLOSED BY` clause and also uses a `TERMINATED BY` clause.

1. Any whitespace at the beginning of the field is skipped.
2. The first non-whitespace character found must be the start of a string that matches the first `ENCLOSED BY` delimiter. If it is not, then the row is rejected.
3. If the first `ENCLOSED BY` delimiter is found, then the search for the second `ENCLOSED BY` delimiter begins.
4. If two of the second `ENCLOSED BY` delimiters are found adjacent to each other, then they are interpreted as a single occurrence of the delimiter and included as part of the data for the field. The search then continues for the second instance of the `ENCLOSED BY` delimiter.
5. If the end of the record is found before the second `ENCLOSED BY` delimiter is found, then the row is rejected.
6. If the second `ENCLOSED BY` delimiter is found, then the parser looks for the `TERMINATED BY` delimiter. If the `TERMINATED BY` delimiter is anything other than `WHITESPACE`, then whitespace found between the end of the second `ENCLOSED BY` delimiter and the `TERMINATED BY` delimiter is skipped over.

> **Caution:**
>
> Only `WHITESPACE` is allowed between the second `ENCLOSED BY` delimiter and the `TERMINATED BY` delimiter. Any other characters will cause an error.

7. The row is *not* rejected if the end of the record is found before the `TERMINATED BY` delimiter is found.

10.4.7.4 Fields Using OPTIONALLY ENCLOSED BY With TERMINATED BY

When data fields use `OPTIONALLY ENCLOSED BY` with `TERMINATED BY`, there is a sequence of processing that SQL*Loader uses for those fields.

The following steps take place when a field uses an `OPTIONALLY ENCLOSED BY` clause and a `TERMINATED BY` clause.

1. Any whitespace at the beginning of the field is skipped.
2. The parser checks to see if the first non-whitespace character found is the start of a string that matches the first `OPTIONALLY ENCLOSED BY` delimiter. If it is not, and the `OPTIONALLY ENCLOSED BY` delimiters are *not* present in the data, then the data for the field is read from the current position of the field up to, but not including, the first occurrence of the `TERMINATED BY` delimiter. If the `TERMINATED BY` delimiter is found in the first column position, then the field is null. If the end of the record is found before the `TERMINATED BY` delimiter, then all data up to the end of the record is considered part of the field.
3. If the first `OPTIONALLY ENCLOSED BY` delimiter is found, then the search for the second `OPTIONALLY ENCLOSED BY` delimiter begins.

4. If two of the second `OPTIONALLY ENCLOSED BY` delimiters are found adjacent to each other, then they are interpreted as a single occurrence of the delimiter and included as part of the data for the field. The search then continues for the second `OPTIONALLY ENCLOSED BY` delimiter.
5. If the end of the record is found before the second `OPTIONALLY ENCLOSED BY` delimiter is found, then the row is rejected.
6. If the `OPTIONALLY ENCLOSED BY` delimiter *is* present in the data, then the parser looks for the `TERMINATED BY` delimiter. If the `TERMINATED BY` delimiter is anything other than `WHITESPACE`, then whitespace found between the end of the second `OPTIONALLY ENCLOSED BY` delimiter and the `TERMINATED BY` delimiter is skipped over.
7. The row is *not* rejected if the end of record is found before the `TERMINATED BY` delimiter is found.

> **Caution:**
>
> Be careful when you specify whitespace characters as the `TERMINATED BY` delimiter and are also using `OPTIONALLY ENCLOSED BY`. SQL*Loader strips off leading whitespace when looking for an `OPTIONALLY ENCLOSED BY` delimiter. If the data contains two adjacent `TERMINATED BY` delimiters in the middle of a record (usually done to set a field in the record to NULL), then the whitespace for the first `TERMINATED BY` delimiter will be used to terminate a field, but the remaining whitespace will be considered as leading whitespace for the next field rather than the `TERMINATED BY` delimiter for the next field. To load a NULL value, you must include the `ENCLOSED BY` delimiters in the data.

10.4.8 Conflicting Field Lengths for Character Data Types

A control file can specify multiple lengths for the character-data fields `CHAR`, `DATE`, and numeric `EXTERNAL`.

If conflicting lengths are specified, then 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
  With predetermined size fields, the lengths of fields are determined by the values you specify. If there is a conflict in specifications, then the field length specification is used.
- Delimited Fields
  With delimited fields, the lengths of fields are determined by field semantics and position specifications.
- Date Field Masks
  The length of `DATE` data type fields depends on the format pattern specified in the mask, but can be overridden by position specifications or length specifications.

10.4.8.1 Predetermined Size Fields

With predetermined size fields, the lengths of fields are determined by the values you specify. If there is a conflict in specifications, then the field length specification is used.

If you specify a starting position and ending position for a predetermined field, then the length of the field is determined by the specifications you provide for the data type. If you specify a

length as part of the data type, and do not give an ending position, then 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 data type specification is used for the length of the field. For example:

```
POSITION(1:10) CHAR(15)
```

In this example, the length of the field is 15.

10.4.8.2 Delimited Fields

With delimited fields, the lengths of fields are determined by field semantics and position specifications.

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 the length can be calculated from the start and end positions, then the maximum length defaults to 255 bytes. The actual length can vary up to that maximum, based on the presence of the delimiter.

If delimiters and also starting and ending positions are specified for the field, 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, then SQL*Loader treats any relatively positioned columns that are not present in the record as null columns, so the remaining fields are null. If either the delimiter or the end of record produces a field that is longer than the maximum, then SQL*Loader rejects the record and returns an error.

Related Topics

- TRAILING NULLCOLS Clause

10.4.8.3 Date Field Masks

The length of `DATE` data type fields depends on the format pattern specified in the mask, but can be overridden by position specifications or length specifications.

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, 2012" would occupy 11 bytes in the record (with byte-length semantics), while "January 31, 2012" 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.

Related Topics

- Categories of Datetime and Interval Data Types
  The SQL*Loader portable value datetime records date and time fields, and the interval data types record time intervals.

10.5 Specifying Field Conditions

A field condition is a statement about a field in a logical record that evaluates as true or false.

- Syntax Diagrams for Field Conditions
  Use the syntax diagrams for field conditions to see how to use field conditions with SQL*Loader.
- Comparing Fields to BLANKS
  The `BLANKS` parameter makes it possible to determine if a field of unknown length is blank.
- Comparing Fields to Literals
  Data fields that are compared to literal strings can have blank padding to the string.

10.5.1 Syntax Diagrams for Field Conditions

Use the syntax diagrams for field conditions to see how to use field conditions with SQL*Loader.

Purpose

Field Condition clauses are used in the `WHEN`, `NULLIF`, and `DEFAULTIF` clauses.

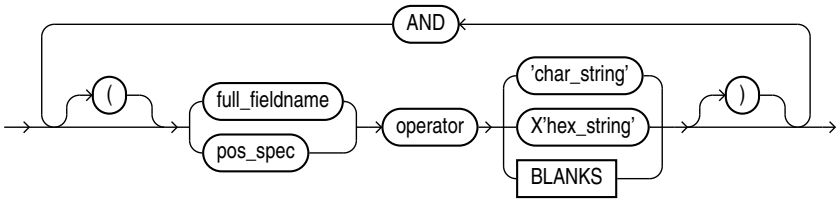
> **Note:**
>
> If a field used in a clause evaluation has a `NULL` value, then that clause will always evaluate to `FALSE`.

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 data file (including filler fields).

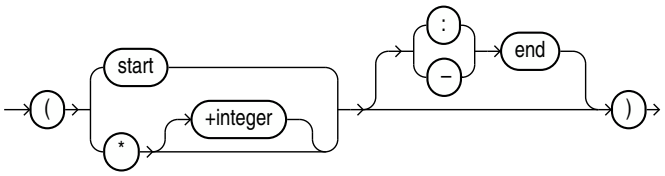
> **Note:**
>
> A field condition cannot be based on fields in a secondary data file (SDF).

Syntax

The syntax for the `field_condition` clause is as follows:



The syntax for the `pos_spec` clause is as follows:



Parameters

The following table describes the parameters used for the field condition clause..

Table 10-4 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, then the length of the field is determined by the length of the comparison string. If the lengths are different, then 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`*, then 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, then 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, then the current record is inserted into the table.

Table 10-4 (Cont.) Parameters for the Field Condition Clause

Parameter	Description
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.

10.5.2 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.

10.5.3 Comparing Fields to Literals

Data fields that are compared to literal strings can have blank padding to the string.

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'.

10.6 Using the WHEN, NULLIF, and DEFAULTIF Clauses

Learn how SQL*Loader processes the WHEN, NULLIF, andDEFAULTIF clauses with scalar fields.

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, then SQL*Loader compares the clause to the evaluated value of the field. The evaluated value takes trimmed whitespace into consideration. For information about trimming blanks and spaces, see:

  Trimming Whitespace

- If the WHEN, NULLIF, or DEFAULTIF clause specifies a position, then SQL*Loader compares the clause to the original logical record in the data file. 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, then 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, then 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, then SQL*Loader checks each field for a NULLIF clause.
4. If a NULLIF clause exists, then SQL*Loader evaluates it.
5. If the NULLIF clause is satisfied, then SQL*Loader sets the field to NULL.
6. If the NULLIF clause is not satisfied, or if there is no NULLIF clause, then 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), then 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, then SQL*Loader checks the field for a DEFAULTIF clause.
8. If a DEFAULTIF clause exists, then SQL*Loader evaluates it.
9. If the DEFAULTIF clause is satisfied, then the field is set to 0 if the field in the data file 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, then 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 you can change your data to take advantage of this processing sequence. NULL detection 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).

Related Topics

- Specifying a NULLIF Clause At the Table Level
  To load a table character field as NULL when it contains certain character strings or hex strings, you can use a NULLIF clause at the table level with SQL*Loader.

10.7 Examples of Using the WHEN, NULLIF, and DEFAULTIF Clauses

These examples explain results for different situations in which you can use the WHEN, NULLIF, and DEFAULTIF clauses.

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 10-3 DEFAULTIF Clause Is Not Evaluated

The control file specifies:

```
(col1 POSITION (1:5),
 col2 POSITION (6:8) CHAR INTEGER EXTERNAL DEFAULTIF col1 = 'aname')
```

The data file contains:

```
aname...
```

In this example, `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 10-4 DEFAULTIF Clause Is Evaluated

The control file specifies:

```
.
.
.
PRESERVE BLANKS
.
.
.
(col1 POSITION (1:5),
 col2 POSITION (6:8) INTEGER EXTERNAL DEFAULTIF col1 = 'aname'
```

The data file contains:

```
aname...
```

In this example, `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 10-5 DEFAULTIF Clause Specifies a Position

The control file specifies:

```
(col1 POSITION (1:5),
 col2 POSITION (6:8) INTEGER EXTERNAL DEFAULTIF (1:5) = BLANKS)
```

The data file contains:

```
.....123
```

In this example, `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 10-6 DEFAULTIF Clause Specifies a Field Name

The control file specifies:

```
(col1 POSITION (1:5),
 col2 POSITION(6:8) INTEGER EXTERNAL DEFAULTIF col1 = BLANKS)
```

The data file contains:

```
.....123
```

In this example, 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.

10.8 Loading Data Across Different Platforms

When a data file 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, then 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 data types. 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* data types. 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 data type implementation may prevent this.

If you cannot use an Oracle Net database link and the data file must be accessed by SQL*Loader running on the target system, then it is advisable to use only the portable SQL*Loader data types (for example, CHAR, DATE, VARCHARC, and numeric EXTERNAL). Data files written using these data types may be longer than those written with native data types. 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. 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.

10.9 Byte Ordering

If you are planning to create input data on a system that has a different byte-ordering scheme than the system on which you plan to run SQL*Loader, then review byte-ordering specifications.

> **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, such as moving data from a big-endian to a little-endian system. If the byte-ordering scheme on each system is the same, then you do not need to read this section.

- Understanding how SQL*Loader Manages Byte Ordering
  SQL*Loader can load data from a data file 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 data file contains certain nonportable data types.
- Byte Order Syntax
  Use the syntax diagrams for `BYTEORDER` to see how to specify byte order of data with SQL*Loader.
- Using Byte Order Marks (BOMs)
  This section describes using byte order marks.
- Suppressing Checks for BOMs
  This section describes suppressing checks for BOMs.

10.9.1 Understanding how SQL*Loader Manages Byte Ordering

SQL*Loader can load data from a data file 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 data file contains certain nonportable data types.

By default, SQL*Loader uses the byte order of the system where it is running as the byte order for all data files. For example, on an Oracle 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 character sets that Oracle supports, 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 data file is only an issue if the data file 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 can identify the byte order of the data, then 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, then SQL*Loader will not correctly load the data into the data file.

Related Topics

- SQL*Loader Case Studies
  To learn how you can use SQL*Loader features, you can run a variety of case studies that Oracle provides.

> **See Also:**
>
> SQL*Loader Case Study 11, Loading Data in the Unicode Character Set, for an example of how SQL*Loader handles byte-swapping.

10.9.2 Byte Order Syntax

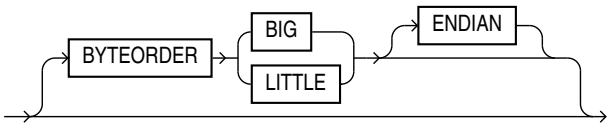
Use the syntax diagrams for `BYTEORDER` to see how to specify byte order of data with SQL*Loader.

Purpose

To specify the byte order of data in the input data files

Syntax

use the following syntax in the SQL*Loader control file:



Usage Notes

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 data files. However, the `BYTEORDER` specification before the `INFILE` parameters applies to the entire list of primary data files.

- The BYTEORDER specification for the primary data files is also used as the default for LOBFILE and SDF data. 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 data types)
  - Character data for data files in the UTF16 character set
  - FLOAT and DOUBLE data types, 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 data types (RAW, VARRAW, or VARRAWC)
  - Graphic data types (GRAPHIC, VARGRAPHIC, or GRAPHIC EXTERNAL)
  - Character data for data files in any character set other than UTF16
  - ZONED or (packed) DECIMAL data types

10.9.3 Using Byte Order Marks (BOMs)

This section describes using byte order marks.

Data files 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 data file that uses the character set UTF16, the values {0xFE,0xFF} in the first two bytes of the file are the BOM indicating that the file contains big-endian data. The values {0xFF,0xFE} are the BOM indicating that the file contains little-endian data.

If the first primary data file uses the UTF16 character set and it also begins with a BOM, then that mark is read and interpreted to determine the byte order for all primary data files. 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 data file. BOMs in data files other than the first primary data file 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 data file.

In summary, the precedence of the byte-order indicators for the first primary data file is as follows:

- BOM in the first primary data file, if the data file 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 data file that uses a UTF8 character set, a BOM of {0xEF,0xBB,0xBF} 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, then it skips it but does not change any byte-order settings for processing the data files.

SQL*Loader first establishes a byte-order setting for the first primary data file using the precedence order just defined. This byte-order setting is used for all primary data files. If another primary data file uses the character set UTF16 and also contains a BOM, then the BOM value is compared to the byte-order setting established for the first primary data file. If the BOM value matches the byte-order setting of the first primary data file, then 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 data file, then SQL*Loader issues an error message and stops processing.

If any LOBFILEs or secondary data files are specified in the control file, then SQL*Loader establishes a byte-order setting for each LOBFILE and secondary data file (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 data file. 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, then 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:

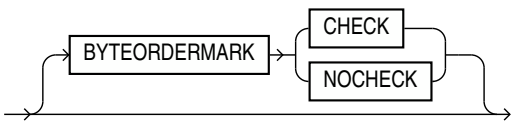
- `BYTEORDER` parameter value specified with the LOBFILE or SDF
- The byte-order setting established for the first primary data file

> **Note:**
>
> If the character set of your data file is a unicode character set and there is a byte-order mark in the first few bytes of the file, then do not use the `SKIP` parameter. If you do, then the byte-order mark will not be read and interpreted as a byte-order mark.

10.9.4 Suppressing Checks for BOMs

This section describes suppressing checks for BOMs.

A data file 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 data file 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 data file as data.

`BYTEORDERMARK CHECK` tells SQL*Loader to check for a BOM. This is the default behavior for a data file 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 data file 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 data files, and also to LOBFILEs and secondary data files (SDFs).
- It is possible to specify a different BYTEORDERMARK value for different data files; however, the BYTEORDERMARK specification before the INFILE parameters applies to the entire list of primary data files.
- The BYTEORDERMARK specification for the primary data files 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 data files can be overridden by specifying BYTEORDERMARK with the LOBFILE or SDF specification.

10.10 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.

Related Topics

- SQL*Loader Case Studies
  To learn how you can use SQL*Loader features, you can run a variety of case studies that Oracle provides.
- Trimming Whitespace
  Blanks, tabs, and other nonprinting characters (such as carriage returns and line feeds) constitute 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.

> **See Also:**
>
> Case study 6, Loading Data Using the Direct Path Load Method, for an example of how to load all-blank fields as NULL with the NULLIF clause, in SQL*Loader Case Studies

10.11 Trimming Whitespace

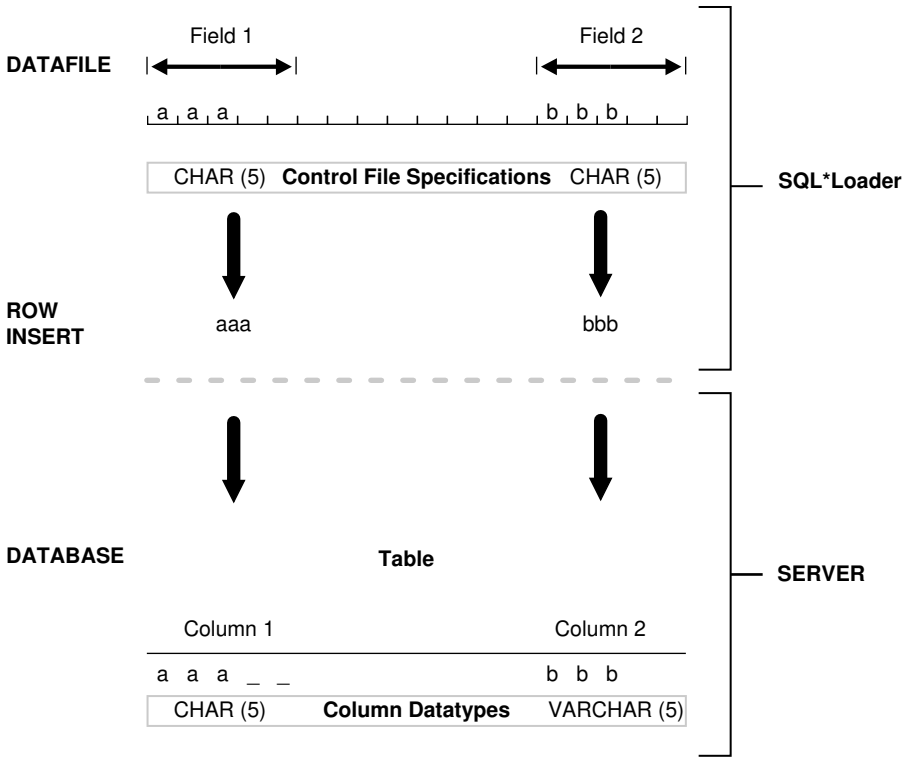
Blanks, tabs, and other nonprinting characters (such as carriage returns and line feeds) constitute whitespace.

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 the figure "Example of Field Conversion, 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 simply tells SQL*Loader how to create the row insert. The data could then be inserted into a `CHAR`, `VARCHAR2`, `NCHAR`, `NVARCHAR2`, 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 the figure "Example of Field Conversion," 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 10-1 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.

The table "Behavior Summary for Trimming Whitespace" 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 for details about how to prevent whitespace trimming.

Table 10-5 Behavior Summary for Trimming Whitespace

Specification	Data	Result	Leading Whitespace Present (When an all-blank field is trimmed, its value is NULL.	Trailing Whitespace Present (When an all-blank field is trimmed, its value is NULL.)
Predetermined size	__aa__	__aa	Yes	No
Terminated	__aa__,	__aa__	Yes	Yes, except for fields that are terminated by whitespace.
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 (Presence of trailing whitespace depends on the current field's specification, as shown by the other entries in the table.)	No	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:

- Data Types for Which Whitespace Can Be Trimmed
  The information in this section applies only to fields specified with one of the character-data data types.
- Specifying Field Length for Data Types for Which Whitespace Can Be Trimmed
  This section describes specifying field length.
- Relative Positioning of Fields
  This section describes the relative positioning of fields.
- Leading Whitespace
  This section describes leading whitespace.
- Trimming Trailing Whitespace
  Trailing whitespace is always trimmed from character-data fields that have a predetermined size.
- Trimming Enclosed Fields
  This section describes trimming enclosed fields.

10.11.1 Data Types for Which Whitespace Can Be Trimmed

The information in this section applies only to fields specified with one of the character-data data types.

- `CHAR` data type
- Datetime and interval data types

- Numeric `EXTERNAL` data types:
  - `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 data file.

10.11.2 Specifying Field Length for Data Types for Which Whitespace Can Be Trimmed

This section describes specifying field length.

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 data type 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, then 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.
- Delimited Fields
  Delimiters are characters that demarcate field boundaries.

10.11.2.1 Predetermined Size Fields

Fields that have a predetermined size are specified with a starting position and ending position, or with a length.

For example:

```
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.

10.11.2.2 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 '|'
```

10.11.3 Relative Positioning of Fields

This section describes the relative positioning of fields.

SQL*Loader determines the starting position of a field in the following situations:

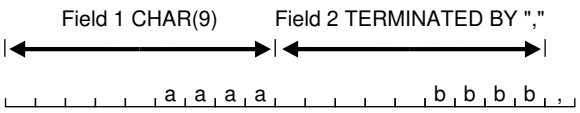
- No Start Position Specified for a Field
  When a starting position is not specified for a field, it begins immediately after the end of the previous field.
- Previous Field Terminated by a Delimiter
  If the previous field (Field 1) is terminated by a delimiter, then the next field begins immediately after the delimiter.
- 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.

10.11.3.1 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.

The following figure illustrates this situation when the previous field (Field 1) has a predetermined size.

Figure 10-2 Relative Positioning After a Fixed Field

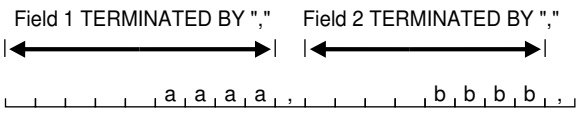


10.11.3.2 Previous Field Terminated by a Delimiter

If the previous field (Field 1) is terminated by a delimiter, then the next field begins immediately after the delimiter.

For example:Figure 10-3.

Figure 10-3 Relative Positioning After a Delimited Field

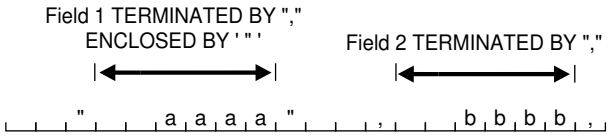


10.11.3.3 Previous Field Has Both Enclosure and Termination Delimiters

When a field is specified with both enclosure delimiters and a termination delimiter, then the next field starts after the termination delimiter.

For example:Figure 10-4. If a nonwhitespace character is found after the enclosure delimiter, but before the terminator, then SQL*Loader generates an error.

Figure 10-4 Relative Positioning After Enclosure Delimiters



10.11.4 Leading Whitespace

This section describes leading whitespace.

In Figure 10-4, 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.

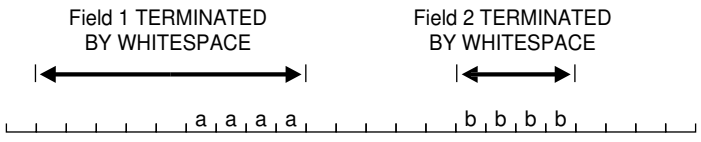
- Previous Field Terminated by Whitespace
  If the previous field is `TERMINATED BY WHITESPACE`, then all whitespace after the field acts as the delimiter.
- Optional Enclosure Delimiters
  Leading whitespace is also removed from a field when optional enclosure delimiters are specified but not present.

10.11.4.1 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 10-5 illustrates this case.

Figure 10-5 Fields Terminated by Whitespace



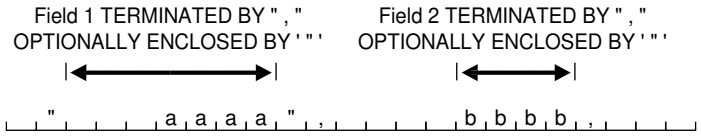
This situation occurs when the previous field is explicitly specified with the `TERMINATED BY WHITESPACE` clause, as shown in the example. It also occurs when you use the global `FIELDS TERMINATED BY WHITESPACE` clause.

10.11.4.2 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, then 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 10-6. (In Field 1 the whitespace is included because SQL*Loader found enclosure delimiters for the field.)

Figure 10-6 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, then 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 10-6.

10.11.5 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.

10.11.6 Trimming Enclosed Fields

This section describes trimming enclosed fields.

If a field is enclosed, or terminated and enclosed, like the first field shown in Figure 10-6, 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.

10.12 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 data type 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, then 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 data type specification for that field, as follows:

```
c1 INTEGER EXTERNAL(10) NO PRESERVE BLANKS
```

10.13 How [NO] PRESERVE BLANKS Works with Delimiter Clauses

The `PRESERVE BLANKS` option is affected by the presence of delimiter clauses

Delimiter clauses affect `PRESERVE BLANKS` in the following cases:

- 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, then 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.

Related Topics

- Trimming Whitespace
  Blanks, tabs, and other nonprinting characters (such as carriage returns and line feeds) constitute whitespace.

10.14 Applying SQL Operators to Fields

This section describes 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 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 data type can be used. SQL strings can be applied to simple scalar column types, and also to user-defined complex types, such as column objects and collections.

The column name and the name of the column in a SQL string bind variable must, with the interpretation of SQL identifier rules, correspond to the same column. But the two names are not required to be written exactly the same way, as in the following example:

```
LOAD DATA
INFILE *
APPEND INTO TABLE XXX
( "Last"   position(1:7)     char   "UPPER(:\"Last\")"
   first   position(8:15)    char   "UPPER(:first || :FIRST || :\"FIRST\")"
)
BEGINDATA
Grant  Phil
Taylor Jason
```

Note the following about the preceding example:

- If, during table creation, a column identifier is declared using double quotation marks because it contains lowercase, or special-case letters, or both (as in the column named "Last" above), then the column name in the bind variable must exactly match the column name used in the CREATE TABLE statement.
- If a column identifier is declared without double quotation marks during table creation (as in the column name first above), then because first, FIRST, and "FIRST" all resolve to FIRST after upper casing is done, any of these written formats in a SQL string bind variable would be acceptable.

Note the following when you are using SQL strings:

- Running SQL strings is not considered to be part of field setting. Instead, when the SQL string is run, it *uses* the result of any field setting and NULLIF or DEFAULTIF clauses. So, the evaluation order is as follows (steps 1 and 2 are a summary of the steps described in "Using the WHEN_NULLIF_ and DEFAULTIF Clauses."):

  1. Field setting is done.

2. Any NULLIF or DEFAULTIF clauses are applied (and that may change the field setting results for the fields that have such clauses). When NULLIF and DEFAULTIF clauses are used with a SQL expression, they affect the field setting results, not the final column results.
3. Any SQL expressions are evaluated using the field results obtained after completion of Steps 1 and 2. The results are assigned to the corresponding columns that have the SQL expressions. (If there is no SQL expression present, then the result obtained from Steps 1 and 2 is assigned to the column.)

- If your control file specifies character input that has an associated SQL string, then 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 must appear 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 require the use of the backslash (escape) character.
- If a SQL string contains a column name that references a column object attribute, then the full object attribute name must be used in the bind variable. Each attribute name in the full name is an individual identifier. Each identifier is subject to the SQL identifier quoting rules, independent of the other identifiers in the full name. For example, suppose you have a column object named CHILD with an attribute name of "HEIGHT_%TILE". (Note that the attribute name is in double quotation marks.) To use the full object attribute name in a bind variable, any one of the following formats would work:

  – :CHILD.\"HEIGHT_%TILE\"

  – :child.\"HEIGHT_%TILE\"

  Enclosing the full name (:\"CHILD.HEIGHT_%TILE\") generates a warning message that the quoting rule on an object attribute name used in a bind variable has changed. The warning is only to suggest to you that the bind variable should be written correctly. It does not indicate that the load will fail. The quoting rule was changed, because enclosing the full name in quotation marks would cause SQL to interpret the name as one identifier, instead of a full column object attribute name consisting of multiple identifiers.
- 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, then the load terminates in error. If the string is recognized, but causes a database error, then the row that caused the error is rejected.
- SQL strings are required when using the EXPRESSION parameter in a field specification.
- The SQL string cannot reference fields that are loaded using OID, SID, REF, or BFILE. Also, the SQL string cannot reference filler fields, or other fields that use SQL strings.
- In direct path mode, a SQL string cannot reference a VARRAY, nested table, or LOB column. This restriction also applies to a VARRAY, nested table, or LOB column that is an attribute of a column object.
- The SQL string cannot be used on RECNUM, SEQUENCE, CONSTANT, or SYSDATE fields.

- The SQL string cannot be used on LOBs, `BFILE`s, `XML` columns, or a file that is an element of a collection.
- In direct path mode, the final result that is returned after evaluation of the expression in the SQL string must be a scalar data type. That is, the expression cannot return an object or collection data type when performing a direct path load.
- Referencing Fields
  To refer to fields in the record, precede the field name with a colon (:).
- Common Uses of SQL Operators in Field Specifications
  If you want to load external data with an implied decimal point, or truncate long fields, then SQL operators in field specifications can help you to manage your data.
- Combinations of SQL Operators
  See how you can combine SQL operators in SQL*Loader to perform multiple steps in data loads.
- 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.
- Interpreting Formatted Fields
  It is possible to use the `TO_CHAR` operator to store formatted dates and numbers.
- Using SQL Strings to Load the ANYDATA Database Type
  The `ANYDATA` database type can contain data of different types.

Related Topics

- Using the WHEN_ NULLIF_ and DEFAULTIF Clauses

10.14.1 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. Note that bind variables enclosed in single quotation marks are treated as text literals, *not* as bind variables.

The following example illustrates how a reference is made to both the current field and to other fields in the control file. It also illustrates how enclosing bind variables in single quotation marks causes them to be treated as text literals. Be sure to read the notes following this example to help you fully understand the concepts it illustrates.

```
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) NULLIF field4.attr2='ZZ' "UPPER(:field4.attr3)",
  attr2  CHAR(2),
  attr3  CHAR(3)  ":field4.attr1 + 1"
 ),
 field5  EXPRESSION "MYFUNC(:FIELD4, SYSDATE)"
)
BEGINDATA
```

```
ABCDEF1234511  ,:field1500YYabc
abcDEF67890    ,:field2600ZZghl
```

Notes About This Example:

- In the following line, `:field1` is *not* enclosed in single quotation marks and is therefore interpreted as a bind variable:

  ```
  field1 POSITION(1:6) CHAR "LOWER(:field1)"
  ```

- In the following line, `':field1'` and `':1'` *are* enclosed in single quotation marks and are therefore treated as text literals and passed unchanged to the `TRANSLATE` function:

  ```
  field3 CHAR(7) "TRANSLATE(:field3, ':field1', ':1')"
  ```

  For more information about the use of quotation marks inside quoted strings, see Specifying File Names and Object Names.

- 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.

- 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 the `field4.attr1` field references the `field4.attr3` field. The `field4.attr1` field is still mapped to the values 500 and NULL (because the `NULLIF field4.attr2='ZZ'` clause is `TRUE` for the second record) in the input records, but the final values stored in its corresponding columns are ABC and GHL.

  The `field4.attr3` field is mapped to the values ABC and GHL in the input records, but the final values stored in its corresponding columns are 500 + 1 = 501 and NULL because the SQL expression references `field4.attr1`. (Adding 1 to a NULL field still results in a NULL field.)

- The `field5` field 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.

10.14.2 Common Uses of SQL Operators in Field Specifications

If you want to load external data with an implied decimal point, or truncate long fields, then SQL operators in field specifications can help you to manage your data.

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)"
  ```

10.14.3 Combinations of SQL Operators

See how you can combine SQL operators in SQL*Loader to perform multiple steps in data loads.

The following examples show how you can apply multiple SQL operators in field specifications with SQL*Loader:

```
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' )"
```

10.14.4 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 data type with a SQL string, a date mask is required. 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` data type.

10.14.5 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.

An example of using the SQL string to load data from a formatted report is shown in case study 7, Extracting Data from a Formatted Report. (See SQL*Loader Case Studies for information on how to access case studies.)

10.14.6 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 called 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 Language Reference* for more information about the `ANYDATA` database type
> - *Oracle Database PL/SQL Packages and Types Reference* for more information about using `ANYDATA` with PL/SQL

10.15 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 data file.

The following parameters are described:

- Loading Data Without Files
  This section describes loading data without files.
- Setting a Column to a Constant Value
  Setting a column to a constant value is the simplest form of generated data.
- 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.
- Setting a Column to the Data File 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.
- 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.
- 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.

- 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.

10.15.1 Loading Data Without Files

This section describes 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 data file—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, then SQL*Loader assumes that data evaluation is necessary, and input records are read.

10.15.2 Setting a Column to a Constant Value

Setting a column to a constant value is the simplest form of generated data.

It does not vary during the load or between loads.

- CONSTANT Parameter

10.15.2.1 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, then 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.

10.15.3 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 data type of the column being loaded.

- EXPRESSION Parameter

10.15.3.1 EXPRESSION Parameter

The combination of column name, `EXPRESSION` parameter, and a SQL string is a complete field specification:

```
column_name EXPRESSION "SQL string"
```

In both conventional path mode and direct path mode, the `EXPRESSION` parameter can be used to load the default value into `column_name`:

```
column_name EXPRESSION "DEFAULT"
```

Note that if `DEFAULT` is used and the mode is direct path, then use of a sequence as a default will not work.

10.15.4 Setting a Column to the Data File 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 data file, starting with record 1. `RECNUM` is incremented as each logical record is assembled. Thus it increments for records that are discarded, skipped, rejected, or loaded. If you use the option `SKIP=10`, then the first record loaded has a `RECNUM` of 11.

- RECNUM Parameter
  The combination of column name and `RECNUM` is a complete column specification.

10.15.4.1 RECNUM Parameter

The combination of column name and `RECNUM` is a complete column specification.

For example:

```
column_name RECNUM
```

10.15.5 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` data type in *Oracle Database SQL Language Reference.*

- SYSDATE Parameter
  The combination of column name and the `SYSDATE` parameter is a complete column specification.

10.15.5.1 SYSDATE Parameter

The combination of column name and the `SYSDATE` parameter is a complete column specification.

For example:

```
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.

10.15.6 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

10.15.6.1 SEQUENCE Parameter

The combination of column name and the `SEQUENCE` parameter is a complete column specification.

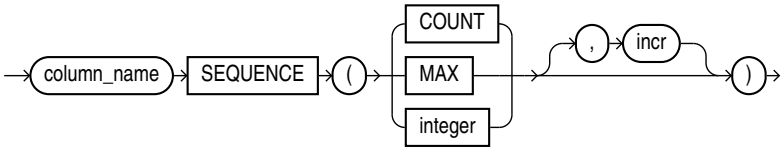


Table 10-6 describes the parameters used for column specification.

Table 10-6 Parameters Used for Column Specification

Parameter	Description
column_name	The name of the column in the database to which to assign the sequence.
`SEQUENCE`	Use the `SEQUENCE` parameter to specify the value for a column.
`COUNT`	The sequence starts with the number of records already in the table plus the increment.
`MAX`	The sequence starts with the current maximum value for the column plus the increment.
integer	Specifies the specific sequence number to begin with.

Table 10-6 (Cont.) Parameters Used for Column Specification

Parameter	Description
incr	The value that the sequence number is to increment after a record is loaded or rejected. This is optional. The default is 1.

If a record is rejected (that is, it has a format error or causes an Oracle error), then 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, then 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, provides an example of using the `SEQUENCE` parameter. (See SQL*Loader Case Studies for information on how to access case studies.)

10.15.7 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

10.15.7.1 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.

11

Loading Objects, LOBs, and Collections with SQL*Loader

You can use SQL*Loader to load column objects in various formats and to load object tables, REF columns, LOBs, and collections.

- Loading Column Objects
  Column objects in the control file are described in terms of their attributes.
- 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.
- Loading REF Columns with SQL*Loader
  SQL*Loader can load system-generated OID `REF` columns, primary-key-based `REF` columns, and unscoped `REF` columns that allow primary keys.
- Loading LOBs with SQL*Loader
  Find out which large object types (LOBs) SQL*Loader can load, and see examples of how to load LOB Data.
- Loading BFILE Columns with SQL*Loader
  The `BFILE` data type stores unstructured binary data in operating system files outside the database.
- Loading Collections (Nested Tables and VARRAYs)
  Like LOBs, collections can be loaded either from a primary data file (data inline) or from secondary data files (data out of line).
- Choosing Dynamic or Static SDF Specifications
  With SQL*Loader, you can specify SDFs either statically (specifying the actual name of the file), or dynamically (using a `FILLER` field as the source of the file name).
- 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.

11.1 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 data file, 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 data types such as 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, then you must specify the full name. For example, if field `fld1` is specified to be a `COLUMN OBJECT` and it contains field `fld2`, then 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
  With stream record formats, you can use SQL*Loader to load records with multi-line fields by specifying a delimitor on column objects.
- Loading Column Objects in Variable Record Format
  You can load column objects in variable record format.
- Loading Nested Column Objects
  You can load nested column objects.
- Loading Column Objects with a Derived Subtype
  You can load column objects with a derived subtype.
- Specifying Null Values for Objects
  You can specify null values for objects.
- Loading Column Objects with User-Defined Constructors
  You can load column objects with user-defined constructors.

11.1.1 Loading Column Objects in Stream Record Format

With stream record formats, you can use SQL*Loader to load records with multi-line fields by specifying a delimitor on column objects.

In stream record format, SQL*Loader forms records by scanning for the record terminator. To show how to use stream record formats, consider the following example, 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 11-1 Loading Column Objects in Stream Record Format

Control File Contents

```
LOAD DATA
INFILE 'example.dat'
INTO TABLE departments
   (dept_no     POSITION(01:03)    CHAR,
    dept_name   POSITION(05:15)    CHAR,
1   dept_mgr    COLUMN OBJECT
      (name     POSITION(17:33)    CHAR,
```

```
        age       POSITION(35:37)     INTEGER EXTERNAL,
        emp_id    POSITION(40:46)     INTEGER EXTERNAL) )
```

Data File (`example.dat`)

```
101 Mathematics  Johnny Quest       30   1024
237 Physics      Albert Einstein   65   0000
```

In the example, note the callout **1** at `dept_mgr COLUMN OBJECT`. You can apply this type of column object specification recursively to describe nested column objects.

11.1.2 Loading Column Objects in Variable Record Format

You can load column objects in variable record format.

Example 11-2 shows a case in which the data is in delimited fields.

Example 11-2 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)) )
```

Data File (sample.dat)

```
3  000034101,Mathematics,Johny Q.,30,1024,
   000039237,Physics,"Albert Einstein",65,0000,
```

> **Note:**
>
> The callouts, in bold, to the left of the example correspond to the following 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, then 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.

11.1.3 Loading Nested Column Objects

You can load nested column objects.

Example 11-3 shows a control file describing nested column objects (one column object nested in another column object).

Example 11-3 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))))
```

Data File (sample.dat)

```
101,Mathematics,Johny Q.,30,1024,"Barbie",650-251-0010,
237,Physics,"Albert Einstein",65,0000,Wife Einstein,654-3210,
```

> **Note:**
>
> The callout, in bold, to the left of the example corresponds to the following note:
>
> 1. This entry specifies a column object nested within a column object.

11.1.4 Loading Column Objects with a Derived Subtype

You can load column objects with a derived subtype.

Example 11-4 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 11-4 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)))
```

Data File (sample.dat)

```
101,Mathematics,Johny Q.,301189453,10249,
237,Physics,"Albert Einstein",128606590,10030,
```

> **Note:**
>
> The callouts, in bold, to the left of the example correspond to the following 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, then this attribute would have resulted in an error, because it is not an attribute of the column's declared type.

11.1.5 Specifying Null Values for Objects

You can specify null values for objects.

Specifying null values for nonscalar data types is somewhat more complex than for scalar data types. 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
  You can specify attribute nulls.
- Specifying Atomic Nulls
  You can specify atomic nulls.

11.1.5.1 Specifying Attribute Nulls

You can specify 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 11-5 demonstrates this.

Example 11-5 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.empid=BLANKS))
```

Data File (sample.dat)

```
2  101            Johny Quest            1024
   237   Physics   Albert Einstein   65   0000
```

> **Note:**
>
> The callouts, in bold, to the left of the example correspond to the following 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.

11.1.5.2 Specifying Atomic Nulls

You can specify 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 11-5, 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 data file).

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 data file (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 11-6.

Example 11-6 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)
```

Data File (sample.dat)

```
101,Mathematics,n,Johny Q.,,1024,"Barbie",608-251-0010,,
237,Physics,,"Albert Einstein",65,0000,,650-654-3210,n,
```

> **Note:**
>
> The callouts, in bold, to the left of the example correspond to the following notes:
>
> 1. The filler field (data file 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.

11.1.6 Loading Column Objects with User-Defined Constructors

You can load 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 11-7 illustrates this difference.

Example 11-7 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,
```

> **Note:**
>
> The callout, in bold, to the left of the example corresponds to the following note:
>
> 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 11-8.

Example 11-8 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 11-7 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 11-9 shows how this can be done.

Example 11-9 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,
```

> **Note:**
>
> The callouts, in bold, to the left of the example correspond to the following note:
>
> 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.

If the control file in Example 11-9 is used in direct path mode, then the following error is reported:

```
SQL*Loader-951: Error calling once/load initialization
ORA-26052: Unsupported type 121 for SQL expression on column EMPLOYEE.
```

11.2 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 11-10 demonstrates loading an object table with primary-key-based object identifiers (OIDs).

Example 11-10 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))
```

Data File (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 data file 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 11-11 demonstrates loading system-generated OIDs with the row objects.

Example 11-11 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))
```

Data File (sample.dat)

```
3  Johny Quest, 18, 007, 21E978406D3E41FCE03400400B403BC3,
   Speed Racer, 16, 000, 21E978406D4441FCE03400400B403BC3,
```

> **Note:**
>
> The callouts, in bold, to the left of the example correspond to the following 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, then the particular record is rejected.
> 3. The OID in the data file 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 Subtypes
  If an object table's row object is based on a nonfinal type, then SQL*Loader allows for any derived subtype to be loaded into the object table.

11.2.1 Loading Object Tables with Subtypes

If an object table's row object is based on a nonfinal type, then SQL*Loader allows for any derived subtype to be loaded into the object table.

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. Use these examples to understand the differences.

Example 11-12 Loading an Object Table with a Subtype

Review the object type definitions, and review the callouts (in **bold**) to understand how the control file is configured.

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))
```

Data File (`sample.dat`)

```
Johny Quest, 18, 007, 32,
Speed Racer, 16, 000, 20,
```

> **Note:**
>
> The callouts in bold, to the left of the example, correspond to the following notes:
>
> 1. The `TREAT AS` clause directs SQL*Loader to treat the object table as if it was 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 is not specified, then using this attribute results in an error, because it is not an attribute of the object table's declared type.

11.3 Loading REF Columns with SQL*Loader

SQL*Loader can load system-generated OID `REF` columns, primary-key-based `REF` columns, and unscoped `REF` columns that allow primary keys.

A REF is an Oracle built-in data type that is a logical "pointer" to an object in an object table. For each of these types of REF columns, you must specify table names correctly for the type.

- Specifying Table Names in a REF Clause
  Use these examples to see how to describe REF clauses in the SQL*Loader control file, and understand case sensitivity.
- System-Generated OID REF Columns
  When you load system-generated `REF` columns, SQL*Loader assumes that the actual OIDs from which the `REF` columns are constructed are in the data file, with the data.
- 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.
- Unscoped REF Columns That Allow Primary Keys
  An unscoped `REF` column that allows primary keys can reference both system-generated and primary key `REF`s.

11.3.1 Specifying Table Names in a REF Clause

Use these examples to see how to describe REF clauses in the SQL*Loader control file, and understand case sensitivity.

> **Note:**
>
> The information in this section applies only to environments in which the release of both SQL*Loader and Oracle Database are 11g release 1 (11.1) or later. It does not apply to environments in which either SQL*Loader, Oracle Database, or both, are at an earlier release.

Example 11-13 REF Clause descriptions in the SQL*Loader Control file

In the SQL*Loader control file, the description of the field corresponding to a `REF` column consists of the column name, followed by a REF clause. The REF clause takes as arguments the table name and any attributes applicable to the type of `REF` column being loaded. The table names can either be specified dynamically (using filler fields), or as constants. The table name can also be specified with or without the schema name.

Whether you specify the table name in the REF clause as a constant, or you specify it by using a filler field, SQL*Loader interprets this specification as interpreted as case-sensitive. If you do not keep this in mind, then the following issues can occur:

- If user `SCOTT` creates a table named `table2` in lowercase without quotation marks around the table name, then it can be used in a REF clause in any of the following ways:
  - `REF(constant 'TABLE2', ...)`
  - `REF(constant '"TABLE2"', ...)`
  - `REF(constant 'SCOTT.TABLE2', ...)`
- If user `SCOTT` creates a table named `"Table2"` using quotation marks around a mixed-case name, then it can be used in a REF clause in any of the following ways:
  - `REF(constant 'Table2', ...)`
  - `REF(constant '"Table2"', ...)`

- `REF(constant 'SCOTT.Table2', ...)`

In both of those situations, if `constant` is replaced with a filler field, then the same values as shown in the examples will also work if they are placed in the data section.

11.3.2 System-Generated OID REF Columns

When you load system-generated `REF` columns, SQL*Loader assumes that the actual OIDs from which the `REF` columns are constructed are in the data file, with 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). Refer to the `ref_spec` SQL*Loader syntax for details.

Example 11-14 Loading System-Generated REF Columns

The following example shows how to load system-generated OID `REF` columns; note the callouts in **bold**:

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))
```

Data File (`sample.dat`)

```
22345, QuestWorld, 21E978406D3E41FCE03400400B403BC3, EMPLOYEES_V2,
23423, Geography, 21E978406D4441FCE03400400B403BC3, EMPLOYEES_V2,
```

> **Note:**
>
> The callout in bold, to the left of the example, corresponds to the following note:
>
> 1. If the specified table does not exist, then the record is rejected. The `dept_mgr` field itself does not map to any field in the data file.

Related Topics

- SQL*Loader Syntax Diagrams
  This appendix describes SQL*Loader syntax in graphic form (sometimes called railroad diagrams or DDL diagrams).

11.3.3 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. Refer to the SQL*Loader syntax for `ref_spec` for details.

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 11-15 Loading Primary Key REF Columns

The following example demonstrates 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))
```

Data File (sample.dat)

```
22345, QuestWorld, 007,
23423, Geography, 000,
```

Related Topics

- SQL*Loader Syntax Diagrams
  This appendix describes SQL*Loader syntax in graphic form (sometimes called railroad diagrams or DDL diagrams).

11.3.4 Unscoped REF Columns That Allow Primary Keys

An unscoped `REF` column that allows primary keys can reference both system-generated and primary key `REF`s.

The syntax for loading data into an unscoped `REF` column is the same syntax you use when loading data into a system-generated OID `REF` column, or into a primary-key-based `REF` column.

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, then 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, then only one object table can be referenced during a single-table load. That is, to load unscoped primary key REFs, some pointing to object table X and some pointing to object table Y, you must 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.

  If you do not use either of these methods, then the data row is rejected with an error message indicating that the referenced table name is invalid.
- SQL*Loader does not support unscoped primary key REFs in collections.
- If you are loading system-generated REFs into this REF column, then any limitations that apply to system-generated OID REF columns also apply.
- If you are loading primary key REFs into this REF column, then any limitations that apply to primary key REF columns also apply.

> **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). SQL*Loader then uses that object table's OID type to determine the REF type that can be referenced in that single-table load.

Example 11-16 Single Load Using Multiple INTO TABLE Clause Method

In this example, the WHEN clauses key off the "CUSTOMERS_PK" data specified by object table names for the unscoped primary key REF tables cust_tbl and cust_no:

```
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'
)
```

11.4 Loading LOBs with SQL*Loader

Find out which large object types (LOBs) SQL*Loader can load, and see examples of how to load LOB Data.

- Loading LOB Data from a Primary Data File
  You can load internal LOBs (`BLOB`s, `CLOB`s, `NCLOB`s) or XML columns from a primary data file.
- 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 data file.
- Loading Data Files that Contain LLS Fields
  If a field in a data file is a LOB location Specifier (LLS) field, then you can indicate this by using the `LLS` clause.

11.4.1 Loading LOB Data from a Primary Data File

You can load internal LOBs (`BLOB`s, `CLOB`s, `NCLOB`s) or XML columns from a primary data file.

To load internal LOBs or `XML` columns from a primary data file, you can use the following standard SQL*Loader formats:

- Predetermined size fields
- Delimited fields
- Length-value pair fields
- LOB Data in Predetermined Size Fields
  See how loading LOBs into predetermined size fields is a very fast and conceptually simple format in which to load LOBs.
- LOB Data in Delimited Fields
  Consider using delimited fields when you want to load LOBs of different sizes within the same column (data file field) with SQL*Loader.
- LOB Data in Length-Value Pair Fields
  To load LOB data organized in length-value pair fields, you can use `VARCHAR`, `VARCHARC`, or `VARRAW` data types.

11.4.1.1 LOB Data in Predetermined Size Fields

See how loading LOBs into predetermined size fields is a very fast and conceptually simple format in which to load LOBs.

> **Note:**
>
> Because the LOBs you are loading can be of different sizes, 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 predetermined size fields, you should use either `CHAR` or `RAW` as the loading data type.

Example 11-17 Loading LOB Data in Predetermined Size Fields

bold
Control File Contents

```
LOAD DATA
INFILE 'sample.dat' "fix 501"
INTO TABLE person_table
   (name       POSITION(01:21)       CHAR,
1  "RESUME"    POSITION(23:500)      CHAR   DEFAULTIF "RESUME"=BLANKS)
```

Data File (sample.dat)

```
Julia Nayer      Julia Nayer
             500 Example Parkway
             jnayer@us.example.com ...
```

> **Note:**
>
> The callout in bold, to the left of the example, corresponds to the following note:
>
> 1. Because the `DEFAULTIF` clause is used, if the data field containing the resume is empty, then the result is an empty LOB rather than a null LOB. However, if a `NULLIF` clause had been used instead of `DEFAULTIF`, then the empty data field would be null.
>
>   You can use SQL*Loader data types other than `CHAR` to load LOBs. For example, when loading `BLOB`s, you would probably want to use the `RAW` data type.

11.4.1.2 LOB Data in Delimited Fields

Consider using delimited fields when you want to load LOBs of different sizes within the same column (data file field) with SQL*Loader.

The delimited field format handles LOBs of different sizes within the same column (data file 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 data file. When the character set of the data file 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, then the specification must consist of characters that are valid in the character set of the input data file. In contrast, if hexadecimal notation is not used, then 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 data file's character set before SQL*Loader searches for the delimiter in the data file.

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`, then 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, then that length is used as a maximum. If no maximum is specified, then 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 for more information about the `CHAR` data type.

Example 11-18 Loading LOB Data in Delimited Fields

Review this example to see how to load LOB data in delimited fields. Note the callouts in **bold**:

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>')
```

Data File (`sample.dat`)

```
Julia Nayer,<startlob>        Julia Nayer
                         500 Example Parkway
                         jnayer@example.com ...   <endlob>
2  |Bruce Ernst, .......
```

> **Note:**
>
> The callouts, in bold, to the left of the example correspond to the following 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, then the maximum would be 507 characters. For more information, refer to character-length semantics.
> 2. If the record separator `'|'` had been placed right after `<endlob>` and followed with the newline character, then 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'`).

Related Topics

- Character-Length Semantics
  Byte-length semantics are the default for all data files except those that use the UTF16 character set (which uses character-length semantics by default).

11.4.1.3 LOB Data in Length-Value Pair Fields

To load LOB data organized in length-value pair fields, you can use `VARCHAR`, `VARCHARC`, or `VARRAW` data types.

Loading data with length-value pair fields provides better performance than using delimited fields. However, this method can reduce flexibility (for example, you must know the LOB length for each LOB before loading).

Example 11-19 Loading LOB Data in Length-Value Pair Fields

bold
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))
```

Data File (`sample.dat`)

```
Julia Nayer,479 Julia Nayer
500 Example Parkway
jnayer@us.example.com... <endrec>
3    Bruce Ernst,000<endrec>
```

> **Note:**
>
> The callouts in bold, to the left of the example, correspond to the following notes:
>
> 1. If the backslash escape character is not supported, then 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, then the length would be 3 characters and the maximum size would be 500 characters. See Character-Length Semantics.
> 3. The length subfield of the `VARCHARC` is 0 (the value subfield is empty). Consequently, the LOB instance is initialized to empty.

Related Topics

- Character-Length Semantics
  Byte-length semantics are the default for all data files except those that use the UTF16 character set (which uses character-length semantics by default).

11.4.2 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 data file.

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 data types.

See Examples of Loading LOB Data from LOBFILEs 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 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 file name).
- Examples of Loading LOB Data from LOBFILEs
  This section contains examples of loading data from different types of fields in LOBFILEs.
- Considerations When Loading LOBs from LOBFILEs
  Be aware of the restrictions and guidelines that apply when you load large object types (LOBs) from `LOBFILES` with SQL*Loader.

11.4.2.1 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 file name).

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, then the two fields typically read the data independently.

11.4.2.2 Examples of Loading LOB Data from LOBFILEs

This section contains examples of loading data from different types of fields in LOBFILEs.

- One LOB for Each File
  When you load large object type (LOB) data, each `LOBFILE` is the source of a single LOB.
- Predetermined Size LOBs
  You can load LOB data using predetermined size LOBs.
- Delimited LOBs
  When you have different sized large object types (LOBs), so you can't use predetermined size LOBs, consider using delimited LOBs with SQL*Loader.
- Length-Value Pair Specified LOBs
  You can obtain better performance by loading large object types (`LOB`s) with length-value pair specification, but you lose some flexibility.

11.4.2.2.1 One LOB for Each File

When you load large object type (LOB) data, each `LOBFILE` is the source of a single LOB.

Use this example to see how you can load LOB data that is organized so that each `LOBFILE` is the source of a single LOB.

Example 11-20 Loading LOB Data with One LOB per LOBFILE

In this example, note that the column or field name is followed by the LOBFILE data type specifications. Note the callouts in `bold`:

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)
```

Data File (`sample.dat`)

```
Johny Quest,jqresume.txt,
Speed Racer,'/private/sracer/srresume.txt',
```

Secondary Data File (`jqresume.txt`)

```
Johny Quest 500 Oracle Parkway ...
```

Secondary Data File (`srresume.txt`)

```
         Speed Racer
     400 Oracle Parkway
        ...
```

> **Note:**
>
> The callouts in bold, to the left of the example, correspond to the following notes:
>
> 1. The filler field is mapped to the 40-byte data field, which is read using the SQL*Loader `CHAR` data type. This assumes the use of default byte-length semantics. If character-length semantics were used, then 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` data type) from the first byte to the EOF character. If no existing LOBFILE is specified, then the `"RESUME"` field is initialized to empty.

11.4.2.2.2 Predetermined Size LOBs

You can load LOB data using predetermined size LOBs.

In Example 11-21, you specify the size of the LOBs to be loaded into a particular column in the control file. During the load, SQL*Loader assumes that any LOB data loaded into that particular column is of the specified size. The predetermined size of the fields allows the data-parser to perform optimally. However, it is often difficult to guarantee that all LOBs are the same size.

Example 11-21 Loading LOB Data Using Predetermined Size LOBs

Control File Contents

```
LOAD DATA
INFILE 'sample.dat'
INTO TABLE person_table
FIELDS TERMINATED BY ','
   (name     CHAR(20),
1  "RESUME"    LOBFILE(CONSTANT '/usr/private/jquest/jqresume.txt')
               CHAR(2000))
```

Data File (sample.dat)

```
Johny Quest,
Speed Racer,
```

Secondary Data File (jqresume.txt)

```
    Johny Quest
500 Oracle Parkway
   ...
    Speed Racer
400 Oracle Parkway
   ...
```

> **Note:**
>
> The callout, in bold, to the left of the example corresponds to the following note:
>
> 1. This entry specifies that SQL*Loader load 2000 bytes of data from the `jqresume.txt` LOBFILE, using the `CHAR` data type, 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, then SQL*Loader would load 2000 characters of data, starting from the first character after the last-loaded character. See Character-Length Semantics.

11.4.2.2.3 Delimited LOBs

When you have different sized large object types (LOBs), so you can't use predetermined size LOBs, consider using delimited LOBs with SQL*Loader.

When you load LOB data instances that are delimited, 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 11-22 Loading LOB Data Using Delimited LOBs

In this example, note the callouts in **bold**:

Control File Contents

```
LOAD DATA
INFILE 'sample.dat'
INTO TABLE person_table
FIELDS TERMINATED BY ','
   (name     CHAR(20),
1  "RESUME"    LOBFILE( CONSTANT 'jqresume') CHAR(2000)
               TERMINATED BY "<endlob>\n")
```

Data File (sample.dat)

```
Johny Quest,
Speed Racer,
```

Secondary Data File (jqresume.txt)

```
    Johny Quest
500 Oracle Parkway
   ... <endlob>
    Speed Racer
400 Oracle Parkway
   ... <endlob>
```

> **Note:**
>
> The callout, in bold, to the left of the example corresponds to the following note:
>
> 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, then you should be sure not to underestimate its value.* The `TERMINATED BY` clause specifies the string that terminates the LOBs. Alternatively, you can use the `ENCLOSED BY` clause. The `ENCLOSED BY` clause allows a bit more flexibility with the relative positioning of the LOBs in the `LOBFILE`, because the LOBs in the `LOBFILE` do not need to be sequential.

11.4.2.2.4 Length-Value Pair Specified LOBs

You can obtain better performance by loading large object types (`LOB`s) with length-value pair specification, but you lose some flexibility.

With length-value pair specified LOBs, each LOB in the LOBFILE is preceded by its length. To load LOB data organized in this way, you can use `VARCHAR`, `VARCHARC`, or `VARRAW` data types.

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 11-23 Loading LOB Data Using Length-Value Pair Specified LOBs

Control File Contents

In the following example, note the callouts in **bold**:

```
LOAD DATA
INFILE 'sample.dat'
INTO TABLE person_table
FIELDS TERMINATED BY ','
   (name          CHAR(20),
1  "RESUME"       LOBFILE(CONSTANT 'jqresume') VARCHARC(4,2000))
```

Data File (`sample.dat`)

```
Johny Quest,
Speed Racer,
```

Secondary Data File (`jqresume.txt`)

```
2      0501Johny Quest
       500 Oracle Parkway
          ...
3      0000
```

> **Note:**
>
> The callouts, in bold, to the left of the example correspond to the following 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, then 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.
> 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.

11.4.2.3 Considerations When Loading LOBs from LOBFILEs

Be aware of the restrictions and guidelines that apply when you load large object types (LOBs) from `LOBFILES` with SQL*Loader.

When you load data using `LOBFILE`s, be aware of the following:

- 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, the result is a record that contains an empty LOB. In the case of an `XML` column, if there is a failure loading the LOB. then a null value is inserted.
- It is not necessary to specify the maximum length of a field corresponding to a LOB column. If a maximum length *is* specified, then 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, then that field is initialized to empty. If the concept of empty does not apply to the particular field type, then 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 for temporary LOB performance.

Related Topics

- Temporary LOB Performance Guidelines

11.4.3 Loading Data Files that Contain LLS Fields

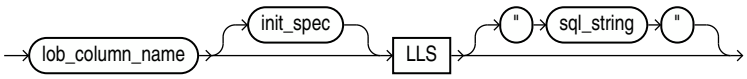
If a field in a data file is a LOB location Specifier (LLS) field, then you can indicate this by using the `LLS` clause.

Purpose

An LLS field contains the file name, offset, and length of the LOB data in the data file. SQL*Loader uses this information to read data for the LOB column.

Syntax

The syntax for the `LLS` clause is as follows:



Usage Notes

The LOB can be loaded in part or in whole and it can start from an arbitrary position and for an arbitrary length. SQL Loader expects the expects the contents of the LLS field to be *`filename.ext.nnn.mmm/`* where each element is defined as follows:

- *`filename.ext`* is the name of the file that contains the LOB.
- *`nnn`* is the offset in bytes of the LOB within the file.
- *`mmm`* is the length of the LOB in bytes. A value of -1 means the LOB is NULL. A value of 0 means the LOB exists, but is empty.
- The forward slash (`/`) terminates the field

If the SQL*Loader parameter, `SDF_PREFIX`, is specified, then SQL*Loader looks for the files in the directory specified by `SDF_PREFIX`. Otherwise, SQL*Loader looks in the same directory as the data file.

An error is reported and the row is rejected if any of the following are true:

- The file name contains a relative or absolute path specification.
- The file is not found, the offset is invalid, or the length extends beyond the end of the file.
- The contents of the field do not match the expected format.
- The data type for the column associated with an LLS field is not a `CLOB`, `BLOB`, or `NCLOB`.

Restrictions

- If an LLS field is referenced by a clause for any other field (for example a `NULLIF` clause) in the control file, then the value used for evaluating the clause is the string in the data file, not the data in the file pointed to by that string.
- The character set for the data in the file pointed to by the `LLS` clause is assumed to be the same character set as the data file.
- The user running SQL*Loader must have read access to the data files.

Example Specification of an LLS Clause

The following is an example of a SQL*Loader control file that contains an `LLS` clause. Note that a data type is not needed on the column specification because the column must be of type `LOB`.

```
LOAD DATA
INFILE *
TRUNCATE
INTO TABLE tklglls
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"' TRAILING NULLCOLS
(col1 , col2 NULLIF col1 = '1' LLS)
BEGINDATA
1,"tklglls1.dat.1.11/"
```

11.5 Loading BFILE Columns with SQL*Loader

The `BFILE` data type stores unstructured binary data in operating system files outside the database.

The Oracle BFILE data type is an Oracle LOB data type that contains a reference to binary data with a maximum size of 4 gigabytes.

A `BFILE` column or attribute stores a file locator that points to the external file containing the data. The file that you want to load as a `BFILE` does not have to exist at the time of loading; it can be created later. To use BFILEs, you must peform some database administration tasks. There are also restrictions on directory objects and BFILE objects. These restrictions include requirements for how you configure the operating system file, and the operating system directory path. With Oracle Database 18c and later releases, symbolic links are not allowed in directory object path names used with BFILE data types. SQL*Loader assumes that the necessary directory objects are already created (a logical alias name for a physical directory on the server's file system).

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. You can provide both arguments as string constants, or these arguments can be dynamically loaded through some other field.

In the following examples of loading `BFILE`s, the first example has only the file name specified dynamically, while the second example demonstrates specifying both the `BFILE` and the directory object dynamically:

Example 11-24 Loading Data Using BFILEs: Only File Name Specified Dynamically

The following are the control file contents. The directory name, `scott_dir1`, is in quotation marks; therefore, the string is used as is, and is not capitalized.

```
LOAD DATA
INFILE sample.dat
INTO TABLE planets
FIELDS TERMINATED BY ','
   (pl_id    CHAR(3),
   pl_name   CHAR(20),
   fname     FILLER CHAR(30),
   pl_pict   BFILE(CONSTANT "scott_dir1", fname))
```

The following are the contents of the data file, `sample.dat`.

```
1,Mercury,mercury.jpeg,
2,Venus,venus.jpeg,
3,Earth,earth.jpeg,
```

Example 11-25 Loading Data Using BFILEs: File Name and Directory Specified Dynamically

The following are the control file contents. Note that `dname` is mapped to the data file field containing the directory name that corresponds to the file being loaded.

```
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),
   dname     FILLER CHAR(20),
   pl_pict   BFILE(dname, fname) )
```

The following are the contents of the data file, `sample.dat`.

```
1, Mercury, mercury.jpeg, scott_dir1,
2, Venus, venus.jpeg, scott_dir1,
3, Earth, earth.jpeg, scott_dir2,
```

Related Topics

- *Oracle Database SecureFiles and Large Objects Developer's Guide*
- *Oracle Database SQL Language Reference*

11.6 Loading Collections (Nested Tables and VARRAYs)

Like LOBs, collections can be loaded either from a primary data file (data inline) or from secondary data files (data out of line).

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.

  If the `COUNT` clause specifies a field in a control file and if that field is set to null for the current row, then the collection that uses that count will be set to empty for the current row as well.

- 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. There are some differences:

- Collection descriptions employ the two mechanisms discussed in the preceding list.
- Collection descriptions can include a secondary data file (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
  There are restrictions for nested tables and `VARRAY`s.
- Secondary Data Files (SDFs)
  When you need to load large nested tables and `VARRAY`s, you can use secondary data files (SDFs). They are similar in concept to primary data files.

> **See Also:**
>
> For details about SDFs, see Secondary Data Files (SDFs).
>
> For details about loading column objects, see Loading Column Objects.

11.6.1 Restrictions in Nested Tables and VARRAYs

There are restrictions for nested tables and `VARRAY`s.

The following restrictions exist for nested tables and `VARRAY`s:

- 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, then SQL*Loader will not automatically split the load into multiple loads and generate a set ID.

Example 11-26 demonstrates loading a `VARRAY` and a nested table.

Example 11-26 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
    )
  )
)
```

Data File (sample.dat)

```
 101,MATH,"Napier",28,2828,"Euclid", 123,9999:0
 210,"Topological Transforms",:2
```

Secondary Data File (SDF) (pr.txt)

```
21034 Topological Transforms
77777 Impossible Proof
```

> **Note:**
>
> The callouts, in bold, to the left of the example correspond to the following 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.

11.6.2 Secondary Data Files (SDFs)

When you need to load large nested tables and VARRAYs, you can use secondary data files (SDFs). They are similar in concept to primary data files.

As with primary data files, 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 data file, the following can be specified for each SDF:

- The record format (fixed, stream, or variable). Also, if stream record format is used, then 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).
- 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 regarding SDFs:

- If a nonexistent SDF is specified as a data source for a particular field, then that field is initialized to empty. If the concept of empty does not apply to the particular field type, then 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
> - OPTIONS Clause
> - sdf_spec

11.7 Choosing Dynamic or Static SDF Specifications

With SQL*Loader, you can specify SDFs either statically (specifying the actual name of the file), or dynamically (using a `FILLER` field as the source of the file name).

With either dynamic or static SDF specification, when the end-of-file (EOF) of an SDF is reached, the file is closed. Further attempts to 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. When 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.

Fynamic 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, then the two fields typically read the data independently.

11.8 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 data file with the data).

The following examples illustrate how to load parent and child tables with user-provided SIDs.

Example 11-27 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))
```

Data File (sample.dat)

```
101,Math,21E978407D4441FCE03400400B403BC3,|
210,"Topology",21E978408D4441FCE03400400B403BC3,|
```

> **Note:**
>
> The callout, in bold, to the left of the example corresponds to the following note:
>
> 1. `mysid` is a filler field that is mapped to a data file field containing the actual set IDs and is supplied as an argument to the `SID` clause.

Example 11-28 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))
```

Data File (sample.dat)

```
21034, "Topological Transforms", 21E978407D4441FCE03400400B403BC3,
77777, "Impossible Proof", 21E978408D4441FCE03400400B403BC3,
```

> **Note:**
>
> The callout, in bold, to the left of the example corresponds to the following note:
>
> 1. The table-level `SID` clause tells SQL*Loader that it is loading the storage table for nested tables. `sidsrc` is the filler field name that is the source of the real set IDs.

- Memory Issues When Loading VARRAY Columns
  There are some memory issues when you load `VARRAY` columns.

11.8.1 Memory Issues When Loading VARRAY Columns

There are some memory issues when you load `VARRAY` columns.

The following list describes some issues to keep in mind when you load `VARRAY` columns:

- `VARRAY`s 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 `VARRAY`s 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 `VARRAY`s or a large number of smaller `VARRAY`s could cause you to run out of memory during the load. If this happens, then specify a smaller value for `BINDSIZE` or `ROWS` and retry the load.

12

Conventional and Direct Path Loads

SQL*Loader provides the option to load data using a conventional path load method, and a direct path load method.

- Data Loading Methods
  SQL*Loader can load data by using either a convention path load, or a direct path load.
- Loading ROWID Columns
  In both conventional path and direct path, you can specify a text value for a `ROWID` column.
- Conventional Path Load
  With conventional path load (the default), SQL*Loader uses the SQL `INSERT` statement and a bind array buffer to load data into database tables.
- Direct Path Loads
  Learn what a SQL*Loader direct path load is, when and how to use it to pass data, and what restrictions apply to this feature.
- Using Direct Path Load
  Learn how you can use the SQL*Loader direct path load method for loading data.
- Optimizing Performance of Direct Path Loads
  Learn how to enable your SQL*Loader direct path loads to run faster, and to use less space.
- Optimizing Direct Path Loads on Multiple-CPU Systems
  If you are performing direct path loads on a multiple-CPU system, then 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.
- Avoiding Index Maintenance
  For both the conventional path and the direct path, SQL*Loader maintains all existing indexes for a table.
- Direct Path Loads, Integrity Constraints, and Triggers
  There can be differences between how you set triggers with direct path loads, compared to conventional path loads
- Optimizing Performance of Direct Path Loads
  Learn how to enable your SQL*Loader direct path loads to run faster, and to use less space.
- General Performance Improvement Hints
  Learn how to enable general performance improvements when using SQL*Loader with parallel data loading.

Related Topics

- SQL*Loader Case Studies
  To learn how you can use SQL*Loader features, you can run a variety of case studies that Oracle provides.

12.1 Data Loading Methods

SQL*Loader can load data by using either a convention path load, or a direct path load.

A conventional path load runs SQL `INSERT` statements to populate tables in Oracle Database. A direct path load eliminates much of the Oracle Database overhead by formatting Oracle data blocks, and then 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.

The tables that you want to be loaded must already exist in the database. SQL*Loader never creates tables. It loads existing tables that either already contain data, or that 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 that you want 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.

Related Topics

- Conventional Path Load
  With conventional path load (the default), SQL*Loader uses the SQL `INSERT` statement and a bind array buffer to load data into database tables.
- Direct Path Loads
  Learn what a SQL*Loader direct path load is, when and how to use it to pass data, and what restrictions apply to this feature.

12.2 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.

12.3 Conventional Path Load

With conventional path load (the default), SQL*Loader uses the SQL `INSERT` statement and a bind array buffer to load data into database tables.

When SQL*Loader performs a conventional path load, it competes equally with all other processes for buffer resources. Using this method can slow the load significantly. Extra overhead is added as SQL statements are generated, passed to Oracle Database, and executed.

Oracle Database looks for partially filled blocks and attempts to fill them on each insert. Although appropriate during normal use, this method can slow bulk loads dramatically.

- Conventional Path Load of a Single Partition
  SQL*Loader uses the partition-extended syntax of the `INSERT` statement.
- When to Use a Conventional Path Load
  To determine when you should use conventional path load instead of direct path load, review the options for your use case scenario.

Related Topics

- Discontinued Conventional Path Loads
  In a conventional path load, data is committed after all data in the bind array is loaded into all tables.

12.3.1 Conventional Path Load of a Single Partition

SQL*Loader uses the partition-extended syntax of the `INSERT` statement.

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, then the row is rejected, and the SQL*Loader log file records an appropriate error message.

12.3.2 When to Use a Conventional Path Load

To determine when you should use conventional path load instead of direct path load, review the options for your use case scenario.

If load speed is most important to you, then you should use direct path load because it is faster than conventional path load. However, certain restrictions on direct path loads can 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

  Note: To use a direct path load (except for 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

  Reason: A direct path load does not support loading of clustered tables.
- When loading a relatively small number of rows into a large indexed table

  Reason: 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

  Reason: 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 causes an Oracle error upon insertion
  - If the record is formatted incorrectly, so that SQL*Loader cannot find field boundaries
  - If the record violates a constraint, or a record tries to make a unique index non-unique

12.4 Direct Path Loads

Learn what a SQL*Loader direct path load is, when and how to use it to pass data, and what restrictions apply to this feature.

- About SQL*Loader Direct Path Load
  The SQL*Loader direct path load option uses the direct path API to pass the data to be loaded to the load engine in the server.
- Data Conversion During Direct Path Loads
  During a SQL*Loader direct path load, data conversion occurs on the client side, rather than on the server side.
- 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).
- Direct Path Load of a Single Partition or Subpartition
  During a direct path load of a single partition, SQL*Loader uses the partition-extended syntax of the `LOAD` statement.
- Advantages of a Direct Path Load
  Direct path loads typically are faster than using conventional path loads.
- Restrictions on Using Direct Path Loads
  To use the direct path load method, your tables and segments must meet certain requirements. Some features are not available with Direct Path Loads.
- Restrictions on a Direct Path Load of a Single Partition
  When you want to use a direct path load of a single partition, the partition that you specify for direct path load must meet additional requirements.
- When to Use a Direct Path Load
  Learn under what circumstances you should run SQL*Loader with direct path load.
- Integrity Constraints
  All integrity constraints are enforced during direct path loads, although not necessarily at the same time.
- Field Defaults on the Direct Path
  Default column specifications defined in the database are not available when you use direct path loading.
- Loading into Synonyms
  You can use SQL*Loader to load data into a synonym for a table during a direct path load, but the synonym must point directly either to a table, or to a view on a simple table.

12.4.1 About SQL*Loader Direct Path Load

The SQL*Loader direct path load option uses the direct path API to pass the data to be loaded to the load engine in the server.

When you use the direct path load feature of SQL*Loader, then istead 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 Database data blocks, and to 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.

Related Topics

- Discontinued Direct Path Loads
  In a direct path load, the behavior of a discontinued load varies depending on the reason the load was discontinued.

12.4.2 Data Conversion During Direct Path Loads

During a SQL*Loader direct path load, data conversion occurs on the client side, rather than on the server side.

As an implication of client side data conversion, this means that NLS parameters in the database 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 you can 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 "Setting the Date Format in the SQL*Loader Control File"), or you can set an `NLS_DATE_FORMAT` environment variable, as shown in "Setting an `NLS_DATE_FORMAT` Environment Variable." .

Example 12-1 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 12-2 Setting an `NLS_DATE_FORMAT` Environment Variable

On Unix Bourne or Korn shell:

```
% NLS_DATE_FORMAT='YYYYMMDD'
% export NLS_DATE_FORMAT
```

On Unix C shell (`csh`):

```
%setenv NLS_DATE_FORMAT='YYYYMMDD'
```

12.4.3 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, then 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, then the entire load is rolled back.

12.4.4 Direct Path Load of a Single Partition or Subpartition

During a direct path load of a single partition, SQL*Loader uses the partition-extended syntax of the `LOAD` statement.

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, to initialize and then finish the load, several calls to Oracle Database are required at the beginning and end of the load. Also, certain DML locks are required during load initialization. When the load completes, these DML locks are released. The following operations occur during the load:

- Index keys are built and put into a sort
- Space management routines are used to obtain new extents, when needed, and to adjust the upper boundary (high-water mark) for a data savepoint.

For more information about protecting data, see "Using Data Saves to Protect Against Data Loss.

Related Topics

- Using Data Saves to Protect Against Data Loss
  When you have a savepoint, if you encounter an instance failure during a SQL*Loader load, then use the `SKIP` parameter to continue the load.

12.4.5 Advantages of a Direct Path Load

Direct path loads typically are faster than using conventional path loads.

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 does not need to run any SQL `INSERT` statements; therefore, the processing load on Oracle Database is reduced.
- A direct path load calls on Oracle Database to lock tables and indexes at the start of the load, and release those locks when the load is finished. A conventional path load issues an Oracle Database call 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 the Oracle Database buffer cache. This process method minimizes contention with other Oracle Database 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 that you specify to load is empty, the presorting option eliminates the sort and merge phases of index-building. The index is filled in as data arrives.
- Protection against instance failure does not require redo log file entries during direct path loads. Therefore, no time is required to log the load when:
  - Oracle Database has the SQL `NOARCHIVELOG` parameter enabled
  - The SQL*Loader `UNRECOVERABLE` clause is enabled
  - The object being loaded has the SQL `NOLOGGING` parameter set

Related Topics

- 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.

12.4.6 Restrictions on Using Direct Path Loads

To use the direct path load method, your tables and segments must meet certain requirements. Some features are not available with Direct Path Loads.

The following conditions must be satisfied for you to use the direct path load method:

- Tables that you want to load cannot be clustered.
- Tables that you want to load cannot have Oracle Virtual Private Database (VPD) policies active on `INSERT`.
- Segments that you want to load cannot have any active transactions pending.

  To check for active transactions, use the Oracle Enterprise Manager command `MONITOR TABLE` to find the object ID for the tables that you want to load. Then use the command `MONITOR LOCK` to see if there are any locks on the tables.
- For Oracle Database releases earlier than Oracle9*i*, you can perform a SQL*Loader direct path load only when the client and server are the same release. This restriction also means that you cannot perform a direct path load of Oracle9*i* data into an earlier Oracle Database release. For example, you cannot use direct path load to load data from Oracle Database 9*i* Release 1 (9.0.1) into an Oracle 8*i* (8.1.7) Oracle Database.

Beginning with Oracle Database 9*i*, you can perform a SQL*Loader direct path load when the client and server are different releases. However, both releases must be at least Oracle Database 9i Release 1 (9.0.1), and the client release must be the same as or lower than the server release. For example, you can perform a direct path load from an Oracle Database 9*i* Release 1 (9.0.1) database into Oracle Database 9*i* Release 2 (9.2). However, you cannot use direct path load to load data from Oracle Database 10g into an Oracle Database 9*i* release.

The following features are not available with direct path load:

- Loading `BFILE` columns
- Use of `CREATE SEQUENCE` during the load. This is because in direct path loads there is no SQL being generated to fetch the next value, because direct path does not generate `INSERT` statements.

12.4.7 Restrictions on a Direct Path Load of a Single Partition

When you want to use a direct path load of a single partition, the partition that you specify for direct path load must meet additional requirements.

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.

12.4.8 When to Use a Direct Path Load

Learn under what circumstances you should run SQL*Loader with direct path load.

If you are not restricted by views, field defaults, or integrity constraints, then then you should use a direct path load in the following circumstances:

- 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.

12.4.9 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 SQL*Loader 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, then 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.

Related Topics

- Direct Path Loads, Integrity Constraints, and Triggers
  There can be differences between how you set triggers with direct path loads, compared to conventional path loads

12.4.10 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.

12.4.11 Loading into Synonyms

You can use SQL*Loader to load data into a synonym for a table during a direct path load, but the synonym must point directly either to a table, or to a view on a simple table.

Note the following restrictions:

- Direct path mode cannot be used if the view is on a table that has either user-defined types, or XML data.
- In direct path mode, a view cannot be loaded using a SQL*Loader control file that contains SQL expressions.

12.5 Using Direct Path Load

Learn how you can use the SQL*Loader direct path load method for loading data.

- Setting Up for Direct Path Loads
  To create the necessary views required to prepare the database for direct path loads, you must run the setup script `catldr.sql`.
- 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.
- 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.
- 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.
- Preventing Data Loss with Data Saves
  You can use data saves to protect against loss of data due to instance failure.
- Data Recovery During Direct Path Loads
  SQL*Loader provides full support for data recovery when using the direct path load method.
- Loading Long Data Fields
  You can load data that is longer than SQL*Loader's maximum buffer size can load on the direct path by using large object types (LOBs).

- 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.
- Auditing SQL*Loader Operations That Use Direct Path Mode
  You can perform auditing on SQL*Loader direct path loads to monitor and record selected user database actions.

12.5.1 Setting Up for Direct Path Loads

To create the necessary views required to prepare the database for direct path loads, you must run the setup script `catldr.sql`.

You only need to run `catldr.sql` once for each database to which you plan to run direct loads. You can run this script during database installation if you know then that you will be doing direct loads.

12.5.2 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.

For example, to configure the parameter file to start SQL*Loader in direct path load mode, include the following line in the parameter file:

```
DIRECT=TRUE
```

Related Topics

- Minimizing Time and Space Required for Direct Path Loads
  You can control the time and temporary storage used during direct path loads.
- Optimizing Direct Path Loads on Multiple-CPU Systems
  If you are performing direct path loads on a multiple-CPU system, then 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.

12.5.3 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 of SQL*Loader direct loads on systems with limited memory, use the `SINGLEROW` parameter.
- Calculating Temporary Segment Storage Requirements
  To estimate the amount of temporary segment space needed during direct path loads for storing new index keys, use this formula.

12.5.3.1 Improving Performance

To improve performance of SQL*Loader direct loads on systems with limited memory, use the `SINGLEROW` parameter.

> **Note:**
>
> If, during a direct load, you have specified that you want the data to be presorted, and the existing index is empty, then 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

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.

Related Topics

- SINGLEROW Option
- Estimate Index Size and Set Storage Parameters

12.5.3.2 Calculating Temporary Segment Storage Requirements

To estimate the amount of temporary segment space needed during direct path loads for storing new index keys, use this formula.

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, where *number_rows* is the number of rows, *sum_of_column_sizes* is the sum of the column sizes, and *number_of_columns* is the number of columns in the index:

```
key_storage = (number_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, then 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, then only enough space to store the index entries is required, and the value of this constant would be 1.0.

Related Topics

- Presorting Data for Faster Indexing
  You can improve the performance of SQL*Loader direct path loads by presorting your data on indexed columns.

12.5.4 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`.

12.5.5 Preventing Data Loss with Data Saves

You can use data saves to protect against loss of data due to instance failure.

- Using Data Saves to Protect Against Data Loss
  When you have a savepoint, if you encounter an instance failure during a SQL*Loader load, then use the `SKIP` parameter to continue the load.
- Using the ROWS Parameter
  The `ROWS` parameter determines when data saves occur during a direct path load.
- 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.

12.5.5.1 Using Data Saves to Protect Against Data Loss

When you have a savepoint, if you encounter an instance failure during a SQL*Loader load, then use the `SKIP` parameter to continue the load.

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, then before you continue the load, drop those indexes, and then recreate them after the load. See "Data Recovery During Direct Path Loads" 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.)

Related Topics

- Data Recovery During Direct Path Loads
  SQL*Loader provides full support for data recovery when using the direct path load method.

12.5.5.2 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.

12.5.5.3 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.

12.5.6 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.
- Media Recovery and Direct Path Loads
  If redo log file archiving is enabled (you are operating in `ARCHIVELOG` mode), then SQL*Loader logs loaded data when using the direct path, making media recovery possible.
- 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.

12.5.6.1 Media Recovery and Direct Path Loads

If redo log file archiving is enabled (you are operating in `ARCHIVELOG` mode), then 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 RMAN `RECOVER` command.

Related Topics

- Performing Complete Recovery of a Tablespace in *Oracle Database Backup and Recovery User's Guide*

12.5.6.2 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, then 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" for information about how to determine if an index has been left in Index Unusable state.

Related Topics

- 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.

12.5.7 Loading Long Data Fields

You can load data that is longer than SQL*Loader's maximum buffer size can load on the direct path by using large object types (LOBs).

In considering how to load long data fields, note the following:

- To improve performance for loading long data fields as LOBs, Oracle recommends that you use a large `STREAMSIZE` value.
- As an alternative to LOBs, you can also load data that is longer than the maximum buffer size by using the `PIECED` parameter. However, for this scenario, Oracle highly recommends that you use LOBs instad of `PIECED`.

Related Topics

- Loading LOBs with SQL*Loader
  Find out which large object types (LOBs) SQL*Loader can load, and see examples of how to load LOB Data.
- 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. T

12.5.8 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 discovered, then 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.

12.5.9 Auditing SQL*Loader Operations That Use Direct Path Mode

You can perform auditing on SQL*Loader direct path loads to monitor and record selected user database actions.

SQL*Loader uses unified auditing, in which all audit records are centralized in one place.

To set up unified auditing you create a unified audit policy, or alter an existing policy. An audit policy is a named group of audit settings that enable you to audit a particular aspect of user behavior in the database. To create the policy, use the SQL `CREATE AUDIT POLICY` statement.

After creating the audit policy, use the `AUDIT` and `NOAUDIT` SQL statements to, respectively, enable and disable the policy.

Related Topics

- CREATE AUDIT POLICY (Unified Auditing)
- Auditing Oracle SQL*Loader Direct Load Path Events

12.6 Optimizing Performance of Direct Path Loads

Learn how to enable your SQL*Loader direct path loads to run faster, and to use less space.

- Minimizing Time and Space Required for Direct Path Loads
  You can control the time and temporary storage used during direct path loads.
- 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.
- Presorting Data for Faster Indexing
  You can improve the performance of SQL*Loader direct path loads by presorting your data on indexed columns.
- Infrequent Data Saves
  Frequent data saves resulting from a small `ROWS` value adversely affect the performance of a direct path load.
- Minimizing Use of the Redo Log
  One way to speed a direct load dramatically is to minimize use of the redo log.
- 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. T
- Specifying a Value for DATE_CACHE
  To improve performance where the same date or timestamp is used many times during a direct path load, you can use the SQL*Loader date cache.

12.6.1 Minimizing Time and Space Required for 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
- Set `DB_UNRECOVERABLE_SCN_TRACKING=FALSE`. Unrecoverable (nologging) direct writes are tracked in the control file by periodically storing the SCN and Time of the last direct write. If these updates to the control file are adversely affecting performance, then setting the `DB_UNRECOVERABLE_SCN_TRACKING` parameter to `FALSE` may improve performance.

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

12.6.2 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.

12.6.3 Presorting Data for Faster Indexing

You can improve the performance of SQL*Loader direct path loads by presorting your data on indexed columns.

- Advantages of Presorting Data
  Learn about how presorting enables you to increase load performance with SQL*Loader
- SORTED INDEXES Clause
  The `SORTED INDEXES` clause identifies the indexes on which the data is presorted.
- 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.
- Multiple-Column Indexes
  If you specify a multiple-column index in the `SORTED INDEXES` clause, then 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.
- 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.

12.6.3.1 Advantages of Presorting Data

Learn about how presorting enables you to increase load performance with SQL*Loader

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, refer to "Temporary Segment Storage Requirements."

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. As a result, temporary storage is not required during the load, and time is saved.

Related Topics

- Calculating Temporary Segment Storage Requirements
  To estimate the amount of temporary segment space needed during direct path loads for storing new index keys, use this formula.

12.6.3.2 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, for an example. (See SQL*Loader Case Studies for information on how to access case studies.)

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.

12.6.3.3 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.

12.6.3.4 Multiple-Column Indexes

If you specify a multiple-column index in the `SORTED INDEXES` clause, then 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
...             ...
```

12.6.3.5 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.

12.6.4 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).

12.6.5 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, then direct path loads do not generate full image redo.

- 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.
- 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).

12.6.5.1 Disabling Archiving

If archiving is disabled, then direct path loads do not generate full image redo.

Use the SQL `ARCHIVELOG` and `NOARCHIVELOG` parameters to set the archiving mode.

Related Topics

- Managing Archived Redo Log Files in *Oracle Database Administrator's Guide*

12.6.5.2 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, then 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`.

The following is an example of specifying the `UNRECOVERABLE` clause in the control file:

```
UNRECOVERABLE
LOAD DATA
INFILE 'sample.dat'
INTO TABLE emp
(ename VARCHAR2(10), empno NUMBER(4));
```

12.6.5.3 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.

12.6.6 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. T

he 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 data types, and Oracle column data types used. When you are using optimal values for your particular configuration, the elapsed time in the SQL*Loader log file should go down.

> **Note:**
>
> You should monitor process paging activity, because if paging becomes excessive, then 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 stream buffer when you perform direct path loads on multiple-CPU systems.

> **See Also:**
>
> - Optimizing Direct Path Loads on Multiple-CPU Systems
> - COLUMNARRAYROWS
> - STREAMSIZE

12.6.7 Specifying a Value for DATE_CACHE

To improve performance where the same date or timestamp is used many times during a direct path load, you can use the SQL*Loader date cache.

If you are performing a direct path load in which the same date or timestamp values are loaded many times, then 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, then 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 data type 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, then it may cause other problems, such as excessive paging or too much memory usage.
- If most of the input date values are unique, then 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, then revert to the default behavior or set the cache size to 0. The overall performance improvement also depends on the data types 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.

Related Topics

- DATE_CACHE
  The `DATE_CACHE` command-line parameter for SQL*Loader specifies the date cache size (in entries).

12.7 Optimizing Direct Path Loads on Multiple-CPU Systems

If you are performing direct path loads on a multiple-CPU system, then 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 data types to Oracle column data types

  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 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

12.8 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 before beginning of the load.
- Mark selected indexes or index partitions as Index Unusable before beginning 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.

12.9 Direct Path Loads, Integrity Constraints, and Triggers

There can be differences between how you set triggers with direct path loads, compared to conventional path loads

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.

- Integrity Constraints
  During a direct path load with SQL*Loader, some integrity constraints are automatically disabled, while others are not.
- Database Insert Triggers
  Table insert triggers are also disabled when a direct path load begins.
- Permanently Disabled Triggers and Constraints
  SQL*Loader needs to acquire several locks on the table to be loaded to disable triggers and constraints.
- Increasing Performance with Concurrent Conventional Path Loads
  If triggers or integrity constraints pose a problem, but you want faster loading, then you should consider using concurrent conventional path loads.

12.9.1 Integrity Constraints

During a direct path load with SQL*Loader, some integrity constraints are automatically disabled, while others are not.

To better understand the concepts behind how integrity constraints enforce the business rules associated with a database, and to understand the different techniques you can use to prevent the entry of invalid information into tables, refer to "Data Integrity."

- Enabled Constraints
  During direct path load, some constraints remain enabled.
- Disabled Constraints
  During a direct path load, some constraints are disabled.
- Reenable Constraints
  When a SQL*Loader load completes, the integrity constraints will be reenabled automatically if the `REENABLE` clause is specified.

Related Topics

- Data Integrity

12.9.1.1 Enabled Constraints

During direct path load, some constraints remain enabled.

During a direct path load, the constraints that remain enabled are as follows:

- `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.

Even though `UNIQUE` constraints remain enabled during direct path loads, any rows that violate those constraints are loaded anyway (this is different than in conventional path in which such rows would be rejected). When indexes are rebuilt at the end of the direct path load, `UNIQUE` constraints are verified and if a violation is detected, then the index will be left in an Index Unusable state. See Indexes Left in an Unusable State.

12.9.1.2 Disabled Constraints

During a direct path load, some constraints are disabled.

During a direct path load, the following constraints are automatically disabled by default:

- `CHECK` constraints
- Referential constraints (`FOREIGN KEY`)

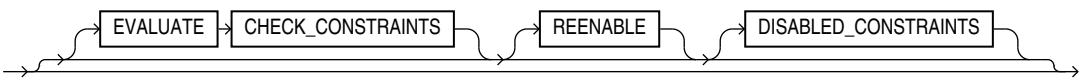
You can override the automatic disabling of `CHECK` constraints by specifying the `EVALUATE CHECK_CONSTRAINTS` clause. SQL*Loader will then evaluate `CHECK` constraints during a direct path load. Any row that violates the `CHECK` constraint is rejected. The following example shows the use of the `EVALUATE 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
```

12.9.1.3 Reenable Constraints

When a SQL*Loader 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, then the exceptions table (default name: `EXCEPTIONS`) must already exist, and you must be able to insert into it. This table contains the `ROWID` values for all rows that violated one of the integrity constraints. It also contains the name of the constraint that was violated.

For instructions on how to create the exceptions table, see `exceptions_clause` under `constraint` in *Oracle Database SQL Language Reference*.

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, then 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, then SQL*Loader automatically reenables the constraint and verifies all new rows. If no errors are found in the new data, then SQL*Loader automatically marks the constraint as validated. If any errors *are* found in the new data, then 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, then 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.

Related Topics

- constraint in *Oracle Database SQL Language Reference*

12.9.2 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.
- When Automatic Constraints Cannot Be Used
  Sometimes an insert trigger cannot be replaced with Oracle's automatic integrity constraints.
- Preparation of Database Triggers
  Before you can use either the insert triggers or automatic constraints method, you must prepare the Oracle Database table
- Using an Update Trigger
  Generally, you can use a database update trigger to duplicate the effects of an insert trigger.
- Duplicating the Effects of Exception Conditions
  If the insert trigger can raise an exception, then more work is required to duplicate its effects.
- Using a Stored Procedure
  If using an insert trigger raises exceptions, then consider using a stored procedure to duplicate the effects of an insert trigger.

12.9.2.1 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.

12.9.2.2 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.

12.9.2.3 Preparation of Database Triggers

Before you can use either the insert triggers or automatic constraints method, you must prepare the Oracle Database table

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.

12.9.2.4 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.

12.9.2.5 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)

12.9.2.6 Using a Stored Procedure

If using an insert trigger raises exceptions, then consider using a stored procedure to duplicate the effects of an insert trigger.

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. 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, then change the "new data" flag to null.

e. If the operations fail, then change the "new data" flag to "bad data."

2. Run 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.

12.9.3 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 ensure 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, then 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 Language Reference.*

12.9.4 Increasing Performance with Concurrent Conventional Path Loads

If triggers or integrity constraints pose a problem, but you want faster loading, then you should consider using concurrent conventional path loads.

That is, use multiple load sessions executing concurrently on a multiple-CPU system. Split the input data files into separate files on logical record boundaries, and then load each such input data file 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.

12.10 Optimizing Performance of Direct Path Loads

Learn how to enable your SQL*Loader direct path loads to run faster, and to use less space.

- About SQL*Loader Parallel Data Loading Models
  There are three basic models of concurrency that you can use to minimize the elapsed time required for data loading.

12.10.1 About SQL*Loader Parallel Data Loading Models

There are three basic models of concurrency that you can use to minimize the elapsed time required for data loading.

The concurrency models are:

- Concurrent conventional path loads
- Intersegment concurrency with the direct path load method
- Intrasegment concurrency with the direct path load method

12.10.2 Concurrent Conventional Path Loads

This topic describes using concurrent conventional path loads.

Using multiple conventional path load sessions executing concurrently is discussed in Increasing Performance with Concurrent Conventional Path Loads, you can use this technique to load the same or different objects concurrently with no restrictions.

12.10.3 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).

12.10.4 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.

12.10.5 Restrictions on Parallel Direct Path Loads

Restrictions are enforced on parallel direct path loads.

The following restrictions are enforced on parallel direct path loads:

- Neither local nor global indexes can be maintained by the load.
- 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, then you must do it manually.

Additionally, the following objects must be disabled on parallel direct path loads. You do not have to take any action to disable them. SQL*Loader disables them before the load begins and re-enables them after the load completes:

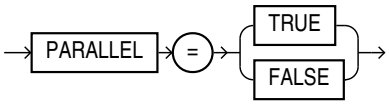
- Referential integrity constraints
- Triggers
- CHECK constraints, unless the `ENABLE_CHECK_CONSTRAINTS` control file option is used

If a parallel direct path load is being applied to a single partition, then you should partition the data first (otherwise, the overhead of record rejection due to a partition mismatch slows down the load).

12.10.6 Initiating Multiple SQL*Loader Sessions

Each SQL*Loader session takes a different data file 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 start three SQL*Loader direct path load sessions on the same table, you would execute each of the following commands at the operating system prompt. After entering each command, you will be prompted for a password.

```
sqlldr USERID=scott CONTROL=load1.ctl DIRECT=TRUE PARALLEL=TRUE
sqlldr USERID=scott CONTROL=load2.ctl DIRECT=TRUE PARALLEL=TRUE
sqlldr USERID=scott 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.

12.10.7 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.
- Using the FILE Parameter
  This topic describes using the FILE parameter.
- 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.

12.10.7.1 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 file name of any valid data file 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.

12.10.7.2 Using the FILE Parameter

This topic describes 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.

12.10.7.3 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, then 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.

12.10.8 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.

12.10.9 PRIMARY KEY and UNIQUE KEY Constraints

This topic describes using the 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.

12.11 General Performance Improvement Hints

Learn how to enable general performance improvements when using SQL*Loader with parallel data loading.

If you have control over the format of the data to be loaded, then 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), then 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 data file 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, then the loader must scan the input data to find the delimiters. If you use positional fields, then 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 data type 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)
    - Data file 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 data file character set and the database character set are the same. If the character sets are the same, then 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.

Related Topics

- Using CONCATENATE to Assemble Logical Records

13
SQL*Loader Express

SQL*Loader express mode allows you to quickly and easily use SQL*Loader to load simple data types.

- What is SQL*Loader Express Mode?
  SQL*Loader express mode lets you quickly perform a load by specifying only a table name when the table columns are all character, number, or datetime data types, and the input data files contain only delimited character data.
- Using SQL*Loader Express Mode
  To activate SQL*Loader express mode, you can simply specify your user name and a table name.
- SQL*Loader Express Mode Parameter Reference
  This section provides descriptions of the parameters available in SQL*Loader express mode.
- SQL*Loader Express Mode Syntax Diagrams
  To understand SQL*Loader express mode options, refer to these graphic form syntax guides (sometimes called railroad diagrams or DDL diagrams).

13.1 What is SQL*Loader Express Mode?

SQL*Loader express mode lets you quickly perform a load by specifying only a table name when the table columns are all character, number, or datetime data types, and the input data files contain only delimited character data.

In express mode, a SQL*Loader control file is not used. Instead, SQL*Loader uses the table column definitions found in the `ALL_TAB_COLUMNS` view to determine the input field order and data types. For most other settings, it assumes default values which you can override with command-line parameters.

> **Note:**
>
> The only valid parameters for use with SQL*Loader express mode are those described in this chapter. Any other parameters will be ignored or may result in an error.

13.2 Using SQL*Loader Express Mode

To activate SQL*Loader express mode, you can simply specify your user name and a table name.

SQL*Loader prompts you for a password, for example:

```
> sqlldr username TABLE=employees
Password:
.
```

```
.
.

SQL*Loader: Release 18.0.0.0.0 - Production on Mon Oct 16 127:19:39 2017
Version 18.1.0.0.0

Copyright (c) 1982, 2017, Oracle and/or its affiliates.  All rights reserved.

Express Mode Load, Table: EMPLOYEES
.
.
.
```

If you activate SQL*Loader express mode by specifying only the `TABLE` parameter, then SQL*Loader uses default settings for a number of other parameters. You can override most of the default values by specifying additional parameters on the command line.

SQL*Loader express mode generates a log file that includes a SQL*Loader control file. The log file also contains SQL scripts for creating the external table and performing the load using a SQL `INSERT AS SELECT` statement. Neither the control file nor the SQL scripts are used by SQL*Loader express mode. They are made available to you in case you want to use them as a starting point to perform operations using regular SQL*Loader or standalone external tables; the control file is for use with SQL*Loader, whereas the SQL scripts are for use with standalone external tables operations.

- Default Values Used by SQL*Loader Express Mode
  Learn how SQL*Loader express loads tables, what defaults it uses, and under what conditions the defaults are changed.
- How SQL*Loader Express Mode Handles Byte Order
  The type of character set used with your data file affects the byte order used with SQL*Loader express.

> **See Also:**
>
> - SQL*Loader Express Mode Parameter Reference
> - SQL*Loader Control File Reference for more information about control files

13.2.1 Default Values Used by SQL*Loader Express Mode

Learn how SQL*Loader express loads tables, what defaults it uses, and under what conditions the defaults are changed.

By default, a load done using SQL*Loader express mode assumes the following, unless you specify otherwise:

- If no data file is specified, then it looks for a file named *`table-name`*`.dat` in the current directory.
- By default, SQL*Loader express uses the external tables load method. However, for some errors, SQL*Loader express mode automatically switches from the default external tables load method to direct path load. An example of when this can occur is if a privilege violation caused the `CREATE DIRECTORY` SQL command to fail.
- SQL*Loader express fields are set up as follows:

  - Names, from table column names (the order of the fields matches the table column order)
  - Types, based on table column types
  - Newline, as the record delimiter
  - Commas, as field delimiters
  - No enclosure
  - Left-right trimming
- The `DEGREE_OF_PARALLELISM` parameter is set to `AUTO`.
- Date and timestamp format use the NLS settings.
- The NLS client character set is used.
- If a table already has data in it, then new data is appended to the table.
- If you do not specify a data file, then the data, log, and bad files take the following default names (note the *`%p`* is replaced with the process ID of the Oracle Database child process):
  - *`table-name.dat`* for the data file
  - *`table-name.log`* for the SQL*Loader log file
  - *`table-name_%p`*`.log_xt` for Oracle Database log files (for example, `emp_17228.log_xt`)
  - *`table-name_%p`*`.bad` for bad files
- If you specify one or more data files, using the `DATA` parameter, then the log and bad files take the following default names (note the *`%p`* is replaced with the process ID of the server child process.):
  - *`table-name`*`.log` for the SQL*Loader log file
  - *`table-name_%p`*`.log_xt` for the Oracle Database log files
  - *`first-data-file_%p`*`.bad` for the bad files

Related Topics

- DATA
  The SQL*Loader express mode `DATA` parameter specifies names of data files containing the data that you want to load.

13.2.2 How SQL*Loader Express Mode Handles Byte Order

The type of character set used with your data file affects the byte order used with SQL*Loader express.

In general, SQL*Loader express mode handles byte order marks in the same way that a load performed using a SQL*Loader control file does.

In summary:

- For data files with a Unicode character set, SQL*Loader express mode checks for a byte order mark at the beginning of the file.
- For a UTF16 data file, if a byte order mark is found, the byte order mark sets the byte order for the data file. If no byte order mark is found, the byte order of the system where SQL*Loader is executing is used for the data file.
- A UTF16 data file can be loaded regardless of whether or not the byte order (endianness) is the same byte order as the system on which SQL*Loader express is running.

- For UTF8 data files, any byte order marks found are skipped.
- A load is terminated if multiple data files are involved and they use different byte ordering.

Related Topics

- Understanding how SQL*Loader Manages Byte Ordering
  SQL*Loader can load data from a data file 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 data file contains certain nonportable data types.

13.3 SQL*Loader Express Mode Parameter Reference

This section provides descriptions of the parameters available in SQL*Loader express mode.

Some of the parameter names are the same as parameters used by regular SQL*Loader, but there may be behavior differences. Be sure to read the descriptions so you know what behavior to expect.

> **Note:**
>
> If parameter values include quotation marks, then it is recommended that you specify them in a parameter file. See "Use of Quotation Marks on the Data Pump Command Line" in Parameters Available in Data Pump Export Command-Line Mode - the issues discussed there are also pertinent to SQL*Loader express mode.

- BAD
  The SQL*Loader express mode BAD parameter specifies the location and name of the bad file.
- CHARACTERSET
  The SQL*Loader express mode `CHARACTERSET` parameter specifies a character set you want to use for the load.
- CSV
  The SQL*Loader express mode `CSV` parameter lets you you specify if CSV format files contain fields with embedded record terminators.
- DATA
  The SQL*Loader express mode `DATA` parameter specifies names of data files containing the data that you want to load.
- DATE_FORMAT
  The SQL*Loader express mode `DATE_FORMAT` parameter specifies a date format that overrides the default value for all date fields.
- DEGREE_OF_PARALLELISM
  The SQL*Loader express mode `DEGREE_OF_PARALLELISM` parameter specifies the degree of parallelism to use for the load.
- DIRECT
  The SQL*Loader express mode `DIRECT` parameter specifies the load method to use, either conventional path or direct path.
- DNFS_ENABLE
  The SQL*Loader express mode `DNFS_ENABLE` parameter lets you enable and disable use of the Direct NFS Client on input data files during a SQL*Loader operation.

- DNFS_READBUFFERS
  The SQL*Loader express mode `DNFS_READBUFFERS` parameter lets you control the number of read buffers used by the Direct NFS Client.
- ENCLOSED_BY
  The SQL*Loader express mode `ENCLOSED_BY` parameter specifies a field enclosure string.
- EXTERNAL_TABLE
  The SQL*Loader express mode `EXTERNAL_TABLE` parameter determines whether to load data using the external tables option.
- FIELD_NAMES
  The SQL*Loader express mode `FIELD_NAMES` parameter overrides the fields being in the order of the columns in the database table.
- LOAD
  The SQL*Loader express mode `LOAD` specifies the number of records that you want to be loaded.
- NULLIF
  The SQL*Loader express mode `NULLIF` parameter specifies a value that is used to determine whether a field is loaded as a NULL column.
- OPTIONALLY_ENCLOSED_BY
  The SQL*Loader express mode `OPTIONALLY_ENCLOSED_BY` specifies an optional field enclosure string.
- PARFILE
  The SQL*Loader express mode `PARFILE` parameter specifies the name of a file that contains commonly used command-line parameters.
- SILENT
  The SQL*Loader express mode `SILENT` parameter suppresses some content that is written to the screen during a SQL*Loader operation.
- TABLE
  The SQL*Loader express mode `TABLE` parameter activates SQL*Loader express mode.
- TERMINATED_BY
  The SQL*Loader express mode `TERMINATED_BY` specifies a field terminator that overrides the default.
- TIMESTAMP_FORMAT
  The `TIMESTAMP_FORMAT` parameter specifies a timestamp format that you want to use for the load.
- TRIM
  The SQL*Loader express mode `TRIM` parameter specifies the type of field trimming that you want to use during the load.
- USERID
  The SQL*Loader express mode `USERID` enables you to provide provide your Oracle username and password, so that you are not prompted for it.

13.3.1 BAD

The SQL*Loader express mode BAD parameter specifies the location and name of the bad file.

Default

The default depends on whether any data files are specified, using the `DATA` parameter.

Purpose

The BAD parameter specifies the location and name of the bad file.

Syntax

```
BAD=[directory/][filename]
```

Usage Notes

The bad file stores records that cause errors during insert or that are improperly formatted. If you specify the BAD parameter, then you must supply either a directory or file name, or both. If you do not specify the BAD parameter, and there are rejected records, then the default file name is used.

The *directory* variable specifies a directory to which the bad file is written. The specification can include the name of a device or a network node.

The *filename* variable specifies a file name recognized as valid on your platform. You must specify only a name (and extension, if you want to use one other than .bad). Any spaces or punctuation marks in the file name must be enclosed in single quotation marks.

The values of *directory* and *filename* are determined as follows:

- If you specify the BAD parameter with a file name, but no directory, then the directory defaults to the current directory.
- If you specify the BAD parameter with a directory, but no file name, then the specified directory is used, and the default is used for the file name and the extension.

The BAD parameter applies to all the files that match the specified DATA parameter, if you specify the DATA parameter. If you do not specify the DATA parameter, then the BAD parameter applies to the one data file (table-name.dat)

> **Caution:**
>
> - If the file name (either the default or one you specify) already exists, then that file name either is overwritten, or a new version is created, depending on your operating system.
> - If multiple data files are being loaded, then Oracle recommends that you either not specify the BAD parameter, or that you specify it with only a directory for the bad file.

Example

The following specification creates a bad file named emp1.bad in the current directory:

```
> sqlldr hr TABLE=employees BAD=emp1
```

13.3.2 CHARACTERSET

The SQL*Loader express mode `CHARACTERSET` parameter specifies a character set you want to use for the load.

Default

The NLS client character set as specified in the `NLS_LANG` environment variable

Purpose

The `CHARACTERSET` parameter specifies a character set, other than the default, to use for the load.

Syntax

```
CHARACTERSET=character_set_name
```

The *character_set_name* variable specifies the character set name. Normally, the specified name must be the name of a character set that is supported by Oracle Database.

Usage Notes

The `CHARACTERSET` parameter specifies the character set of the SQL*Loader input data files. If the `CHARACTERSET` parameter is not specified, then the default character set for all data files is the session character set, which is defined by the `NLS_LANG` environment variable. Only character data (fields of the SQL*Loader data types `CHAR`, `VARCHAR`, `VARCHARC`, numeric `EXTERNAL`, and the datetime and interval data types) is affected by the character set of the data file.

For UTF-16 Unicode encoding, use the name UTF16 rather than AL16UTF16. AL16UTF16, which is the supported 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 data file, the data in the data file 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 data file that has little-endian byte order, then SQL*Loader issues a warning message and processes the data file as little-endian.

The `CHARACTERSET` parameter value is assumed to the be same for all data files.

> **Note:**
>
> 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.

Restrictions

None.

Example

The following example specifies the UTF-8 character set:

```
> sqlldr hr TABLE=employees CHARACTERSETNAME=utf8
```

13.3.3 CSV

The SQL*Loader express mode `CSV` parameter lets you you specify if CSV format files contain fields with embedded record terminators.

Default

If the CSV parameter is not specified on the command line, then SQL*Loader express assumes that the CSV file being loaded contains data that has no embedded characters and no enclosures.

If `CSV=WITHOUT_EMBEDDED` is specified on the command line, then SQL*Loader express assumes that the CSV file being loaded contains data that has no embedded characters and that is optionally enclosed by "".

Purpose

The `CSV` parameter provides options that let you specify whether the comma-separated value (CSV) format file being loaded contains fields in which record terminators are embedded.

Syntax

```
CSV=[WITH_EMBEDDED | WITHOUT_EMBEDDED]
```

- `WITH_EMBEDDED` — This option means that there can be record terminators included (embedded) in a field in the record. The record terminator is newline. The default delimiters are `TERMINTATED by ","` and `OPTIONALLY_ENCLOSED_BY '"'`. Embedded record terminators must be enclosed.
- `WITHOUT_EMBEDDED` — This option means that there are no record terminators included (embedded) in a field in the record. The record terminator is newline. The default delimiters are `TERMINATED BY ","` and `OPTIONALLY_ENCLOSED_BY ' " '`.

Usage Notes

If the CSV file contains many embedded record terminators, then it is possible that performance can be adversely affected by this parameter.

Restrictions

- Normally a file can be processed in parallel (split up and processed by more than one execution server at a time). But in the case of CSV format files with embedded record terminators, the file must be processed by only one execution server. Therefore, parallel processing within a data file is disabled when you set the `CSV` parameter to `CSV=WITH_EMBEDDED`.

Example

The following example processes the data files as CSV format files with embedded record terminators.

```
> sqlldr hr TABLE=employees CSV=WITH_EMBEDDED
```

13.3.4 DATA

The SQL*Loader express mode `DATA` parameter specifies names of data files containing the data that you want to load.

Default

The same name as the table name, but with an extension of `.dat`.

Purpose

The `DATA` parameter specifies names of data files containing the data that you want to load.

Syntax

```
DATA=data-file-name
```

If you do not specify a file extension, then the default is `.dat`.

Usage Notes

The file specification can contain wildcards, but only in the file name and file extension, not in a device or directory name. An asterisk (`*`) represents multiple characters. A question mark (`?`) represents a single character. For example:

```
DATA='emp*.dat'
```

```
DATA='m?emp.dat'
```

To list multiple data file specifications (each of which can contain wild cards), you must separate the file names by commas.

If the file name contains any special characters (for example, spaces, `*`, or `?`), then the entire name must be enclosed within single quotation marks.

The following are three examples of possible valid uses of the DATA parameter (the single quotation marks would only be necessary if the file name contained special characters):

```
DATA='file1','file2','file3','file4','file5','file6'


DATA='file1','file2'
DATA='file3,'file4','file5'
DATA='file6'


DATA='file1'
DATA='file2'
DATA='file3'
DATA='file4'
DATA='file5'
DATA='file6'
```

> **Caution:**
>
> If multiple data files are being loaded, and you also specify the BAD parameter, then Oracle recommends that you specify only a directory for the bad file, not a file name. If you specify a file name, and a file with that name already exists, then that file either is overwritten, or a new version is created, depending on your operating system.

Example

Assume that the current directory contains data files with the names emp1.dat, emp2.dat, m1emp.dat, and m2emp.dat and you issue the following command:

```
> sqlldr hr TABLE=employees DATA='emp*','m1emp'
```

The command loads the emp1.dat, emp2.dat, and m1emp.dat files. The m2emp.dat file is not loaded because it did not match any of the wildcard criteria.

13.3.5 DATE_FORMAT

The SQL*Loader express mode DATE_FORMAT parameter specifies a date format that overrides the default value for all date fields.

Default

If the DATE_FORMAT parameter is not specified, then the NLS_DATE_FORMAT, NLS_LANGUAGE, or NLS_DATE_LANGUAGE environment variable settings (if defined for the SQL*Loader session) are used. If the NLS_DATE_FORMAT is not defined, then dates are assumed to be in the default format defined by the NLS_TERRITORY setting.

Purpose

The DATE_FORMAT parameter specifies a date format that overrides the default value for all date fields.

Syntax

```
DATE_FORMAT=mask
```

The *mask* is a date format mask, which normally is enclosed in double quotation marks.

Example

If the date in the data file was June 25, 2019, then the date format would be specified in the following format:

```
> sqlldr hr TABLE=employees DATE_FORMAT="DD-Month-YYYY"
```

13.3.6 DEGREE_OF_PARALLELISM

The SQL*Loader express mode `DEGREE_OF_PARALLELISM` parameter specifies the degree of parallelism to use for the load.

Default

```
AUTO
```

Purpose

The `DEGREE_OF_PARALLELISM` parameter specifies the degree of parallelism to use for the load.

Syntax

```
DEGREE_OF_PARALLELISM=[degree-num|DEFAULT|AUTO|NONE]
```

Options:

- *degree-num* specifies degree of parallelism. It must be a whole number value from 1 to *n*.
- `DEFAULT` specifies that you want to use the default parallelism *of the database* (not the default parameter value of `AUTO`).
- `AUTO` specifies that Oracle Database automatically sets the degree of parallelism for the load. If the `DEGREE_OF_PARALLELISM` parameter is not specified,. then this is the default option for parallelism.
- `NONE` specifies that you do not want the load performed in parallel. A value of `NONE` is the same as a value of `1`.

Restrictions

- The `DEGREE_OF_PARALLELISM` parameter is ignored if you force the load method to be conventional or direct path (the `NONE` option is used). Any time you specify the `DEGREE_OF_PARALLELISM` parameter, for any value, you receive a message reminding you of this.
- If the load is a default external tables load and an error occurs that causes SQL*Loader express mode to use direct path load instead, then the job is not performed in parallel, even if you had specified a degree of parallelism or had accepted the external tables default of `AUTO`. A message is displayed alerting you to this change.

Example

The following example loads the data without using parallelism:

```
> sqlldr hr TABLE=employees DEGREE_OF_PARALLELISM=NONE
```

Related Topics

- Parallel Execution Concepts

13.3.7 DIRECT

The SQL*Loader express mode `DIRECT` parameter specifies the load method to use, either conventional path or direct path.

Default

No default.

Purpose

The `DIRECT` parameter specifies the load method to use, either conventional path or direct path.

Syntax

```
DIRECT=[TRUE|FALSE]
```

A value of `TRUE` specifies a direct path load. A value of `FALSE` specifies a conventional path load.

Usage Notes

This parameter overrides the SQL*Loader express mode default load method of external tables.

For some errors, SQL*Loader express mode automatically switches from the default external tables load method to direct path load. An example of when this can occur is if a privilege violation caused the `CREATE DIRECTORY` SQL command to fail.

If you use the `DIRECT` parameter to specify a conventional or direct path load, then the following regular SQL*Loader parameters are valid to use in express mode:

- `BINDSIZE`
- `COLUMNARRAYROWS` (direct path loads only)
- `DATE_CACHE`
- `ERRORS`
- `MULTITHREADING` (direct path loads only)
- `NO_INDEX_ERRORS` (direct path loads only)
- `RESUMABLE`
- `RESUMABLE_NAME`
- `RESUMABLE_TIMEOUT`

- ROWS
- SKIP
- STREAMSIZE

Example

In the following example, SQL*Loader uses the direct path load method for the load instead of external tables:

```
> sqlldr hr TABLE=employees DIRECT=TRUE
```

13.3.8 DNFS_ENABLE

The SQL*Loader express mode DNFS_ENABLE parameter lets you enable and disable use of the Direct NFS Client on input data files during a SQL*Loader operation.

Default

TRUE

Purpose

The DNFS_ENABLE parameter lets you enable and disable use of the Direct NFS Client on input data files during a SQL*Loader operation.

The Direct NFS Client is an API that can be implemented by file servers to allow improved performance when Oracle accesses files on those servers.

Syntax

The syntax is as follows:

```
DNFS_ENABLE=[TRUE|FALSE]
```

Usage Notes

SQL*Loader uses the Direct NFS Client interfaces by default when it reads data files over 1 GB. For smaller files, the operating system's I/O interfaces are used. To use the Direct NFS Client on *all* input data files, use DNFS_ENABLE=TRUE.

To disable use of the Direct NFS Client for all data files, specify DNFS_ENABLE=FALSE.

The DNFS_ENABLE parameter can be used in conjunction with the DNFS_READBUFFERS parameter, which can specify the number of read buffers used by the Direct NFS Client.

13.3.9 DNFS_READBUFFERS

The SQL*Loader express mode DNFS_READBUFFERS parameter lets you control the number of read buffers used by the Direct NFS Client.

Default

4

Purpose

The DNFS_READBUFFERS parameter lets you control the number of read buffers used by the Direct NFS Client. The Direct NFS Client is an API that can be implemented by file servers to allow improved performance when Oracle accesses files on those servers.

Syntax

The syntax is as follows:

```
DNFS_READBUFFERS = n
```

Usage Notes

Using values larger than the default can compensate for inconsistent I/O from the Direct NFS Client file server, but using larger values can also result in increased memory usage.

To use this parameter without also specifying the DNFS_ENABLE parameter, the input file must be larger than 1 GB.

13.3.10 ENCLOSED_BY

The SQL*Loader express mode ENCLOSED_BY parameter specifies a field enclosure string.

Default

The default is that there is no enclosure character.

Purpose

The ENCLOSED_BY parameter specifies a field enclosure string.

Syntax

```
ENCLOSED_BY=['string'|x'hex-string']
```

The enclosure character must be a string or a hexadecimal string.

Usage Notes

The same string must be used to signify both the beginning and the ending of the enclosure.

Example

In the following example, the field data is enclosed by the '/' character (forward slash).

```
> sqlldr hr TABLE=employees ENCLOSED_BY='/'
```

13.3.11 EXTERNAL_TABLE

The SQL*Loader express mode EXTERNAL_TABLE parameter determines whether to load data using the external tables option.

Default

```
EXECUTE
```

Purpose

The EXTERNAL_TABLE parameter instructs SQL*Loader whether to load data using the external tables option.

Syntax

```
EXTERNAL_TABLE=[NOT_USED | GENERATE_ONLY | EXECUTE]
```

There are three possible values:

- NOT_USED — 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 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 — the default value in SQL*Loader express mode. 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 log file.

Usage Notes

The external table option uses directory objects in the database to indicate where all data files 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 data files, 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 data file or output file, then SQL*Loader will generate the SQL statement to create one. Therefore, when the EXECUTE option is specified, you must have the CREATE ANY DIRECTORY privilege. If you want the directory object to be deleted at the end of the load, then you must also have the DROP 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 attempts to use directory objects that already exist, and that you have privileges to access. However, if SQL*Loader does not find the matching directory object, then it 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 issue, 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.

Example

```
sqlldr hr TABLE=employees EXTERNAL_TABLE=NOT_USED
```

13.3.12 FIELD_NAMES

The SQL*Loader express mode `FIELD_NAMES` parameter overrides the fields being in the order of the columns in the database table.

Default

`NONE`

Purpose

The `FIELD_NAMES` parameter is used to override the fields being in the order of the columns in the database table. (By default, SQL*Loader Express uses the table column definitions found in the `ALL_TAB_COLUMNS` view to determine the input field order and data types.)

An example of when this parameter could be useful is when the data in the input file is not in the same order as the columns in the table. In such a case, you can include a field name record (similar to a column header row for a table) in the data file and use the `FIELD_NAMES` parameter to notify SQL*Loader to process the field names in the first record to determine the order of the fields.

Syntax

```
FIELD_NAMES=[ALL | ALL_IGNORE | FIRST | FIRST_IGNORE | NONE]
```

The valid options for this parameter are as follows:

- `ALL` — The field name record is processed for every data file.
- `ALL_IGNORE` — Ignore the first (field names) record in all the data files and process the data records normally.
- `FIRST` — In the first data file, process the first (field names) record. For all other data files, there is no field names record, so the data file is processed normally.
- `FIRST_IGNORE` — In the first data file, ignore the first (field names) record and use table column order for the field order.
- `NONE` — There are no field names records in any data file, so the data files are processed normally. This is the default.

Usage Notes

- If any field name has mixed case or special characters (for example, spaces), then you must use either the `OPTIONALLY_ENCLOSED_BY` parameter, or the `ENCLOSED_BY` parameter to indicate that case should be preserved, and that special characters should be included as part of the field name.

Example

If you are loading a CSV file that contains column headers into a table, and the fields in each row in the input file are in the same order as the columns in the table, then you could use the following:

```
> sqlldr hr TABLE=employees CSV=WITHOUT_EMBEDDED FIELD_NAMES=FIRST_IGNORE
```

13.3.13 LOAD

The SQL*Loader express mode `LOAD` specifies the number of records that you want to be loaded.

Default

All records are loaded.

Purpose

The `LOAD` parameter specifies the number of records that you want to be loaded.

Syntax

```
LOAD=n
```

Usage Notes

To test that all parameters you have specified for the load are set correctly, use the `LOAD` parameter to specify a limited number of records rather than loading all records. No error occurs if fewer than the maximum number of records are found.

Example

The following example specifies that a maximum of 10 records be loaded:

```
> sqlldr hr TABLE=employees LOAD=10
```

For external tables method loads (the default load method for express mode), only successfully loaded records are counted toward the total. So if there are 15 records in the file and records 2 and 4 are bad, then the following records are loaded into the table, for a total of 10 records - 1, 3, 5, 6, 7, 8, 9, 10, 11, and 12.

For conventional and direct path loads, both successful and unsuccessful load attempts are counted toward the total. So if there are 15 records in the file and records 2 and 4 are bad, then only the following 8 records are actually loaded into the table - 1, 3, 5, 6, 7, 8, 9, and 10.

13.3.14 NULLIF

The SQL*Loader express mode `NULLIF` parameter specifies a value that is used to determine whether a field is loaded as a NULL column.

Default

The default is that no `NULLIF` checking is done.

Syntax

```
NULLIF = "string"
```

Or:

```
NULLIF != "string"
```

Usage Notes

SQL*Loader checks the specified value against the value of the field in the record. If there is a match using the equal (=) or not equal (!=) specification, then the field is set to `NULL` for that row. Any field that has a length of 0 after blank trimming is also set to `NULL`.

Example

In the following example, if there are any fields whose value is a period, then those fields are set to NULL in their respective rows.

```
> sqlldr hr TABLE=employees NULLIF="."
```

13.3.15 OPTIONALLY_ENCLOSED_BY

The SQL*Loader express mode `OPTIONALLY_ENCLOSED_BY` specifies an optional field enclosure string.

Default

The default is that there is no optional field enclosure character.

Purpose

The `OPTIONALLY_ENCLOSED_BY` parameter specifies an optional field enclosure string.

Syntax

```
OPTIONALLY_ENCLOSED_BY=['string'| x'hex-string']
```

The enclosure character is a string or a hexadecimal string.

Usage Notes

You must use the same string to signify both the beginning and the ending of the enclosure.

Examples

The following example specifies the optional enclosure character as a double quotation mark (").

```
> sqlldr hr TABLE=employees OPTIONALLY_ENCLOSED_BY='"'
```

The following example specifies the optional enclosure character in hexadecimal format:

```
> sqlldr hr TABLE=employees OPTIONALLY_ENCLOSED_BY=x'22'
```

13.3.16 PARFILE

The SQL*Loader express mode `PARFILE` parameter specifies the name of a file that contains commonly used command-line parameters.

Default

There is no default

Syntax

```
PARFILE=parameter_file_name
```

Usage Notes

If any parameter values contain quotation marks, then Oracle recommends that you use a parameter file.

> **Note:**
>
> Although it is not usually important, on some systems it can be necessary to have no spaces around the equal sign (=) in the parameter specifications.

Restrictions

- For security reasons, Oracle recommends that you do not include your `USERID` password in a parameter file. After you specify the parameter file at the command line, SQL*Loader prompts you for the password. For example:

  ```
  > sqlldr hr TABLE=employees PARFILE=daily_report.par
  Password:
  ```

Example

Suppose you have the following parameter file, `test.par`:

```
table=employees
data='mydata*.dat'
enclosed_by='"'
```

When you run the following command, any fields enclosed by double quotation marks, in any data files that match `mydata*.dat`, are loaded into table `employees`:

```
> sqlldr hr PARFILE=test.par
Password:
```

13.3.17 SILENT

The SQL*Loader express mode `SILENT` parameter suppresses some content that is written to the screen during a SQL*Loader operation.

Default

\\If this parameter is not specified, then no content is suppressed.

Purpose

The `SILENT` parameter suppresses some of the content that is written to the screen during a SQL*Loader operation.

Syntax

The syntax is as follows:

SILENT={HEADER | FEEDBACK | ERRORS | DISCARDS | PARTITIONS | ALL}
Use the appropriate values to suppress one or more of the following (if more than one option is specified, they must be separated by commas):

- `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" messages and the status messages for the load 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. This option is ignored in express mode.
- `PARTITIONS` — Disables writing the per-partition statistics to the log file during a direct load of a partitioned table. This option is meaningful only in a forced direct path operation.
- `ALL` — Implements all of the suppression options.

Example

For example, you can suppress the header and feedback messages that normally appear on the screen with the following command-line argument:

```
> sqlldr hr TABLE=employees SILENT=HEADER, FEEDBACK
```

13.3.18 TABLE

The SQL*Loader express mode `TABLE` parameter activates SQL*Loader express mode.

Default

There is no default.

Syntax

```
TABLE=[schema-name.]table-name
```

Usage Notes

If the schema name or table name includes lower case characters, spaces, or other special characters, then the names must be enclosed in double quotation marks and that entire string enclosed within single quotation marks. For example:

```
TABLE='"hr.Employees"'
```

Restrictions

The `TABLE` parameter is valid only in SQL*Loader express mode.

Example

The following example loads the table employees in express mode:

```
> sqlldr hr TABLE=employees
```

13.3.19 TERMINATED_BY

The SQL*Loader express mode `TERMINATED_BY` specifies a field terminator that overrides the default.

Default

By default, comma is the field terminator.

Purpose

The `TERMINATED_BY` parameter specifies a field terminator that overrides the default.

Syntax

```
TERMINATED_BY=['string'| x'hex-string' | WHITESPACE]
```

The field terminator must be a string or a hexadecimal string.

Usage Notes

If you specify `TERMINATED_BY=WHITESPACE`, then 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 method allows field values to be delimited by varying amounts of whitespace.

If you specify `TERMINATED_BY=WHITESPACE`, then null fields cannot contain just blanks or other whitespace, because the blanks and whitespace are skipped, which can result in an error being reported. With this option, if you have null fields in the data, then consider using another string to indicate the null field, and use the `NULLIF` parameter to indicate the `NULLIF` string. For example, you can use the string `"NoData"` to indicate a null field, and then insert the string

"NoData" in the data to indicate a null field. Specify NULLIF="NoData" to tell SQL*Loader to set fields with the string "NoData" to NULL.

Example

In the following example, fields are terminated by the | character.

```
> sqlldr hr TABLE=employees TERMINATED_BY="|"
```

13.3.20 TIMESTAMP_FORMAT

The TIMESTAMP_FORMAT parameter specifies a timestamp format that you want to use for the load.

Default

The default is taken from the value of the NLS_TIMESTAMP_FORMAT environment variable. If NLS_TIMESTAMP_FORMAT is not set up, then timestamps use the default format defined in the NLS_TERRITORY environment variable, with 6 digits of fractional precision.

Syntax

```
TIMESTAMP_FORMAT="timestamp_format"
```

Example

The following is an example of specifying a timestamp format:

```
> sqlldr hr TABLE=employees TIMESTAMP_FORMAT="MON-DD-YYYY HH:MI:SSXFF AM"
```

13.3.21 TRIM

The SQL*Loader express mode TRIM parameter specifies the type of field trimming that you want to use during the load.

Default

The default for conventional and direct path loads is LDRTRIM. The default for external tables loads is LRTRIM.

Purpose

The TRIM parameter specifies the type of field trimming that you want to use during the load. Use TRIM to specify that you want spaces trimmed from the beginning of a text field, or the end of a text field, or both. Spaces include blanks and other nonprinting characters, such as tabs, line feeds, and carriage returns.

Syntax

```
TRIM=[LRTRIM | NOTRIM | LTRIM | RTRIM |LDRTRIM]
```

Options:

- LRTRIM specifies that you want both leading and trailing spaces trimmed.

- `NOTRIM` specifies that you want no characters trimmed from the field. This setting generally yields the fastest performance.
- `LTRIM` specifies that you want leading spaces trimmed.
- `RTRIM` specifies that you want trailing spaces trimmed.
- `LDRTRIM` is the same as `NOTRIM`unless the field is a delimited field with `OPTIONALLY_ENCLOSED_BY` specified, and the optional enclosures are missing for a particular instance. In that case spaces are trimmed from the left.

Usage Notes

If you specify trimming for a field that is all spaces, then the field is set to `NULL`.

Restrictions

- Only `LDRTRIM` is supported for forced conventional path and forced direct path loads. Any time you specify the `TRIM` parameter, for any value, you receive a message reminding you of this.
- If the load is a default external tables load and an error occurs that causes SQL*Loader express mode to use direct path load instead, then `LDRTRM` is used as the trimming method, even if you specified a different method or had accepted the external tables default of `LRTRIM`. A message is displayed alerting you to this change.

  To use `NOTRIM`, use a control file with the `PRESERVE BLANKS` clause.

Example

The following example reads the fields, trimming all spaces on the right (trailing spaces).

```
> sqlldr hr TABLE=employees TRIM=RTRIM
```

13.3.22 USERID

The SQL*Loader express mode `USERID` enables you to provide provide your Oracle username and password, so that you are not prompted for it.

Default

None.

Purpose

The `USERID` parameter enables you to to provide your Oracle username and password.

Syntax

```
USERID = [username | / | SYS]
```

Usage Notes

If you do not specify the `USERID` parameter, then you are prompted for it. If only a slash is used, then `USERID` defaults to your operating system login.

If you connect as user `SYS`, then you must also specify `AS SYSDBA` in the connect string.

Restrictions

- Because the string, `AS SYSDBA`, contains a blank, some operating systems can require that you place the entire connect string in quotation marks, or marked as a literal by some other method. Some operating systems also require that you precede quotation marks on the command line using an escape character, such as backslashes.

  Refer to your operating system documentation for information about special and reserved characters on your system.

Example

The following example starts the job for user `hr`:

```
> sqlldr USERID=hr TABLE=employees
  Password:
```

13.4 SQL*Loader Express Mode Syntax Diagrams

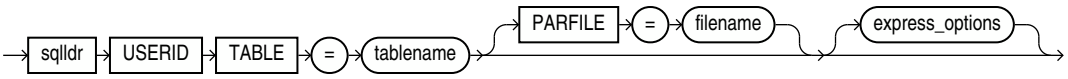
To understand SQL*Loader express mode options, refer to these graphic form syntax guides (sometimes called railroad diagrams or DDL diagrams).

Understanding Graphic Syntax Notation

For information about the syntax notation used, see:

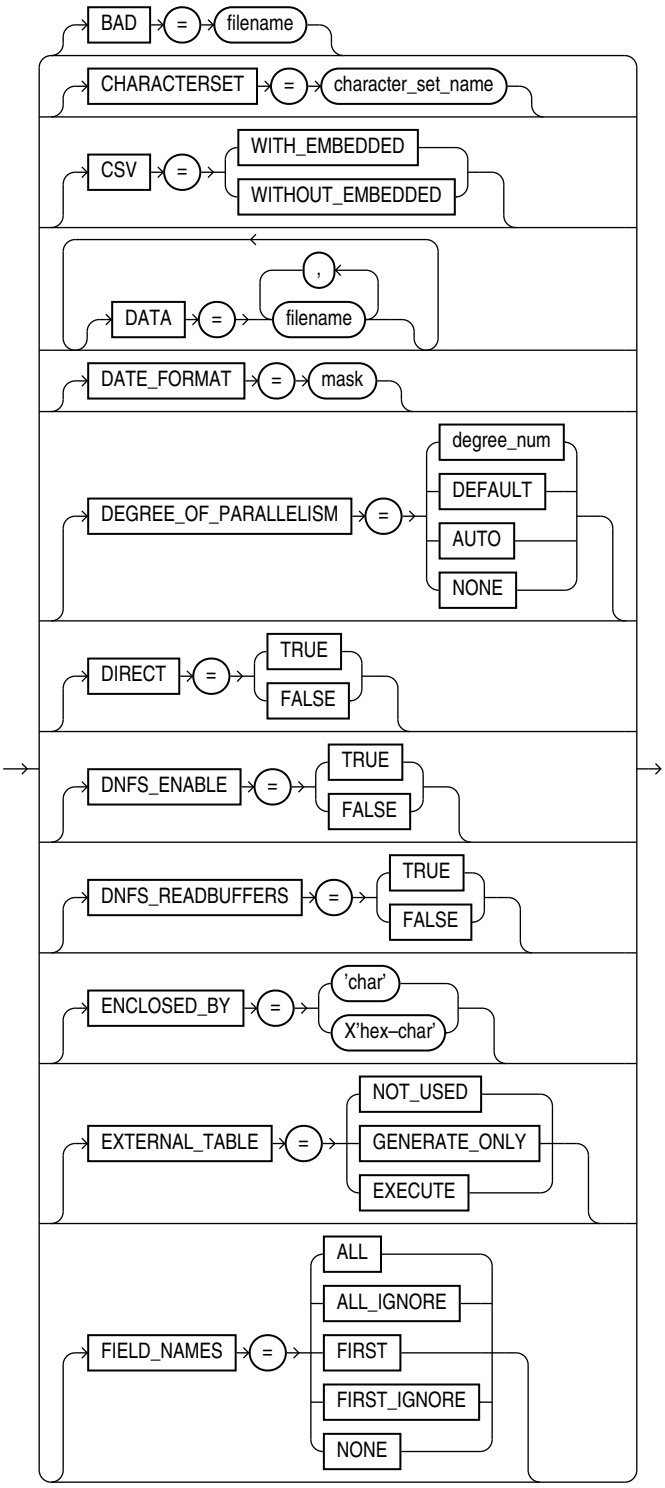
How to Read Syntax Diagrams

express_init

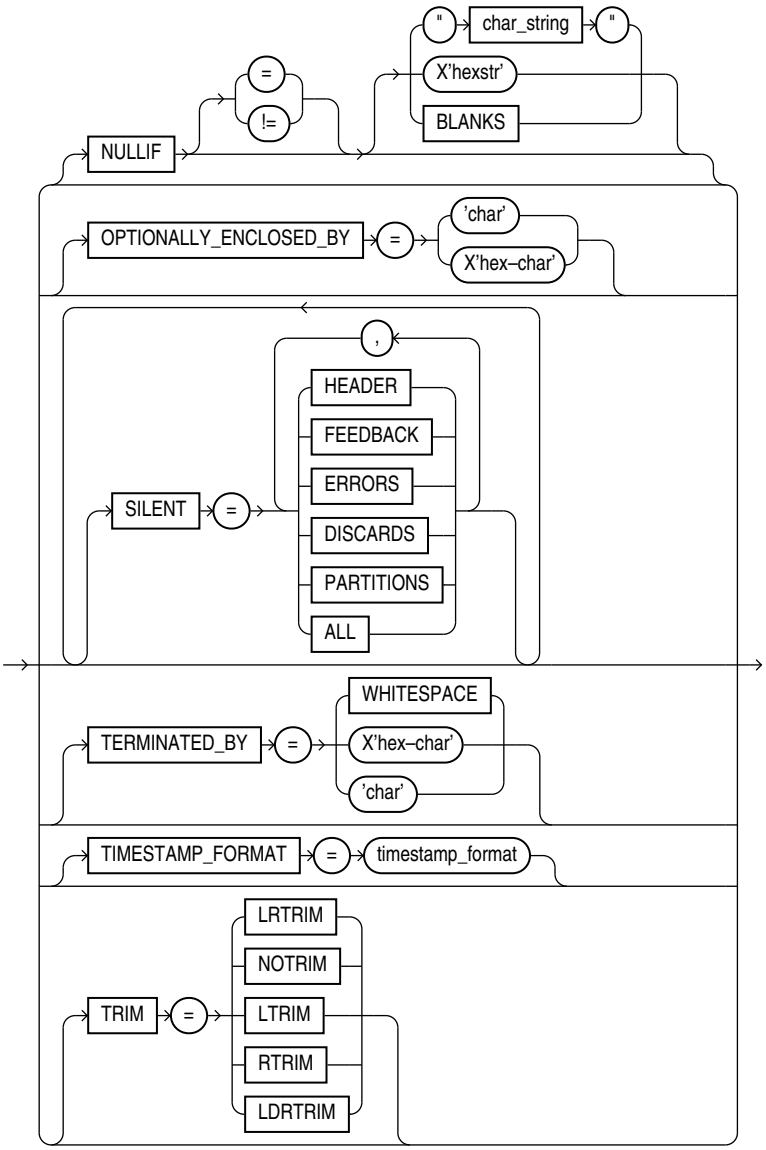


The following syntax diagrams show the parameters included in `express_options` in the previous syntax diagram. SQL*Loader express mode parameters shown in the following syntax diagrams are all optional and can appear in any order on the SQL*Loader command line. Therefore, they are presented in simple alphabetical order.

express_options



express_options_cont



Part III
External Tables

To use external tables successfully, find out about external table concepts, and see examples of what options are available to you to use external tables with Oracle Database.

- 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.
- The ORACLE_LOADER Access Driver
  Learn how to control the way external tables are accessed by using the ORACLE_LOADER access driver parameters to modify the default behavior of the access driver.
- The ORACLE_DATAPUMP Access Driver
  The `ORACLE_DATAPUMP` access driver provides a set of access parameters that are unique to external tables of the type `ORACLE_DATAPUMP`.
- ORACLE_BIGDATA Access Driver
  With the `ORACLE_BIGDATA` access driver, you can access data stored in object stores as if that data was stored in tables in an Oracle Database.
- External Tables Examples
  Learn from these examples how to use the `ORACLE_LOADER`, `ORACLE_DATAPUMP`, `ORACLE_HDFS`, and `ORACLE_HIVE` access drivers to query data in Oracle Database and Big Data.

14
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.

- How Are External Tables Created?
  External tables are created using the SQL `CREATE TABLE...ORGANIZATION EXTERNAL` statement.
- Location of Data Files and Output Files
  Data files and output files must be located on the server. You must have a directory object that specifies the location from which to read and write files.
- CREATE_EXTERNAL_PART_TABLE Procedure
  This procedure creates an external partitioned table on files in the Cloud. This procedure enables you to run queries on external data in Oracle Autonomous Database, using the `ORACLE_BIGDATA` driver.
- CREATE_EXTERNAL_TABLE Procedure
  This procedure creates an external table on files in the Cloud or from files in a directory. This enables you to run queries on external data from Oracle Database.
- Location of Data Files and Output Files
  Data files and output files must be located on the server. You must have a directory object that specifies the location from which to read and write files.
- Access Parameters for External Tables
  To modify the default behavior of the access driver for external tables, specify access parameters.
- Data Type Conversion During External Table Use
  If source and target data types do not match, then conversion errors can occur when Oracle Database reads from external tables, and when it writes to external tables.

14.1 How Are External Tables Created?

External tables are created using the SQL `CREATE TABLE...ORGANIZATION EXTERNAL` statement.

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" for more information about how load behavior differs between SQL*Loader and external tables.

As of Oracle Database 12c Release 2 (12.2.0.1), you can partition data contained in external tables, which allows you to take advantage of the same performance improvements provided when you partition tables stored in a database (for example, partition pruning).

> **Note:**
>
> External tables can be used as inline external tables in SQL statements, thus eliminating the need to create an external table as a persistent database object in the data dictionary. For additional information, see *Oracle Database SQL Language Reference*.

When you create an external table, you specify the following attributes:

- `TYPE` — specifies the type of external table. Each type of external table is supported by its own access driver.
  - `ORACLE_LOADER` — this is the default access driver. It loads data from external tables to internal tables. The data must come from text data files. (The `ORACLE_LOADER` access driver cannot perform unloads; that is, it cannot move data from an internal table to an external table.)
  - `ORACLE_DATAPUMP` — this 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 binary dump files of the external table. The `ORACLE_DATAPUMP` access driver can write dump files only as part of creating an external table with the SQL `CREATE TABLE AS SELECT` statement. After the dump file is created, it can be read any number of times, but it cannot be modified (that is, no DML operations can be performed).
  - `ORACLE_HIVE` — extracts data stored in Apache HIVE.
  - `ORACLE_BIGDATA` — this access driver enables you to access data stored in object stores as if that data was stored in tables in an Oracle Database.
- `DEFAULT DIRECTORY` — specifies the default directory to use for all input and output files that do not explicitly name a directory object. The location is specified with a directory object, not a directory path. You must create the directory object before you create the external table; otherwise, an error is generated. See Location of Data Files and Output Files for more information.
- `ACCESS PARAMETERS` — describe the external data source and implement 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. Access parameters are optional. See Access Parameters.
- `LOCATION` — specifies the data files for the external table.

  For `ORACLE_LOADER`, `ORACLE_BIGDATA`, and `ORACLE_DATAPUMP`, the files are named in the form *`directory:file`*. The *`directory`* portion is optional. If it is missing, then the default directory is used as the directory for the file. If you are using the `ORACLE_LOADER` access driver, then you can use wildcards in the file name: an asterisk (*) signifies multiple characters, a question mark (?) signifies a single character.

The following examples briefly show the use of attributes for each of the access drivers.

Example 14-1 Specifying Attributes for the ORACLE_LOADER Access Driver

The following example uses the `ORACLE_LOADER` access driver to show the use of each of these attributes (it assumes that the default directory `def_dir1` already exists):

```
SQL> CREATE TABLE emp_load
  2    (employee_number      CHAR(5),
  3     employee_dob         CHAR(20),
  4     employee_last_name   CHAR(20),
  5     employee_first_name  CHAR(15),
  6     employee_middle_name CHAR(15),
  7     employee_hire_date   DATE)
  8  ORGANIZATION EXTERNAL
  9    (TYPE ORACLE_LOADER
 10     DEFAULT DIRECTORY def_dir1
 11     ACCESS PARAMETERS
 12       (RECORDS DELIMITED BY NEWLINE
 13        FIELDS (employee_number      CHAR(2),
 14                employee_dob         CHAR(20),
 15                employee_last_name   CHAR(18),
 16                employee_first_name  CHAR(11),
 17                employee_middle_name CHAR(11),
 18                employee_hire_date   CHAR(10) date_format DATE mask "mm/dd/
yyyy"
 19               )
 20       )
 21     LOCATION ('info.dat')
 22    );

Table created.
```

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.

Example 14-2 Specifying Attributes for the ORACLE_DATAPUMP Access Driver

This 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` sample 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.
```

Example 14-3 Specifying Attributes for the ORACLE_BIGDATA Access Driver

```
CREATE TABLE tab_from_csv
     (
      c0 number,
```

```
 c1 varchar2(20)
)
ORGANIZATION external
(
 TYPE oracle_bigdata
 DEFAULT DIRECTORY data_pump_dir
 ACCESS PARAMETERS
 (
  com.oracle.bigdata.fileformat=csv
 )
 location
 (
  'data.csv'
 )
)REJECT LIMIT 1
;
```

Related Topics

- Behavior Differences Between SQL*Loader and External Tables
  Oracle recommends that you review the differences between loading data with external tables, using the `ORACLE_LOADER` access driver, and loading data with SQL*Loader conventional and direct path loads.
- *Oracle Database Administrator's Guide* Managing External Tables

14.2 Location of Data Files and Output Files

Data files and output files must be located on the server. You must have a directory object that specifies the location from which to read and write files.

> **Note:**
>
> The information in this section about directory objects does not apply to data files for the `ORACLE_HDFS` access driver or `ORACLE_HIVE` access driver. With the `ORACLE_HDFS` driver, the location of data is specified with a list of URIs for a directory or for a file, and there is no directory object associated with a URI. The `ORACLE_HIVE` driver does not specify a data source location; it reads the Hive metastore table to get that information, so no directory object is needed.

The access driver runs inside the database server. This behavior 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 requires access to files that the access driver can load.
- The server must create and write the output files created by the access driver: the log file, bad file, discard file, and also any dump files created by the `ORACLE_DATAPUMP` access driver.

To specify the location from which to read and write files, the access driver requires that you use a directory object. 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';
```

DBAs or any user can create directory objects with the `CREATE ANY DIRECTORY` privilege.

> **Note:**
>
> To use external tables in an Oracle Real Applications Cluster (Oracle RAC) configuration, you must ensure that the directory object path is on a cluster-wide file system.

After a directory is created, the user creating the directory object must grant `READ` and `WRITE` privileges on the directory to other users. These privileges must be explicitly granted, rather than assigned by using 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 Oracle Database `SYS` user is the only user that can own directory objects, but the `SYS` user can grant other Oracle Database users the privilege to create directory objects.`READ` or `WRITE` permission to a directory object means that only Oracle Database reads or writes that file on your behalf. You are not given direct access to those files outside of Oracle Database, unless you have the appropriate operating system privileges. Similarly, Oracle Database requires permission from the operating system to read and write files in the directories.

14.3 CREATE_EXTERNAL_PART_TABLE Procedure

This procedure creates an external partitioned table on files in the Cloud. This procedure enables you to run queries on external data in Oracle Autonomous Database, using the `ORACLE_BIGDATA` driver.

Use Case

Starting with Oracle Database 19c, when you are using the `ORACLE_BIGDATA` driver with object stores, you are now able to select column values from a path in external tables. This feature enables you to query and load files in object storage that are partitioned, which represent the partition columns for the table.

Syntax

```
DBMS_CLOUD.CREATE_EXTERNAL_PART_TABLE (
    table_name           IN VARCHAR2,
    credential_name      IN VARCHAR2,
    partitioning_clause  IN CLOB,
    column_list          IN CLOB,
    field_list           IN CLOB DEFAULT,
    format               IN CLOB DEFAULT);
```

Parameters

Parameter	Description
table_name	The name of the external table. For example: 'mysales'
credential_name	The name of the credential to access the Cloud Object Storage. When resource principal is enabled, you can use 'OCI$RESOURCE_PRINCIPAL' as the credential_name
partitioning_clause	Specifies the complete partitioning clause, including the location information for individual partitions. If you use the partitioning_clause parameter, then the file_url_list parameter is not allowed.
file_uri_list	There are two options for the file_uri_list parameter: • A comma-delimited list of individual file URIs without wildcards. • A single file URI with wildcards. The wildcards can only be after the last slash "/". If you use the parameter file_url_list, then the partitioning_clause parameter is not allowed. The specification should be the root folder in a nested path, where are multiple files within a folder structure that have the same schema. For example: https://objectstorage.us-phoenix-1.oraclecloud.com/n/namespace-string/b/mybucket/0/sales/month=jan2022.csv https://objectstorage.us-phoenix-1.oraclecloud.com/n/namespace-string/b/mybucket/0/sales/month=feb2022.csv In this case, the root folder for the sales table is /0/sales
column_list	Comma-delimited list of column names and data types for the external table. This parameter has the following requirements, depending on the type of the data files specified with the file_url_list parameter: • The column_list parameter is required with unstructured files. Using unstructured files, for example with CSV text files, the column_list parameter must specify all the column names and data types inside the data file as well as the partition columns derived from the object name. • The column_list parameter is optional with structured files. For example, with Avro, ORC, or Parquet data files, the column_list is not required. When the column_list is not included, the format parameter partition_columns option must include specifications for both column names (name) and data types (type). For example: 'product varchar2(100), units number, country varchar2(100), year number, month varchar2(2)',
field_list	Identifies the fields in the source files and their data types. The default value is NULL, meaning the fields and their data types are determined by the column_list parameter. This argument's syntax is the same as the field_list clause in regular Oracle Database external tables. The field_list is not required for structured files, such as Apache Parquet files..

Parameter	Description
format	The format option partition_columns specifies the DBMS_CLOUD.CREATE_EXTERNAL_PART_TABLE column names and data types of partition columns when the partition columns are derived from the file path, depending on the type of data file, structured or unstructured: • When the DBMS_CLOUD.CREATE_EXTERNAL_PART_TABLE includes the column_list parameter and the data files are unstructured, such as with CSV text files, partition_columns does not include the data type. For example, use a format such as the following for this type of partition_columns specification: '"partition_columns":["state","zipcode"]' The data type is not required because it is specified in the DBMS_CLOUD.CREATE_EXTERNAL_PART_TABLE column_list parameter. • When the DBMS_CLOUD.CREATE_EXTERNAL_PART_TABLE does not include the column_list parameter and the data files are structured, such as Avro, ORC, or Parquet files, the partition_columns option includes both the column name, name sub-clause, and the data type, type sub-clause. For example, the following shows a partition_columns specification: '"partition_columns":[{"name":"country", "type":"varchar2(10)"}, {"name":"year", "type":"number"}, {"name":"month", "type":"varchar2(10)"}]' If the data files are unstructured and the type sub-clause is specified with partition_columns, the type sub-clause is ignored. For object names that are not based on hive format, the order of the partition_columns specified columns must match the order as they appear in the object name in the file path specified in the file_url_list parameter.

Usage Notes

- You cannot call this procedure with both partitioning_clause and file_url_list parameters.
- Specifying the column_list parameter is optional with structured data files, including Avro, Parquet, or ORC data files. If column_list is not specified, the format parameter partition_columns option must include both name and type.
- The column_list parameter is required with unstructured data files, such as CSV text files.
- The procedure DBMS_CLOUD.CREATE_EXTERNAL_PART_TABLE supports external partitioned files in the supported cloud object storage services, including:
  - Oracle Cloud Infrastructure Object Storage
  - Azure Blob Storage

  - Amazon S3-Compatible, including: Oracle Cloud Infrastructure Object Storage, Google Cloud Storage, and Wasabi Hot Cloud storage.
  - GitHub Repository

- When you call `DBMS_CLOUD.CREATE_EXTERNAL_PART_TABLE` with the `file_url_list` parameter, the types for columns specified in the Cloud Object Store file name must be one of the following types:

  ```
  VARCHAR2(n)
  NUMBER(n)
  NUMBER(p,s)
  NUMBER
  DATE
  TIMESTAMP(9)
  ```

- The default record delimiter is `detected newline`. With `detected newline`, `DBMS_CLOUD` tries to automatically find the correct newline character to use as the record delimiter. `DBMS_CLOUD` first searches for the Windows newline character `\r\n`. If it finds the Windows newline character, this is used as the record delimiter for all files in the procedure. If a Windows newline character is not found, `DBMS_CLOUD` searches for the UNIX/Linux newline character `\n`, and if it finds one it uses `\n` as the record delimiter for all files in the procedure. If the source files use a combination of different record delimiters, you may encounter an error such as, "`KUP-04020: found record longer than buffer size supported`". In this case, you need to either modify the source files to use the same record delimiter or only specify the source files that use the same record delimiter.
- The external partitioned tables that you create with `DBMS_CLOUD.CREATE_EXTERNAL_PART_TABLE` include two invisible columns, `file$path` and `file$name`. These columns help identify which file a record is coming from.
  - `file$path`: Specifies the file path text up to the beginning of the object name.
  - `file$name`: Specifies the object name, including all the text that follows the bucket name.

Examples

Example using the `partitioning_clause` parameter:

```
BEGIN
   DBMS_CLOUD.CREATE_EXTERNAL_PART_TABLE(
      table_name =>'PET1',
      credential_name =>'OBJ_STORE_CRED',
      format => json_object('delimiter' value ',', 'recorddelimiter' value
'newline', 'characterset' value 'us7ascii'),
      column_list => 'col1 number, col2 number, col3 number',
      partitioning_clause => 'partition by range (col1)
                              (partition p1 values less than (1000) location
                                  ( ''&base_URL//file_11.txt'')
                               ,
                               partition p2 values less than (2000) location
                                  ( ''&base_URL/file_21.txt'')
                               ,
                               partition p3 values less than (3000)
location
                                  ( ''&base_URL/file_31.txt'')
                               )'
```

```
        );
    END;
/
```

Example using the `file_uri_list` and `column_list` parameters with unstructured data files:

```
BEGIN
  DBMS_CLOUD.CREATE_EXTERNAL_PART_TABLE(
   table_name => 'MYSALES',
   credential_name => 'DEF_CRED_NAME',
   file_uri_list     => 'https://objectstorage.us-phoenix-1.oraclecloud.com/n/namespace-
string/b/bucketname/o/*.csv',
   column_list       => 'product varchar2(100), units number, country varchar2(100), year
number, month varchar2(2)',
   field_list        => 'product, units', --[Because country, year and month are not in the
file, they are not listed in the field list]
   format            => '{"type":"csv", "partition_columns":["country","year","month"]}');
END;
/
```

Example using the `file_uri_list` without the `column_list` parameter with structured data files:

```
BEGIN
  DBMS_CLOUD.CREATE_EXTERNAL_PART_TABLE(
  table_name => 'MYSALES',
  credential_name => 'DEF_CRED_NAME',
  DBMS_CLOUD.CREATE_EXTERNAL_PART_TABLE(
    table_name      => 'MYSALES',
    credential_name => 'DEF_CRED_NAME',
    file_uri_list   => 'https://objectstorage.us-phoenix-1.oraclecloud.com/n/namespace-
string/b/bucketname/o/*.parquet',
    format          =>
        json_object('type' value 'parquet', 'schema' value 'first',
                    'partition_columns' value
                          json_array(
                                json_object('name' value 'country', 'type' value
'varchar2(100)'),
                                json_object('name' value 'year', 'type' value 'number'),
                                json_object('name' value 'month', 'type' value 'varchar2(2)')
                          )
         )
    );
END;
/
```

Example with a partitioned Apache Parquet source. You can run this example, because the data is public.

In this case, data is organized into months. The resource principal was enabled, as shown below. However, because this is a public data source, it is not required.

> **Note:**
>
> The list of columns is not required, because it is derived from the Parquet source. You do need to specify the data type for month, because there is no column list.

```
BEGIN
  DBMS_CLOUD.CREATE_EXTERNAL_PART_TABLE(
   credential_name => 'OCI$RESOURCE_PRINCIPAL',
   table_name => 'sales',
   file_uri_list => 'https://objectstorage.us-ashburn-1.oraclecloud.com/n/
c4u04/b/moviestream_gold/o/custsales/*.parquet',
   format => '{"type":"parquet","partition_columns":
[{name:"month","type":"varchar2(20)"}]}'
   );
END;
/
mgubar: Finally, here is the generated ddl:
CREATE TABLE sales
   ( "DAY_ID" TIMESTAMP (6),
      "GENRE_ID" NUMBER(19,0),
      "MOVIE_ID" NUMBER(19,0),
      "CUST_ID" NUMBER(19,0),
      "APP" VARCHAR2(4000 BYTE),
      "DEVICE" VARCHAR2(4000 BYTE),
      "OS" VARCHAR2(4000 BYTE),
      "PAYMENT_METHOD" VARCHAR2(4000 BYTE),
      "LIST_PRICE" BINARY_DOUBLE,
      "DISCOUNT_TYPE" VARCHAR2(4000 BYTE),
      "DISCOUNT_PERCENT" BINARY_DOUBLE,
      "ACTUAL_PRICE" BINARY_DOUBLE,
      "MONTH" VARCHAR2(20 BYTE)
   )
   ORGANIZATION EXTERNAL
    ( TYPE ORACLE_BIGDATA
      DEFAULT DIRECTORY "DATA_PUMP_DIR"
      ACCESS PARAMETERS
       ( com.oracle.bigdata.fileformat=parquet
com.oracle.bigdata.filename.columns=["month"]
com.oracle.bigdata.file_uri_list="https://objectstorage.us-
ashburn-1.oraclecloud.com/n/c4u04/b/moviestream_gold/o/custsales/*.parquet"
com.oracle.bigdata.credential.schema="ADMIN"
com.oracle.bigdata.credential.name="OCI$RESOURCE_PRINCIPAL"
com.oracle.bigdata.trimspaces=notrim
  )
    )
   REJECT LIMIT 0
  PARTITION BY LIST ("MONTH")
 (PARTITION "P1" VALUES (('2019-01'))
      LOCATION
       ( 'https://objectstorage.us-ashburn-1.oraclecloud.com/n/c4u04/b/
moviestream_gold/o/custsales/month=2019-01/*.parquet'
       ),
 PARTITION "P2" VALUES (('2019-02'))
      LOCATION
```

```
       ( 'https://objectstorage.us-ashburn-1.oraclecloud.com/n/c4u04/b/
moviestream_gold/o/custsales/month=2019-02/*.parquet'
       ),
 PARTITION "P3" VALUES (('2019-03'))
      LOCATION
       ( 'https://objectstorage.us-ashburn-1.oraclecloud.com/n/c4u04/b/
moviestream_gold/o/custsales/month=2019-03/*.parquet'
       ),
 PARTITION "P4" VALUES (('2019-04'))
      LOCATION
       ( 'https://objectstorage.us-ashburn-1.oraclecloud.com/n/c4u04/b/
moviestream_gold/o/custsales/month=2019-04/*.parquet'
       ),
 ...
 PARTITION "P24" VALUES (('2020-12'))
      LOCATION
       ( 'https://objectstorage.us-ashburn-1.oraclecloud.com/n/c4u04/b/
moviestream_gold/o/custsales/month=2020-12/*.parquet'
       ))
  PARALLEL ;
```

Example of not requiring a field list. Parquet is a structured file. Because the file is Parquet, the field list is derived from the structured file.

```
CREATE TABLE ADMIN.EXT_CUSTSALES
   ( DAY_ID TIMESTAMP (6),
      GENRE_ID NUMBER(19,0),
      MOVIE_ID NUMBER(19,0),
      CUST_ID NUMBER(19,0),
      APP VARCHAR2(4000 BYTE),
      DEVICE VARCHAR2(4000 BYTE),
      OS VARCHAR2(4000 BYTE),
      PAYMENT_METHOD VARCHAR2(4000 BYTE),
      LIST_PRICE BINARY_DOUBLE,
      DISCOUNT_TYPE VARCHAR2(4000 BYTE),
      DISCOUNT_PERCENT BINARY_DOUBLE,
      ACTUAL_PRICE BINARY_DOUBLE
   ) DEFAULT COLLATION USING_NLS_COMP
   ORGANIZATION EXTERNAL
    ( TYPE ORACLE_BIGDATA
      DEFAULT DIRECTORY DATA_PUMP_DIR
      ACCESS PARAMETERS
       ( com.oracle.bigdata.fileformat=parquet
com.oracle.bigdata.trimspaces=notrim
  )
      LOCATION
       ( 'https://objectstorage.us-ashburn-1.oraclecloud.com/n/c4u04/b/
moviestream_landing/o/sales_sample/*.parquet'
       )
    )
   REJECT LIMIT UNLIMITED
  PARALLEL ;
```

Related Topics

- ORACLE_LOADER Access Driver field_list under field_definitions Clause
- DBMS_CLOUD Package Format Format Options in *Oracle Database PL/SQL Packages and Types Reference*

14.4 CREATE_EXTERNAL_TABLE Procedure

This procedure creates an external table on files in the Cloud or from files in a directory. This enables you to run queries on external data from Oracle Database.

Use Case

Starting with Oracle Database 19c, when you are using the `ORACLE_BIGDATA` driver with object stores, you are now able to select column values from a path in external tables.

Syntax

```
DBMS_CLOUD.CREATE_EXTERNAL_TABLE (
    table_name        IN VARCHAR2,
    credential_name   IN VARCHAR2,
    file_uri_list     IN CLOB,
    column_list       IN CLOB,
    field_list        IN CLOB DEFAULT,
    format            IN CLOB DEFAULT);
```

Parameters

Parameter	Description
`table_name`	The name of the external table.
`credential_name`	The name of the credential to access the Cloud Object Storage. This parameter is not used when you specify a directory with `file_uri_list`.
`file_uri_list`	Comma-delimited list of source file URIs. You can use wildcards in the file names in your URIs. The character "*" can be used as the wildcard for multiple characters, the character "?" can be used as the wildcard for a single character. The format of the URIs depend on the Cloud Object Storage service you are using. For details see: DBMS_CLOUD URI Formats
`column_list`	Comma-delimited list of column names and data types for the external table.
`field_list`	Identifies the fields in the source files and their data types. The default value is NULL meaning the fields and their data types are determined by the `column_list` parameter. This argument's syntax is the same as the `field_list` clause in regular Oracle external tables. For more information about `field_list` see: ORACLE_LOADER Access Driver field_list under field_definitions Clause.

Parameter	Description
format	The options describing the format of the source files. For the list of the options and how to specify the values see: DBMS_CLOUD Package Format Options in *Oracle Database PL/SQL Packages and Types Reference* For Avro or Parquet format files, see: DBMS_CLOUD CREATE_EXTERNAL_TABLE Procedure for Avro or Parquet Files in *Oracle Database PL/SQL Packages and Types Reference*

Usage Notes

The procedure DBMS_CLOUD.CREATE_EXTERNAL_TABLE supports external partitioned files in the supported cloud object storage services, including:

- Oracle Cloud Infrastructure Object Storage
- Azure Blob Storage
- Amazon S3

The credential is a table level property; therefore, the external files must be on the same object store.

See DBMS_CLOUD URI Formats in *Oracle Database PL/SQL Packages and Types Reference*

Example

```
BEGIN
   DBMS_CLOUD.CREATE_EXTERNAL_TABLE(
      table_name =>'WEATHER_REPORT_DOUBLE_DATE',
      credential_name =>'OBJ_STORE_CRED',
      file_uri_list =>'&base_URL/
Charlotte_NC_Weather_History_Double_Dates.csv',
      format => json_object('type' value 'csv', 'skipheaders' value '1'),
      field_list => 'REPORT_DATE DATE''mm/dd/yy'',
                     REPORT_DATE_COPY DATE ''yyyy-mm-dd'',
                     ACTUAL_MEAN_TEMP,
                     ACTUAL_MIN_TEMP,
                     ACTUAL_MAX_TEMP,
                     AVERAGE_MIN_TEMP,
                     AVERAGE_MAX_TEMP,
                     AVERAGE_PRECIPITATION',
      column_list => 'REPORT_DATE DATE,
                     REPORT_DATE_COPY DATE,
                     ACTUAL_MEAN_TEMP NUMBER,
                     ACTUAL_MIN_TEMP NUMBER,
                     ACTUAL_MAX_TEMP NUMBER,
                     AVERAGE_MIN_TEMP NUMBER,
                     AVERAGE_MAX_TEMP NUMBER,
                     AVERAGE_PRECIPITATION NUMBER');
   END;
/

SELECT * FROM WEATHER_REPORT_DOUBLE_DATE where
   actual_mean_temp > 69 and actual_mean_temp < 74
```

Related Topics

- DBMS_CLOUD in *Oracle Database PL/SQL Packages and Types Reference*

14.5 Location of Data Files and Output Files

Data files and output files must be located on the server. You must have a directory object that specifies the location from which to read and write files.

> **Note:**
>
> The information in this section about directory objects does not apply to data files for the `ORACLE_HDFS` access driver or `ORACLE_HIVE` access driver. With the `ORACLE_HDFS` driver, the location of data is specified with a list of URIs for a directory or for a file, and there is no directory object associated with a URI. The `ORACLE_HIVE` driver does not specify a data source location; it reads the Hive metastore table to get that information, so no directory object is needed.

The access driver runs inside the database server. This behavior 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 requires access to files that the access driver can load.
- The server must create and write the output files created by the access driver: the log file, bad file, discard file, and also any dump files created by the `ORACLE_DATAPUMP` access driver.

To specify the location from which to read and write files, the access driver requires that you use a directory object. 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';
```

DBAs or any user can create directory objects with the `CREATE ANY DIRECTORY` privilege.

> **Note:**
>
> To use external tables in an Oracle Real Applications Cluster (Oracle RAC) configuration, you must ensure that the directory object path is on a cluster-wide file system.

After a directory is created, the user creating the directory object must grant `READ` and `WRITE` privileges on the directory to other users. These privileges must be explicitly granted, rather than assigned by using 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 Oracle Database SYS user is the only user that can own directory objects, but the SYS user can grant other Oracle Database users the privilege to create directory objects.READ or WRITE permission to a directory object means that only Oracle Database reads or writes that file on your behalf. You are not given direct access to those files outside of Oracle Database, unless you have the appropriate operating system privileges. Similarly, Oracle Database requires permission from the operating system to read and write files in the directories.

14.6 Access Parameters for External Tables

To modify the default behavior of the access driver for external tables, specify 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. Oracle provides the following access drivers for use with external tables: ORACLE_LOADER , ORACLE_DATAPUMP, ORACLE_HDFS, and ORACLE_HIVE.

> **Note:**
>
> These access parameters are collectively referred to as the opaque_format_spec in the SQL CREATE TABLE...ORGANIZATION EXTERNAL statement. The ACCESS parameter clause allows SQL comments.

Related Topics

- The ORACLE_LOADER Access Driver
- The ORACLE_DATAPUMP Access Driver
- CREATE TABLE in*Oracle Database SQL Language Reference*

> **See Also:**
>
> *Oracle Database SQL Language Reference* for information about specifying opaque_format_spec when using the SQL CREATE TABLE statement

14.7 Data Type Conversion During External Table Use

If source and target data types do not match, then conversion errors can occur when Oracle Database reads from external tables, and when it writes to external tables.

Conversion Errors When Reading External Tables

When you select rows from an external table, the access driver performs any transformations necessary to make the data from the data source match the data type of the corresponding column in the external table. Depending on the data and the types of transformations required, the transformation can encounter errors.

To illustrate the types of data conversion problems that can occur when reading from an external table, suppose you create the following external table, KV_TAB_XT, with two columns: KEY, whose data type is VARCHAR2(4), and VAL, whose data type is NUMBER.

```
SQL> CREATE TABLE KV_TAB_XT (KEY VARCHAR2(4), VAL NUMBER)
2 ORGANIZATION EXTERNAL
3 (DEFAULT DIRECTORY DEF_DIR1 LOCATION ('key_val.csv'));
```

The external table KV_TAB_XT uses default values for the access parameters. The following is therefore true:

- Records are delimited by new lines.
- The data file and the database have the same character set.
- The fields in the data file have the same name and are in the same order as the columns in the external table.
- The data type of the field is CHAR(255).
- Data for each field is terminated by a comma.

The records in the data file for the KV_TAB_XT external table should have the following:

- A string, up to 4 bytes long. If the string is empty, then the value for the field is NULL.
- A terminating comma.
- A string of numeric characters. If the string is empty, then the value for this field is NULL.
- An optional terminating comma.

When the access driver reads a record from the data file, it verifies that the length of the value of the KEY field in the data file is less than or equal to 4, and it attempts to convert the value of the VAL field in the data file to an Oracle Database number.

If the length of the value of the KEY field is greater than 4, or if there is a non-numeric character in the value for VAL, then the ORACLE_LOADER access driver rejects the row. The result is that a copy of the row is written to the bad file, and an error message is written to the log file.

All access drivers must handle conversion from the data type of fields in the source for the external table and the data type for the columns of the external tables. The following are some examples of the types of conversions and checks that access drivers perform:

- Convert character data from character set used by the source data to the character set used by the database.
- Convert from character data to numeric data.
- Convert from numeric data to character data.
- Convert from character data to a date or timestamp.
- Convert from a date or timestamp to character data.
- Convert from character data to an interval data type.
- Convert from an interval data type to a character data.
- Verify that the length of data value for a character column does not exceed the length limits of that column.

When the access driver encounters an error doing the required conversion or verification, it can decide how to handle the error. When the ORACLE_LOADER and ORACLE_DATAPUMP access drivers encounter errors, they reject the record, and write an error message to the log file. In

that event it is as if that record were not in the data source. When the ORACLE_HDFS and ORACLE_HIVE access drivers encounter errors, the value of the field in which the error is encountered is set to NULL. This action is consistent with the behavior of how Hive handles errors in Hadoop.

Even after the access driver has converted the data from the data source to match the data type of the external table columns, the SQL statement that is accessing the external table could require additional data type conversions. If any of these additional conversions encounter an error, then the entire statement fails. (The exception to this is if you use the DML error logging feature in the SQL statement to handle these errors.) These conversions are the same as any that typically can be required when running a SQL statement. For example, suppose you change the definition of the KV_TAB_XT external table to only have columns with character data types, and then you run an INSERT statement to load data from the external table into another table that has a NUMBER data type for column VAL:

```
SQL> CREATE TABLE KV_TAB_XT (KEY VARCHAR2(20), VAL VARCHAR2(20))
2 ORGANIZATION EXTERNAL
3 (DEFAULT DIRECTORY DEF_DIR1 LOCATION ('key_val.csv'));
4 CREATE TABLE KV_TAB (KEY VARCHAR2(4), VAL NUMBER);
5 INSERT INTO KV_TAB SELECT * FROM KV_TAB_XT;
```

In this example, the access driver will not reject a record if the data for VAL contains a non-numeric character, because the data type of VAL in the external table is now VARCHAR2 (instead of NUMBER). However, SQL processing now must handle the conversion from character data type in KV_TAB_XT to number data type in KV_TAB. If there is a non-numeric character in the value for VAL in the external table, then SQL raises a conversion error, and rolls back any rows that were inserted. To avoid conversion errors in SQL Oracle recommends that you make the data types of the columns in the external table match the data types expected by other tables or functions that will be using the values of those columns.

Conversion Errors When Writing to External Tables

The ORACLE_DATAPUMP access driver allows you to use a CREATE TABLE AS SELECT statement to unload data into an external table. Data conversion occurs if the data type of a column in the SELECT expression does not match the data type of the column in the external table. If SQL encounters an error while converting the data type, then SQL stops the statement, and the data file will not be readable.

To avoid problems with conversion errors that cause the operation to fail, the data type of the column in the external table should match the data type of the column in the source table or expression used to write to the external table. This is not always possible, because external tables do not support all data types. In these cases, the unsupported data types in the source table must be converted into a data type that the external table can support. The following CREATE TABLE statement shows an example of this conversion:

```
CREATE TABLE LONG_TAB_XT (LONG_COL CLOB)
ORGANIZATION EXTERNAL...SELECT TO_LOB(LONG_COL) FROM LONG_TAB;
```

The source table named LONG_TAB has a LONG column. Because of that, the corresponding column in the external table being created, LONG_TAB_XT, 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.

> **Note:**
>
> All forms of `LONG` data types (`LONG`, `LONG RAW`, `LONG VARCHAR`, `LONG VARRAW`) were deprecated in Oracle8i Release 8.1.6. For succeeding releases, the `LONG` data type was provided for backward compatibility with existing applications. In new applications developed with later releases, Oracle strongly recommends that you use `CLOB` and `NCLOB` data types for large amounts of character data.

15
The ORACLE_LOADER Access Driver

Learn how to control the way external tables are accessed by using the ORACLE_LOADER access driver parameters to modify the default behavior of the access driver.

- About the ORACLE_LOADER Access Driver
  The `ORACLE_LOADER` access driver provides a set of access parameters unique to external tables of the type `ORACLE_LOADER`.
- access_parameters Clause
  The `access_parameters` clause contains comments, record formatting, and field formatting information.
- record_format_info Clause
  Learn how to parse, label and manage record information with the `record_format_info` clause and its subclauses.
- field_definitions Clause
  In the `field_definitions` clause, you use the `FIELDS` parameter to name the fields in the data file and specify how to find them in records.
- column_transforms Clause
  The optional `ORACLE_LOADER` access drive `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 data file.
- Parallel Loading Considerations for the ORACLE_LOADER Access Driver
  The `ORACLE_LOADER` access driver attempts to divide large data files into chunks that can be processed separately.
- Performance Hints When Using the ORACLE_LOADER Access Driver
  This topic describes some performance hints when using the `ORACLE_LOADER` access driver.
- Restrictions When Using the ORACLE_LOADER Access Driver
  This section lists restrictions to be aware of when you use the `ORACLE_LOADER` access driver.
- 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.

15.1 About the ORACLE_LOADER Access Driver

The `ORACLE_LOADER` access driver provides a set of access parameters unique to external tables of the type `ORACLE_LOADER`.

You can use the access parameters to modify the default behavior of the access driver. 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.

To use the external table management features that the ORACLE_LOADER access parameters provide, you must have some knowledge of the file format and record format (including character sets and field data types) of the data files on your platform. You must also

know enough about SQL to be able to create an external table, and to perform queries against it.

You can find it helpful to use the `EXTERNAL_TABLE=GENERATE_ONLY` parameter in SQL*Loader to obtain 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. You can edit and customize these SQL statements. You can perform the actual load later without the use of SQL*Loader by executing these statements in SQL*Plus.

Note:

- It is sometimes difficult to understand `ORACLE_LOADER` access driver parameter syntax without reference to other `ORACLE_LOADER` access driver parameters. If you have difficulty understanding the syntax of a particular parameter, then refer to it in context with other referenced parameters.
- Be aware that in ORACLE_LOADER access driver parameter examples that show a `CREATE TABLE...ORGANIZATION EXTERNAL` statement, followed by an example of contents of the data file for the external table, the contents of the data file in the example are not part of the `CREATE TABLE` statement. They are present in the example only 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, then it must be enclosed in double quotation marks.

Related Topics

- EXTERNAL_TABLE
  The `EXTERNAL_TABLE` parameter instructs SQL*Loader whether to load data using the external tables option.
- 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.
- *Oracle Database Administrator's Guide*

15.2 access_parameters Clause

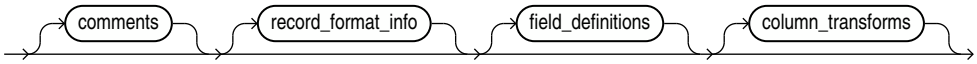
The `access_parameters` clause contains comments, record formatting, and field formatting information.

Default

None.

Syntax

The syntax for the `access_parameters` clause is as follows:



Purpose

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 data types for fields in the data source can be different from the columns in the external table

The access driver ensures that data from the data source is processed so that it matches the definition of the external table.

> **Note:**
>
> These access parameters are collectively referred to as the `opaque_format_spec` in the SQL `CREATE TABLE...ORGANIZATION EXTERNAL` statement.

> **See Also:**
>
> *Oracle Database SQL Language Reference* for information about specifying `opaque_format_spec` when using the SQL `CREATE TABLE...ORGANIZATION EXTERNAL` statement

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.

field_definitions

The `field_definitions` clause is used to describe the fields in the data file. If a data file 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.

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 data file. This is done using the following transforms: `NULL`, `CONSTANT`, `CONCAT`, and `LOBFILE`. For a full description of the syntax, see column_transforms Clause.

15.3 record_format_info Clause

Learn how to parse, label and manage record information with the `record_format_info` clause and its subclauses.

- Overview of 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.
- FIXED Length
  Use the `record_format_info FIXED` clause to identify the records in external tables as all having a fixed size of length bytes.
- VARIABLE size
  Use the `record_format_info VARIABLE` clause to indicate that the records have a variable length
- DELIMITED BY
  Use the `record_format_info DELIMITED BY` clause to delimit the end-of-record character.
- XMLTAG
  Use the `record_format_info XMLTAG` clause to specify XML tags that are used to load subdocuments from an XML document.
- CHARACTERSET
  Use the `record_format_info CHARACTERSET` clause to specify the character set of the data file.
- EXTERNAL VARIABLE DATA
  To load dump files into the Oracle SQL Connector for HDFS that are generated with the `ORACLE_DATAPUMP` access driver, use the `EXTERNAL VARIABLE DATA` clause.
- PREPROCESSOR
  To specify your own preprocessor program that you want to run for every data file, use the `record_format_info PREPROCESSOR` clause.
- LANGUAGE
  The `LANGUAGE` clause allows you to specify a language name (for example, `FRENCH`), from which locale-sensitive information about the data can be derived.
- TERRITORY
  The `TERRITORY` clause allows you to specify a territory name to further determine input data characteristics.
- 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 data file.
- BYTEORDERMARK [CHECK | NOCHECK]
  Use the `record_format_info BYTEORDERMARK` clause to specify whether the data file should be checked for the presence of a byte-order mark (BOM).

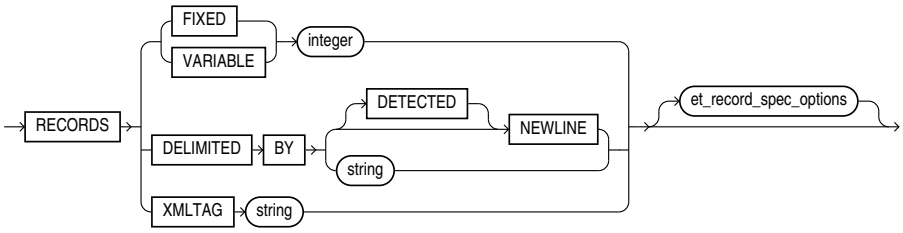
- STRING SIZES ARE IN
  Use the `record_format_info` `STRING SIZES ARE IN` clause to indicate whether the lengths specified for character strings are in bytes or characters.
- LOAD WHEN
  Use the `record_format_info` `LOAD WHEN` clause to identify the records that should be passed to the database.
- BADFILE | NOBADFILE
  Use the `record_format_info` `BADFILE` clause to name the file to which records are written when they cannot be loaded because of errors.
- DISCARDFILE | NODISCARDFILE
  Use the `record_format_info` `DISCARDFILE` clause to name the file to which records are written that fail the condition in the `LOAD WHEN` clause.
- LOGFILE | NOLOGFILE
  Use the `record_format_info` `LOGFILE` clause to name the file that contains messages generated by the external tables utility while it was accessing data in the data file.
- SKIP
  Use the `record_format_info` `SKIP` clause to skip the specified number of records in the data file before loading.
- FIELD NAMES
  Use the `record_format_info` `FIELD NAMES` clause to specify field order in data files.
- READSIZE
  The `READSIZE` parameter specifies the size of the read buffer used to process records.
- DISABLE_DIRECTORY_LINK_CHECK
  The `DISABLE_DIRECTORY_LINK_CHECK` parameter directs the `ORACLE_LOADER` access driver to bypass the symbolic link check.
- DATE_CACHE
- string
  A string is a quoted series of characters or hexadecimal digits.
- condition_spec
  The `condition_spec` specifies one or more conditions that are joined by Boolean operators.
- [directory object name:] [filename]
  The `[directory object name:] [filename]` clause is used to specify the name of an output file (`BADFILE`, `DISCARDFILE`, or `LOGFILE`).
- condition
- IO_OPTIONS clause
- DNFS_DISABLE | DNFS_ENABLE
- DNFS_READBUFFERS
  The `DNFS_READBUFFERS` parameter of the `record_format_info` clause is used to control the number of read buffers used by the Direct NFS Client.

15.3.1 Overview of record_format_info Clause

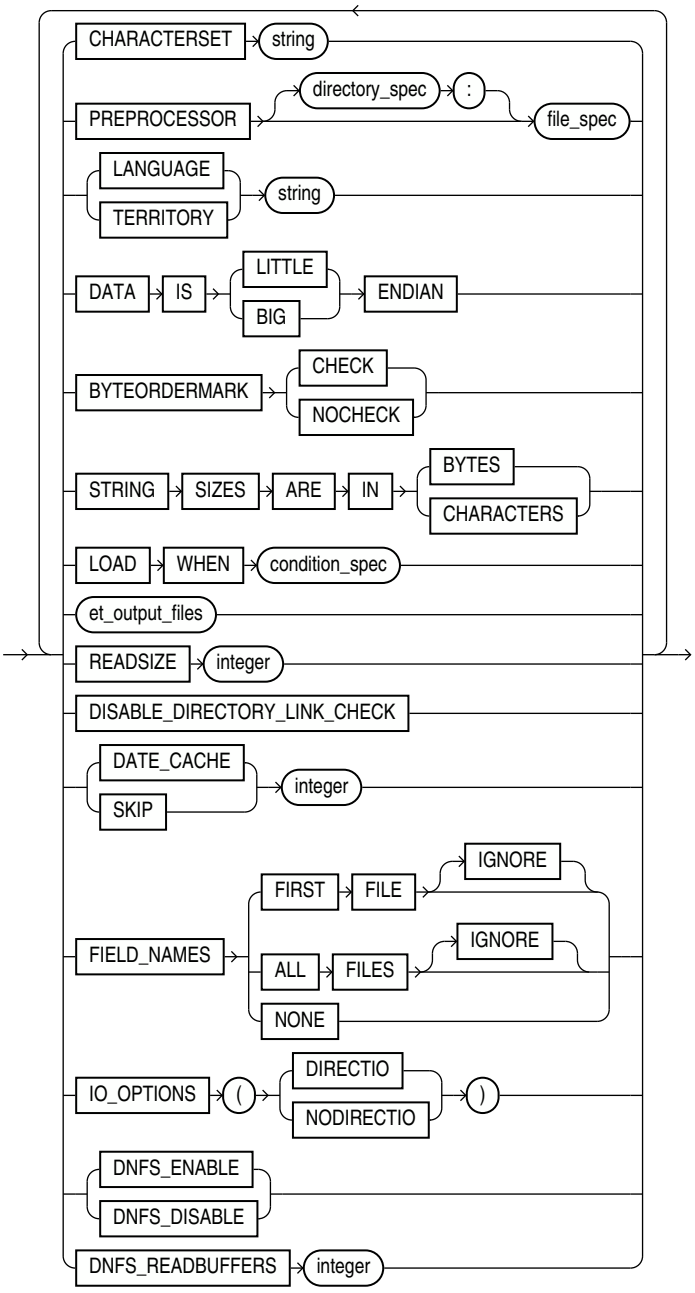
The `record_format_info` clause contains information about the record, such as its format, the character set of the data, and what rules are used to exclude records from being loaded.

The `PREPROCESSOR` clause allows you to optionally specify the name of a user-supplied program that will run and modify the contents of a data file so that the `ORACLE_LOADER` access driver can parse it.

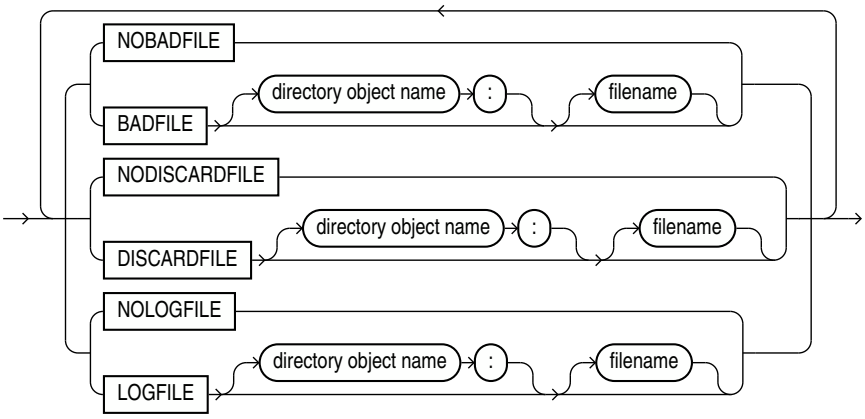
The `record_format_info` clause is optional. The syntax for the `record_format_info` clause is as follows:



The `et_record_spec_options` clause allows you to optionally specify additional formatting information. You can specify as many of the formatting options as you want, in any order. The syntax of the options is as follows:



The following `et_output_files` diagram shows the options for specifying the bad, discard, and log files. For each of these clauses, you must supply either a directory object name or a file name, or both.



15.3.2 FIXED Length

Use the `record_format_info FIXED` clause to identify the records in external tables as all having a fixed size of length bytes.

Default

None.

Purpose

Enables you to identify the records in external tables as all having a fixed size of length bytes.

Usage Notes

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.

Example

The following is an example of using `FIXED` records. In this example, we assume that there is a 1-byte newline character at the end of each record in the data file. After the create table command using `FIXED`, you see an example of the data file that you can load with 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
```

15.3.3 VARIABLE size

Use the `record_format_info VARIABLE` clause to indicate that the records have a variable length

Default

None.

Purpose

Use the `VARIABLE` clause 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.

Example

In the following example of using `VARIABLE` records, there is a 1-byte newline character at the end of each record in the data file. After the SQL example, you see an example of a data file 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,
```

15.3.4 DELIMITED BY

Use the `record_format_info DELIMITED BY` clause to delimit the end-of-record character.

Default

None

Purpose

The `DELIMITED BY` clause is used to indicate the character that identifies the end of a record.

If you specify `DELIMITED BY NEWLINE` then the actual value used is platform-specific. On Unix or Linux operating systems, `NEWLINE` is assumed to be '`\n`'. On Microsoft Windows operating systems, `NEWLINE` is assumed to be '`\r\n`'.

If you are unsure what record delimiter was used when a data file was created, then running an external table query with `DELIMITED BY NEWLINE` can result in files that are incorrectly loaded.

The query can be run without identifying what record delimiter was used when the data file was created. For example, you can work on a Unix or Linux operating system and use a file that was created in Windows format. If you specify `RECORDS DELIMITED BY NEWLINE` on the UNIX or Linux operating system, the delimiter is automatically assumed to be '`\n`'. However, because the file was created in Windows format, in which the records are delimited by '`\r\n`', the file is incorrectly uploaded to the UNIX or Linux operating system.

To resolve problems of different record delimiters, use this syntax:

```
RECORDS DELIMITED BY DETECTED NEWLINE
```

With this syntax, the `ORACLE_LOADER` access driver scans the data looking first for a Windows delimiter ('`\r\n`'). If a Windows delimiter is not found, then the access driver looks for a Unix or Linux delimiter ('`\n`'). The first delimiter found is the one used as the record delimiter.

After a record delimiter is found, the access driver identifies that delimiter as the end of the record. For this reason, if the data contains an embedded delimiter character in a field before the end of the record, then you cannot use the `DETECTED` keyword. This is because the `ORACLE_LOADER` access driver incorrectly assumes that the delimiter in the field denotes the end of the record. As a result, the current and all subsequent records in the file cannot parse correctly.

You cannot mix newline delimiters in the same file. When the `ORACLE_LOADER` access driver finds the first delimiter, then that is the delimiter that it identifies for the records in the file. The access driver then processes all subsequent records in the file by using the same newline character as the delimiter..

If you specify `DELIMITED BY` *`string`*, then *`string`* can be either text or a series of hexadecimal digits enclosed within quotation marks and prefixed by `0X` or `X`. If the string is text, then the text is converted to the character set of the data file, and the result is used for identifying record boundaries.

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 data file.
- Some characters in the delimiter string cannot be translated into the character set of the data file.

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, then 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.

> **Note:**
>
> 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.

Example

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
```

Related Topics

- string
  A string is a quoted series of characters or hexadecimal digits.

15.3.5 XMLTAG

Use the `record_format_info XMLTAG` clause to specify XML tags that are used to load subdocuments from an XML document.

Default

None

Purpose

You can use the `XMLTAG` clause of the `ORACLE_LOADER` access driver to specify XML tags that are used to load subdocuments from an XML document. The access driver searches the data file for documents enclosed by the tags you identify with the clause, and loads those documents as separate rows in the external table.

The `XMLTAG` clause accepts a list of one or more strings. The strings are used to build tags that `ORACLE_LOADER` uses to search for subdocuments in the data file. The tags specified in the access parameters do not include the "<" and ">" delimiters.

The `ORACLE_LOADER` access driver starts at the beginning of the file, and looks for the first occurrence of any of the tags listed in the `XMLTAG` clause. When it finds a match, it then searches for the corresponding closing tag. For example, if the tag is "`ORDER_ITEM`", then `ORACLE_LOADER` looks for the text string "`<ORDER_ITEM>`", starting at the beginning of the file. When it finds an occurrence of "`<ORDER_ITEM>`" it then looks for "`</ORDER_ITEM>`". Everything found between the <ORDER_ITEM> and </ORDER_ITEM> tags is part of the document loaded for the row. `ORACLE_LOADER` then searches for the next occurrence of any of the tags, starting from the first character after the closing tag.

The `ORACLE_LOADER` access driver is not parsing the XML document to the elements that match the tag names; it is only doing a string search through a text file. If the external table is being accessed in parallel, then `ORACLE_LOADER` splits large files up so that different sections are read independently. When it starts reading a section of the data file, it starts looking for one of the tags specified by `XMLTAG`. If it reaches the end of a section and is still looking for a matching end tag, then `ORACLE_LOADER` continues reading into the next section until the matching end tag is found.

Restrictions When Using XMLTAG

- The XMLTAG clause cannot be used to load data files that have elements nested inside of documents of the same element. For example, if a data file being loaded with XMLTAG('FOO') contains the following data:

  ```
  <FOO><BAR><FOO></FOO></BAR></FOO>
  ```

  then ORACLE_LOADER extracts everything between the first <FOO> and the first </FOO> as a document, which does not constitute a valid document.

  Similarly, if XMLTAG("FOO","BAR") is specified and the data file contains the following:

  ```
  <FOO><BAR></BAR></FOO>
  ```

  then <BAR> and </BAR> are loaded, but as the document for "FOO".

- The limit on how large an extracted sub-document can be is determined by the READSIZE access parameter. If the ORACLE_LOADER access driver sees a subdocument larger than READSIZE, then it returns an error.

Example Use of the XMLTAG Clause

Suppose you create an external table T_XT as follows:

```
CREATE TABLE "T_XT"
(
  "C0" VARCHAR2(2000)
)
ORGANIZATION external
(
  TYPE oracle_loader
  DEFAULT DIRECTORY DMPDIR
  ACCESS PARAMETERS
  (
    RECORDS
    XMLTAG ("home address", "work address"," home phone ")
    READSIZE 1024
    SKIP 0
    FIELDS NOTRIM
    MISSING FIELD VALUES ARE NULL
    (
      "C0" (1:2000) CHAR(2000)
    )
  )
  location
  (
    't.dat'
  )
)REJECT LIMIT UNLIMITED
/
exit;
```

Assume the contents of the data file are as follows:

```
<first name>Lionel</first name><home address>23 Oak St, Tripoli, CT</home
address><last name>Rice</last name>
```

You could then perform the following SQL query:

```
SQL> SELECT C0 FROM T_XT;

C0
--------------------------------------------------------------------
<home address>23 Oak St, Tripoli, CT</home address>
```

15.3.6 CHARACTERSET

Use the `record_format_info CHARACTERSET` clause to specify the character set of the data file.

Default

None.

Purpose

The `CHARACTERSET` *`string`* clause identifies the character set of the data file. If a character set is not specified, then the data is assumed to be in the default character set for the database.

> **Note:**
>
> The settings of NLS environment variables on the client have no effect on the character set used for the database.

Related Topics

- string
  A string is a quoted series of characters or hexadecimal digits.
- *Oracle Database Globalization Support Guide*

15.3.7 EXTERNAL VARIABLE DATA

To load dump files into the Oracle SQL Connector for HDFS that are generated with the `ORACLE_DATAPUMP` access driver, use the `EXTERNAL VARIABLE DATA` clause.

Default

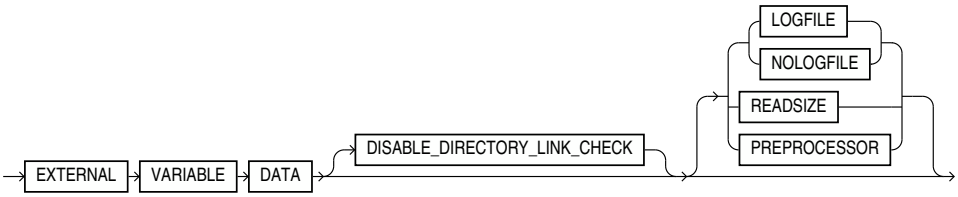
None.

Purpose

When you specify the `EXTERNAL VARIABLE DATA` clause, the `ORACLE_LOADER` access driver is used to load dump files that were generated with the `ORACLE_DATAPUMP` access driver.

> **Note:**
>
> The `EXTERNAL VARIABLE DATA` clause is valid only for use with the Oracle SQL Connector for Hadoop Distributed File System (HDFS). See *Oracle Big Data Connectors User's Guide* for more information about the Oracle SQL Connector for HDFS.

Syntax and Description



You can only use the following access parameters with the `EXTERNAL VARIABLE DATA` clause:

- LOGFILE | NOLOGFILE
- READSIZE
- PREPROCESSOR

> **Note:**
>
> The parameter DISABLE_DIRECTORY_LINK_CHECK is desupported.

Example

In the following example of using the `EXTERNAL VARIABLE DATA` clause, the following scenario is true:

- The `deptxt1.dmp` dump file was previously generated by the `ORACLE_DATAPUMP` access driver.
- The `tkexcat` program specified by the `PREPROCESSOR` parameter is a user-supplied program used to manipulate the input data.

```
CREATE TABLE deptxt1
(
   deptno  number(2),
   dname   varchar2(14),
   loc     varchar2(13)
)
ORGANIZATION EXTERNAL
(
  TYPE ORACLE_LOADER
  DEFAULT DIRECTORY dpump_dir
  ACCESS PARAMETERS
  (
    EXTERNAL VARIABLE DATA
```

```
    LOGFILE 'deptxt1.log'
    READSIZE=10000
    PREPROCESSOR tkexcat
  )
  LOCATION ('deptxt1.dmp')
)
REJECT LIMIT UNLIMITED
;
```

Related Topics

- LOGFILE | NOLOGFILE
  Use the `record_format_info` `LOGFILE` clause to name the file that contains messages generated by the external tables utility while it was accessing data in the data file.
- READSIZE
  The `READSIZE` parameter specifies the size of the read buffer used to process records.
- PREPROCESSOR
  To specify your own preprocessor program that you want to run for every data file, use the `record_format_info` `PREPROCESSOR` clause.

15.3.8 PREPROCESSOR

To specify your own preprocessor program that you want to run for every data file, use the `record_format_info` `PREPROCESSOR` clause.

Default

None.

Purpose

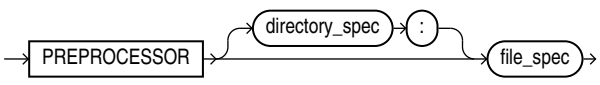
> **Caution:**
>
> There are security implications to consider when using the `PREPROCESSOR` clause.

If the file you want to load contains data records that are not in a format supported by the `ORACLE_LOADER` access driver, then use the `PREPROCESSOR` clause to specify a user-supplied preprocessor program that will execute for every data file. Note that the program specification must be enclosed in a shell script if it uses arguments (see the description of `file_spec`).

The preprocessor program converts the data to a record format supported by the access driver and then writes the converted record data to standard output (stdout), which the access driver reads as input.

Syntax

The syntax of the `PREPROCESSOR` clause is as follows:



directory_spec

Specifies the directory object containing the name of the preprocessor program to execute for every data file. The user accessing the external table must have the `EXECUTE` privilege for the directory object that is used. If `directory_spec` is omitted, then the default directory specified for the external table is used.

> **Caution:**
>
> For security reasons, to store preprocessor programs, Oracle strongly recommends that you use a separate directory. Do not use the default directory. Do not store any other files in the directory in which preprocessor programs are stored.

To maintain security, the preprocessor program must reside in a directory object, so that access to it can be controlled . Your operating system administrator must create a directory corresponding to the directory object, and and must verify that the operating system Oracle user for the database has access to that directory. Database administrators then must ensure that only approved users are granted permissions to the directory object associated with the directory path. Although multiple database users can have access to a directory object, only those with the `EXECUTE` privilege can run a preprocessor in that directory. No existing database user with read-write privileges to a directory object will be able to use the preprocessing feature. As a DBA, you can prevent preprocessors from ever being used by never granting the `EXECUTE` privilege to anyone for a directory object. Refer to *Oracle Database SQL Language Reference* for information about how to grant the EXECUTE privilege.

file_spec

The name of the preprocessor program. It is appended to the path name associated with the directory object that is being used (either the `directory_spec` or the default directory for the external table). The `file_spec` cannot contain an absolute or relative directory path.

If the preprocessor program requires any arguments (for example, `gunzip -c`), then you must specify the program name and its arguments in an executable shell script (or on Microsoft Windows operating systems, in a batch (`.bat`) file). Shell scripts and batch files have certain requirements, as discussed in the following sections.

It is important to verify that the correct version of the preprocessor program is in the operating system directory.

The following is an example of specifying the `PREPROCESSOR` clause without using a shell or batch file:

```
SQL> CREATE TABLE xtab (recno varchar2(2000))
     2    ORGANIZATION EXTERNAL (
     3    TYPE ORACLE_LOADER
     4    DEFAULT DIRECTORY data_dir
     5    ACCESS PARAMETERS (
     6    RECORDS DELIMITED BY NEWLINE
     7    PREPROCESSOR execdir:'zcat'
     8    FIELDS (recno char(2000)))
     9    LOCATION ('foo.dat.gz'))
    10    REJECT LIMIT UNLIMITED;
Table created.
```

Using Shell Scripts With the PREPROCESSOR Clause on Linux Operating Systems

To use shell scripts on Linux, the following conditions must be true:

- The shell script must reside in `directory_spec`.
- The full path name must be specified for system commands such as `gunzip`.
- The preprocessor shell script must have `EXECUTE` permissions.
- The data file listed in the external table `LOCATION` clause should be referred to by `$1`.

The following example shows how to specify a shell script on the `PREPROCESSOR` clause when creating an external table.

```
SQL> CREATE TABLE xtab (recno varchar2(2000))
     2    ORGANIZATION EXTERNAL (
     3    TYPE ORACLE_LOADER
     4    DEFAULT DIRECTORY data_dir
     5    ACCESS PARAMETERS (
     6    RECORDS DELIMITED BY NEWLINE
     7    PREPROCESSOR execdir:'uncompress.sh'
     8    FIELDS (recno char(2000)))
     9    LOCATION ('foo.dat.gz'))
   10    REJECT LIMIT UNLIMITED;
Table created.
```

Using Batch Files With The PREPROCESSOR Clause on Windows Operating Systems

To use shell scripts on Microsoft Windows, the following conditions must be true:

- The batch file must reside in `directory_spec`.
- The full path name must be specified for system commands such as `gunzip`.
- The preprocessor batch file must have EXECUTE permissions.
- The first line of the batch file should contain `@echo off`. The reason for this requirement is that when the batch file is run, the default is to display the commands being executed, which has the unintended side-effect of the echoed commands being treated as input to the external table access driver.
- To represent the input from the location clause, `%1` should be used. (Note that this differs from Unix and Linux-style shell scripts where the location clause is referenced by `$1`.)
- A full path should be specified to any executables in the batch file (`sed.exe` in the following example). Note also that the MKS Toolkit may not exist on all Microsoft Windows installations, so commands such as `sed.exe` may not be available.

  The batch file used on Microsoft Windows must have either a `.bat` or `.cmd` extension. Failure to do so (for example, trying to specify the preprocessor script as `sed.sh`) results in the following error:

  ```
  SQL> select * from foo ;
  select * from foo
  *
  ERROR at line 1:

  ORA-29913: error in executing ODCIEXTTABLEFETCH callout
  ORA-29400: data cartridge error
  ```

```
   KUP-04095: preprocessor command
   C:/Temp\sed.sh encountered error
   "CreateProcess Failure for Preprocessor:
   C:/Temp\sed.sh, errorcode: 193
```

The following is a simple example of using a batch file with the external table `PREPROCESSOR` option on Windows. In this example a batch file uses the stream editor (`sed.exe`) utility to perform a simple transformation of the input data.

```
SQL> create table deptXT (deptno char(2),
  2  dname char(14),
  3  loc char(13)
  4  )
  5  organization external
  6  (
  7  type ORACLE_LOADER
  8  default directory def_dir1
  9  access parameters
 10  (
 11  records delimited by newline
 12  badfile 'deptXT.bad'
 13  logfile 'deptXT.log'
 14  preprocessor exec_dir:'sed.bat'
 15  fields terminated by ','
 16  missing field values are null
 17  )
 18  location ('deptXT.dat')
 19  )
 20  reject limit unlimited ;

Table created.

select * from deptxt ;

Where deptxt.dat contains:

20,RESEARCH,DALLAS
30,SALES,CHICAGO
40,OPERATIONS,BOSTON
51,OPERATIONS,BOSTON
```

The preprocessor program `sed.bat` has the following content:

```
@echo off
c:/mksnt/mksnt/sed.exe -e 's/BOSTON/CALIFORNIA/' %1
```

The `PREPROCESSOR` option passes the input data (`deptxt.dat`) to `sed.bat`. If you then select from the `deptxt` table, the results show that the `LOC` column in the last two rows, which used to be `BOSTON`, is now `CALIFORNIA`.

```
SQL> select * from deptxt ;

DE DNAME          LOC
-- -------------- -------------
```

```
20 RESEARCH,     DALLAS
30 SALES         CHICAGO
40 OPERATIONS    CALIFORNIA
51 OPERATIONS    CALIFORNIA

4 rows selected.
```

Usage Notes for Parallel Processing with the PREPROCESSOR Clause

External tables treat each data file specified on the `LOCATION` clause as a single granule. To make the best use of parallel processing with the `PREPROCESSOR` clause, Oracle recommends that the data that you want to load is split into multiple files (granules). Note that external tables limits the degree of parallelism to the number of data files present. For example, if you specify a degree of parallelism of 16, but have only 10 data files, then in effect the degree of parallelism is 10; this is because 10 child processes are busy, and 6 are idle. To process data more efficiently, avoid idle child processes. If you do specify a degree of parallelism, then try to ensure that the degree of parallelism you specify is no larger than the number of data files, so that all child processes are kept busy. Refer to *Oracle Database VLDB and Partitioning Guide* for more information about granules of parallelism.

Also note that you cannot use the same preprocessor script that you use for file system files to process object store data. If you want to use the preprocessor for object store data, then you must write a preprocessor script that can access the object store data, and modify the data. For example, on Linux or Unix systems, in this case, `$1` represents a source such as `https://www.yoururl.example.com/yourdata`:

```
@echo off
#!/bin/sh/your_script_or_plsql_function_to_display_objectstore_contents($1) |
sed -e 's/BOSTON/CALIFORNIA/'
```

With this syntax, the preprocessor obtains your data, and sends it to `stdout`, and pipes it for the access driver to read.

Restrictions When Using the PREPROCESSOR Clause

- The `PREPROCESSOR` clause is not available on databases that use the Oracle Database Vault feature.
- The `PREPROCESSOR` clause does not work in conjunction with the `COLUMN TRANSFORMS` clause.
- Using Parallel Processing with the PREPROCESSOR Clause
  Using parallel processing with the `PREPROCESSOR` clause.
- Restrictions When Using the PREPROCESSOR Clause
  Describes restrictions when using the `PREPROCESSOR` clause.

Related Topics

- Guidelines for Securing the ORACLE_LOADER Access Driver
- *Oracle Database SQL Language Reference* GRANT

15.3.8.1 Using Parallel Processing with the PREPROCESSOR Clause

Using parallel processing with the `PREPROCESSOR` clause.

External tables treat each data file specified on the `LOCATION` clause as a single granule. To make the best use of parallel processing with the `PREPROCESSOR` clause, the data to be loaded

should be split into multiple files (granules). This is because external tables *limits* the degree of parallelism *to* the number of data files present. For example, if you specify a degree of parallelism of 16, but have only 10 data files, then in effect the degree of parallelism is 10 because 10 slave processes will be busy and 6 will be idle. It is best to not have any idle slave processes. So if you do specify a degree of parallelism, then ideally it should be *no larger* than the number of data files so that all slave processes are kept busy.

See Also:

- *Oracle Database VLDB and Partitioning Guide* for more information about granules of parallelism

15.3.8.2 Restrictions When Using the PREPROCESSOR Clause

Describes restrictions when using the `PREPROCESSOR` clause.

- The `PREPROCESSOR` clause is not available on databases that use the Oracle Database Vault feature.
- The `PREPROCESSOR` clause does not work in conjunction with the `COLUMN TRANSFORMS` clause.

15.3.9 LANGUAGE

The `LANGUAGE` clause allows you to specify a language name (for example, `FRENCH`), from which locale-sensitive information about the data can be derived.

The following are some examples of the type of information that can be derived from the language name:

- Day and month names and their abbreviations
- Symbols for equivalent expressions for A.M., P.M., A.D., and B.C.
- Default sorting sequence for character data when the `ORDER BY` SQL clause is specified
- Writing direction (right to left or left to right)
- Affirmative and negative response strings (for example, `YES` and `NO`)

See Also:

Oracle Database Globalization Support Guide for a listing of Oracle-supported languages

15.3.10 TERRITORY

The `TERRITORY` clause allows you to specify a territory name to further determine input data characteristics.

For example, in some countries a decimal point is used in numbers rather than a comma (for example, 531.298 instead of 531,298).

> **See Also:**
>
> *Oracle Database Globalization Support Guide* for a listing of Oracle-supported territories

15.3.11 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 data file.

Purpose

Indicates the endianness of data whose byte order may vary depending on the platform that generated the data file.

Usage Notes

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 data type in the UTF16 character set
- Any string specified by `RECORDS DELIMITED BY` *`string`*, and in the UTF16 character set

Microsoft Windows-based platforms generate little-endian data. Big-endian platforms include Oracle Solaris and IBM zSeries Based Linux. 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 data files can have a mark at the beginning of the file indicating the endianness of the data. If present, then this mark overrides the `DATA IS...ENDIAN` clause.

15.3.12 BYTEORDERMARK [CHECK | NOCHECK]

Use the `record_format_info BYTEORDERMARK` clause to specify whether the data file should be checked for the presence of a byte-order mark (BOM).

Default

`CHECK`

Syntax

```
BYTEORDERMARK [CHECK | NOCHECK]
```

Purpose

The BYTEORDERMARK clause is used to specify whether the data file 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 data file should not be checked for a BOM and that all the data in the data file should be read as data.

BYTEORDERMARK CHECK indicates that the data file should be checked for a BOM. This is the default behavior for a data file in a Unicode character set.

Usage Notes

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 data file, 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 data file, and the data is actually big-endian, then an error is returned because you specified little-endian.
- If a BOM is not found, and no endianness is specified with the DATA IS...ENDIAN parameter, then the endianness of the platform is used.
- If BYTE ORDER MARK NOCHECK is specified, and the DATA IS...ENDIAN parameter specified an endianness, then that endian value is used. Otherwise, the endianness of the platform is used.

Related Topics

- Understanding how SQL*Loader Manages Byte Ordering
  SQL*Loader can load data from a data file 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 data file contains certain nonportable data types.

15.3.13 STRING SIZES ARE IN

Use the record_format_info STRING SIZES ARE IN clause to indicate whether the lengths specified for character strings are in bytes or characters.

Default

None.

Syntax

```
STRING SIZES ARE IN [BYTES | CHARACTERS]
```

Purpose

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, then 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, then 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.

15.3.14 LOAD WHEN

Use the `record_format_info LOAD WHEN` clause to identify the records that should be passed to the database.

Default

Syntax

The syntax of the `LOAD WHEN` clause is as follows, where *`condition_spec`* are condition specifications:

```
LOAD WHEN condition_spec
```

Purpose

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, then 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 use case is helpful where the records in the file that you do not want to be loaded cannot be parsed into the current record definition without errors.

Example

The following is an examples of using `LOAD WHEN`:

```
LOAD WHEN (empid != BLANKS)
LOAD WHEN ((dept_id = "SPORTING GOODS" OR dept_id = "SHOES") AND total_sales != 0)
```

Related Topics

- condition_spec
  The `condition_spec` specifies one or more conditions that are joined by Boolean operators.

15.3.15 BADFILE | NOBADFILE

Use the `record_format_info BADFILE` clause to name the file to which records are written when they cannot be loaded because of errors.

Default

Create a bad file with default name. See Purpose for details.

Syntax

```
BADFILE name | NOBADFILE
```

Purpose

The BADFILE clause names the file to which records are written when they cannot be loaded because of errors. For example, a record would be written to the bad file if a field in the data file could not be converted to the data type of a column in the external table. 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 the BADFILE clause, then you must supply either a directory object name or file name, or both. See [directory object name:] [filename].

If you specify NOBADFILE, then a bad file is not created.

If neither BADFILE nor NOBADFILE is specified, then the default is to create a bad file if at least one record is rejected. The name of the file is the table name followed by _*%p*, where *%p* is replaced with the PID of the process creating the file. The file is given an extension of .bad. If the table name contains any characters that could be interpreted as directory navigation (for example, %, /, or *), then those characters are not included in the output file name.

Records that fail the LOAD WHEN clause are not written to the bad file, but instead are written to the discard file. 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.

Related Topics

- [directory object name:] [filename]
  The [*directory object name*:] [*filename*] clause is used to specify the name of an output file (BADFILE, DISCARDFILE, or LOGFILE).

15.3.16 DISCARDFILE | NODISCARDFILE

Use the record_format_info DISCARDFILE clause to name the file to which records are written that fail the condition in the LOAD WHEN clause.

Default

Create a discard file with default name. See Purpose for details.

Syntax

```
DISCARDFILE name | NODISCARDFILE
```

Purpose

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 for discard 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, then you must supply either a directory object name or file name, or both. See [directory object name:] [filename].

If you specify NODISCARDFILE, then a discard file is not created.

If neither DISCARDFILE nor NODISCARDFILE is specified, then the default is to create a discard file if at least one record fails the LOAD WHEN clause. The name of the file is the table name

followed by _%p, where %p is replaced with the PID of the process creating the file. The file is given an extension of .dcs. If the table name contains any characters that could be interpreted as directory navigation (for example, %, /, or *), then those characters are not included in the file name.

Related Topics

- [directory object name:] [filename]
  The [*directory object name*:] [*filename*] clause is used to specify the name of an output file (BADFILE, DISCARDFILE, or LOGFILE).

15.3.17 LOGFILE | NOLOGFILE

Use the record_format_info LOGFILE clause to name the file that contains messages generated by the external tables utility while it was accessing data in the data file.

Default

Use an existing file, or create a log file with default name. See Purpose for details.

Syntax

```
LOGFILE name | NOLOGFILE
```

Purpose

The LOGFILE clause names the file that contains messages generated by the external tables utility while it was accessing data in the data file. If a log file already exists by the same name, then 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. The NOLOGFILE clause is used to prevent creation of a log file.

If you specify LOGFILE, then you must supply either a directory object name or file name, or both. See [directory object name:] [filename].

If you specify NOLOGFILE, then a log file is not created.

If neither LOGFILE nor NOLOGFILE is specified, then the default is to create a log file. The name of the file is the table name followed by _%p, where %p is replaced with the PID of the process creating the file. The file is given an extension of .log. If the table name contains any characters that could be interpreted as directory navigation (for example, %, /, or *), then those characters are not included in the file name.

Related Topics

- [directory object name:] [filename]
  The [*directory object name*:] [*filename*] clause is used to specify the name of an output file (BADFILE, DISCARDFILE, or LOGFILE).

15.3.18 SKIP

Use the record_format_info SKIP clause to skip the specified number of records in the data file before loading.

Default

None (0)

Syntax

The syntax is as follows, where *num* is the number of records to skip (Default 0).

```
SKIP = num
```

Purpose

The SKIP clause skips the specified number of records in the data file before loading. You can specify this clause only when nonparallel access is being made to the data. If there is more than one data file in the same location for the same table, then the SKIP clause causes the ORACLE_LOADER driver to skip the specified number of records in the first data file only.

15.3.19 FIELD NAMES

Use the record_format_info FIELD NAMES clause to specify field order in data files.

Default

```
NONE
```

Syntax

```
FIELD NAMES {FIRST FILE | FIRST IGNORE | ALL FILES | ALL IGNORE| NONE}
```

The FIELD NAMES options are:

- FIRST FILE — Indicates that the first data file contains a list of field names for the data in the first record. This list uses the same delimiter as the data in the data file. This record is read and used to set up the mapping between the fields in the data file and the columns in the target table. This record is skipped when the data is processed. This option can be useful if the order of the fields in the data file is different from the order of the columns in the table.
- FIRST IGNORE — Indicates that the first data file contains a list of field names for the data in the first record, but that the information should be ignored. This record is skipped when the data is processed, but is not used for setting up the fields.
- ALL FILES — Indicates that all data files contain a list of field names for the data in the first record. The ordering of the fields in the datafiles can be in any order. The order is specified by the first row in each file, which specifies to the access driver that the fields are in a different order than the columns in the external table.
- ALL IGNORE — Indicates that all data files contain a list of field names for the data in the first record, but that the information should be ignored. This record is skipped when the data is processed in every data file, but it is not used for setting up the fields.
- NONE — Indicates that the data file contains normal data in the first record. This is the default option.

Purpose

Use the FIELD NAMES clause to specify the field order of data files for the first row of the data file using one of the options. For example, if FIELD NAMES FIRST FILE is specified, then only the first data file has the row header. If FIELD NAMES ALL FILES is specified, then all data files will have the row header.

Restrictions

- The `FIELD NAMES` clause does not trim whitespace between field names in data files.

  For example, if a data file has field names `deptno, dname, loc` (with whitespace between field names) then specifying `FIELD NAMES` can fail with "KUP-04117: Field name LOC was not found in the access parameter field list or table."
- Field names in data files cannot use quotations. For example, the following column field names are not supported:

  ```
  deptno,"dname",loc
  ```
- Embedded delimiters are not supported in the first column header row.

Example

Typically fields in a data file where you want to generate a table with columns (COL1, COL2, COL3) are in the same order in the data file as they will be in the table. However, in the following example, the ordering of data file fields is diffferent in `deptxt1.dat` and `deptxt2.dat`. Specifying `FIELD NAMES ALL FILES` enables data fields in differing field name order in one or more datafiles to be queried correctly:

```
[admin@example]$ cat /tmp/deptxt1.dat
deptno,dname,loc
10,ACCOUNTING,NEW YORK
20,RESEARCH,DALLAS
30,SALES,CHICAGO
40,OPERATIONS,BOSTON

[admin@example]$ cat /tmp/deptxt2.dat
dNamE,lOc,DEPTNO
ACCOUNTING,NEW YORK,11
RESEARCH,DALLAS,21
SALES,CHICAGO,31
OPERATIONS,BOSTON,41

[admin@example]$ sql @xt

Connected.

Directory created.

SQL> create table deptXT
   2  (
   3      deptno    number(2),
   4      dname    varchar2(14),
   5      loc    varchar2(13)
   6  )
   7  organization external
   8  (
   9    type ORACLE_LOADER
  10    DEFAULT DIRECTORY DATA_DIR
  11    access parameters
  12    (
  13       records delimited by newline
  14       field names all files
```

```
  15       logfile 'deptxt.log'
  16       badfile 'deptxt.bad'
  17       fields terminated by ','
  18       missing field values are null
  19     )
  20     location ('deptxt?.dat')
  21   )
  22   reject limit unlimited
  23   ;

Table created.

SQL> Rem returns all 8 rows
SQL> select deptno, dname, loc from deptxt order by deptno;

DEPTNO     DNAME          LOC
---------- -------------- -------------
     10    ACCOUNTING     NEW YORK
     11    ACCOUNTING     NEW YORK
     20    RESEARCH       DALLAS
     21    RESEARCH       DALLAS
     30    SALES          CHICAGO
     31    SALES          CHICAGO
     40    OPERATIONS     BOSTON
     41    OPERATIONS     BOSTON
```

15.3.20 READSIZE

The `READSIZE` parameter specifies the size of the read buffer used to process records.

The size of the read buffer must be at least as big as the largest input record the access driver will encounter. 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 data file 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.

The amount of memory available for allocation is another limit because additional buffers might be allocated. The additional buffer is used to correctly complete the processing of any records that may have been split (either in the data; at the delimiter; or if multi character/byte delimiters are used, in the delimiter itself).

15.3.21 DISABLE_DIRECTORY_LINK_CHECK

The `DISABLE_DIRECTORY_LINK_CHECK` parameter directs the `ORACLE_LOADER` access driver to bypass the symbolic link check.

By default, the `ORACLE_LOADER` access driver checks before opening data and log files to ensure that the directory being used is not a symbolic link. The `DISABLE_DIRECTORY_LINK_CHECK` parameter (which takes no arguments) directs the access driver to bypass this check, allowing you to use files for which the parent directory may be a symbolic link.

> **Note:**
>
> Use of this parameter involves security risks because symbolic links can potentially be used to redirect the input/output of the external table load operation.

15.3.22 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 data type 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.

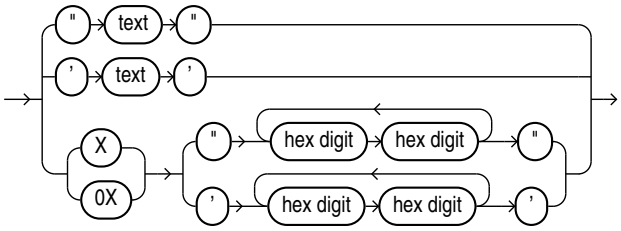
> **See Also:**
>
> Specifying a Value for the Date Cache

15.3.23 string

A string is a quoted series of characters or hexadecimal digits.

Syntax

The syntax for a `string` is as follows:



Purpose

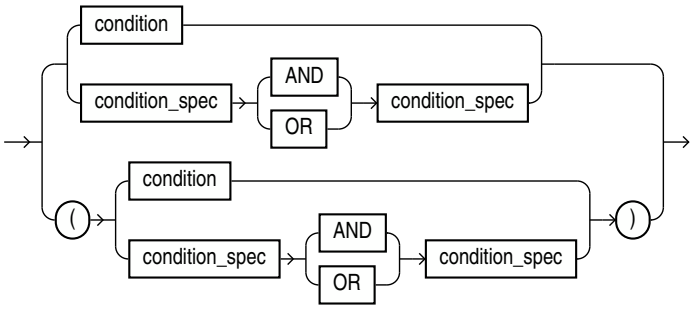
If it is a series of characters, then those characters will be converted into the character set of the data file. If it is a series of hexadecimal digits, then there must be an even number of hexadecimal digits. The hexadecimal digits are converted into their binary translation, and the translation is 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.

15.3.24 condition_spec

The `condition_spec` specifies one or more conditions that are joined by Boolean operators.

This clause is an expression that evaluates to either true or false. The conditions and Boolean operators are evaluated from left to right. (Boolean operators are applied after the conditions are evaluated.) To override the default order of evaluation of Boolean operators, you can use parentheses. 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

15.3.25 [directory object name:] [filename]

The `[directory object name:] [filename]` clause is used to specify the name of an output file (`BADFILE`, `DISCARDFILE`, or `LOGFILE`).

Syntax

```
[directory object name:] [filename]
```

- *`directory object name`*: The alias for the operating system directory on the database server for reading and writing files.

- *filename*: The name of the file that you want to create in the directory object.

To help make file names unique in parallel loads, the access driver does some symbol substitution. The symbol substitutions supported for the Linux, Unix, and Microsoft Windows operating systems are as follows (other platforms can 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 a filename specified as 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 you specify bad_data_%a.bad as the file name, then the agent creates a file named bad_data_003.bad.

- %% is replaced by %. If there is a need to have a percent sign in the file name, 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.

Purpose

Specifies the name of an output file (BADFILE, DISCARDFILE, or LOGFILE).

Usage Notes

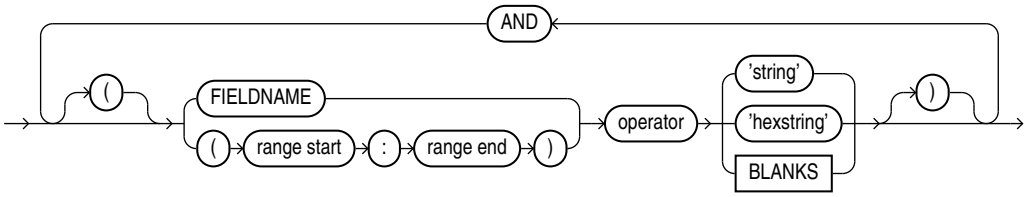
To use this clause, you must supply either a directory object name or file name, or both. The directory object name is the name of a directory object where the user accessing the external table has privileges to write. If the directory object name is omitted, then the value specified for the DEFAULT DIRECTORY clause in the CREATE TABLE...ORGANIZATION EXTERNAL statement is used.

If %p or %a is not used to create unique file names for output files, and an external table is being accessed in parallel, then it is possible that output files can be corrupted, or that agents may be unable to write to the files.

If you do not specify BADFILE (or DISCARDFILE or LOGFILE), then the SQL_LOADER 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, then a default extension is used. For bad files, the default extension is .bad; for discard files, the default is .dsc; and for log files, the default is .log.

15.3.26 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, then it will be translated into the character set of the data file. If the field has a noncharacter data type, then no data type conversion is performed on either the field value or the string. The syntax for a condition is as follows:



- range start : range end
  The `range start:range end` clause describes a range of bytes or characters in the record, which you want to use for a condition.

15.3.26.1 range start : range end

The `range start:range end` clause describes a range of bytes or characters in the record, which you want to use for a condition.

The value entered for the `STRING SIZES ARE` clause determines whether `range` refers to bytes, or to 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. The `range start:range end` clause must be enclosed in parentheses. For example: `(10:13)`.

> **Note:**
>
> The data file should not mix binary data (including data types 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.

The following are some examples of using `condition`:

```
LOAD WHEN empid != BLANKS
LOAD WHEN (10:13) = 0x'00000830'
LOAD WHEN PRODUCT_COUNT = "MISSING"
```

15.3.27 IO_OPTIONS clause

The `IO_OPTIONS` clause allows you to specify I/O options used by the operating system for reading the data files. The only options available for specification are `DIRECTIO` (the default) and `NODIRECTIO`.

The `DIRECTIO` option is used by default, so an attempt is made to open the data file and read it using direct I/O. If successful, then the operating system and NFS server (if the file is on an NFS server) do not cache the data read from the file. This can improve the read performance for the data file, especially if the file is large. If direct I/O is not supported for the data file being read, then the file is opened and read but the `DIRECTIO` option is ignored.

If the `IO_OPTIONS` clause is specified with the `NODIRECTIO` option, then direct I/O is not used to read the data files.

If the `IO_OPTIONS` clause is not specified at all, then the default `DIRECTIO` option is used.

15.3.28 DNFS_DISABLE | DNFS_ENABLE

Use these parameters to enable and disable use of the Direct NFS Client on input data files during an external tables operation.

The Direct NFS Client is an API that can be implemented by file servers to allow improved performance when Oracle accesses files on those servers.

External tables uses the Direct NFS Client interfaces by default when it reads data files over 1 gigabyte. For smaller files, the operating system's I/O interfaces are used. To use the Direct NFS Client on *all* input data files, use `DNFS_ENABLE`.

To disable use of the Direct NFS Client for all data files, specify `DNFS_DISABLE`.

See Also:

- *Oracle Grid Infrastructure Installation Guide for Linux* for information about enabling Direct NFS Client Oracle Disk Manager Control of NFS

15.3.29 DNFS_READBUFFERS

The `DNFS_READBUFFERS` parameter of the `record_format_info` clause is used to control the number of read buffers used by the Direct NFS Client.

Use `DNFS_READBUFFERS` to control the number of read buffers used by the Direct NFS Client. The Direct NFS Client is an API that can be implemented by file servers to allow improved performance when Oracle accesses files on those servers.

The default value for `DNFS_READBUFFERS` is 4.

Using larger values might compensate for inconsistent I/O from the Direct NFS Client file server, but it may result in increased memory usage.

See Also:

- *Oracle Grid Infrastructure Installation Guide for Linux* for information about enabling Direct NFS Client Oracle Disk Manager Control of NFS

15.4 field_definitions Clause

In the `field_definitions` clause, you use the `FIELDS` parameter to name the fields in the data file and specify how to find them in records.

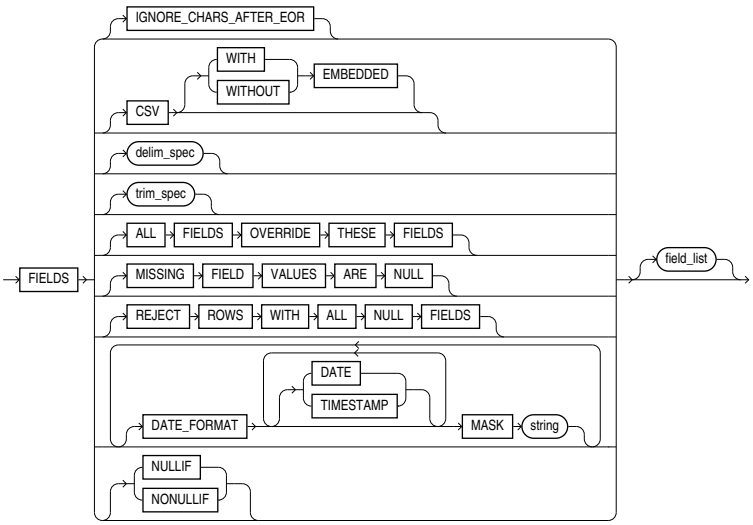
If the `field_definitions` clause is omitted, then the following is assumed:

- The fields are delimited by ','
- The fields are of data type `CHAR`
- The maximum length of the field is 255
- The order of the fields in the data file 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 data file, `info.dat`, 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:



IGNORE_CHARS_AFTER_EOR

This optional parameter specifies that if extraneous characters are found after the **last** end-of-record **but before the end of the file** that do not satisfy the record definition, they will be ignored.

Error messages are written to the external tables log file if all four of the following conditions apply:

- The `IGNORE_CHARS_AFTER_EOR` parameter is set or the field allows free formatting. (Free formatting means either that the field is variable length or the field is specified by a delimiter or enclosure characters and is also variable length).
- Characters remain after the **last** end-of-record **in the file**.
- The access parameter `MISSING FIELD VALUES ARE NULL` is not set.
- The field does not have absolute positioning.

The error messages that get written to the external tables log file are as follows:

```
KUP-04021: field formatting error for field Col1
KUP-04023: field start is after end of record
KUP-04101: record 2 rejected in file /home/oracle/datafiles/example.dat
```

CSV

To direct external tables to access the data files as comma-separated-values format files, use the `FIELDS CSV` clause. This assumes that the file is a stream record format file with the normal carriage return string (for example, `\n` on UNIX or Linux operating systems and either `\n` or `\r\n` on Windows operating systems). Record terminators can be included (embedded) in data values. The syntax for the `FIELDS CSV` clause is as follows:

```
FIELDS CSV [WITH EMBEDDED | WITHOUT EMBEDDED] [TERMINATED BY ','] [OPTIONALLY ENCLOSED
BY '""']
```

The following are key points regarding the `FIELDS CSV` clause:

- The default is to not use the `FIELDS CSV` clause.
- The `WITH EMBEDDED` and `WITHOUT EMBEDDED` options specify whether record terminators are included (embedded) in the data. The `WITH EMBEDDED` option is the default.
- If `WITH EMBEDDED` is used, then embedded record terminators must be enclosed, and intra-datafile parallelism is disabled for external table loads.
- The `TERMINATED BY ','` and `OPTIONALLY ENCLOSED BY '""'` options are the defaults and do not have to be specified. You can override them with different termination and enclosure characters.
- When the `CSV` clause is used, a delimiter specification is not allowed at the field level and only delimitable data types are allowed. Delimitable data types include `CHAR`, datetime, interval, and numeric `EXTERNAL`.
- The `TERMINATED BY` and `ENCLOSED BY` clauses cannot be used at the field level when the CSV clause is specified.
- When the `CSV` clause is specified, the default trimming behavior is `LDRTRIM`. You can override this by specifying one of the other external table trim options (`NOTRIM`, `LRTRIM`, `LTRIM`, or `RTRIM`).
- The `CSV` clause must be specified after the `IGNORE_CHARS_AFTER_EOR` clause and before the `delim_spec` clause.

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.

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.

ALL FIELDS OVERRIDE

The `ALL FIELDS OVERRIDE` clause tells the access driver that all fields are present and that they are in the same order as the columns in the external table. You only need to specify fields

that have a special definition. This clause must be specified after the optional `trim_spec` clause and before the optional `MISSING FIELD VALUES ARE NULL` clause.

The following is a sample use of thee `ALL FIELDS OVERRIDE` clause. The only field that had to be specified was the hiredate, which required a data format mask. All the other fields took default values.

```
FIELDS TERMINATED BY "," OPTIONALLY ENCLOSED BY '"' LDRTRIM
ALL FIELDS OVERRIDE
REJECT ROWS WITH ALL NULL FIELDS
(
 HIREDATE CHAR(20) DATE_FORMAT DATE MASK "DD-Month-YYYY"
)
```

MISSING FIELD VALUES ARE NULL

`MISSING FIELD VALUES ARE NULL` sets to null any fields for which position is not explicitly stated and there is not enough data to fill them. For a full description see MISSING FIELD VALUES ARE NULL.

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, then 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."

DATE_FORMAT

The `DATE_FORMAT` clause allows you to specify a datetime format mask once at the fields level, and have it apply to all fields of that type which do not have their own mask specified. The datetime format mask must be specified after the optional `REJECT ROWS WITH ALL NULL FIELDS` clause and before the `fields_list` clause.

The `DATE_FORMAT` can be specified for the following datetime types: `DATE`, `TIME`, `TIME WITH TIME ZONE`, `TIMESTAMP`, and `TIMESTAMP WITH TIME ZONE`.

The following example shows a sample use of the `DATE_FORMAT` clause that applies a date mask of `DD-Month-YYYY` to any `DATE` type fields:

```
FIELDS TERMINATED BY "," OPTIONALLY ENCLOSED BY '"' LDRTRIM
REJECT ROWS WITH ALL NULL FIELDS
DATE_FORMAT DATE MASK "DD-Month-YYYY"

    (
       EMPNO,
       ENAME,
       JOB,
       MGR,
       HIREDATE CHAR(20),
       SAL,
       COMM,
       DEPTNO,
       PROJNO,
       ENTRYDATE CHAR(20)
    )
```

NULLIF | NO NULLIF

The NULLIF clause applies to all character fields (for example, CHAR, VARCHAR, VARCHARC, external NUMBER, and datetime).

The syntax is as follows:

```
NULLIF {=|!=}{"char_string"|x'hex_string'|BLANKS}
```

If there is a match using the equal or not equal specification for a field, then the field is set to NULL for that row.

The char_string and hex_string must be enclosed in single or double quotation marks.

If a NULLIF specification is specified at the field level, it overrides this NULLIF clause.

If there is a field to which you do not want the NULLIF clause to apply, you can specify NO NULLIF at the field level (as shown in the following example).

The NULLIF clause must be specified after the optional REJECT ROWS WITH ALL NULL FIELDS clause and before the fields_list clause.

The following is an example of using the NULLIF clause. The MGR field is set to NO NULLIF which means that the NULLIF="NONE" clause will not apply to that field.

```
FIELDS TERMINATED BY "," OPTIONALLY ENCLOSED BY '"' LDRTRIM
REJECT ROWS WITH ALL NULL FIELDS
NULLIF = "NONE"
(
  EMPNO,
  ENAME,
  JOB,
  MGR
 )
```

field_list Clause

The field_list clause identifies the fields in the data file and their data types. For a full description of the syntax, see field_list.

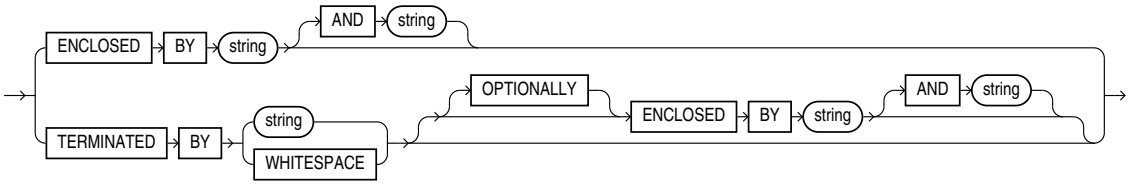
- delim_spec
  The delim_spec clause is used to find the end (and if ENCLOSED BY is specified, the start) of a field.
- trim_spec
  The trim_spec clause is used to specify that spaces should be trimmed from the beginning of a text field, the end of a text field, or both.
- MISSING FIELD VALUES ARE NULL
  The effect of MISSING FIELD VALUES ARE NULL depends on whether POSITION is used to explicitly state field positions.
- field_list
  The field_definitions field_list clause identifies the fields in the data file and their data types.
- pos_spec Clause
  The ORACLE_LOADER pos_spec clause indicates the position of the column within the record.

- datatype_spec Clause
  The ORACLE_LOADER datatype_spec clause describes the data type of a field in the data file if the data type is different than the default.
- init_spec Clause
  The init_spec clause for external tables is used to specify when a field should be set to NULL, or when it should be set to a default value.
- LLS Clause

15.4.1 delim_spec

The delim_spec clause is used to find the end (and if ENCLOSED BY is specified, the start) of a field.

Syntax



Usage Notes

If you specify ENCLOSED BY, then the ORACLE_LOADER 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 non-blank character. After the first non-blank 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. 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 non-blank character, then the access driver looks for the terminating delimiter. In this case, leading blanks are trimmed.

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.

To find out more about the access driver's default trimming behavior, refer to "Trimming Whitespace." You can override this behavior by using with `LTRIM` and `RTRIM`.

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, then it could contain a single quotation mark by specifying two single quotation marks in a row, as shown in the word don't in the following example:

```
'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
  See how to create an external table that uses terminating delimiters, and a data file with terminating delimiters.
- Example: External Table with Enclosure and Terminator Delimiters
  See how to create an external table that uses both enclosure and terminator delimiters.
- Example: External Table with Optional Enclosure Delimiters
  See how to create an external table that uses optional enclosure delimiters.

Related Topics

- Trimming Whitespace
  Blanks, tabs, and other nonprinting characters (such as carriage returns and line feeds) constitute whitespace.

15.4.1.1 Example: External Table with Terminating Delimiters

See how to create an external table that uses terminating delimiters, and a data file with terminating delimiters.

This table is created to use terminating delimiters. It is followed by an example of a data file that can be used to load the table.

```
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
```

15.4.1.2 Example: External Table with Enclosure and Terminator Delimiters

See how to create an external table that uses both 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 data file 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)
```

15.4.1.3 Example: External Table with Optional Enclosure Delimiters

See how to create an external table that uses optional enclosure delimiters.

This table is an external table that is created to use optional enclosure delimiters. Note that `LRTRIM` is used to trim leading and trailing blanks from fields. The example is followed by an examplle of a data file that can be used to load the table.

```
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)
```

15.4.2 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.

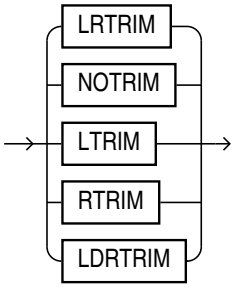
Description

Directs the `ORACLE_LOADER` access driver to trim spaces from the beginning of a text field, the end of a text field, or both. Spaces include blanks and other non-printing characters, such as tabs, line feeds, and carriage returns.

Default

The default is `LDRTRIM`. Specifying `NOTRIM` yields the fastest performance.

Syntax



Options

- `NOTRIM` Indicates that you want no characters trimmed from the field.
- `LRTRIM` Indicates that you want both leading and trailing spaces trimmed.
- `LTRIM` Indicates that you want leading spaces trimmed.
- `RTRIM` Indicates that you want trailing spaces trimmed.
- `LDRTRIM` Provides 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 are trimmed from the left.

Usage Notes

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 data file 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
```

15.4.3 MISSING FIELD VALUES ARE NULL

The effect of `MISSING FIELD VALUES ARE NULL` depends on whether `POSITION` is used to explicitly state field positions.

For example:

- The default behavior is that if field position is not explicitly stated and there is not enough data in a record for all fields, then the record is rejected. You can override this behavior by using `MISSING FIELD VALUES ARE NULL` to define as NULL any fields for which there is no data available.
- If field position is explicitly stated, then fields for which there are no values are always defined as NULL, regardless of whether `MISSING FIELD VALUES ARE NULL` is used.

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 data file. If the `MISSING FIELD VALUES ARE NULL` clause were 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 data file that can be used to load it.

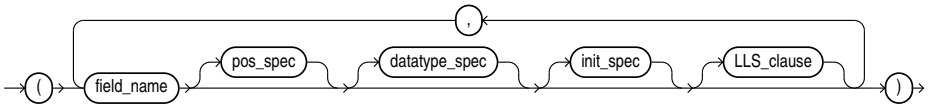
```
CREATE TABLE emp_load (first_name CHAR(15), last_name CHAR(20), year_of_birth INT)
  ORGANIZATION EXTERNAL (TYPE ORACLE_LOADER DEFAULT DIRECTORY ext_tab_dir
                         ACCESS PARAMETERS (FIELDS TERMINATED BY ","
                                            MISSING FIELD VALUES ARE NULL)
                         LOCATION ('info.dat'));

Alvin,Tolliver,1976
Baer,Kenneth
Mary,Dube,1973
```

15.4.4 field_list

The `field_definitions field_list` clause identifies the fields in the data file and their data types.

Syntax

The syntax for the `field_list` clause is as follows:



The `field list` clauses are as follows:

- **field_name**: A string identifying the name of a field in the data file. If the string is not within quotation marks, then 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**: Indicates the position of the column within the record. For a full description of the syntax, see pos_spec Clause.
- **datatype_spec**: Indicates the data type of the field. If `datatype_spec` is omitted, then the access driver assumes the data type is `CHAR(255)`. For a full description of the syntax, see datatype_spec Clause.
- **init_spec**: Indicates when a field is `NULL` or has a default value. For a full description of the syntax, see init_spec Clause.
- **LLS**: When `LLS` is specified for a field, `ORACLE_LOADER` does not load the value of the field into the corresponding column. Instead, it use the information in the value to determine where to find the value of the field. See LLS Clause.

Purpose

The `field_list` clause identifies the fields in the data file and their data types. Evaluation criteria for the `field_list` clause are as follows:

- If no data type is specified for a field, then 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 data file are assumed to be in the same order as the fields in the external table. The data type 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 data type 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 data file 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.

Example

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 data file 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
```

15.4.5 pos_spec Clause

The `ORACLE_LOADER` `pos_spec` clause indicates the position of the column within the record.

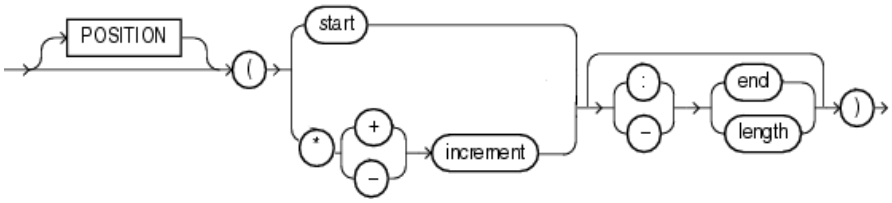
The setting of the `STRING SIZES ARE IN` clause determines whether `pos_spec` refers to byte positions or character positions. Using character positions with varying-width character sets takes significantly longer than using character positions with fixed-width character sets. Binary and multibyte character data should not be present in the same data file when `pos_spec` is used for character positions. If they are, then the results are unpredictable.

- pos_spec Clause Syntax
  The syntax for the `ORACLE_LOADER` `pos_spec` clause is as follows.

- start
  The `pos_spec` clause `start` parameter indicates the number of bytes from the beginning of the record to where the field begins.
- *
  The `pos_spec` clause `*` parameter indicates that the field begins at the first byte after the end of the previous field.
- increment
  The `pos_spec` clause `increment` parameter positions the start of the field is a fixed number of bytes from the end of the previous field.
- end
  The `pos_spec` clause `end` parameter indicates the absolute byte offset into the record for the last byte of the field.
- length
  The `pos_spec` clause `length` parameter value indicates that the end of the field is a fixed number of bytes from the start.

15.4.5.1 pos_spec Clause Syntax

The syntax for the `ORACLE_LOADER` `pos_spec` clause is as follows.



15.4.5.2 start

The `pos_spec` clause `start` parameter indicates the number of bytes from the beginning of the record to where the field begins.

The `start` parameter enables you to position the start of the field at an absolute spot in the record, rather than relative to the position of the previous field.

15.4.5.3 *

The `pos_spec` clause `*` parameter indicates that the field begins at the first byte after the end of the previous field.

The `*` parameter is useful with `ORACLE_LOADER` where you have a varying-length field followed by a fixed-length field. This option cannot be used for the first field in the record.

15.4.5.4 increment

The `pos_spec` clause `increment` parameter positions the start of the field is a fixed number of bytes from the end of the previous field.

The `increment` parameter positions the start of the field at a fixed number of bytes 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). To move the start after the current position, use `*+`*`increment`*

15.4.5.5 end

The `pos_spec` clause `end` parameter indicates the absolute byte offset into the record for the last byte of the field.

Use the `end` parameter to set the absolute byte 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.

15.4.5.6 length

The `pos_spec` clause `length` parameter value indicates that the end of the field is a fixed number of bytes from the start.

Use the `length` parameter when you want to set fixed-length fields when the start is specified with `*`. The following example shows various ways of using `pos_spec`. It is followed by an example of a data file that you can use 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
     (RECORDS CHARACTERSET we8iso8859p1
      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
```

In this example, the declared `RECORDS CHARACTERSET`, `we8iso8859p1`, is not a multi-byte character set. It is guaranteed that every character is represented as single byte. The `POSITION` clause calculations to determine where the data field starts and ends (including the `*` and `+` operators) are based on bytes rather than characters (that is, characters must only require 1 byte to represent them, such as the Oracle character set `WE8ISO8859P1`). If you use a variable length character set (for example, Unicode variants, `JIS X 0208-1990`, or other multibyte character sets, where the field data contains one or more multibyte characters), then the calculations will be incorrect.

15.4.6 datatype_spec Clause

The ORACLE_LOADER datatype_spec clause describes the data type of a field in the data file if the data type is different than the default.

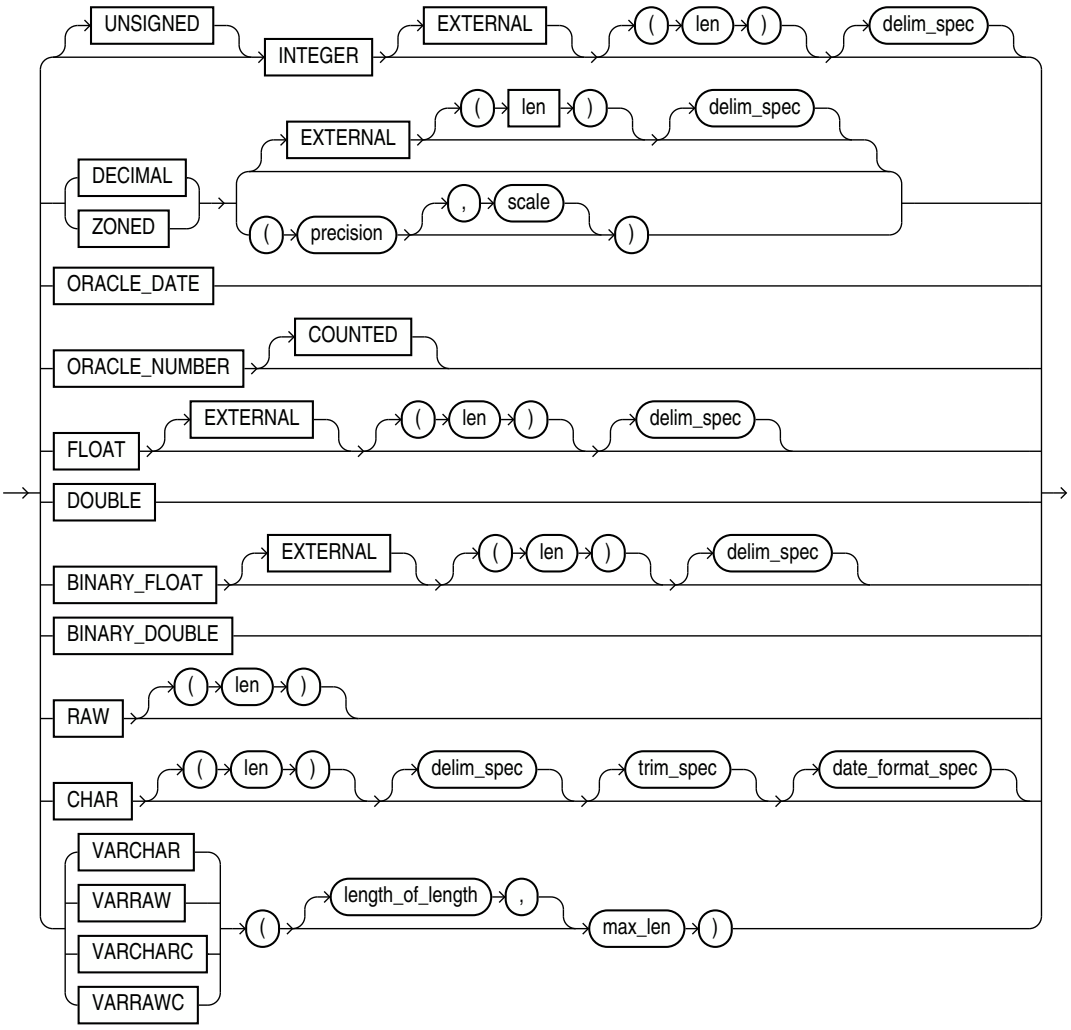
The data type of the field can be different than the data type of a corresponding column in the external table. The access driver handles the necessary conversions.

- datatype_spec Clause Syntax
  The syntax for the ORACLE_LOADER datatype_spec clause is as follows:
- [UNSIGNED] INTEGER [EXTERNAL] [(len)]
  The datatype_spec clause [UNSIGNED] INTEGER [EXTERNAL] [(len)] defines a field as an integer.
- 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.
- ORACLE_DATE
  ORACLE_DATE is a field containing a date in the Oracle binary date format.
- ORACLE_NUMBER
  ORACLE_NUMBER is a field containing a number in the Oracle number format.
- Floating-Point Numbers
  The following four data types, DOUBLE, FLOAT, BINARY_DOUBLE, and BINARY_FLOAT are floating-point numbers.
- DOUBLE
  The DOUBLE clause indicates that the field is the same format as the C language DOUBLE data type on the platform where the access driver is executing.
- FLOAT [EXTERNAL]
  The FLOAT clause indicates that the field is the same format as the C language FLOAT data type on the platform where the access driver is executing.
- BINARY_DOUBLE
  The datatype_spec clause value BINARY_DOUBLE is a 64-bit, double-precision, floating-point number data type.
- BINARY_FLOAT
  The datatype_spec clause value BINARY_FLOAT is a 32-bit, single-precision, floating-point number data type.
- RAW
  The RAW clause is used to indicate that the source data is binary data.
- CHAR
  The datatype_spec clause data type CHAR clause is used to indicate that a field is a character data type.
- 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.
- VARCHAR and VARRAW
  The datatype_spec clause VARCHAR data type defines character data, and the VARRAW data type defines binary data.

- VARCHARC and VARRAWC
  The datatype_spec clause VARCHARC data type defines character data, and the VARRAWC data type defines binary data.

15.4.6.1 datatype_spec Clause Syntax

The syntax for the `ORACLE_LOADER 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 data file.

Related Topics

- init_spec Clause
  The `init_spec` clause for external tables is used to specify when a field should be set to `NULL`, or when it should be set to a default value.
- *Oracle Database SQL Language Reference*

15.4.6.2 [UNSIGNED] INTEGER [EXTERNAL] [(len)]

The `datatype_spec` clause `[UNSIGNED] INTEGER [EXTERNAL] [(len)]` defines a field as an integer.

This clause defines a field as an integer. If `EXTERNAL` is specified, then the number is a character string. If `EXTERNAL` is not specified, then 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, then 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, then the default value is 255.

The default value of the `[UNSIGNED] INTEGER [EXTERNAL] [(len)]` data type is determined as follows:

- If no length specified, then the default length is 1.
- If no length is specified and the field is delimited with a `DELIMITED BY NEWLINE` clause, then the default length is 1.
- If no length is specified and the field is delimited with a `DELIMITED BY` clause, then the default length is 255 (unless the delimiter is `NEWLINE`, as stated above).

15.4.6.3 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, then 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.

15.4.6.4 ORACLE_DATE

`ORACLE_DATE` is a field containing a date in the Oracle binary date format.

This is the format used by the `DTYDAT` data type in Oracle Call Interface (OCI) programs. The field is a fixed length of 7.

15.4.6.5 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.

15.4.6.6 Floating-Point Numbers

The following four data types, `DOUBLE`, `FLOAT`, `BINARY_DOUBLE`, and `BINARY_FLOAT` are floating-point numbers.

The following four data types, `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 data types used by default for the `DOUBLE` and `FLOAT` data types 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 Language Reference* for more information about floating-point numbers

15.4.6.7 DOUBLE

The `DOUBLE` clause indicates that the field is the same format as the C language `DOUBLE` data type 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 data type may not be portable between certain platforms.

15.4.6.8 FLOAT [EXTERNAL]

The `FLOAT` clause indicates that the field is the same format as the C language `FLOAT` data type on the platform where the access driver is executing.

The `FLOAT` clause indicates that the field is the same format as the C language `FLOAT` data type 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 data type 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.

15.4.6.9 BINARY_DOUBLE

The `datatype_spec` clause value `BINARY_DOUBLE` is a 64-bit, double-precision, floating-point number data type.

Each `BINARY_DOUBLE` value requires 9 bytes, including a length byte. See the information in the note provided for the `FLOAT` data type for more details about floating-point numbers.

15.4.6.10 BINARY_FLOAT

The `datatype_spec` clause value `BINARY_FLOAT` is a 32-bit, single-precision, floating-point number data type.

Each `BINARY_FLOAT` value requires 5 bytes, including a length byte. See the information in the note provided for the `FLOAT` data type for more details about floating-point numbers.

15.4.6.11 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.

15.4.6.12 CHAR

The datatype_spec clause data type `CHAR` clause is used to indicate that a field is a character data type.

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 data type `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, then 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 receive an error stating that the field in the data file exceeds maximum length.

The following example shows the use of the `CHAR` clause.

```
SQL> CREATE TABLE emp_load
  2    (employee_number      CHAR(5),
  3     employee_dob         CHAR(20),
  4     employee_last_name   CHAR(20),
  5     employee_first_name  CHAR(15),
  6     employee_middle_name CHAR(15),
  7     employee_hire_date   DATE)
  8  ORGANIZATION EXTERNAL
  9    (TYPE ORACLE_LOADER
 10     DEFAULT DIRECTORY def_dir1
```

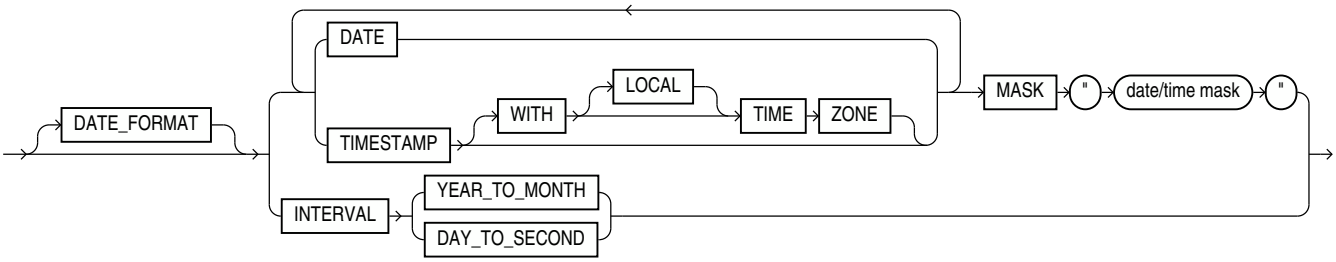
```
 11     ACCESS PARAMETERS
 12       (RECORDS DELIMITED BY NEWLINE
 13        FIELDS (employee_number      CHAR(2),
 14                employee_dob         CHAR(20),
 15                employee_last_name   CHAR(18),
 16                employee_first_name  CHAR(11),
 17                employee_middle_name CHAR(11),
 18                employee_hire_date   CHAR(10) date_format DATE mask "mm/dd/
yyyy"
 19               )
 20       )
 21     LOCATION ('info.dat')
 22    );

Table created.
```

15.4.6.13 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 data type and only when a character string field is mapped into a date column.

The syntax for the `date_format_spec` clause is as follows:



For detailed information about the correct way to specify date and time formats, see *Oracle Database SQL Reference*.

- 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 data type.
- TIMESTAMP
  The `TIMESTAMP` clause indicates that a field contains a formatted timestamp.
- INTERVAL
  The `INTERVAL` clause indicates that a field contains a formatted interval.

Related Topics

- *Oracle Database SQL Language Reference*

15.4.6.13.1 DATE

The `DATE` clause indicates that the string contains a date.

15.4.6.13.2 MASK

The `MASK` clause is used to override the default globalization format mask for the data type.

If a date mask is not specified, then the settings of NLS parameters for the database (not the session settings) for the appropriate globalization parameter for the data type are used. The `NLS_DATABASE_PARAMETERS` view shows these settings.

- `NLS_DATE_FORMAT` for `DATE` data types
- `NLS_TIMESTAMP_FORMAT` for `TIMESTAMP` data types
- `NLS_TIMESTAMP_TZ_FORMAT` for `TIMESTAMP WITH TIME ZONE` data types

Note the following:

- The database setting for the `NLS_NUMERIC_CHARACTERS` initialization parameter (that is, from the `NLS_DATABASE_PARAMETERS` view) governs the decimal separator for implicit conversion from character to numeric data types.
- A group separator is not allowed in the default format.

15.4.6.13.3 TIMESTAMP

The `TIMESTAMP` clause indicates that a field contains a formatted timestamp.

15.4.6.13.4 INTERVAL

The `INTERVAL` clause indicates that a field contains a formatted 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`.

The following example shows a sample use of a complex `DATE` character string and a `TIMESTAMP` character string. It is followed by a sample of the data file that can be used to load it.

```
SQL> CREATE TABLE emp_load
  2    (employee_number      CHAR(5),
  3     employee_dob         CHAR(20),
  4     employee_last_name   CHAR(20),
  5     employee_first_name  CHAR(15),
  6     employee_middle_name CHAR(15),
  7     employee_hire_date   DATE,
  8     rec_creation_date    TIMESTAMP WITH TIME ZONE)
  9  ORGANIZATION EXTERNAL
 10    (TYPE ORACLE_LOADER
 11     DEFAULT DIRECTORY def_dir1
 12     ACCESS PARAMETERS
 13       (RECORDS DELIMITED BY NEWLINE
 14        FIELDS (employee_number      CHAR(2),
 15                employee_dob         CHAR(20),
 16                employee_last_name   CHAR(18),
 17                employee_first_name  CHAR(11),
 18                employee_middle_name CHAR(11),
 19                employee_hire_date   CHAR(22) date_format DATE mask "mm/dd/yyyy hh:mi:ss
AM",
```

```
 20                rec_creation_date    CHAR(35) date_format TIMESTAMP WITH TIME ZONE mask
"DD-MON-RR HH.MI.SSXFF AM TZH:TZM"
 21               )
 22       )
 23     LOCATION ('infoc.dat')
 24    );

Table created.


SQL> SELECT * FROM emp_load;

EMPLO EMPLOYEE_DOB         EMPLOYEE_LAST_NAME   EMPLOYEE_FIRST_ EMPLOYEE_MIDDLE
----- -------------------- -------------------- --------------- ---------------
EMPLOYEE_
---------
REC_CREATION_DATE
---------------------------------------------------------------------------
56    november, 15, 1980   baker                mary            alice
01-SEP-04
01-DEC-04 11.22.03.034567 AM -08:00

87    december, 20, 1970   roper                lisa            marie
01-JAN-02
01-DEC-02 02.03.00.678573 AM -08:00


2 rows selected.
```

The `info.dat` file looks like the following. Note that this is 2 long records. There is one space between the data fields (`09/01/2004`, `01/01/2002`) and the time field that follows.

```
56november, 15, 1980  baker            mary       alice      09/01/2004 08:23:01 AM01-
DEC-04 11.22.03.034567 AM -08:00
87december, 20, 1970  roper            lisa       marie      01/01/2002 02:44:55 PM01-
DEC-02 02.03.00.678573 AM -08:00
```

15.4.6.14 VARCHAR and VARRAW

The `datatype_spec` clause `VARCHAR` data type defines character data, and the `VARRAW` data type defines binary data.

The `VARCHAR` data type 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` for information about how to specify whether the count is interpreted as a count of characters or count of bytes.

The `VARRAW` data type 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 `VARIABLE 2` clause in the `ACCESS PARAMETERS` clause specifies the size of the binary field that contains the length.

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, then 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 data file. 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 content of the data file, info.dat, is shown following the example.

```
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
     (RECORDS
        VARIABLE 2
        DATA IS BIG ENDIAN
        CHARACTERSET US7ASCII
      FIELDS (first_name VARCHAR(2,12),
              last_name VARCHAR(2,20),
              resume VARCHAR(4,10000),
              picture VARRAW(4,100000)))
    LOCATION ('info.dat'));
```

Contents of info.dat Data File

The contents of the data file used in the example are as follows:.

```
0005Alvin0008Tolliver0000001DAlvin Tolliver's Resume etc.
0000001013f4690a30bc29d7e40023ab4599ffff
```

It is important to understand that, for the purposes of readable documentation, the binary values for the count bytes and the values for the raw data are shown in the data file in italics, with 2 characters per binary byte. The values in an actual data file would be in binary format, not ASCII. Therefore, if you attempt to use this example by cutting and pasting, then you will receive an error.

Related Topics

- STRING SIZES ARE IN
  Use the record_format_info STRING SIZES ARE IN clause to indicate whether the lengths specified for character strings are in bytes or characters.

15.4.6.15 VARCHARC and VARRAWC

The datatype_spec clause VARCHARC data type defines character data, and the VARRAWC data type defines binary data.

The `VARCHARC` data type 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` for information about how to specify whether the count is interpreted as a count of characters, or acount 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` data type 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 data file. 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
```

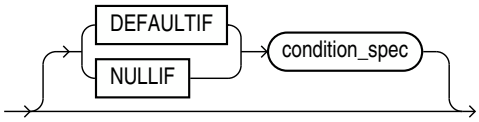
Related Topics

- STRING SIZES ARE IN
  Use the `record_format_info STRING SIZES ARE IN` clause to indicate whether the lengths specified for character strings are in bytes or characters.

15.4.7 init_spec Clause

The `init_spec` clause for external tables 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:

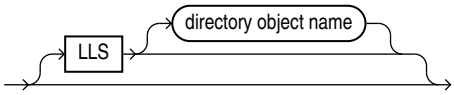


Only one NULLIF clause and only one DEFAULTIF clause can be specified for any field. These clauses behave as follows:

- If NULLIF *condition_spec* is specified and it evaluates to TRUE, then the field is set to NULL.
- If DEFAULTIF *condition_spec* is specified and it evaluates to TRUE, then the value of the field is set to a default value. The default value depends on the data type of the field, as follows:
  - For a character data type, the default value is an empty string.
  - For a numeric data type, the default value is a 0.
  - For a date data type, 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.

15.4.8 LLS Clause

If a field in a data file is a LOB location Specifier (LLS) field, then you can indicate this by using the LLS clause. An LLS field contains the file name, offset, and length of the LOB data in the data file. SQL*Loader uses this information to read data for the LOB column. The LLS clause for ORACLE_LOADER has the following syntax:



When the LLS clause is used, ORACLE_LOADER does not load the value of the field into the corresponding column. Instead, it uses the information in the value to determine where to find the value of the field. The LOB can be loaded in part or in whole and it can start from an arbitrary position and for an arbitrary length. ORACLE_LOADER expects the contents of the field to be *filename.ext.nnn.mmm/* where each element is defined as follows:

- *filename.ext* is the name of the file that contains the LOB
- *nnn* is the offset in bytes of the LOB within the file
- *mmm* is the length of the LOB in bytes A value of -1 means the LOB is NULL. A value of 0 means the lob exists, but is empty.
- The forward slash (/) terminates the field

The LLS clause has an optional DIRECTORY clause which specifies an Oracle directory object:

- If DIRECTORY is specified, then the file must exist there and you must have READ access to that directory object.
- If DIRECTORY is not specified, then the file must exist in the same directory as the data file.

An error is returned and the row rejected if any of the following are true:

- The file name contains a relative or absolute path specification.
- The file is not found, the offset is invalid, or the length extends beyond the end of the file.
- The contents of the field do not match the expected format.
- The data type for the column associated with an `LLS` field is not a `CLOB`, `BLOB` or `NCLOB`.

If an `LLS` field is referenced by a clause for any other field (for example a `NULLIF` clause), then in the access parameters, the value used for evaluating the clause is the string in the data file, not the data in the file pointed to by that string.

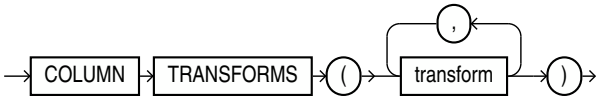
The character set for the data in the file pointed to by the `LLS` clause is assumed to be the same character set as the data file.

15.5 column_transforms Clause

The optional `ORACLE_LOADER` access drive `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 data file.

Syntax

The syntax for the `column_transforms` clause is as follows:



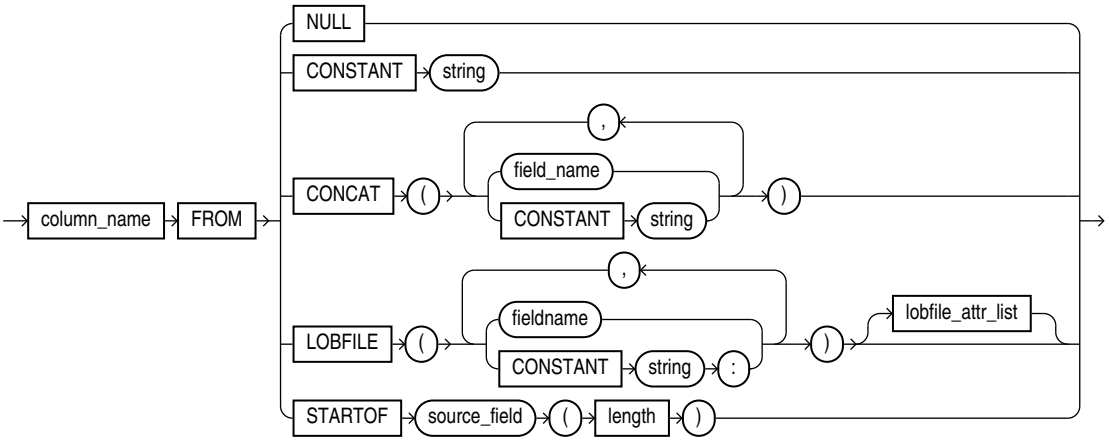
> **Note:**
>
> The `COLUMN TRANSFORMS` clause does not work in conjunction with the `PREPROCESSOR` clause.

- 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.

15.5.1 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 data file. The `LOBFILE` transform is used to load data into a field for a record from another data file. Each of these transforms is explained further in the following sections.

- column_name FROM
  The `column_name` uniquely identifies a column in the external table that you want to be loaded.
- NULL
  When the `NULL` transform is specified, every value of the field is set to `NULL` for every record.
- CONSTANT
  The `CONSTANT` clause transform uses the value of the string specified as the value of the column in the record.
- CONCAT
  The `CONCAT` transform concatenates constant strings and fields in the data file together to form one string.
- 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.
- lobfile_attr_list
  The `lobfile_attr_list` lists additional attributes of the LOBFILE.
- STARTOF source_field (length)
  The `STARTOF` keyword allows you to create an external table in which a column can be a substring of the data in the source field.

15.5.1.1 column_name FROM

The `column_name` uniquely identifies a column in the external table that you want 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 data file.

15.5.1.2 NULL

When the `NULL` transform is specified, every value of the field is set to `NULL` for every record.

15.5.1.3 CONSTANT

The `CONSTANT` clause 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 data type of the column. This conversion will be done for every row.

The character set of the string used for data type conversions is the character set of the database.

15.5.1.4 CONCAT

The `CONCAT` transform concatenates constant strings and fields in the data file together to form one string.

Only fields that are character data types 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.

15.5.1.5 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 file name in the form *`directory object:filename`*. The following rules apply to use of the `LOBFILE` transform:

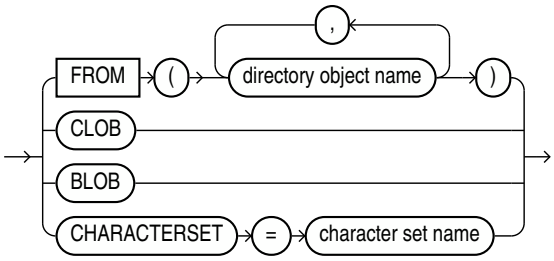
- Both the directory object and the file name can be either a constant string or the name of a field in the field clause.
- If a constant string is specified, then that string is used to find the `LOBFILE` for every row in the table.
- If a field name is specified, then the value of that field in the data file is used to find the `LOBFILE`.
- If a field name is specified for either the directory object or the file name 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, then 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.

15.5.1.6 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 data file is assumed.

15.5.1.7 STARTOF source_field (length)

The STARTOF keyword allows you to create an external table in which a column can be a substring of the data in the source field.

The length is the length of the substring, beginning with the first byte. It is assumed that length refers to a byte count and that the external table column(s) being transformed use byte length and not character length semantics. (Character length semantics might give unexpected results.)

Only complete character encodings are moved; characters are never split. So if a substring ends in the middle of a multibyte character, then the resulting string will be shortened. For example, if a length of 10 is specified, but the 10th byte is the first byte of a multibyte character, then only the first 9 bytes are returned.

The following example shows how you could use the STARTOF keyword if you only wanted the first 4 bytes of the department name (dname) field:

```
SQL> CREATE TABLE dept (deptno  NUMBER(2),
  2                     dname   VARCHAR2(14),
  3                     loc     VARCHAR2(13)
  4                          )
  5  ORGANIZATION EXTERNAL
  6  (
  7    DEFAULT DIRECTORY def_dir1
  8    ACCESS PARAMETERS
  9    (
 10      RECORDS DELIMITED BY NEWLINE
 11      FIELDS TERMINATED BY ','
```

```
 12      (
 13        deptno           CHAR(2),
 14        dname_source     CHAR(14),
 15        loc              CHAR(13)
 16      )
 17      column transforms
 18      (
 19         dname FROM STARTOF dname_source (4)
 20      )
 21    )
 22    LOCATION ('dept.dat')
 23  );

Table created.
```

If you now perform a `SELECT` operation from the `dept` table, only the first four bytes of the `dname` field are returned:

```
SQL> SELECT * FROM dept;

    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCO           NEW YORK
        20 RESE           DALLAS
        30 SALE           CHICAGO
        40 OPER           BOSTON

4 rows selected.
```

15.6 Parallel Loading Considerations for the ORACLE_LOADER Access Driver

The `ORACLE_LOADER` access driver attempts to divide large data files 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 data file 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.

15.7 Performance Hints When Using the ORACLE_LOADER Access Driver

This topic describes some performance hints when using the `ORACLE_LOADER` access driver.

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 data files. 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 (you can use the `READSIZE` clause in the access parameters to specify the size of the buffers). 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.

Performance can also sometimes be increased with use of date cache functionality. By using the date cache to specify the number of unique dates anticipated during the load, you can reduce the number of date conversions done when many duplicate date or timestamp values are present in the input data. The date cache functionality provided by external tables is identical to the date cache functionality provided by SQL*Loader. See DATE_CACHE 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 data file match the character set of the database is faster than a character set conversion.
- Having data types in the data file match the data types in the database is faster than data type 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.

15.8 Restrictions When Using the ORACLE_LOADER Access Driver

This section lists restrictions to be aware of when you use the `ORACLE_LOADER` access driver.

Specifically:

- Exporting and importing of external tables with encrypted columns is not supported.

- 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 data type 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.
- SQL strings cannot be specified in access parameters for the `ORACLE_LOADER` access driver. As a workaround, you can use the `DECODE` clause in the `SELECT` clause of the statement that is reading the external table. Alternatively, you can create a view of the external table that uses the `DECODE` clause and select from that view rather than the external table.
- The use of the backslash character (\) within strings is not supported in external tables. See Use of the Backslash Escape Character.
- 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, then it must be enclosed in double quotation marks.

15.9 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, then 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
- DNFS_ENABLE
- DNFS_DISABLE
- DNFS_READBUFFERS
- 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

16

The ORACLE_DATAPUMP Access Driver

The `ORACLE_DATAPUMP` access driver provides a set of access parameters that are unique to external tables of the type `ORACLE_DATAPUMP`.

- Using the ORACLE_DATAPUMP Access Driver
  To modify the default behavior of the access driver, use ORACLE_DATAPUMP access parameters . The information that 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.
- access_parameters Clause
  When you create the `ORACLE_DATAPUMP` access driver external table, you can specify certain parameters in an `access_parameters` clause.
- Unloading and Loading Data with the ORACLE_DATAPUMP Access Driver
  As part of creating an external table with a SQL `CREATE TABLE AS SELECT` statement, the `ORACLE_DATAPUMP` access driver can write data to a dump file.
- Supported Data Types
  The `ORACLE_DATAPUMP` access driver resolves many data types automatically during loads and unloads.
- Unsupported Data Types
  You can use the `ORACLE_DATAPUMP` access driver to unload and reload data for some of the unsupported data types
- Performance Hints When Using the ORACLE_DATAPUMP Access Driver
  Learn how to improve `ORACLE_DATAPUMP` access driver performance.
- Restrictions When Using the ORACLE_DATAPUMP Access Driver
  Be aware of restrictions that apply to accessing external tables with the `ORACLE_DATAPUMP` access driver.
- Reserved Words for the ORACLE_DATAPUMP Access Driver
  If you use words in identifiers that are reserved by the `ORACLE_DATAPUMP` access driver, then they must be enclosed in double quotation marks.

16.1 Using the ORACLE_DATAPUMP Access Driver

To modify the default behavior of the access driver, use ORACLE_DATAPUMP access parameters . The information that 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.

To use the ORACLE_DATAPUMP access driver successfully, you must know a little about the file format and record format (including character sets and field data types) of the data files on your platform. You must also be able to use SQL to create an external table, and to perform queries against the table you create.

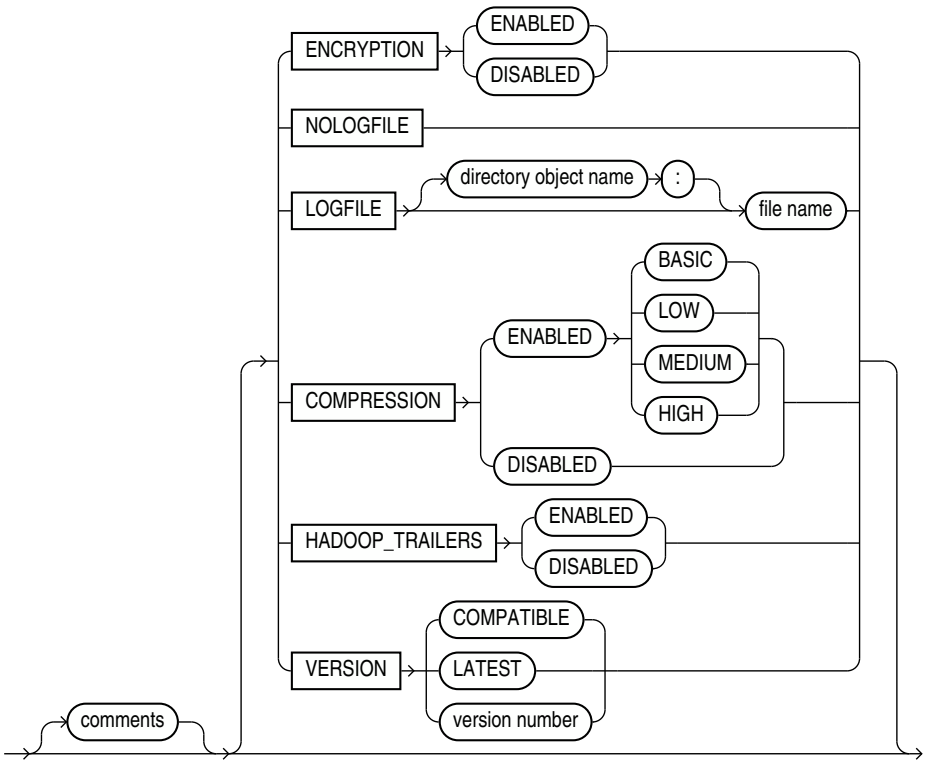
> **Note:**
>
> - It is sometimes difficult to describe syntax without using other syntax that is not documented until later in the chapter. If it is not clear what some syntax is supposed to do, then you can skip ahead and read about that particular element.
> - When identifiers (for example, column 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, then it must be enclosed in double quotation marks. See Reserved Words for the ORACLE_DATAPUMP Access Driver.

16.2 access_parameters Clause

When you create the `ORACLE_DATAPUMP` access driver external table, you can specify certain parameters in an `access_parameters` clause.

This clause is optional, as are its individual parameters. For example, you can specify `LOGFILE`, but not `VERSION`, or vice versa. The syntax for the `access_parameters` clause is as follows.

> **Note:**
>
> These access parameters are collectively referred to as the `opaque_format_spec` in the SQL `CREATE TABLE...ORGANIZATION EXTERNAL` statement.



- Comments
  The `ORACLE_DATAPUMP` access driver `comments` access parameter enables you to place comments with external tables
- COMPRESSION
  The `ORACLE_DATAPUMP` access driver `compression` access parameter specifies whether and how data is compressed before the external table data is written to the dump file set.
- ENCRYPTION
  The `ORACLE_DATAPUMP` access driver `encryption` access parameter specifies whether to encrypt data before it is written to the dump file set.
- LOGFILE | NOLOGFILE
  The `ORACLE_DATAPUMP` access driver `LOGFILE|NOLOGFILE` access parameter specifies the name of the log file that contains any messages generated while the dump file was being accessed.t.
- VERSION Clause
  The `ORACLE_DATAPUMP` access driver `version` clause access parameter enables you to specify generating a dump file that can be read with an earlier Oracle Database release.
- HADOOP_TRAILERS Clause
  The `ORACLE_DATAPUMP` access driver provides a `HADOOP_TRAILERS` clause that specifies whether to write Hadoop trailers to the dump file.
- Effects of Using the SQL ENCRYPT Clause
  Review the requirements and guidelines for external tables when you encrypt columns using the `ORACLE_DATAPUMP` access driver `ENCRYPT` clause.

Related Topics

- CREATE TABLE

> **See Also:**
>
> *Oracle Database SQL Language Reference* `CREATE TABLE` for information about specifying `opaque_format_spec` when using the SQL `CREATE TABLE...ORGANIZATION EXTERNAL` statement.

16.2.1 Comments

The `ORACLE_DATAPUMP` access driver `comments` access parameter enables you to place comments with external tables

Purpose

Comments are lines that begin with two hyphens followed by text.

Restrictions

Comments must be placed *before* any access parameters.

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.

16.2.2 COMPRESSION

The `ORACLE_DATAPUMP` access driver `compression` access parameter specifies whether and how data is compressed before the external table data is written to the dump file set.

Default

`DISABLED`

Purpose

Specifies whether to compress data (and optionally, which compression algorithm to use) before the data is written to the dump file set.

Syntax and Description

```
COMPRESSION [ENABLED {BASIC | LOW| MEDIUM | HIGH} | DISABLED]
```

- If `ENABLED` is specified, then all data is compressed for the entire unload operation. You can additionally specify one of the following compression options:
  - `BASIC` - Offers a good combination of compression ratios and speed; the algorithm used is the same as in previous versions of Oracle Data Pump.
  - `LOW` - Least impact on unload throughput and suited for environments where CPU resources are the limiting factor.
  - `MEDIUM` - Recommended for most environments. This option, like the `BASIC` option, provides a good combination of compression ratios and speed, but it uses a different algorithm than `BASIC`.
  - `HIGH` - Best suited for unloads over slower networks where the limiting factor is network speed.

> **Note:**
>
> To use these compression algorithms, the `COMPATIBLE` initialization parameter must be set to at least 12.0.0. This feature requires that the Oracle Advanced Compression option is enabled.

  The performance of a compression algorithm is characterized by its CPU usage and by the compression ratio (the size of the compressed output as a percentage of the uncompressed input). These measures vary on the size and type of inputs as well as the speed of the compression algorithms used. The compression ratio generally increases from low to high, with a trade-off of potentially consuming more CPU resources.

  Oracle recommends that you run tests with the different compression levels on the data in your environment. The only way to ensure that the exported dump file set compression level meets your performance and storage requirements is to test a compression level based on your environment, workload characteristics, and size and type of data.

- If `DISABLED` is specified, then no data is compressed for the upload operation.

Example

In the following example, the `COMPRESSION` parameter is set to `ENABLED`. Therefore, all data written to the `dept.dmp` dump file will be in compressed format.

```
CREATE TABLE deptXTec3
 ORGANIZATION EXTERNAL (TYPE ORACLE_DATAPUMP DEFAULT DIRECTORY def_dir1
 ACCESS PARAMETERS (COMPRESSION ENABLED) LOCATION ('dept.dmp'));
```

16.2.3 ENCRYPTION

The `ORACLE_DATAPUMP` access driver `encryption` access parameter specifies whether to encrypt data before it is written to the dump file set.

Default

`DISABLED`

Purpose

Specifies whether to encrypt data before it is written to the dump file set.

Syntax and Description

```
ENCRYPTION [ENABLED | DISABLED]
```

If `ENABLED` is specified, then all data is written to the dump file set in encrypted format.

If `DISABLED` is specified, then no data is written to the dump file set in encrypted format.

Restrictions

This parameter is used only for export operations.

Example

In the following example, the `ENCRYPTION` parameter is set to `ENABLED`. Therefore, all data written to the `dept.dmp` file will be in encrypted format.

```
CREATE TABLE deptXTec3
 ORGANIZATION EXTERNAL (TYPE ORACLE_DATAPUMP DEFAULT DIRECTORY def_dir1
 ACCESS PARAMETERS (ENCRYPTION ENABLED) LOCATION ('dept.dmp'));
```

16.2.4 LOGFILE | NOLOGFILE

The `ORACLE_DATAPUMP` access driver `LOGFILE|NOLOGFILE` access parameter specifies the name of the log file that contains any messages generated while the dump file was being accessed.t.

Default

If `LOGFILE` is not specified, then 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, then the access driver reopens that log file and appends the new log information to the end.

Purpose

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.

Syntax and Description

```
NOLOGFILE
```

or

```
LOGFILE [directory_object:]logfile_name
```

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, then an error is returned. See File Names for LOGFILE for information about using substitution variables to create unique file names during parallel loads or unloads.

Example

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'));
```

- Log File Naming in Parallel Loads

16.2.4.1 Log File Naming in Parallel Loads

The access driver does some symbol substitution to help make file names 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 file name, 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 file name, 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 file names 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 no extension is supplied for the file, then a default extension of .log is used. If the name generated is not a valid file name, then an error is returned and no data is loaded or unloaded.

16.2.5 VERSION Clause

The ORACLE_DATAPUMP access driver version clause access parameter enables you to specify generating a dump file that can be read with an earlier Oracle Database release.

Default

COMPATIBLE

Purpose

Specifies the version of database that can read the dump file.

The legal values for the VERSION parameter are as follows:

- COMPATIBLE - This value is the default value. The version of the metadata corresponds to the database compatibility level as specified on the COMPATIBLE initialization parameter.

  Note: Database compatibility must be set to 9.2 or later.
- LATEST - The version of the metadata and resulting SQL DDL corresponds to the database release, regardless of its compatibility level.
- *version_string* - A specific database release (for example, 11.2.0).This value cannot be lower than 9.2.

You can use the VERSION clause to create a dump file set that is compatible with a previous release of Oracle Database. The VERSION clause enables you to identify the compatibility version of objects that you export. COMPATIBLE indicates that the source and target database releases versions are compatible. However, if the source and target databases are not compatible (for example, when you unload data from an Oracle Database 19c release to an Oracle Database 11g (Release 11.2) database, where compatibility is set to 11.2), then you can specify the VERSION clause to indicate the compatibility level of the dump file is readable by the earlier release Oracle Database

Syntax and Description

```
VERSION=[COMPATIBLE | LATEST | version_string]
```

Example

If you use the access parameter VERSION clause to specify 11.2, then an Oracle Database 11g Release 11.2 database is the earliest Oracle Database release that can read the dump file. Oracle Databases with compatibility set to 11.2 or later can read the dump file. However, if you set the VERSION clause to 19, then only Oracle Database 19c and later Oracle Database releases can read the dump file that you generate.

16.2.6 HADOOP_TRAILERS Clause

The ORACLE_DATAPUMP access driver provides a HADOOP_TRAILERS clause that specifies whether to write Hadoop trailers to the dump file.

Default

DISABLED

Purpose

Specifies whether to write Hadoop trailers to the dump file.

Syntax and Description

```
HADOOP_TRAILERS [ENABLED|DISABLED]
```

When the `HADOOP_TRAILERS` clause is set to `ENABLED`, Hadoop trailers are written to the dump file. Hadoop trailers include information about locations and sizes of different parts of the file. The information is written in a dump trailer block at the end of the file, and at the end of the stream data, instead of at the beginning.

16.2.7 Effects of Using the SQL ENCRYPT Clause

Review the requirements and guidelines for external tables when you encrypt columns using the `ORACLE_DATAPUMP` access driver `ENCRYPT` clause.

Purpose

The `ENCRYPT` clause lets you use the Transparent Data Encryption (TDE) feature to encrypt the dump file

If you specify the SQL `ENCRYPT` clause when you create an external table, then keep the following in mind:

- The columns for which you specify the `ENCRYPT` clause will be encrypted before being written into the dump file.
- If you move the dump file to another database, then the same encryption password must be used for both the encrypted columns in the dump file, and for the external table used to read the dump file.
- If you do not specify a password for the correct encrypted columns in the external table on the second database, then an error is returned. If you do not specify the correct password, then garbage data is written to the dump file.
- The dump file that is produced must be at release 10.2 or higher. Otherwise, an error is returned.

Syntax and Description

See *Oracle Database SQL Language Reference* for more information about using the `ENCRYPT` clause on a `CREATE TABLE` statement

Related Topics

- *Oracle Database SQL Language Reference* CREATE TABLE

16.3 Unloading and Loading Data with the ORACLE_DATAPUMP Access Driver

As part of creating an external table with a SQL `CREATE TABLE AS SELECT` statement, the `ORACLE_DATAPUMP` access driver can write data to a dump file.

The data in the file is written in a binary format that can only be read by the `ORACLE_DATAPUMP` access driver. Once the dump file is created, it cannot be modified (that is, no data manipulation language (DML) operations can be performed on it). However, the file can be read any number of times and 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 data type 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
  This topic describes parallel loading and unloading.

- Combining Dump Files
  Dump files populated by different external tables can all be specified in the LOCATION clause of another external table.

16.3.1 Parallel Loading and Unloading

This topic describes 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 is 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 is 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.
```

When the ORACLE_DATAPUMP access driver is used to 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. The degree of parallelization is not tied to the number of files in the LOCATION clause when reading from ORACLE_DATAPUMP external tables.

16.3.2 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 data types 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
```

16.4 Supported Data Types

The `ORACLE_DATAPUMP` access driver resolves many data types automatically during loads and unloads.

When you use external tables to move data between databases, you may encounter the following situations:

- 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 data types 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 data type that is not supported for external tables, then you receive an error. This is demonstrated in the following example, in which the unsupported data type, 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
```

16.5 Unsupported Data Types

You can use the ORACLE_DATAPUMP access driver to unload and reload data for some of the unsupported data types

An external table supports a subset of all possible data types for columns. In particular, it supports character data types (except LONG), the RAW data type, all numeric data types, and all date, timestamp, and interval data types.

The unsupported data types for which you can use the ORACLE_DATAPUMP access driver to unload and reload data include the following:

- BFILE
- LONG and LONG RAW
- Final object types
- Tables of final object types
- Unloading and Loading BFILE Data Types
  The BFILE data type has two pieces of information stored in it: the directory object for the file and the name of the file within that directory object.
- Unloading LONG and LONG RAW Data Types
  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.
- 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.

- Tables of Final Object Types
  Object tables have an object identifier that uniquely identifies every row in the table.

16.5.1 Unloading and Loading BFILE Data Types

The `BFILE` data type 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 file name 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` data types.

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 file name 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
```

16.5.2 Unloading LONG and LONG RAW Data Types

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 data types.

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
```

16.5.3 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.
```

16.5.4 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
```

16.6 Performance Hints When Using the ORACLE_DATAPUMP Access Driver

Learn how to improve `ORACLE_DATAPUMP` access driver performance.

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 data files. 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.

16.7 Restrictions When Using the ORACLE_DATAPUMP Access Driver

Be aware of restrictions that apply to accessing external tables with the `ORACLE_DATAPUMP` access driver.

The restrictions that apply to using the `ORACLE_DATAPUMP` access driver with external tables includes the following:

- Encrypted columns: Exporting and importing of external tables with encrypted columns is not supported.
- 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 data type 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.
- `LONG` columns: An external table cannot load data into a `LONG` column.
- Handling of byte-order marks during a load: In an external table load for which the data file 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 data file 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 data file. If no byte-order mark is present, then the byte order of the server platform is used.
- Backslash escape characters: The external tables feature does not support the use of the backslash (`\`) escape character within strings.

- Reserved words: 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, then it must be enclosed in double quotation marks.

Related Topics

- Use of the Backslash Escape Character
  SQL*Loader and external tables use different conventions to identify single quotation marks as an enclosure character.

16.8 Reserved Words for the ORACLE_DATAPUMP Access Driver

If you use words in identifiers that are reserved by the `ORACLE_DATAPUMP` access driver, then they must be enclosed in double quotation marks.

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, then it must be enclosed in double quotation marks. The following are the reserved words for the `ORACLE_DATAPUMP` access driver:

- `BADFILE`
- `COMPATIBLE`
- `COMPRESSION`
- `DATAPUMP`
- `DEBUG`
- `ENCRYPTION`
- `INTERNAL`
- `JOB`
- `LATEST`
- `LOGFILE`
- `NOBADFILE`
- `NOLOGFILE`
- `PARALLEL`
- `TABLE`
- `VERSION`
- `WORKERID`

17

ORACLE_BIGDATA Access Driver

With the ORACLE_BIGDATA access driver, you can access data stored in object stores as if that data was stored in tables in an Oracle Database.

ORACLE_BIGDATA currently supports access to Oracle Object Store, Amazon S3, and Azure Blob Storage. You can also use this driver to query local data, which is useful for testing and smaller data sets.

- Using the ORACLE_BIGDATA Access Driver
  You can use the ORACLE_BIGDATA driver to access data located in external object stores.
- How to Create a Credential for Object Stores
  Credential objects enable you to access an external object store.
- Object Store Access Parameters
  You can use these access parameters to specify properties about the files residing in object stores.

Related Topics

- External Tables Examples
  Learn from these examples how to use the ORACLE_LOADER, ORACLE_DATAPUMP, ORACLE_HDFS, and ORACLE_HIVE access drivers to query data in Oracle Database and Big Data.

17.1 Using the ORACLE_BIGDATA Access Driver

You can use the ORACLE_BIGDATA driver to access data located in external object stores.

There are two steps required to access data in an object store:

- Create a credential object (not required for public buckets).
  A **credential object** stores object store credentials in an encrypted format. The identity specified by the credential must have access to the underlying data in the object store.
- In-line external tables are supported. These external tables are simply expressed as part of a query.
  Create an external table or query using an in-line external table. The access driver type must be ORACLE_BIGDATA. The CREATE TABLE statement must reference the credential object, which provides authentication to access the object store. The table you create also requires a LOCATION clause, which provides the URI to the files within the object store.

  For public buckets, the CREDENTIAL is not required.

17.2 How to Create a Credential for Object Stores

Credential objects enable you to access an external object store.

To create your credential object, use either the DBMS_CREDENTIAL.CREATE_CREDENTIAL or DBMS_CLOUD.CREATE_CREDENTIAL. This object contains the username and password information needed to access the object store. This credential password must match the authentication token (**auth token**) created for the username in your cloud service.

> **Note:**
>
> You must have the `DBMS_CLOUD` package installed.

- Creating the Credential Object with DBMS_CREDENTIAL.CREATE_CREDENTIAL
  The `DBMS_CLOUD` subprogram `DBMS_CREDENTIAL.CREATE_CREDENTIAL` enables you to authenticate access to an external object store.
- Creating the Credential Object with DBMS_CLOUD.CREATE_CREDENTIAL
  The `DBMS_CLOUD` subprogram `DBMS_CLOUD.CREATE_CREDENTIAL` enables you to authenticate access to an external object store.
- How to Define the Location Clause for Object Storage
  Use these examples to see how you can specify the object store URI, depending on its source.
- Understanding ORACLE_BIGDATA Access Parameters
  To use `ORACLE_BIGDATA`, you provide information in an access parameter to indicate how to access and parse the data.

Related Topics

- My Oracle Support Note 2748362.1

17.2.1 Creating the Credential Object with DBMS_CREDENTIAL.CREATE_CREDENTIAL

The `DBMS_CLOUD` subprogram `DBMS_CREDENTIAL.CREATE_CREDENTIAL` enables you to authenticate access to an external object store.

These examples show how to use `DBMS_CREDENTIAL.CREATE_CREDENTIAL`.

Example 17-1 Cloud Service Credentials

In the following example, *`my_credential`* is the Oracle Cloud Infrastructure user name, *`username`* is the account username, *`password`* is the Oracle Cloud Infrastructure auth token:

```
execute dbms_credential.create_credential(
   credential_name => 'my_credential',
   username        => 'username',
   password        => 'password'
);
```

Example 17-2 Native Oracle Cloud Infrastructure Credentials

In the following example, *`my_credential`* is the Oracle Cloud Infrastructure user name, *`user_ocid`* is the Oracle Cloud Identifier (OCID), *`tenancy_ocid`* is the Oracle Cloud tenancy identifier, *`private_key`* is the SSH private key, and *`fingerprint`* is the public key fingerprint:

```
execute dbms_credential.create_credential(
   credential_name => 'my_credential',
   username => 'user_ocid',
   password => '',
   key =>
```

```
'{"tenancy_ocid":"tenancy_ocid","private_key":"private_key","fingerprint":"fin
gerprint"}');
```

After you create the credential, specify the credential object name in the parameter `com.oracle.bigdata.credential.name`, At the time of this release, the credential must be in the same schema as the table

Related Topics

- CREATE_CREDENTIAL Procedure

17.2.2 Creating the Credential Object with DBMS_CLOUD.CREATE_CREDENTIAL

The `DBMS_CLOUD` subprogram `DBMS_CLOUD.CREATE_CREDENTIAL` enables you to authenticate access to an external object store.

These examples show how to use `DBMS_CLOUD.CREATE_CREDENTIAL`.

> **Note:**
>
> The credential parameter cannot be an OCI resource principal, Azure service principal, Amazon Resource Name (ARN), or a Google service account.

Example 17-3 Native Oracle Cloud Infrastructure Credentials

In the following example, *my_credential* is the Oracle Cloud Infrastructure user name, *password* is the Oracle Cloud Infrastructure auth token, *user_ocid* is the Oracle Cloud Identifier (OCID), *tenancy_ocid* is the Oracle Cloud tenancy identifier, *private_key* is the SSH private key, and *fingerprint* is the public key fingerprint:

```
execute dbms_cloud.create_credential DBMS_CLOUD.CREATE_CREDENTIAL(
    credential_name => 'my_credential',
    username => 'user_ocid@example.com',
    password => 'password'
    key =>
'{"tenancy_ocid":"tenancy_ocid","private_key":"private_key","fingerprint":"fin
gerprint"}');
```

After you create the credential, specify the credential object name in the parameter `com.oracle.bigdata.credential.name`. At the time of this release, the credential must be in the same schema as the table.

Related Topics

- CREATE_CREDENTIAL Procedure

17.2.3 How to Define the Location Clause for Object Storage

Use these examples to see how you can specify the object store URI, depending on its source.

`LOCATION` is a URI pointing to data in the object store. Currently supported object stores are Oracle Object Store, Amazon S3 and Azure Blob Storage.

In the examples, the following variables are used:

- *region* – tenancy region
- *host* – a server host name
- *port* – a port number assigned to the service, listening on a host
- *container* – name of a container resource
- *namespace* – namespace in a region
- *bucket* – a globally unique name for a resource
- *objectname* – a unique identifier for an object in a bucket
- *filename* – object store filename

Note the following prerequisites for defining the location:

- The credential object is required for private object store access. If the credential parameter is omitted, then the object must be in a public bucket.

  The user ID associated with this credential must have access to read the data from object storage.
- If you are testing access for data in object storage using local storage, then you must specify an Oracle directory object in the location, similar to what you do for `ORACLE_LOADER` data sources.

Example 17-4 Native Oracle Cloud Infrastructure Object Storage

```
location ('https://objectstorage.region.oraclecloud.com/n/namespace/b/
bucket/o/objectname')
```

Example 17-5 Oracle Cloud Infrastructure Object Storage

```
location ('https://swiftobjectstorage.region.oraclecloud.com/v1/namespace/
bucket/filename'
```

Example 17-6 Hosted-Style URI format

```
location ('https://bucket.host/objectname')
```

Example 17-7 Path-style URI Format

```
location ('https://host/bucket/objectname')
```

For example, an Amazon path style URI can take the following format:

```
location ('https://s3-us-west-2.amazonaws.com/adwc/filename')
```

Example 17-8 Azure BLOB Storage Location Format

```
location ('https://host:port/container/blob')
```

For example, an Azure path style URI can take the following format:

```
location ('https://exampleacount.blob.core.windows.net/examplecontainer/
exampleblob')
```

17.2.4 Understanding ORACLE_BIGDATA Access Parameters

To use `ORACLE_BIGDATA`, you provide information in an access parameter to indicate how to access and parse the data.

To access the external object store, you define the file format type in the access parameter `com.oracle.bigdata.fileformat`, using one of the following values: `csv`, `textfile`, `avro`, `parquet`, or `orc`:

```
com.oracle.bigdata.fileformat=[csv|textfile|avro|parquet|orc]
```

You can also use `ORACLE_BIGDATA` to access local files for testing, or for simple querying. In this case, the `LOCATION` field value is the same as what you would use for `ORACLE_LOADER`. You can use an Oracle directory object followed by the name of the file in the `LOCATION` field. For local files, a credential object is not required. However, you must have privileges over on the directory object in order to access the file.

Related Topics

- ORACLE_BIGDATA Access Parameters in *Oracle Big Data SQL User's Guide*

17.3 Object Store Access Parameters

You can use these access parameters to specify properties about the files residing in object stores.

- Syntax Rules for Specifying Properties
  The properties are set using keyword-value pairs in the SQL `CREATE TABLE ACCESS PARAMETERS` clause and in the configuration files.
- ORACLE_BIGDATA Access Parameters
  There is a set of access parameters that are common to all file formats. There are also parameters that are unique to a specific file format.
- GATHER_EXTERNAL_TABLE_STATS
  This is the PL/SQL interface for manually gathering statistics on external tables (`ORACLE_HDFS`, `ORACLE_HIVE`, `ORACLE_BIGDATA`).

17.3.1 Syntax Rules for Specifying Properties

The properties are set using keyword-value pairs in the SQL `CREATE TABLE ACCESS PARAMETERS` clause and in the configuration files.

The syntax must obey these rules:

- The format of each keyword-value pair is a **keyword**, a colon or equal sign, and a **value**. The following are valid keyword-value pairs:

  ```
  keyword=value
  keyword:value
  ```

The value is everything from the first non-whitespace character after the separator to the end of the line. Whitespace between the separator and the value is ignored. Trailing whitespace for the value is retained.

- A property definition can be on one line or multiple lines.
- A line terminator is a line feed, a carriage return, or a carriage return followed by line feeds.
- When a property definition spans multiple lines, then precede the line terminators with a backslash (escape character), except on the last line. In this example, the value of the `Keyword1` property is `Value part 1 Value part 2 Value part 3`.

```
Keyword1=  Value part 1 \
           Value part 2 \
           Value part 3
```

- You can create a **logical line** by stripping each physical line of leading whitespace and concatenating the lines. The parser extracts the property names and values from the logical line.
- You can embed special characters in a property name or property value by preceding a character with a backslash (escape character), indicating the substitution. The following table describes the special characters.

Table 17-1 Special Characters in Properties

Escape Sequence	Character
`\b`	Backspace (`\u0008`)
`\t`	Horizontal tab (`\u0009`)
`\n`	Line feed (`\u000a`)
`\f`	Form feed (`\u000c`)
`\r`	Carriage return (`\u000d`)
`\"`	Double quote (`\u0022`)
`\'`	Single quote (`\u0027`)
`\\`	Backslash (`\u005c`) When multiple backslashes are at the end of the line, the parser continues the value to the next line only for an odd number of backslashes.
`\uxxxx`	2-byte, big-endian, Unicode code point. When a character requires two code points (4 bytes), the parser expects `\u` for the second code point.

17.3.2 ORACLE_BIGDATA Access Parameters

There is a set of access parameters that are common to all file formats. There are also parameters that are unique to a specific file format.

Common Access Parameters

The following table lists parameters that are common to all file formats accessed through `ORACLE_BIGDATA`. The first column identifies each access parameter common to all data file types. The second column describes each parameter.

Table 17-2 Common Access Parameters

Common Access Parameter	Description

com.oracle.bigdata.credential.name	Specifies the credential object to use when accessing data files in an object store.
	This access parameter is required for object store access. It is not needed for access to files through a directory object or for data stored in public buckets.
	The name specified for the credential must be the name of a credential object in the same schema as the owner of the table. Granting a user SELECT or READ access to this table means that credential will be used to access the table.
	Use DBMS_CREDENTIAL.CREATE_CREDENTIAL in the DBMS_CREDENTIAL PL/SQL package to create the credential object:
	`exec dbms_credential.create_credential(credential_name => 'MY_CRED',username =>'<username>', password => '<password>');`
	In the CREATE TABLE statement, set the value of the credential parameter to the name of the credential object.
	`com.oracle.bigdata.credential.name=MY_CRED`

com.oracle.bigdata.fileformat	Specifies the format of the file. The value of this parameter identifies the reader that will process the file. Each reader can support additional access parameters that may or may not be supported by other readers. **Valid values:** parquet, orc, textfile, avro, csv **Default:** PARQUET
com.oracle.bigdata.log.opt	Specifies whether log messages should be written to a log file. When none is specified, then no log file is created. If the value is normal, then log file is created when the file reader decides to write a message. It is up to the file reader to decide what is written to the log file. **Valid values:** normal, none **Default:** none.
com.oracle.bigdata.log.qc	Specifies the name of the log file created by the parallel query coordinator. This parameter is used only when com.oracle.bigdata.log.opt is set to normal. The valid values are the same as specified for com.oracle.bigdata.log.qc in ORACLE_HIVE and ORACLE_HDFS.
com.oracle.bigdata.log.exec	Specifies the name of the log file created during query execution. This value is used (and is required) only when com.oracle.bigdata.log.exec is set to normal. The valid values are the same as specified for in ORACLE_HIVE and ORACLE_HDFS. **Valid values:** normal, none **Default:** none.

Avro-Specific Access Parameters

In addition to common access parameters, there are some that are only valid for the Avro file format. The first column in this table identifies the access parameters specific to the Avro file

format and the second column describes the parameter. There is only one Avro-specific parameter at this time.

Table 17-3 Avro-Specific Access Parameters

Avro-Specific Parameter	Description
`com.oracle.bigdata.avro.decimaltpe`	Specifies the representation of a decimal stored in the byte array. **Valid values:** `int`, `integer`, `str`, `string` **Default:** If this parameter is not used, an Avro decimal column is read assuming byte arrays store the numerical representation of the values (that is default to `int`) as the Avro specification defines.

Parquet-Specific Access Parameters

Some access parameters are only valid for the Parquet file format. The first column in this table identifies the access parameters specific to the Parquet file format and the second column describes the parameter.

Table 17-4 Parquet-Specific Access Parameters

Parquet-Specific Access Parameter	Description
`com.oracle.bigdata.prq.binary_as_string`	This is a Boolean property that specifies if the binary is stored as a string. **Valid values:** `true`, `t`, `yes`, `y`, `1`, `false`, `f`, `no`, `n`, `0` **Default:** `true`
`com.oracle.bigdata.prq.int96_as_timestamp`	This is a Boolean property that specifies if `int96` represents a timestamp. **Valid values:** `true`, `t`, `yes`, `y`, `1`, `false`, `f`, `no`, `n`, `0` **Default:** `true`

Textfile and CSV-Specific Access Parameters

The text file and comma-separated value (`csv`) file formats are similar to the hive text file format. It reads text and `csv` data from delimited files. `ORACLE_BIGDATA` automatically detects the line terminator (either `\n`, `\r`, or `\r\n`). By default, it assumes the fields in the file are separated by commas, and the order of the fields in the file match the order of the columns in the external table.

Example 17-9 CSV Data File

This is a simple `csv` example. The data file has comma-separated values, with optional enclosing quotes.

```
-----Source csv data in t.dat
t.dat:
```

```
1,"abc",
2,xyx,


----------Create an external table over the csv source data in t.dat
CREATE TABLE t
(
 c0 number,
 c1 varchar2(20)
)
ORGANIZATION external
(
 TYPE oracle_bigdata
 DEFAULT DIRECTORY DMPDIR
 ACCESS PARAMETERS
 (
  com.oracle.bigdata.fileformat=csv
 )
 location
 (
  't.dat'
 )
)REJECT LIMIT 1
;


-------Select data from external table
select c0, c1 from t;

 C0    C1
----  -----
 1     abc
 2     xyz
```

Example 17-10 CSV Data File

This example shows how to create an external table over a `csv` data source, which has '`|`' as the field separator, the data file compressed with `gzip`, blanks as null, and a date format.

```
-----The source csv data in t.dat
t.dat:

 1|  |
 2|Apr-99-30|


-------Create an external table over the csv data source in t.dat
CREATE TABLE t(
 c0 number,
 c1 date
)
ORGANIZATION external
(
 TYPE oracle_bigdata
 DEFAULT DIRECTORY DMPDIR
 ACCESS PARAMETERS
```

```
 (
  com.oracle.bigdata.fileformat=textfile
  com.oracle.bigdata.compressiontype=gzip
  com.oracle.bigdata.csv.rowformat.separatorcharacter='|'
  com.oracle.bigdata.blankasnull=true
  com.oracle.bigdata.dateformat="MON-RR-DD HH:MI:SS"
 )
 location
 (
  't.dat.gz'
  )
 )REJECT LIMIT 1
;


--Select csv data from external table
QL> select c0, c1 from t;

 C0       C1
------ ---------
  1
  2    30-APR-99
```

Example 17-11 JSON Data File

This is a JSON file where each row is a JSON document. The external table reaches each row. Queries use Oracle SQL JSON functions to parse the data.

```
{"id":"72","name":"George","lat":40.76727216,"lon":-73.99392888,"segments":
["wealthy","middle-aged"],"age":45}
{"id":"79","name":"Lucy","lat":40.71911552,"lon":-74.00666661,"segments":
["married","yes"],"age":33}


— Create the external table over Json source
CREATE TABLE people (
 doc varchar2(4000)
)
ORGANIZATION EXTERNAL (
  TYPE ORACLE_BIGDATA
  DEFAULT DIRECTORY DEFAULT_DIR
  ACCESS PARAMETERS(
    com.oracle.bigdata.credential.name=MY_CRED
    com.oracle.bigdata.fileformat=textfile
    com.oracle.bigdata.csv.rowformat.fields.terminator='\n’    <— notice the
      delimiter is a new line (that is, not a comma).  So, it will read to
the end of the line and get
      the whole document.
    )
  LOCATION ('https://swftobjectstorage.us-phoenix-1.oraclecloud.com/v1/
mybucket/people.json')
```

```
)
REJECT LIMIT UNLIMITED;


----Select data from external table
select s.doc.id,
  s.doc.name,
  s.doc.segments[0] ---notice we're getting the first item in the array of
possible individuals
from the table people;

id     Name     segments[0]
----   -----    -----------------------------------
  72   George   wealthy
  79   Lucy     married
```

Textfile and CSV-Specific Access Parameters

Table 17-5 Textfile and CSV-Specific Access Parameters

Textfile-Specific Access Parameter	Description
`com.oracle.bigdata.buffersize`	Specifies the size of the input/output (I/O) buffer used for reading the file. The value is the size of the buffer in kilobytes. Note that the buffer size is also the largest size that a record can be. If a format reader encounters a record larger than this value, then it will return an error. **Default:** `1024`
`com.oracle.bigdata.blankasnull`	When set to `true`, loads fields consisting of spaces as null. **Valid values**: `true`, `false` **Default**: `false` **Example**: `com.oracle.bigdata.blankasnull=true`
`com.oracle.bigdata.characterset`	Specifies the character set of source files. **Valid values**: `UTF-8` **Default**: `UTF-8` **Example**: `com.oracle.bigdata.characterset=UTF-8`
`com.oracle.bigdata.compressiontype`	If this parameter is specified, then the code tries to decompress the data according to the compression scheme specified. **Valid values:** `gzip`, `bzip2` , `zlib`, `detect` **Default:** no compression If `detect` is specified, then the format reader tries to determine which of the supported compression methods was used to compress the file.

Table 17-5 (Cont.) Textfile and CSV-Specific Access Parameters

Textfile-Specific Access Parameter	Description
com.oracle.bigdata.conversionerrors	If a row has data type conversion errors, then the related columns are stored as null, or the row is rejected. **Valid values**: reject_record, store_null **Default**: store_null **Example**: com.oracle.bigdata.conversionerrors=reject_record
com.oracle.bigdata.csv.rowformat.nulldefinedas	Specifies the character used to indicate the value of a field is NULL. If the parameter is not specified, then there is no value.
com.oracle.bigdata.csv.rowformat.fields.terminator	Specifies the character used to separate the field values. The character value must be wrapped in single-quotes. Example: '|'. **Default:** ','
com.oracle.bigdata.csv.rowformat.fields.escapedby	Specifies the character used to escape any embedded field terminators or line terminators in the value for fields. The character value must be wrapped in single quotes. Example: '\'.
com.oracle.bigdata.dateformat	Specifies the date format in the source file. The format option Auto checks for the following formats: J, MM-DD-YYYYBC, MM-DD-YYYY, YYYYMMDD HHMISS, YYMMDD HHMISS, YYYY.DDD, YYYY-MM-DD **Default**: yyyy-mm-dd hh24:mi:ss **Example**: com.oracle.bigdata.dateformat="MON-RR-DDHH:MI:SS"
com.oracle.bigdata.fields	Specifies the order of fields in the data file. The values are the same as for com.oracle.bigdata.fields in ORACLE_HDFS, with one exception – in this case, the data type is optional. Because the data file is text, the text file reader ignores the data types for the fields, and assumes all fields are text. Because the data type is optional, this parameter can be a list of field names.
com.oracle.bigdata.ignoreblanklines	Blank lines are ignored when set to true. **Valid values**: true, false **Default**: false **Example**: com.oracle.bigdata.ignoreblanklines=true

Table 17-5 (Cont.) Textfile and CSV-Specific Access Parameters

Textfile-Specific Access Parameter	Description
com.oracle.bigdata.ignoremissingcolumns	Missing columns are stored as null. **Valid values**: true **Default**: true **Example**: com.oracle.bigdata.ignoremissingcolumns=true
com.oracle.bigdata.quote	Specifies the quote character for the fields. The quote characters are removed during loading when specified. **Valid values**: character **Default**: Null, meaning no quote **Example:** com.oracle.bigdata.csv.rowformat.quotecharacter='"'
com.oracle.bigdata.rejectlimit	The operation errors out after specified number of rows are rejected. This only applies when rejecting records due to conversion errors. **Valid values**: number **Default**: 0 **Example:** com.oracle.bigdata.rejectlimit=2
com.oracle.bigdata.removequotes	Removes any quotes that are around any field in the source file. **Valid values**: true, false **Default**: false **Example:**com.oracle.bigdata.removequotes=true
com.oracle.bigdata.csv.skip.header	Specifies how many rows should be skipped from the start of the files. **Valid values**: number **Default**: 0, if not specified **Example:** com.oracle.bigdata.csv.skip.header=1
com.oracle.bigdata.timestampformat	Specifies the timestamp format in the source file. The format option AUTO checks for the following formats: YYYY-MM-DD HH:MI:SS.FF, YYYY-MM-DD HH:MI:SS.FF3, MM/DD/YYYY HH:MI:SS.FF3 **Valid values**: auto **Default**: yyyy-mm-dd hh24:mi:ss.ff **Example:** com.oracle.bigdata.timestamptzformat="auto"

Table 17-5 (Cont.) Textfile and CSV-Specific Access Parameters

Textfile-Specific Access Parameter	Description
com.oracle.bigdata.timestampltzformat	Specifies the timestamp with local timezone format in the source file. The format option AUTO checks for the following formats: DD Mon YYYY HH:MI:SS.FF TZR, MM/DD/YYYY HH:MI:SS.FF TZR, YYYY-MM-DD HH:MI:SS+/-TZR, YYYY-MM-DD HH:MI:SS.FF3, DD.MM.YYYY HH:MI:SS TZR **Valid values**: auto **Default**: yyyy-mm-dd hh24:mi:ss.ff **Example:** com.oracle.bigdata.timestampltzformat="auto"
com.oracle.bigdata.timestamptzformat	Specifies the timestamp with timezone format in the source file. The format option AUTO checks for the following formats: DD Mon YYYY HH:MI:SS.FF TZR, MM/DD/YYYY HH:MI:SS.FF TZR, YYYY-MM-DD HH:MI:SS+/-TZR, YYYY-MM-DD HH:MI:SS.FF3, DD.MM.YYYY HH:MI:SS TZR **Valid values**: auto **Default**: yyy-mm-dd hh24:mi:ss.ff **Example:** com.oracle.bigdata.timestamptzformat="auto"
com.oracle.bigdata.trimspaces	Specifies how the leading and trailing spaces of the fields are trimmed. **Valid values**: rtrim, ltrim, notrim, ltrim, ldrtrim **Default**: notrim **Example:** com.oracle.bigdata.trimspaces=rtrim
com.oracle.bigdata.truncatecol	If the data in the file is too long for a field, then this option truncates the value of the field rather than rejecting the row or setting the field to NULL. **Valid values**: true, false **Default**: false **Example**: com.oracle.bigdata.truncatecol=true

17.3.3 GATHER_EXTERNAL_TABLE_STATS

This is the PL/SQL interface for manually gathering statistics on external tables (ORACLE_HDFS, ORACLE_HIVE, ORACLE_BIGDATA).

The behavior and parameters are identical to that of dbms_stats.gather_table_stats, with the exception that the owner of the table must be the session user running the procedure, and the stats gathered using this procedure persist after a restart. This procedure cannot be used on external tables that are automatically synced from Hive.

See GATHER_TABLE_STATS Procedure

Syntax

```
PROCEDURE gather_external_table_stats( tabname varchar2, partname varchar2
default null, estimate_percent number default
dbms_stats.DEFAULT_ESTIMATE_PERCENT, block_sample boolean default FALSE,
method_opt varchar2 default dbms_stats.DEFAULT_METHOD_OPT, degree number default
dbms_stats.DEFAULT_DEGREE_VALUE, granularity varchar2 default
dbms_stats.DEFAULT_GRANULARITY, cascade boolean default
dbms_stats.DEFAULT_CASCADE, stattab varchar2 default null, statid varchar2
default null, statown varchar2 default null, no_invalidate boolean default
dbms_stats.to_no_invalidate_type(dbms_stats.get_param('NO_INVALIDATE')), stattype
varchar2 default 'DATA', force boolean default FALSE, options varchar2 default
dbms_stats.DEFAULT_OPTIONS )

PROCEDURE gather_external_table_stats(
    tabname varchar2,
    partname varchar2 default null,
    estimate_percent number default dbms_stats.DEFAULT_ESTIMATE_PERCENT,
    block_sample boolean default FALSE,
    method_opt varchar2 default dbms_stats.DEFAULT_METHOD_OPT,
    degree number default dbms_stats.DEFAULT_DEGREE_VALUE,
    granularity varchar2 default dbms_stats.DEFAULT_GRANULARITY,
    cascade boolean default dbms_stats.DEFAULT_CASCADE,
    stattab varchar2 default null,
    statid varchar2 default null,
    statown varchar2 default null,
    no_invalidate boolean default
dbms_stats.to_no_invalidate_type(dbms_stats.get_param('NO_INVALIDATE')),
    stattype varchar2 default 'DATA',
    force boolean default FALSE,
    options varchar2 default dbms_stats.DEFAULT_OPTIONS
  )
```

18
External Tables Examples

Learn from these examples how to use the `ORACLE_LOADER`, `ORACLE_DATAPUMP`, `ORACLE_HDFS`, and `ORACLE_HIVE` access drivers to query data in Oracle Database and Big Data.

- Using the ORACLE_LOADER Access Driver to Create Partitioned External Tables
  This topic describes using the `ORACLE_LOADER` access driver to create partitioned external tables.
- Using the ORACLE_LOADER Access Driver to Create Partitioned Hybrid Tables
  This topic describes using the `ORACLE_LOADER` access driver to create partitioned hybrid tables.
- Using the ORACLE_DATAPUMP Access Driver to Create Partitioned External Tables
  The example in this section shows how to create a subpartitioned external table.
- Using the ORACLE_BIGDATA Access Driver to Create Partitioned External Tables
  The example in this section shows how to create a subpartitioned external table.
- Loading LOBs From External Tables
  External tables are particularly useful for loading large numbers of records from a single file, so that each record appears in its own row in the table.
- Loading CSV Files From External Tables
  This topic provides examples of how to load CSV files from external tables under various conditions.

18.1 Using the ORACLE_LOADER Access Driver to Create Partitioned External Tables

This topic describes using the `ORACLE_LOADER` access driver to create partitioned external tables.

Example 18-1 Using ORACLE_LOADER to Create a Partitioned External Table

This example assumes there are four data files with the following content:

```
p1a.dat:
1, AAAAA Plumbing,01372,
28, Sparkly Laundry,78907,
13, Andi's Doughnuts,54570,

p1b.dat:
51, DIY Supplies,61614,
87, Fast Frames,22201,
89, Friendly Pharmacy,89901,

p2.dat:
121, Pleasant Pets,33893,
130, Bailey the Bookmonger,99915,
105, Le Bistrot du Chat Noir,94114,
```

```
p3.dat:
210, The Electric Eel Diner,07101,
222, Everyt'ing General Store,80118,
231, Big Rocket Market,01754,
```

There are three fields in the data file: `CUSTOMER_NUMBER`, `CUSTOMER_NAME` and `POSTAL_CODE`. The external table uses range partitioning on `CUSTOMER_NUMBER` to create three partitions.

- Partition 1 is for `customer_number` less than 100
- Partition 2 is for `customer_number` less than 200
- Partition 3 is for `customer_number` less than 300

Note that the first partition has two data files while the other partitions only have one. The following is the output from SQLPlus for creating the file.

```
SQL> create table customer_list_xt
  2    (CUSTOMER_NUMBER number, CUSTOMER_NAME VARCHAR2(50), POSTAL_CODE
CHAR(5))
  3  organization external
  4    (type oracle_loader default directory def_dir1)
  5  partition by range(CUSTOMER_NUMBER)
  6  (
  7    partition p1 values less than (100) location('p1a.dat', 'p1b.dat'),
  8    partition p2 values less than (200) location('p2.dat'),
  9    partition p3 values less than (300) location('p3.dat')
 10  );

Table created.
SQL>
```

The following is the output from `SELECT *` for the entire table:

```
SQL> select customer_number, customer_name, postal_code
  2    from customer_list_xt
  3    order by customer_number;

CUSTOMER_NUMBER CUSTOMER_NAME                                      POSTA
--------------- -------------------------------------------------- -----
              1  AAAAA Plumbing                                     01372
             13  Andi's Doughnuts                                  54570
             28  Sparkly Laundry                                   78907
             51  DIY Supplies                                      61614
             87  Fast Frames                                       22201
             89  Friendly Pharmacy                                 89901
            105  Le Bistrot du Chat Noir                           94114
            121  Pleasant Pets                                     33893
            130  Bailey the Bookmonger                             99915
            210  The Electric Eel Diner                            07101
            222  Everyt'ing General Store                          80118
            231  Big Rocket Market                                 01754

12 rows selected.

SQL>
```

The following query should only read records from the first partition:

```
SQL> select customer_number, customer_name, postal_code
  2    from customer_list_xt
  3    where customer_number < 20
  4    order by customer_number;

CUSTOMER_NUMBER CUSTOMER_NAME                                      POSTA
--------------- -------------------------------------------------- -----
              1  AAAAA Plumbing                                     01372
             13  Andi's Doughnuts                                  54570

2 rows selected.

SQL>
```

The following query specifies the partition to read as part of the `SELECT` statement.

```
SQL> select customer_number, customer_name, postal_code
  2    from customer_list_xt partition (p1)
  3    order by customer_number;

CUSTOMER_NUMBER CUSTOMER_NAME                                      POSTA
--------------- -------------------------------------------------- -----
              1  AAAAA Plumbing                                     01372
             13  Andi's Doughnuts                                  54570
             28  Sparkly Laundry                                   78907
             51  DIY Supplies                                      61614
             87  Fast Frames                                       22201
             89  Friendly Pharmacy                                 89901

6 rows selected.

SQL>
```

18.2 Using the ORACLE_LOADER Access Driver to Create Partitioned Hybrid Tables

This topic describes using the `ORACLE_LOADER` access driver to create partitioned hybrid tables.

Hybrid Partitioned Tables is a feature that extends Oracle Partitioning by allowing some partitions to reside in database segments and some partitions in external files or sources. This significantly enhances functionality of partitioning for Big Data SQL where large portions of a table can reside in external partitions.

Example 18-2 Example

Here is an example of a statement for creating a partitioned hybrid l table:

```
CREATE TABLE hybrid_pt (time_id date, customer number)
  TABLESPACE TS1
  EXTERNAL PARTITION ATTRIBUTES (TYPE ORACLE_LOADER
                                 DEFAULT DIRECTORY data_dir0
                                 ACCESS PARAMETERS(FIELDS TERMINATED BY ',')
```

```
                                    REJECT LIMIT UNLIMITED)
PARTITION by range (time_id)
(
 PARTITION century_18 VALUES LESS THAN ('01-01-1800')
    EXTERNAL,                                                  <-- empty
external partition
  PARTITION century_19 VALUES LESS THAN ('01-01-1900')
    EXTERNAL DEFAULT DIRECTORY data_dir1 LOCATION ('century19_data.txt'),
  PARTITION century_20 VALUES LESS THAN ('01-01-2000')
    EXTERNAL LOCATION ('century20_data.txt'),
  PARTITION year_2000 VALUES LESS THAN ('01-01-2001') TABLESPACE TS2,
  PARTITION pmax VALUES LESS THAN (MAXVALUE)
);
```

In this example, the table contains both internal and external partitions. The default tablespace for internal partitions in the table is `TS1`. An `EXTERNAL PARTITION ATTRIBUTES` clause is added for specifying parameters that apply, at the table level, to the external partitions in the table. The clause is mandatory for hybrid partitioned tables. In this case, external partitions are accessed through the `ORACLE_LOADER` access driver, and the parameters required by the access driver are specified in the clause. At the partition level, an `EXTERNAL` clause is specified in each external partition, along with any external parameters applied to the partition.

In this example, `century_18`, `century_19`, and `century_20` are external partitions. `century_18` is an empty partition since it does not contain a location. The default directory for partition `century_19` is`data_dir1`, overriding the table level default directory. The partition has a location `data_dir1:century19_data.txt`. Partition`century_20` has location `data_dir0:century20_data.txt`, since the table level default directory is applied to a location when a default directory is not specified in a partition. Partitions `year_2000` and `pmax` are internal partitions. Partition `year_2000`has a `tablespace TS2`. When a partition has no`EXTERNAL` clause or external parameters specified in it, it is assumed to be an internal partition by default.

18.3 Using the ORACLE_DATAPUMP Access Driver to Create Partitioned External Tables

The example in this section shows how to create a subpartitioned external table.

It also shows how to use a virtual column to partition the table.

Example 18-3 Using the ORACLE_DATAPUMP Access Driver to Create Partitioned External Tables

In this example, the dump files used are the same as those created in the previous example using the `ORACLE_LOADER` access driver. However, in this example, in addition to partitioning the data using `customer_number`, the data is subpartitioned using `postal_code`. For every partition, there is a subpartition where the `postal_code` is less than 50000 and another subpartition for all other values of `postal_code`. With three partitions, each containing two subpartitions, a total of six files is required. To create the files, use the SQL `CREATE TABLE AS SELECT` statement to select the correct rows for the partition and then write those rows into the file for the `ORACLE_DATAPUMP` driver.

The following statement creates a file with data for the first subpartition (`postal_code` less than 50000) of partition `p1` (`customer_number` less than 100).

```
SQL> create table customer_list_dp_p1_sp1_xt
  2  organization external
  3    (type oracle_datapump default directory def_dir1
location('p1_sp1.dmp'))
  4  as
  5    select customer_number, customer_name, postal_code
  6      from customer_list_xt partition (p1)
  7      where to_number(postal_code) < 50000;

Table created.

SQL>
```

This statement creates a file with data for the second subpartition (all other values for postal_code) of partition `p1` (customer_number less than 100).

```
SQL> create table customer_list_dp_p1_sp2_xt
  2  organization external
  3    (type oracle_datapump default directory def_dir1
location('p1_sp2.dmp'))
  4  as
  5    select customer_number, customer_name, postal_code
  6      from customer_list_xt partition (p1)
  7      where to_number(postal_code) >= 50000;

Table created.
```

The files for other partitions are created in a similar fashion, as follows:

```
SQL> create table customer_list_dp_p2_sp1_xt
  2  organization external
  3    (type oracle_datapump default directory def_dir1
location('p2_sp1.dmp'))
  4  as
  5    select customer_number, customer_name, postal_code
  6      from customer_list_xt partition (p2)
  7      where to_number(postal_code) < 50000;

Table created.

SQL>
SQL> create table customer_list_dp_p2_sp2_xt
  2  organization external
  3    (type oracle_datapump default directory def_dir1
location('p2_sp2.dmp'))
  4  as
  5    select customer_number, customer_name, postal_code
  6      from customer_list_xt partition (p2)
  7      where to_number(postal_code) >= 50000;

Table created.
```

```
SQL>
SQL> create table customer_list_dp_p3_sp1_xt
  2  organization external
  3    (type oracle_datapump default directory def_dir1
location('p3_sp1.dmp'))
  4  as
  5    select customer_number, customer_name, postal_code
  6      from customer_list_xt partition (p3)
  7      where to_number(postal_code) < 50000;

Table created.

SQL>
SQL> create table customer_list_dp_p3_sp2_xt
  2  organization external
  3    (type oracle_datapump default directory def_dir1
location('p3_sp2.dmp'))
  4  as
  5    select customer_number, customer_name, postal_code
  6      from customer_list_xt partition (p3)
  7      where to_number(postal_code) >= 50000;

Table created.

SQL>
```

You can select from each of these external tables to verify that it has the data you intended to write out. After you have executed the SQL `CREATE TABLE AS SELECT` statement, you can drop these external tables.

To use a virtual column to partition the table, create the partitioned `ORACLE_DATAPUMP` table. Again, the table is partitioned on the `customer_number` column and subpartitioned on the `postal_code` column. The `postal_code` column is a character field that contains numbers, but this example partitions it based on the numeric value, not a character string. In order to do this, create a virtual column, `postal_code_num`, whose value is the `postal_code` field converted to a `NUMBER` data type. The `SUBPARTITION` clause uses the virtual column to determine the subpartition for the row.

```
SQL> create table customer_list_dp_xt
  2  (customer_number    number,
  3   CUSTOMER_NAME      VARCHAR2(50),
  4   postal_code        CHAR(5),
  5   postal_code_NUM    as (to_number(postal_code)))
  6  organization external
  7    (type oracle_datapump default directory def_dir1)
  8  partition by range(customer_number)
  9  subpartition by range(postal_code_NUM)
 10  (
 11    partition p1 values less than (100)
 12     (subpartition p1_sp1 values less than (50000) location('p1_sp1.dmp'),
 13      subpartition p1_sp2 values less than (MAXVALUE)
location('p1_sp2.dmp')),
 14    partition p2 values less than (200)
 15     (subpartition p2_sp1 values less than (50000) location('p2_sp1.dmp'),
```

```
 16      subpartition p2_sp2 values less than (MAXVALUE)
location('p2_sp2.dmp')),
 17    partition p3 values less than (300)
 18     (subpartition p3_sp1 values less than (50000) location('p3_sp1.dmp'),
 19      subpartition p3_sp2 values less than (MAXVALUE)
location('p3_sp2.dmp'))
 20  );

Table created.

SQL>
```

If you select all rows, then the data returned is the same as was returned in the previous example using the `ORACLE_LOADER` access driver.

```
SQL> select customer_number, customer_name, postal_code
  2    from customer_list_dp_xt
  3    order by customer_number;

customer_number CUSTOMER_NAME                                      POSTA
--------------- -------------------------------------------------- -----
              1  AAAAA Plumbing                                     01372
             13  Andi's Doughnuts                                  54570
             28  Sparkly Laundry                                   78907
             51  DIY Supplies                                      61614
             87  Fast Frames                                       22201
             89  Friendly Pharmacy                                 89901
            105  Le Bistrot du Chat Noir                           94114
            121  Pleasant Pets                                     33893
            130  Bailey the Bookmonger                             99915
            210  The Electric Eel Diner                            07101
            222  Everyt'ing General Store                          80118
            231  Big Rocket Market                                 01754

12 rows selected.

SQL>
```

The `WHERE` clause can limit the rows read to a subpartition. The following query should only read the first subpartition of the first partition.

```
SQL> select customer_number, customer_name, postal_code
  2    from customer_list_dp_xt
  3    where customer_number < 20 and postal_code_NUM < 39998
  4    order by customer_number;

customer_number CUSTOMER_NAME                                      POSTA
--------------- -------------------------------------------------- -----
              1  AAAAA Plumbing                                     01372

1 row selected.

SQL>
```

You could also specify a specific subpartition in the query, as follows:

```
SQL> select customer_number, customer_name, postal_code
  2    from customer_list_dp_xt subpartition (p2_sp2) order by
customer_number;

customer_number CUSTOMER_NAME                                      POSTA
--------------- -------------------------------------------------- -----
            105  Le Bistrot du Chat Noir                           94114
            130  Bailey the Bookmonger                             99915

2 rows selected.

SQL>
```

18.4 Using the ORACLE_BIGDATA Access Driver to Create Partitioned External Tables

The example in this section shows how to create a subpartitioned external table.

In the following example, we create a table called `SALES_EXTENDED_EXT` that has access to the table `sales_extended.parquet`, and the table `t.dat` that has access to the object store `tab_from_csv_oss`.

Example 18-4 Using the ORACLE_BIGDATA Access Driver to create

```
CREATE TABLE "SALES_EXTENDED_EXT"
        ("PROD_ID" NUMBER(10,0),
     "CUST_ID" NUMBER(10,0),
     "TIME_ID" VARCHAR2(4000 BYTE),
     "CHANNEL_ID" NUMBER(10,0),
     "PROMO_ID" NUMBER(10,0),
     "QUANTITY_SOLD" NUMBER(10,0),
     "AMOUNT_SOLD" NUMBER(10,2),
     "GENDER" VARCHAR2(4000 BYTE),
     "CITY" VARCHAR2(4000 BYTE),
     "STATE_PROVINCE" VARCHAR2(4000 BYTE),
     "INCOME_LEVEL" VARCHAR2(4000 BYTE)
        )
        ORGANIZATION EXTERNAL
         ( TYPE ORACLE_BIGDATA
           DEFAULT DIRECTORY "DATA_PUMP_DIR"
           ACCESS PARAMETERS
           ( com.oracle.bigdata.credential.name=oss
             com.oracle.bigdata.fileformat=PARQUET
       )
           LOCATION
            ( 'https://objectstorage.eu-frankfurt-1.oraclecloud.com/n/
adwc4pm/b/parquetfiles/o//sales_extended.parquet'
            )
         )
        REJECT LIMIT UNLIMITED
       PARALLEL ;
```

```
CREATE TABLE tab_from_csv_oss
     (
      c0 number,
      c1 varchar2(20)
     )
     ORGANIZATION external
     (
      TYPE oracle_bigdata
      DEFAULT DIRECTORY data_pump_dir
      ACCESS PARAMETERS
       (
        com.oracle.bigdata.fileformat=csv
        com.oracle.bigdata.credential.name=oci_swift
       )
       location
       (
        'https://objectstorage.us-sanjose-1.oraclecloud.com/n/axffbtla8jep/b/
misc/o/t.dat'
       )
     )REJECT LIMIT 1
     ;
```

18.5 Loading LOBs From External Tables

External tables are particularly useful for loading large numbers of records from a single file, so that each record appears in its own row in the table.

The following example shows how to perform such a load.

Example 18-5 Loading LOBs From External Tables

Suppose you define an external table, `my_ext_table`, as follows:

```
CREATE TABLE my_ext_table ( id NUMBER, author VARCHAR2(30), created DATE,
text CLOB )
ORGANIZATION EXTERNAL (
  TYPE ORACLE_LOADER
  DEFAULT DIRECTORY MY_DIRECTORY
  ACCESS PARAMETERS (
    RECORDS DELIMITED BY 0x'0A'
    FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
      ( id         CHAR(10),
        author     CHAR(30),
        created    DATE "YYYY-MM-DD",
        text       CHAR(131071)
      )
    )
    LOCATION (
      MY_DIRECTORY:'external.dmp'
    )
);
```

The contents of the `external.dmp` file are as follows:

```
1,Roger,2015-08-08,The quick brown fox jumps over the lazy dog
2,John,2012-01-01,"The angry aligator, acting alone, ate the antelope"
```

The second line in the dump file requires quotation marks around the full text string; otherwise the field would be terminated at the comma.

> **Note:**
>
> Although not a problem in the dump file being used in this example, if something in the full text string contained quotation marks, then you would enclose it in another set of quotation marks, as follows for the word `alone`:
>
> ```
> 2,John,2012-01-01,"The angry aligator, acting ""alone"", ate the
> antelope"
> ```

If the full text might contain the record delimiter character (0x'0A', or newline), you can specify a separate file for each document. External tables do not support filler fields, so instead you must use a `COLUMN TRANSFORMS` clause to specify that the `fname` field contains the name of the external file:

```
DROP TABLE my_ext_table2;

CREATE TABLE my_ext_table2 ( id NUMBER, author VARCHAR2(30), created DATE,
text CLOB )
ORGANIZATION EXTERNAL (
  TYPE ORACLE_LOADER
  DEFAULT DIRECTORY MY_DIRECTORY
  ACCESS PARAMETERS (
    RECORDS DELIMITED BY 0x'0A'
    FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
      ( id         CHAR(10),
        author     CHAR(30),
        created    DATE "YYYY-MM-DD",
        fname      char(100)
      )
    COLUMN TRANSFORMS (text FROM LOBFILE(fname) FROM (MY_DIRECTORY) )
    )
    LOCATION (
      'loader.txt'
    )
);
```

> **Note:**
>
> The `FROM (MY_DIRECTORY)` clause is not actually necessary since it has already been specified as the default directory. However it is being shown here for example purposes because if the `loader.txt` file and the individual CLOB files were in different locations, it would be needed.

Once the data is in an external table, you can either leave it there and perform normal table operations (DML and most DDL) on the external table, or you can use the external table as a staging table to get the data into a normal table. To create a new normal (non-external) table, you could use the following SQL statement:

```
CREATE TABLE normaltable AS SELECT * FROM externaltable;
```

You can similarly use the following SQL statement to insert data into the new normal table:

```
INSERT INTO normaltable AS SELECT * FROM externaltable;
```

18.6 Loading CSV Files From External Tables

This topic provides examples of how to load CSV files from external tables under various conditions.

Some of the examples build on previous examples.

Example 18-6 Loading Data From CSV Files With No Access Parameters

This example requires the following conditions:

- The order of the columns in the table must match the order of fields in the data file.
- The records in the data file must be terminated by newline.
- The field in the records in the data file must be separated by commas (if field values are enclosed in quotation marks, then the quotation marks are *not* removed from the field).
- There cannot be any newline characters in the middle of a field.

The data for the external table is as follows:

```
events_all.csv
Winter Games,10-JAN-2010,10,
Hockey Tournament,18-MAR-2009,3,
Baseball Expo,28-APR-2009,2,
International Football Meeting,2-MAY-2009,14,
Track and Field Finale,12-MAY-2010,3,
Mid-summer Swim Meet,5-JUL-2010,4,
Rugby Kickoff,28-SEP-2009,6,
```

The definition of the external table is as follows:

```
SQL> CREATE TABLE EVENTS_XT_1
  2  (EVENT       varchar2(30),
  3   START_DATE  date,
  4   LENGTH      number)
```

```
  5  ORGANIZATION EXTERNAL
  6  (default directory def_dir1 location ('events_all.csv'));

Table created.
```

The following shows a `SELECT` operation on the external table `EVENTS_XT_1`:

```
SQL> select START_DATE, EVENT, LENGTH
  2    from EVENTS_XT_1
  3    order by START_DATE;

START_DAT EVENT                              LENGTH
--------- ------------------------------ ----------
18-MAR-09 Hockey Tournament                       3
28-APR-09 Baseball Expo                           2
02-MAY-09 International Football Meeting         14
28-SEP-09 Rugby Kickoff                           6
10-JAN-10 Winter Games                           10
12-MAY-10 Track and Field Finale                  3
05-JUL-10 Mid-summer Swim Meet                    4

7 rows selected.

SQL>
```

Example 18-7 Default Date Mask For the Session Does Not Match the Format of Data Fields in the Data File

This example is the same as the previous example, except that the default date mask for the session does not match the format of date fields in the data file. In the example below, the session format for dates is `DD-Mon-YYYY` whereas the format of dates in the data file is `MM/DD/YYYY`. If the external table definition does not have a date mask, then the `ORACLE_LOADER` access driver uses the session date mask to attempt to convert the character data in the data file to a date data type. ou specify an access parameter for the date mask to use for all fields in the data file that are used to load date columns in the external table.

The following is the contents of the data file for the external table:

```
events_all_date_fmt.csv
Winter Games,1/10/2010,10
Hockey Tournament,3/18/2009,3
Baseball Expo,4/28/2009,2
International Football Meeting,5/2/2009,14
Track and Field Finale,5/12/2009,3
Mid-summer Swim Meet,7/5/2010,4
Rugby Kickoff,9/28/2009,6
```

The definition of the external table is as follows:

```
SQL> CREATE TABLE EVENTS_XT_2
  2  (EVENT       varchar2(30),
  3   START_DATE  date,
  4   LENGTH      number)
  5  ORGANIZATION EXTERNAL
  6  (default directory def_dir1
```

```
  7   access parameters (fields date_format date mask "mm/dd/yyyy")
  8   location ('events_all_date_fmt.csv'));

Table created.

SQL>
```

The following shows a `SELECT` operation on the external table `EVENTS_XT_2`:

```
SQL> select START_DATE, EVENT, LENGTH
  2    from EVENTS_XT_2
  3    order by START_DATE;

START_DAT EVENT                              LENGTH
--------- ------------------------------ ----------
18-MAR-09 Hockey Tournament                       3
28-APR-09 Baseball Expo                           2
02-MAY-09 International Football Meeting         14
12-MAY-09 Track and Field Finale                  3
28-SEP-09 Rugby Kickoff                           6
10-JAN-10 Winter Games                           10
05-JUL-10 Mid-summer Swim Meet                    4

7 rows selected.
```

Example 18-8 Data is Split Across Two Data Files

This example is that same as the first example in this section except for the following:

- The data is split across two data files.
- Each data file has a row containing the names of the fields.
- Some fields in the data file are enclosed by quotation marks.

The `FIELD NAMES ALL FILES` tells the access driver that the first row in each file contains a row with names of the fields in the file. The access driver matches the names of the fields to the names of the columns in the table. This means the order of the fields in the file can be different than the order of the columns in the table. If a field name in the first row is not enclosed in quotation marks, then the access driver uppercases the name before trying to find the matching column name in the table. If the field name is enclosed in quotation marks, then it does not change the case of the names before looking for a matching name.

Because the fields are enclosed in quotation marks, the access parameter requires the `CSV WITHOUT EMBEDDED RECORD TERMINATORS` clause. This clause states the following:

- Fields in the data file are separated by commas.
- If the fields are enclosed in double quotation marks, then the access driver removes them from the field value.
- There are no new lines embedded in the field values (this option allows the access driver to skip some checks that can slow the performance of `SELECT` operations on the external table).

The two data files are as follows:

```
events_1.csv
```

```
"EVENT","START DATE","LENGTH",
"Winter Games", "10-JAN-2010", "10"
"Hockey Tournament", "18-MAR-2009", "3"
"Baseball Expo", "28-APR-2009", "2"
"International Football Meeting", "2-MAY-2009", "14"


events_2.csv

Event,Start date,Length,
Track and Field Finale, 12-MAY-2009, 3
Mid-summer Swim Meet, 5-JUL-2010, 4
Rugby Kickoff, 28-SEP-2009, 6
```

The external table definition is as follows:

```
SQL> CREATE TABLE EVENTS_XT_3
  2  ("START DATE"  date,
  3   EVENT         varchar2(30),
  4   LENGTH        number)
  5  ORGANIZATION EXTERNAL
  6  (default directory def_dir1
  7   access parameters (records field names all files
  8                      fields csv without embedded record terminators)
  9   location ('events_1.csv', 'events_2.csv'));

Table created.
```

The following shows the result of a `SELECT` operation on the `EVENTS_XT_3` external table:

```
SQL> select "START DATE", EVENT, LENGTH
  2    from EVENTS_XT_3
  3    order by "START DATE";

START DAT EVENT                              LENGTH
--------- ------------------------------ ----------
18-MAR-09 Hockey Tournament                       3
28-APR-09 Baseball Expo                           2
02-MAY-09 International Football Meeting         14
12-MAY-09 Track and Field Finale                  3
28-SEP-09 Rugby Kickoff                           6
10-JAN-10 Winter Games                           10
05-JUL-10 Mid-summer Swim Meet                    4

7 rows selected.
```

Example 18-9 Data Is Split Across Two Files and Only the First File Has a Row of Field Names

This example is the same as example 3 except that only the 1st file has a row of field names. The first row of the second file has real data. The RECORDS clause changes to "field names first file".

The two data files are as follows:

```
events_1.csv (same as for example 3)

"EVENT","START DATE","LENGTH",
"Winter Games", "10-JAN-2010", "10"
"Hockey Tournament", "18-MAR-2009", "3"
"Baseball Expo", "28-APR-2009", "2"
"International Football Meeting", "2-MAY-2009", "14"


events_2_no_header_row.csv

Track and Field Finale, 12-MAY-2009, 3
Mid-summer Swim Meet, 5-JUL-2010, 4
Rugby Kickoff, 28-SEP-2009, 6
```

The external table definition is as follows:

```
SQL> CREATE TABLE EVENTS_XT_4
  2  ("START DATE"  date,
  3   EVENT         varchar2(30),
  4   LENGTH        number)
  5  ORGANIZATION EXTERNAL
  6  (default directory def_dir1
  7   access parameters (records field names first file
  8                      fields csv without embedded record terminators)
  9   location ('events_1.csv', 'events_2_no_header_row.csv'));

Table created.
```

The following shows a `SELECT` operation on the `EVENTS_XT_4` external table:

```
SQL> select "START DATE", EVENT, LENGTH
  2    from EVENTS_XT_4
  3    order by "START DATE";

START DAT EVENT                              LENGTH
--------- ------------------------------ ----------
18-MAR-09 Hockey Tournament                       3
28-APR-09 Baseball Expo                           2
02-MAY-09 International Football Meeting         14
12-MAY-09 Track and Field Finale                  3
28-SEP-09 Rugby Kickoff                           6
10-JAN-10 Winter Games                           10
05-JUL-10 Mid-summer Swim Meet                    4

7 rows selected.
```

Example 18-10 The Order of the Fields in the File Match the Order of the Columns in the Table

This example has the following conditions:

- The order of the fields in the file match the order of the columns in the table.

- Fields are separated by newlines and are optionally enclosed in double quotation marks.
- There are fields that have embedded newlines in their value and those fields are enclosed in double quotation marks.

The contents of the data files are as follows:

```
event_contacts_1.csv

Winter Games, 10-JAN-2010, Ana Davis,
Hockey Tournament, 18-MAR-2009, "Daniel Dube
Michel Gagnon",
Baseball Expo, 28-APR-2009, "Robert Brown"
Internation Football Meeting, 2-MAY-2009,"Pete Perez
Randall Barnes
Melissa Gray",


event_contacts_2.csv

Track and Field Finale, 12-MAY-2009, John Taylor,
Mid-summer Swim Meet, 5-JUL-2010, "Louise Stewart
Cindy Sanders"
Rugby Kickoff, 28-SEP-2009, "Don Nguyen
Ray Lavoie"
```

The table definition is as follows. The `CSV WITH EMBEDDED RECORD TERMINATORS` clause tells the access driver how to handle fields enclosed by double quotation marks that also have embedded new lines.

```
SQL> CREATE TABLE EVENTS_CONTACTS_1
  2  (EVENT       varchar2(30),
  3   START_DATE  date,
  4   CONTACT     varchar2(120))
  5  ORGANIZATION EXTERNAL
  6  (default directory def_dir1
  7   access parameters (fields CSV with embedded record terminators)
  8   location ('event_contacts_1.csv', 'event_contacts_2.csv'));

Table created.
```

The following shows the result of a `SELECT` operation on the `EVENT_CONTACTS_1` external table:

```
SQL> column contact format a30
SQL> select START_DATE, EVENT, CONTACT
  2    from EVENTS_CONTACTS_1
  3    order by START_DATE;

START_DAT EVENT                          CONTACT
--------- ------------------------------ ------------------------------
18-MAR-09 Hockey Tournament              Daniel Dube
                                         Michel Gagnon

28-APR-09 Baseball Expo                  Robert Brown
02-MAY-09 Internation Football Meeting   Pete Perez
```

```
                                          Randall Barnes
                                          Melissa Gray

12-MAY-09 Track and Field Finale          John Taylor
28-SEP-09 Rugby Kickoff                   Don Nguyen
                                          Ray Lavoie

10-JAN-10 Winter Games                    Ana Davis
05-JUL-10 Mid-summer Swim Meet            Louise Stewart
                                          Cindy Sanders

7 rows selected.
```

Example 18-11 Not All Fields in the Data File Use Default Settings for the Access Parameters

This example shows what to do when most field in the data file use default settings for the access parameters but a few do not. Instead of listing the setting for all fields, this example shows how you can set attributes for just the fields that are different from the default. The differences are as follows:

- there are two date fields, one of which uses the session format, but `registration_deadline` uses a different format
- `registration_deadline` also uses a value of `NONE` to indicate a null value.

The content of the data file is as follows:

```
events_reg.csv

Winter Games,10-JAN-2010,10,12/1/2009,
Hockey Tournament,18-MAR-2009,3,3/11/2009,
Baseball Expo,28-APR-2009,2,NONE
International Football Meeting,2-MAY-2009,14,3/1/2009
Track and Field Finale,12-MAY-2010,3,5/10/010
Mid-summer Swim Meet,5-JUL-2010,4,6/20/2010
Rugby Kickoff,28-SEP-2009,6,NONE
```

The table definition is as follows. The `ALL FIELDS OVERRIDE` clause allows you to specify information for that field while using defaults for the remaining fields. The remaining fields have a data type of `CHAR(255)` and the field data is terminated by a comma with a trimming option of `LDRTRIM`.

```
SQL> CREATE TABLE EVENT_REGISTRATION_1
  2  (EVENT                varchar2(30),
  3   START_DATE           date,
  4   LENGTH               number,
  5   REGISTRATION_DEADLINE date)
  6  ORGANIZATION EXTERNAL
  7  (default directory def_dir1
  8   access parameters
  9  (fields all fields override
 10     (REGISTRATION_DEADLINE CHAR (10) DATE_FORMAT DATE MASK "mm/dd/yyyy"
 11                           NULLIF REGISTRATION_DEADLINE = 'NONE'))
 12   location ('events_reg.csv'));
```

```
Table created.
```

The following shows the result of a `SELECT` operation on the `EVENT_REGISTRATION_1` external table:

```
SQL> select START_DATE, EVENT, LENGTH, REGISTRATION_DEADLINE
  2    from EVENT_REGISTRATION_1
  3    order by START_DATE;

START_DAT EVENT                          LENGTH REGISTRAT
--------- ------------------------------ ---------- ---------
18-MAR-09 Hockey Tournament                     3 11-MAR-09
28-APR-09 Baseball Expo                         2
02-MAY-09 International Football Meeting       14 01-MAR-09
28-SEP-09 Rugby Kickoff                         6
10-JAN-10 Winter Games                         10 01-DEC-09
12-MAY-10 Track and Field Finale                3 10-MAY-10
05-JUL-10 Mid-summer Swim Meet                  4 20-JUN-10

7 rows selected.
```

Part IV
Other Utilities

Other Oracle data management utilities include the ADR Command Interpreter, `DBVERIFY`, Oracle LogMiner, the `DBMS_METADATA` API, and the legacy data movement utilities.

- Cloud Premigration Advisor Tool
  To evaluate the compatibility of the source database before you migrate to an Oracle Cloud database, use the Cloud Premigration Advisor Tool (CPAT).
- ADRCI: ADR Command Interpreter
  The Automatic Diagnostic Repository Command Interpreter (ADRCI) utility is a command-line tool that you use to manage Oracle Database diagnostic data.
- DBVERIFY: Offline Database Verification Utility
  DBVERIFY is an external command-line utility that performs a physical data structure integrity check.
- DBNEWID Utility
  DBNEWID is a database utility that can change the internal database identifier (DBID) and the database name (DBNAME) for an operational database.
- 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.
- Using the Metadata APIs
  Using the `DBMS_METADATA` APIs.
- Original Export
  The original Export utility (`exp`) was used to write data from Oracle Database into an operating system file in binary format.
- Original Import
  The original Import utility (`imp`) imports dump files that were created using the original Export utility (`exp`).

19

Cloud Premigration Advisor Tool

To evaluate the compatibility of the source database before you migrate to an Oracle Cloud database, use the Cloud Premigration Advisor Tool (CPAT).

- What is the Cloud Premigration Advisor Tool
  The Cloud Premigration Advisor Tool (CPAT) is a migration assistant that analyzes database metadata in an Oracle Database, and provides information to assist you to move data to Oracle Autonomous Database in Oracle Cloud.
- Prerequisites for Using the Cloud Premigration Advisor Tool
  Ensure that you have the required Java environment, user permissions and security set up to run the Cloud Premigration Advisor Tool (CPAT).
- Downloading and Configuring Cloud Premigration Advisor Tool
  Download the most recent update to the Cloud Premigration Advisor Tool (CPAT), extract it to a directory, and set up environment variables.
- Getting Started with the Cloud Premigration Advisor Tool (CPAT)
  After you download Oracle SQLcl or CPAT, ensure that your source database has the required Java home, set up environment variables, and decide what kinds of checks you want to perform.
- Connection Strings for Cloud Premigration Advisor Tool
  The Cloud Premigration Advisor Tool (CPAT) accepts standard Oracle JDBC format connection strings.
- Required Command-Line Strings for Cloud Premigration Advisor Tool
  Depending on your use case, some strings are required to run the Cloud Premigration Advisor Tool (CPAT).
- FULL Mode and SCHEMA Mode
  The Cloud Premigration Advisor Tool (CPAT) can run against the entire instance, or against a schema.
- Interpreting Cloud Premigration Advisor Tool (CPAT) Report Data
  Reports generated by CPAT contain summary information, and details for each check that is performed successfully.
- Command-Line Syntax and Properties
  Use the Cloud Premigration Advisor Tool (CPAT) properties to specify the checks and other operations you want to perform in CPAT command-line syntax.
- List of Checks Performed By the Premigration Advisor Tool
  Review information about the checks you find in a Premigration Advisor Tool report.
- Best Practices for Using the Premigration Advisor Tool
  These Cloud Premigration Advisor Tool (CPAT) tips can help you use CPAT more effectively.

19.1 What is the Cloud Premigration Advisor Tool

The Cloud Premigration Advisor Tool (CPAT) is a migration assistant that analyzes database metadata in an Oracle Database, and provides information to assist you to move data to Oracle Autonomous Database in Oracle Cloud.

The purpose of the Cloud Premigration Advisor Tool (CPAT) is to help plan successful migrations to Oracle Databases in the Oracle Cloud or on-premises. It analyzes the compatibility of the source database with your database target and chosen migration method, and suggests a course of action for potential incompatibilities. CPAT provides you with information to consider for different migration tools.

Running the Cloud Premigration Advisor Tool does not require any changes to the source database. It does not require adding users, or granting roles, or loading packages.

How the Cloud Premigration Advisor Tool Works

The Cloud Premigration Advisor Tool performs source database metadata checks, and provides you with information for your migration. It does not perform the actual migration. You use that information as part of your migration plan. CPAT runs using Java 7 or later releases, Java 8 Java Runtime Environment (JRE) preferred.

> **Note:**
>
> Installing and running CPAT does not modify Oracle Database. CPAT does not create any users, any packages, or require granting any roles or privileges. CPAT treats the database as `READ ONLY`.

A **check** is something that can be determined programmatically about a database, database object, user, or component. Checks are intended to determine the suitability of the database and database schema for moving to a particular Oracle Cloud Database deployment option. For example: Oracle Autonomous Database on Shared Exadata Infrastructure (ADB-S), using a particular migration method, such as Oracle Data Pump.

The source database is the database that you want to analyze for suitability to migrate to an Oracle Autonomous Database. The target is either a particular Oracle Autonomous Database, or a generic Oracle Autonomous Database deployment option that you can select when you run CPAT.

You start CPAT by running it either as Java command-line tool, or as a SQL command-line tool, using SQLcl. You then specify a source database and an Oracle Autonomous Database target, or specify `DEFAULT` for other Oracle Cloud Infrastructure (OCI) target databases, such as Exadata Cloud@Customer, Exadata Cloud Service, or an on-premises database. CPAT performs a number of checks on the source database and schema contents. These checks are guided by the target that you select, and your intended migration option.

After CPAT completes the source database checks, it generates a report indicating what was found. Reports contain both summary information and details for each check including the check result: **Passing**, **Review Suggested**, **Review Required**, or **Action Required**. In addition, CPAT identifies additional metadata in the source database that can be relevant for the migration.

The check results are compiled and presented in a report. The report can be a machine-readable report (`JSON`), a human-readable format (plain text, or HTML, or both). If the you do

not specifiy a specific report type on the command line with `--reportformat`, then by default CPAT will generate both Text and HTML reports. These reports can also be used directly by other Oracle migration products and features, such as Oracle Zero Downtime Migration (ZDM) Cloud Service, and the Oracle Cloud Infrastructure (OCI) Database Migration Service. You can specify

Premigration Advisor Tool Properties

You can specify how CPAT runs, and what checks it performs, by specifying properties in the command line to provide information for its analysis checks.

Cloud Premigration Advisor Tool Reports

CPAT recommends any relevant actions, such as using certain migration commands, setting certain database parameters, or performing SQL scripts on either the source or target instance, because the checks can be performed on target deployment options, as well as actual database targets, the reports use the term "Locus" instead of "Target" when something needs to be completed on either the Source or Target database. When the report recommends that you use particular parameters and commands, Oracle strongly recommends that you follow the guidance in the report.

Related Topics

- Cloud Premigration Advisor Tool (CPAT) Analyzes Databases for Suitability of Cloud Migration (Doc ID 2758371.1)

19.2 Prerequisites for Using the Cloud Premigration Advisor Tool

Ensure that you have the required Java environment, user permissions and security set up to run the Cloud Premigration Advisor Tool (CPAT).

Java Runtime Environment (JRE) Requirement

You must have Java 7 or later installed on the server or client where you run CPAT. Oracle recommends that you use Java 8 Java Runtime Environment (JRE).

CPAT looks for a JRE using the environment variables `JAVA_HOME` and `ORACLE_HOME`. If your source Oracle Database is later than Oracle 12c Release 1 (12.1.0.2), then a version of the Java JRE that can run CPAT is available in the Oracle home. If you are migrating from an earlier release of Oracle Database, or if you want to specify to use a later Java release Oracle home, then ensure that the environment variable is set to an appropriate Java home for CPAT.

If you use a thick Oracle Call Interface-based JDBC connect string, then CPAT currently expects the following environment variables to be set: `ORACLE_SID`, `ORACLE_HOME`, and `LD_LIBRARY_PATH`.

> **Note:**
>
> Oracle recommends that you set `ORACLE_SID`, `ORACLE_HOME`, and `LD_LIBRARY_PATH` by using the `oraenv` script available within the Oracle Database home.

More details on connect strings and associated environment variables can be found in the Advanced Usage Notes section titled Connection Strings.

User Privileges on the Source Database

When you specify a user to connect to the source database for checks, and provide that user with the CPAT `--username` property, the user name that you specify must be granted the `SELECT ANY DICTIONARY` privilege, and be granted `SELECT` on `SYSTEM.DUM$COLUMNS` and `SYSTEM.DUM$DATABASE`.

Access to the `DUM$` tables is needed only if the source and target character sets indicate that Oracle Database Migration Assistant for Unicode (DMU) is required.

> **Note:**
>
> Installing and running CPAT does not modify the Oracle Database. CPAT creates no users or packages, and CPAT does not grant any roles or privileges. The CPAT access to the database is `READ ONLY`. It only checks database metadata; no application or business data is checked.

Security Configuration

- Use the `--outdir` property to set the output location of CPAT logs and uses a secure location on your server or client.
- Set the user file creation mode mask (`umask`) on Linux and Unix systems so that the default values for the `r|w|x` privileges on CPAT scripts are restricted to authorized users.

19.3 Downloading and Configuring Cloud Premigration Advisor Tool

Download the most recent update to the Cloud Premigration Advisor Tool (CPAT), extract it to a directory, and set up environment variables.

To run CPAT, download the latest version from My Oracle Support, as described in the procedure.

If you cannot access My Oracle Support, then you can use Oracle SQLcl and the SQLcl command `- MIGRATEADVSOR`. You can download SQLcl from the following URL:

https://www.oracle.com/database/sqldeveloper/.

1. Read the My Oracle Support note about CPAT, and download and extract the CPAT patch from the following URL:

   Cloud Premigration Advisor Tool (CPAT) Analyzes Databases for Suitability of Cloud Migration (Doc ID 2758371.1).

   You require an Oracle account to log in to My Oracle Support.

2. Ensure that you have Java installed, and the `JAVA_HOME` user environment variable and other environment variables are set.

   After you download and unzip CPAT, ensure that you have an appropriate Java Runtime Environment (JRE) installed on the machine where CPAT is run. The minimum JRE version required for CPAT is Java 7.

   CPAT searches for a JRE home using the environment variables `JAVA_HOME` and `ORACLE_HOME`. If the version of Java in ORACLE_HOME is Java 6 or an earlier release,

which should only be the case with an Oracle Database 12g Release 1 or earlier home, then set `JAVA_HOME` to point to a Java 7 (or higher) JRE. To upgrade Java in an ORACLE_HOME, visit https://support.oracle.com and search for Document 2366614.1 (patch id 25803774) for Oracle Database 11g databases, or Document 2495017.1 (patch id 27301652) for Oracle Database 12.1 databases.

To set `JAVA_HOME` on a Microsoft Windows system:

a. Right click My Computer and select Properties.

b. On the Advanced tab, select Environment Variables, and then edit `JAVA_HOME` to point to the location of the of the Java Runtime Environment (JRE).

   For example:

   ```
   C:\Program Files\Java\jdk1.8\jre
   ```

   JRE is part of the Java Development Kit (JDK), but you can download it separately.

To set `JAVA_HOME` on a Linux or Unix system (Korn or Bash shell):

```
export JAVA_HOME=jdk-install-dir
export PATH=$JAVA_HOME/bin:$PATH
```

> **Note:**
>
> On Linux and Unix, systems, Oracle recommends that you set the `ORACLE_SID`, `ORACLE_HOME`, and `LD_LIBRARY_PATH` variables using the `oraenv` script that comes with Oracle Database.

If you want to use CPAT without defining ORACLE_HOME, and you don't need to use the Oracle Call interface JDBC connection string, then ensure that `JAVA_HOME` is set to a Java 7 (or higher) JRE. When possible, Oracle recommends that you use a Java 8 or higher JRE. Among other benefits, the functionality included in `OJDBC8` jars simplifies wallet-based connections such as those used when connecting to Oracle Cloud instances.

Related Topics

- Cloud Premigration Advisor Tool (CPAT) Analyzes Databases for Suitability of Cloud Migration (Doc ID 2758371.1)

19.4 Getting Started with the Cloud Premigration Advisor Tool (CPAT)

After you download Oracle SQLcl or CPAT, ensure that your source database has the required Java home, set up environment variables, and decide what kinds of checks you want to perform.

The workflow for using the Cloud Premigration Advisor tool (CPAT) is as follows:

1. Determine the type of Cloud database to which you want to migrate.
2. Run CPAT to generate a CPAT properties file using the `gettargetprops`. This switch gathers the properties of the target database, if one has been created. The target

properties are used when analyzing the source database to focus, and limits the checks that are run to those required for the target database.

3. Run CPAT with the options required for your migration scenario. You can run CPAT to test different migration scenarios. If you do run CPAT repeatedly, then to distinguish between the tests, Oracle recommends using the `--outfileprefix` and `--outdir` switches to keep the outputs organized, and to keep reports from being overwritten.

The CPAT patch distribution kit contains `premigration.sh` for running CPAT on Linux and Unix platforms, and `premigration.cmd` for running CPAT on Microsoft Windows platforms. CPAT can be run from any host with network access to the database instance that you want to analyze.

> **Note:**
>
> Running the premigration script on the server doesn't modify Oracle Database. CPAT itself creates no users or packages, and requires granting no roles or privileges. CPAT treats the database as `READ ONLY`. It only checks database metadata; no application or business data is checked.

In this example, `premigration.sh` is used (use `premigration.cmd` on Microsoft Windows systems)

Example 19-1 Generating a CPAT Properties File

This example checks whether your source database is ready to migrate to an Oracle Autonomous Database Shared for Transaction Processing and Mixed Workloads (ATP-S), you generate a properties file for the requirements:

```
premigration.sh --connectstring \
'jdbc:oracle:thin:@db_tp_tunnel?TNS_ADMIN=/path/to/wallets/Wallet1' --
username ADMIN \
--gettargetprops --outdir migration
```

The output of that command is as follows:

```
Enter password for ADMIN user: Cloud Premigration Advisor Tool Version 22.10.0
Cloud Premigration Advisor Tool generated properties file location: /home/oracle/
migration/configprops/atps_premigration_advisor_analysis.properties
```

> **Note:**
>
> When CPAT is run with the `--username` switch, the Oracle user name you specify must have the `SELECT ANY DICTIONARY` privilege, and must be granted `SELECT` on `SYSTEM.DUM$COLUMNS` and `SYSTEM.DUM$DATABASE`. Access to the `DUM$` tables is needed only if the source and target character sets indicate that Oracle Database Migration Assistant for Unicode (DMU) is required.

19.5 Connection Strings for Cloud Premigration Advisor Tool

The Cloud Premigration Advisor Tool (CPAT) accepts standard Oracle JDBC format connection strings.

Using standard Oracle JDBC format connection strings means that you can use either the thick" or the "thin" Oracle JDBC driver for connections.

Table 19-1 Example JDBC Connection Strings

Connection Description	Connection String	Notes
Thin client	`jdbc:oracle:thin:@host:port:sid`	Replace the variables *host*, *port* and *sid* with the host the connection port, and the system identifier for your source.
Thin client with PDB Service	jdbc:oracle:thin:@*host*:*port*/*pdb-service-name*	Replace the variables *host*, *port* and *pdb-service-name* with the host the connection port, and the PDB service name for your source.
Thin with AWS RDS	`jdbc:oracle:thin:@database-1.xxx.us-east-1.rds.amazonaws.com:port:sid`	Consult the Amazon Web Services Relational Database (AWS RDS) documentation for instructions on finding your database's endpoint and port details.
Operating system authentication	`jdbc:oracle:oci:@`	The CPAT command line must also include the property `--sysdba`
Operating system authentication with PDB	`jdbc:oracle:oci:@`	The CPAT command line must also include the properties `--sysdba` and `--pdbname` *pdb-name*, where *pdb-name* is the name of the PDB.
Wallet-based with Java 8 JRE	`jdbc:oracle:thin:@service-name?TNS_ADMIN=path-to-wallet`	The `TNS_ADMIN` connection property specifies the following, represented by *path-to-wallet*: The location of `tnsnames.ora`. The location of Oracle Wallet (`ewallet.sso`, `ewallet.p12`) or Java KeyStore (JKS) files (`truststore.jks`, `keystore.jks`). The location of `ojdbc.properties`. This file contains the connection properties required to use Oracle Wallets or Java KeyStore (JKS). For more information about using a keystore, see the Oracle Autonomous Database documentation.

Additional Connection String Information

Using the `--pdbname` property is only required when the connection string is for `CDB$ROOT`.

If you use keystore connection strings such as `jdbc:oracle:thin:@service-name?TNS_ADMIN=path-to-wallet`, then JDBC requires that *one* of the following is true:

- An `ojdbc.properties` file is located in the Wallet directory, and it contains `oracle.net.wallet_location property` with a value such as `oracle.net.wallet_location=(SOURCE=(METHOD=FILE)(METHOD_DATA=(DIRECTORY=${TNS_ADMIN})))`
- The `JAVA_TOOL_OPTIONS` environment variable is set with the appropriate values, such as the following:

  ```
  export JAVA_TOOLS_OPTIONS='-Doracle.net.tns_admin=path-to-wallet-dir -Doracle.net.wallet_location=(SOURCE=(METHOD=FILE)(METHOD_DATA=(DIRECTORY=path-to-wallet-dir)))'
  ```

Related Topics

- Oracle Database Insider: Migrating from AWS RDS to Oracle Autonomous Database via Data Pump
- Using Oracle Autonomous Database on Shared Exadata Infrastructure: Using a JDBC URL Connection String with JDBC Thin Driver and Wallets

19.6 Required Command-Line Strings for Cloud Premigration Advisor Tool

Depending on your use case, some strings are required to run the Cloud Premigration Advisor Tool (CPAT).

When using CPAT to connect to a database for source analysis, there are three required properties in the command string: One that specifies the cloud target (`targetcloud`), one that specifies the connection string (`connectstring`), and a user authentication string, provided either with the `sysdba` or `username` property.

The first two command properties must always be

- `--targetcloud` *`type`* (or `-t` *`type`*), where *`type is the Oracle Cloud target type`*
- `--connectstring` *`jdbc-connect-string`*, or `-c` *`jdbc-connect-string`*, where *`jdbc-connect-string`* is the JDBC connection string you use to connect to the migration source Oracle Database.

The other required property provides user credentials, and so it depends on what user credentials you use to start the analysis:

- For operating system authentication by user account, or authorization on the local system by using the `SYS` user, you use`--sysdba`, or `-d`. This starts CPAT by connecting to the source database with `AS SYSDBA`. This authentication option is also required if you connect as a user that has been granted `SYSDBA` but not the other privileges required by CPAT.
- For authentication by user account, where you are not using a wallet or operating system authentication, use `--username` *`name`*, or `-u` *`name`*, where *`name`* is the user account name you use to log in to the source system. As it runs, CPAT prompts you for the password for that user. The user name that you provide must be a user account granted `SYSDBA` and `ADMIN` privileges.

  If you authenticate CPAT with the `username` property, then the Oracle user name that you specify must have the `SELECT ANY DICTIONARY` privilege, and must be granted `SELECT` on `SYSTEM.DUM$COLUMNS` and `SYSTEM.DUM$DATABASE`. Access to the `DUM$` tables is needed only if the source and target character sets indicate that Oracle Database Migration Assistant for Unicode (DMU) is required.

19.7 FULL Mode and SCHEMA Mode

The Cloud Premigration Advisor Tool (CPAT) can run against the entire instance, or against a schema.

FULL Mode

`FULL` mode is the default mode. In this mode, CPAT runs any check relevant to the migration methods and the Cloud target types you choose, and analyzes data in all schemas that are not maintained by Oracle. In `FULL` mode, `SCHEMA`, `INSTANCE`, and `UNIVERSAL` scope checks are run.

> **Note:**
>
> Even in `FULL` mode, CPAT by default excludes checking data in schemas known to be maintained by Oracle. The use of the `--excludeschemas` property does not change CPAT's default `FULL` mode.

SCHEMA Mode

SCHEMA mode is set with the `--schemas` property. When `--schemas` is set, and `--full` is not also specified, then CPAT runs in `SCHEMA` mode. In `SCHEMA` mode, `SCHEMA` and `UNIVERSAL` scope checks are run. `INSTANCE` scope checks are not run.

Controlling CPAT Modes

The CPAT mode is controlled by the use of two options properties:

- The schemas property (`--schemas '`*`schemaname`*`' ['`*`schemaname`*`''`*`schemaname`*`']`, runs checks against the schemas that you list, in a space-delimited schema name list of one or more schema names, where the names are specified within single straight quotes. In schema mode, `SCHEMA` and `UNIVERSAL` scope checks are run. `INSTANCE` scope checks are not run.
- The Full property (`--full`) runs checks against the entire source database instance.

If you do not specify a value for the `--schemas` property, then the default is `FULL` mode.

If you specify `--schemas` on the command line, then CPAT runs in `SCHEMA` mode unless you also specify `--full` in the command line. If both properties are used, then `SCHEMA`, `INSTANCE`, and `UNIVERSAL` scope checks are run, but only on the list of schemas in the `-schemas` list.

If a schema name is lowercase, mixed case, or uses special characters, then use double quotation marks as well as single quotation marks to designate the schema name. For example:

```
premigration.sh --schemas 'PARdUS' '"ComEDIT"' '"faciem.$meam"' --targetcloud
ATPS --connectstring jdbc-connect-string"
```

19.8 Interpreting Cloud Premigration Advisor Tool (CPAT) Report Data

Reports generated by CPAT contain summary information, and details for each check that is performed successfully.

Each check includes the following information in the Premigration Advisor report:

- **Description**: This field describes what the check is looking for, or why the check is being performed.
- **Impact**: This field describes the consequences of a result other than **Passing**.
- **Action**: This check describes what, if anything, you should do before migration to correct issues, if the check result is not **Passing**.

Each check CPAT runs is given a report status of **Passing**, **Review Suggested, Review Required**, or **Action Required**.

The overall result of the CPAT report will be the most severe result of all checks performed. For example, if 30 checks have the status **Passing**, one check has a **Review Required** status, then the overall result will be **Review Required**.

The current definitions of each of the CPAT check results are as follows:

Table 19-2 Premigration Advisor Tool (CPAT) Check Result Definitions

Check	Definition
Passing	Indicates that the migration should succeed, and that there should be no difference in behavior of applications.
Review Suggested	Indicates that migration should succeed, and that applications likely will have no functional difference. However, database administrators should evaluate each check with this status to look for potential issues before migration.
Review Required	Indicates that migration may succeed (at least in part), but that either you cannot expect everything to work exactly as it did in the source database, or that a database administrator must complete additional work after migration to bring the target instance into alignment with the source database.
Action Required	Indicates something that likely would cause the migration to be unsuccessful. Checks with this result typically must be resolved before attempting migration.
Failed	The Cloud Premigration Advisor was unable to complete its analysis. Please contact Oracle Support Services.

Note: A CPAT result of **Action Required** does not necessarily mean that, for instance, Oracle Data Pump import will terminate prematurely while importing the data. It means that there will likely be errors during import which can indicate not all data has been migrated. It is imperative that an administrator familiar with both the database and the applications supported by the database examine the results of any checks that are not **Passing**.

Why are Checks sometimes marked as "skipped"

Checks marked in the Premigration Advisor report as `Skipped` should have completed during the CPAT analysis for properties provided in the CPAT command (for example, `--targetcloud --migrationmethod`, or other report value), but were not run in this particular Premigration Advisor report.

Either one of these two cases are the cause of a "Skipped" status:

- The check *should* be run but it is impossible to run at the time the report is generated, either due to the current contents or configuration of the source database. In this case, the check result will be **Review Suggested** or more severe.
- The check does not need to be completed at the time of the report, due to the current contents or configuration of the source database. The check result in this case will be **Passing**.

19.9 Command-Line Syntax and Properties

Use the Cloud Premigration Advisor Tool (CPAT) properties to specify the checks and other operations you want to perform in CPAT command-line syntax.

- Premigration Advisor Tool Command-Line Syntax
  You run the Premigration Advisor Tool as a command-line shell script.
- Premigration Advisor Tool Command-Line Properties
  Review the Premigration Advisor Tool properties to construct a command tree and options for your Oracle Database migration scenario.

19.9.1 Premigration Advisor Tool Command-Line Syntax

You run the Premigration Advisor Tool as a command-line shell script.

Prerequisites

- You must have Java Development Kit (JDK) 7 or later installed in your source environment. Oracle recommends that you use Java 8 Runtime Environment (JRE).

  JDK 8 is installed with every release starting with Oracle Database 12c Release 2 (12.2). For any release earlier than 12.2, you must either run Premigration Advisor Tool (CPAT) using the Java release in the target Oracle Database, or you must install JDK 8 on your source database server.

Java File Path

Obtain the latest CPAT zip file from My Oracle Support. The application and deployment instructions for the application are available from My Oracle Support note 2758371.1. Because CPAT is a Java-based tool, it requires that an appropriate Java Runtime Environment (JRE) is installed on the machine where the tool is run.

For thin clients, CPAT searches for a Java Runtime Environment (JRE) using the environment variables `JAVA_HOME` and `ORACLE_HOME`. The JRE should be in one of these paths.

For thick clients, CPAT uses an Oracle Call Interface (OCI) based JDBC connect string. With this type of connection string, CPAT connects to the database typically by using the environment variables: `ORACLE_SID`, `ORACLE_HOME`, and `LD_LIBRARY_PATH`.

> **Note:**
>
> You only need to set the `ORACLE_SID` if you use operating system authentication for the user running CPAT. If necessary, the CPAT script can set `LD_LIBRARY_PATH` by itself, so in most cases, you only need to set an `ORACLE_HOME` environment variable.

Syntax

The Premigration Advisor Tool command syntax is case-sensitive. You can pass properties either as character strings or as text strings, as noted for each command property.

The syntax takes the following format, where *character* is a single case-sensitive character, *command-string* is a case-sensitive string, and *value* is an input option or value specified by the command property.

Shell command:

```
./premigration.sh [-character [value] | --command-string value]]
```

Multiple properties can be concatenated in the command syntax, using either the character flag or the full name of a property.

19.9.2 Premigration Advisor Tool Command-Line Properties

Review the Premigration Advisor Tool properties to construct a command tree and options for your Oracle Database migration scenario.

- analysisprops
  The Premigration Advisor Tool property `analysisprops` specifies the path and name of a properties file for the source database.
- connectstring
  The Premigration Advisor Tool property `connectstring` provides the JDBC connect string for the source database.
- excludeschemas
  The Premigration Advisor Tool property `excludeschemas` specifies a list of schemas that you want to exclude from analysis for migration.
- full
  The Premigration Advisor Tool (CPAT) property `full` specifies that the full set of checks are run, even when `--schemas` is used.
- gettargetprops
  The Premigration Advisor Tool property `gettargetprops` reads the connection properties for the migration target database instance for analysis against the source database instance.
- help
  The Premigration Advisor Tool property `help` prints out the command line help information, and exits.
- logginglevel
  The Premigration Advisor Tool property `logginglevel` specifies the level of issues recorded in the logging file.
- maxrelevantobjects
  The Premigration Advisor Tool property `maxrelevantobjects` specifies the maximum number of relevant objects included in all reports.
- maxtextdatarows
  The Premigration Advisor Tool property `maxtextdatarows` specifies a limit to the number of relevant object rows displayed in text reports (does not apply to JSON reports).

- migrationmethod
  The Premigration Advisor Tool property `migrationmethod` specifies the type of method or tooling that you intend to use to migrate to Oracle Cloud.
- outdir
  The Premigration Advisor Tool property `outdir` specifies the directory path where you want premigration analysis log files and report files to be generated.
- outfileprefix
  The Premigration Advisor Tool property `outfileprefix` specifies a prefix for the Premigration Advisor Tool reports.
- pdbname
  The Premigration Advisor Tool property `pdbname` specifies the name of a source PDB on a CDB for which you want CPAT to generate a report.
- reportformat
  The Premigration Advisor Tool (CPAT) property `reportformat` specifies the format of CPAT report output.
- schemas
  The Premigration Advisor Tool property `schemas` specifies a list of schemas that you want to analyze for migration.
- sqltext
  The Premigration Advisor Tool property `sqltext` specifies to show the SQL used for CPAT checks in TEXT reports
- sysdba
  The Premigration Advisor Tool property `sysdba` is used to force `AS SYSDBA` when connecting to the database.
- targetcloud
  The Premigration Advisor Tool property `targetcloud` specifies the type of Oracle Cloud database to which you want to migrate.
- username
  The Premigration Advisor Tool property `username` specifies the username to use when connecting to the source database.
- version
  The Premigration Advisor Tool property `version` prints out the current version of CPAT, and then exits.
- updatecheck
  The Premigration Advisor Tool property `updatecheck` prints the current version of CPAT, checks to see if there is a more recent version available, and then exits.

19.9.2.1 analysisprops

The Premigration Advisor Tool property `analysisprops` specifies the path and name of a properties file for the source database.

Property	Description
property type	character, string
Syntax	`-a|--analysisprops --property-file-name`

Description

The Premigration Advisor Tool `analysisprops` property specifies the path and name of a properties file that you have generated previously for the source database by using the Premigration Advisor Tool command-line property `--gettargetprops`. You use this properties file with the Premigration Advisor Tool to analyze properties of the database .

Usage Notes

In the command string, you must also specify the options `--connectString` (`-c`) to the source database, and `--targetcloud` (`-t`) to specify the type of Cloud database to which you want to migrate.

Examples

In this example, you obtain the properties file `premigration_advisor_analysis.properties` from the target instance, and identify that file to use with `analysisprops`:

```
./premigration.sh --connectstring jdbc:oracle:oci:@ --targetcloud ATPD --
sysdba \
--analysisprops premigration_advisor_analysis.properties
```

19.9.2.2 connectstring

The Premigration Advisor Tool property `connectstring` provides the JDBC connect string for the source database.

Property	Description
property type	character, string
Syntax	`-c, --connectstring` *`connect-string`* `[--pdbname` *`pdb-name`*`]`
Default value	None

Description

The `connectstring` property specifies the JDBC connect string for the source database. If the connect string is for a CDB, then you must also specify a PDB name using the `--pdbname` switch, using `--pdbname` *`pdb-name`*, where *`pdb-name`* is the name of the PDB containing the source database.

CPAT connections have the following steps:

1. Connect to and obtain properties from the target instance using primigration.sh. This connection requires connnection information for the target instance, but does not require `--targetcloud`. It is this step that creates the `premigration_advisor_analysis` properties file. `connectstring` is required.
2. If necessary, connect to the computer where you will analyze the source instance, and copy the `premigration_advisor_analysis.properties` file to that computer.
3. Generate a CPAT report by running `premigration.sh` with the connection information for the source instance.

If you have a properties file that has Cloud service/lockdown information about the target, then `--targetcloud` is not required. If you do not provide a properties file, or if the properties file doesn't specify the Cloud service, then to obtain the most relevant information, you must use `--targetcloud` or `-t` to specify a target cloud. If you don't specify a target cloud using `--targetcloud` or `-t`, then the default is a Cloud target with no known Cloud service/lockdown profile set on the PDB target.

> **Note:**
>
> The restrictions enforced by a lockdown profile are for the entire PDB, and affect all users on that PDB, including `SYS` and `SYSTEM`.

Examples

In the following example, the PDB name is `sales1`, and *`connect-string`* indicates where the connection string is placed.

```
premigration.sh -c connect-string --pdbname sales1
```

19.9.2.3 excludeschemas

The Premigration Advisor Tool property `excludeschemas` specifies a list of schemas that you want to exclude from analysis for migration.

Property	Description
property type	string

Syntax	`--excludeschemas schemaname ['schemaname' 'schemaname' ...]`
	where `schemaname` is the name of one or more schema names, separated by spaces.
	Schema names are assumed to be case sensitive. For example, use `SYSTEM`, not `system`. If a schema name is lowercase, mixed case, or uses special characters, then use double quotation marks as well as single quotation marks to designate the schema name. For example:
	`--excludeschemas '"MixedCase"' '"Special.Char$"'`

Description

The Premigration Advisor Tool `excludeschemas` property specifies the schemas that you want to *exclude* from analysis for their readiness to migrate to the Cloud.

Usage Notes

Use to indicate the schemas on which you do not want premigration checks to be performed. If `excludeschemas` is omitted, and `schemas` is not used, then all schemas in the database will be analyzed. The `excludeschemas` property cannot be used in conjunction with `schemas`.

In the command string, you must also specify the options `--connectString` (`-c`) to the source database, and `--targetcloud` (`-t`) to specify the type of Cloud database to which you want to migrate.

19.9.2.4 full

The Premigration Advisor Tool (CPAT) property `full` specifies that the full set of checks are run, even when `--schemas` is used.

Property	Description
property type	character, string
Syntax	`-f|--full`

Description

Each CPAT check has a defined scope. If the scope of a check is `INSTANCE`, then that check will not be run unless you override that defined scope by selecting `FULL`. The CPAT `full` property forces the full set of checks to be run on the source database, even when `--schemas` has also been specified in the command string to limit the scope of checks.

Usage Notes

The option you use with CPAT should also be used with Oracle Data Pump. If you intend to use Oracle Data Pump with `FULL` mode, then you should run CPAT with the `full` property. If you intend to use Oracle Data Pump in `SCHEMA` mode, then run CPAT in `schema` mode.

Examples

Suppose you have 100 schemas in your source database instance, but you want to migrate only three schemas, s1, s2 and s3, to Autonomous Transaction Processing Dedicated (ATP-D).

In this case, you do not need to analyze all the schemas, but you do want to run `INSTANCE SCOPED` checks on all three schemas. You can do this by running CPAT with `--schemas s1 s2 s3 --full`

19.9.2.5 gettargetprops

The Premigration Advisor Tool property `gettargetprops` reads the connection properties for the migration target database instance for analysis against the source database instance.

Property	Description
property type	string
Syntax	`-g|--gettargetprops property`

Description

The Premigration Advisor Tool `gettargetprops` property specifies that CPAT collects the connection parameters for the migration target instance. CPAT collects properties of the migration target instance, so that it can then analyze those properties on the source database instance.

Usage Notes

These properties are typically set by tools that use CPAT in their migration flow, and use these properties to specify to CPAT that certain migration operations have been or will be performed

during migration. Generate the properties file with the `--gettargetprops` switch and `targetconnection` parameters

For more information, run `premigration.sh --help`, or `premigration.com --help` on Microsoft Windows systems.

Examples

```
./premigration.sh --gettargetprops --connectstring
jdbc:oracle:thin:@atpd_high?TNS_ADMIN=/path/wallet . . .
```

19.9.2.6 help

The Premigration Advisor Tool property `help` prints out the command line help information, and exits.

Property	Description
property type	string
Syntax	`-h|--help`

Description

The Premigration Advisor Tool `help` property prints out the command-line help instructions, and causes the advisor to exit.

Usage Notes

Use this option to obtain help information about the version of the Premigration Advisor Tool that you are running.

Examples

```
premigration.sh --help
```

19.9.2.7 logginglevel

The Premigration Advisor Tool property `logginglevel` specifies the level of issues recorded in the logging file.

Property	Description
property type	string
Syntax	`-l|--logginglevel -[severe|warning|info|config|fine|finer|finest]`
Default	If you do not provide this property in the command string, then the default is `fine`.

Description

The Premigration Advisor Tool `logginglevel` property specifies the severity of issues that you want to have logged in the Premigration Advisor Tool Report

Usage Notes

Use to indicate which type of checks you want to perform on the target database or databases. Log properties:

- `severe`
- `warning`
- `info`
- `config`
- `fine`
- `finer`
- `finest`

19.9.2.8 maxrelevantobjects

The Premigration Advisor Tool property `maxrelevantobjects` specifies the maximum number of relevant objects included in all reports.

Property	Description
property type	string
Syntax	`-M|--maxrelevantobjects` *`maximum-relevant-objects`*

Description

The Premigration Advisor Tool `maxrelevantobjects` property specifies the maximum number of relevant objects displayed in premigration advisor reports, specified by a numeric value. For `TEXT` reports, this property overrides the `maxtextdatarows` property.

> **Note:**
>
> If you specify a limit to the number of objects reported, then there can be objects that can affect your migration that are not published in reports.

Usage Notes

The purpose of this property is to place limits on the report that CPAT generates:

- Limit the size of a CPAT report
- Limit the memory CPAT uses
- Exclude inclusion of objects that may contain proprietary or confidential table, column or other information in the report.

Examples

```
premigration.sh  -maxrelevantobjects 5 -outfileprefix limit -targettype adws -
analysisprops /usr/example/CPAT/
cloud_premigration_advisor_analysis.properties
```

19.9.2.9 maxtextdatarows

The Premigration Advisor Tool property `maxtextdatarows` specifies a limit to the number of relevant object rows displayed in text reports (does not apply to JSON reports).

Property	Description
property type	string
Syntax	`-n|--maxtextdatarows` *`maximum-number-of-data-rows`*
Default	All rows in data tables (no maximum).

Description

The Premigration Advisor Tool `maxtextdatarows` property specifies the maximum number of relevant object rows that are included in the `TEXT` reports, and provides a message indicating that rows after the maximum row number is reached are not displayed. If this property is not set, then all relevant objects are included (no maximum). This property does not apply to JSON reports

Usage Notes

Where there is a conflict in property settings, `maxrelevantobjects` overrides the setting for `maxtextdatarows` for Premigration Advisor TEXT report files.

Examples

19.9.2.10 migrationmethod

The Premigration Advisor Tool property `migrationmethod` specifies the type of method or tooling that you intend to use to migrate to Oracle Cloud.

Property	Description
property type	string
Syntax	`-m|--migrationmethod -['datapump'|'goldengate']`
Default	If no value is supplied, then the default is `datapump`.

Description

The Premigration Advisor Tool `migrationmethod` property specifies the type of migration method or tooling that you intend to use to migrate databases to the Cloud. The migration method is used to influence what checks are done on the source database. Anything found in the source database that is incompatible with the migration method will be included in the generated report.

Usage Notes

Use to indicate which type of checks you want to perform on the target database or databases.

Option	Description
`datapump`	Specifies that the Preupgrade Advisor Tool performs checks for using Oracle Data Pump to perform migrations to the Oracle Cloud deployment you select.

Option	Description
`goldengate`	Specifies that the Preupgrade Advisor Tool performs checks for using Oracle GoldenGate to perform migrations to the Oracle Cloud deployment you select.

Examples

In the following example, *`connect-string`* indicates where the connection string is placed. The target Oracle Cloud database is Autonomous Transaction Processing Shared, and the migration method selected is Oracle GoldenGate.

```
premigration.cmd --connectstring some-string --targetcloud atps --username
SYSTEM -migrationmethod 'goldengate'
```

19.9.2.11 outdir

The Premigration Advisor Tool property `outdir` specifies the directory path where you want premigration analysis log files and report files to be generated.

Property	Description
property type	string
Syntax	`-o|--outdir` *`directory-path`* where *`directory-path`* is the path for the log file and report directory.

Description

The Premigration Advisor Tool `outdir` property specifies where the log files and report files should be created.

Usage Notes

If the path you provide is not an absolute path then the Premigration Advisor Tool specifies the directory relative to the file path location from which CPAT was started. If you do not specify an output file name, then the default file name is `premigration`. CPAT creates the filename, if it does not exist.

Examples

In the following example, *`connect-string`* indicates where the connection string is placed. The target PDB is `trend1`, the Oracle Cloud database is Autonomous Data Warehouse Dedicated, and the output directory path is `/users/analytic/adwd-migr`.

```
premigration.cmd --connectstring connect-string --targetcloud adwd --username
SYSTEM --pdbname trend1 -outdir /users/analytic/adwd-migr
```

19.9.2.12 outfileprefix

The Premigration Advisor Tool property `outfileprefix` specifies a prefix for the Premigration Advisor Tool reports.

Property	Description
property type	string
Syntax	`-P|--outfileprefix` *`prefix-string`*

Description

The Premigration Advisor Tool `outfileprefix` property specifies a prefix that you want to place on the output reports generated for the source database. Without a prefix, the standard name for a Premigration Advisor Tool report or log is `premigration_advisor`.

Usage Notes

Use a prefix to distinguish different report outputs. For example, you can use a prefix to distinguish the reports for a database where you generate one report for a migration using Oracle GoldenGate, and another report for a migration using Oracle Data Pump, or generate separate reports for each of the PDBs in a CDB.

Examples

In the following example, the prefix string is `cdb4`, *`connect-string`* indicates where the connection string is placed, and the migration target Oracle Cloud database is Autonomous Transaction Processing Shared. The reports for this command are `cdb4_premigration_advisor_report.txt` and `cdb4_premigration_advisor.log`.

```
./premigration.sh -c connect-string --targetcloud atps -P cdb4
```

19.9.2.13 pdbname

The Premigration Advisor Tool property `pdbname` specifies the name of a source PDB on a CDB for which you want CPAT to generate a report.

Property	Description
property type	string
Syntax	`-p|--pdbname` *`pdbname`*

Description

The name of a PDB to connect to. Applicable only when the source database connect string is for a CDB.

Usage Notes

You only need to use this property when the source database connect string is for a CDB.

Examples

In the following example, connect-string indicates where the connection string is placed for the source CDB. The source PDB is `trend4`, and the target is an Oracle Cloud Autonomous Data Warehouse Dedicated database.

```
premigration.cmd --connectstring connect-string --targetcloud adwd --username
SYSTEM --pdbname trend4
```

19.9.2.14 reportformat

The Premigration Advisor Tool (CPAT) property `reportformat` specifies the format of CPAT report output.

Property	Description
property type	string
Syntax	`-r|--reportformat -format [format format]` where *format* is a report format. The CPAT supports a machine-readable report in JSON format, and human-readable formats HTML or TEXT. Multiple formats are space-delimited. If you do not specifiy a specific report type on the command line with --reportformat, then by default CPAT will generate both Text and HTML reports.

Description

At the time of this release, the Premigration Advisor Tool can generate reports in either JSON or text format. Use the `reportformat` property to specify which report outputs you require.

Usage Notes

Use to indicate which type of report output you want to generate. If this property is not specified, then the default is TEXT.

> **Note:**
>
> Oracle recommends that you specify both text and JSON reports, and that you always save reports and log files. If you encounter an issue during migration, then it is important to include all possible information to assist with the resolution of the issue, including the log file, and both the text and JSON reports.

Option	Description
`json`	Specifies that the Preupgrade Advisor Tool produces a report in JSON format.
`text`	Specifies that the Preupgrade Advisor Tool produces a report in text file format.

Examples

In the following example, report outputs in JSON and text formats are specified for a report where the target is an Oracle Cloud Autonomous Data Warehouse Dedicated database. The reports generated are `premigration_advisor_report.json` `premigration_advisor_report.txt`.

```
premigration.cmd --connectstring connect-string --targetcloud adwd --username
SYSTEM --sqltext
```

19.9.2.15 schemas

The Premigration Advisor Tool property `schemas` specifies a list of schemas that you want to analyze for migration.

Property	Description
property type	string
Syntax	`-s|--schemas 'schemaname' ['schemaname' 'schemaname' ...]` where `schemaname` is the name of one or more schema names, separated by spaces.

Description

The Premigration Advisor Tool `schemas` property specifies the schemas that you want to check for their readiness to migrate to the Cloud. The migration method is used to influence what checks are done on the source database. Anything found in the source database that is incompatible with the migration method will be included in the generated report.

Usage Notes

Use to restrict the report to a specific list of schemas on which you want to perform checks. In schema mode, `SCHEMA` and `UNIVERSAL` scope checks are run. `INSTANCE` scope checks are not run. If you do not specify `schemas`, and `excludeschemas` is not used, then the default is to run with the `full` property. All schemas in the database will be analyzed, except for the schemas managed by Oracle. This can result in your receiving a report that lists problems in schemas that you do not intend to migrate to the Cloud target.

> **Note:**
>
> The option you use with CPAT should also be used with Oracle Data Pump. If you intend to use Oracle Data Pump with `FULL` mode, then you should run CPAT with the `full` property. If you intend to use Oracle Data Pump in `SCHEMA` mode, then run CPAT in `schema` mode.

The `schemas` property cannot be used in conjunction with `excludeschemas`. Limiting the scope of schemas that you check can be particularly useful if the source instance hosts multiple applications, each of which you may want to migrate to different Oracle Autonomous Database instances.

> **Note:**
>
> If you specify the `--full` property, then it forces the full set of checks to be run on the source database, overriding the restrictions that otherwise are in force when you limit the scope of checks with `--schemas`.

Schema names are assumed to be case sensitive. For example, use `SYSTEM`, not `system`. If a schema name is lowercase, mixed case, or uses special characters, then use double quotation marks as well as single quotation marks to designate the schema name. For example:

```
--schemas '"MixedCase"' '"Special.Char$"'
```

Examples

In the following example, a report is generated for the schemas `ADMIN` and `MixedCase` where the target is an Oracle Cloud Autonomous Data Warehouse Dedicated database, and *`connect-string`* represents the connection string to the source database.

```
premigration.cmd --connectstring connect-string --targetcloud atps --username
ADMIN -s 'SYSTEM' '"MixedCase"'
```

19.9.2.16 sqltext

The Premigration Advisor Tool property `sqltext` specifies to show the SQL used for CPAT checks in TEXT reports

Property	Description
property type	string
Syntax	`-S|--sqltext`

Description

The Premigration Advisor Tool `sqltext` property overides the default to hide SQL that was run for CPAT checks in TEXT reports. This property does not apply to JSON reports. It does not take any options.

Usage Notes

CPAT performs checks on the database using SQL statements. CPAT reports can be generated in both TEXT and JSON format. By default the SQL that was executed for each check is *not* included in the TEXT report. To have the SQL shown in the TEXT report, you can use this parameter.

Examples

```
premigration.cmd --connectstring connect-string --targetcloud adwd --username
SYSTEM --sqltext
```

19.9.2.17 sysdba

The Premigration Advisor Tool property `sysdba` is used to force `AS SYSDBA` when connecting to the database.

Property	Description
property type	character, string
Syntax	`-d,--sysdba`

Description

The Premigration Advisor Tool `sysdba` property specifies that the Premigration Advisor Tool connects to the source database AS SYSDBA. .

Usage Notes

If you are using operating aystem authentication, or the SYS user then you must use `--sysdba`.You also must use `--sysdba` to connect as a user who has been granted `SYSDBA`, but not the other privileges required by CPAT to perform checks.

Examples

```
./premigration.sh --connectstring jdbc:oracle:oci:@ --targetcloud ATPD --
sysdba --analysisprops premigration_advisor_analysis.properties
```

19.9.2.18 targetcloud

The Premigration Advisor Tool property `targetcloud` specifies the type of Oracle Cloud database to which you want to migrate.

Property	Description
property type	string
Syntax	`-t | --targetcloud` *cloudtype*
Default	DEFAULT indicates a target with no known lockdown profile.

Description

This option is used The Premigration Advisor Tool `targetcloud` property specifies the type of Cloud database to which you want to migrate. In a configuration file, you can set this value to a different value for each database that you want to check.

Usage Notes

Use to identify the type of cloud to which you are migrating, which affects the kinds of checks performed on the source database.

Option	Description
`'ATPD'`	Oracle Autonomous Database Transaction Processing Dedicated
`'ATPS'`	Oracle Autonomous Database Serverless
`'ADWD'`	Oracle Autonomous Data Warehouse Dedicated
`'ADWS'`	Oracle Autonomous Data Warehouse Serverless.
'DEFAULT'	Use for targets such as Oracle Autonomous Database on Exadata Cloud@Customer or Oracle Autonomous Database Cloud Service, where typically there is no predefined lockdown profile

Examples

```
./premigration.sh --targetcloud atps --outfileprefix ATPS_RUN_01 --outdir /
path/CPAT_output --reportformat TEXT JSON ...
```

19.9.2.19 username

The Premigration Advisor Tool property `username` specifies the username to use when connecting to the source database.

Property	Description
property type	string
Syntax	`-u|--username` *user-name*

Description

The `--username` switch provides CPAT with the user to connect to the source database.

Usage Notes

The user name you specify must have the `SELECT ANY DICTIONARY` privilege, and be granted `SELECT` on `SYSTEM.DUM$COLUMNS` and `SYSTEM.DUM$DATABASE`. When connecting to the target database, use the `ADMIN` user, or another user with the `PDB_DBA` role.

Examples

```
premigration --connectstring jdbc:oracle:thin:@example.oracle.com:1521/
ORCLPDB1 --username ADMIN -t atps
```

19.9.2.20 version

The Premigration Advisor Tool property `version` prints out the current version of CPAT, and then exits.

Property	Description
property type	string
Syntax	`-v|--version`

Description

The Premigration Advisor Tool `version` property enables you to print out the version number of the Premigration Advisor Tool, and the date it was released.

Usage Notes

Use this option to obtain information about the version of the Preupgrade Advisor Tool that you are running.

Examples

```
premigration.sh -v
Premigration Advisor Application Version: 22.10.0 (production)
Build date: 2022/10/18 10:55:43
Build hash: 53950fd


premigration.com --version
Premigration Advisor Application Version: 22.10.0 (production)
Build date: 2022/10/18 10:55:43
Build hash: 53950fd
```

19.9.2.21 updatecheck

The Premigration Advisor Tool property `updatecheck` prints the current version of CPAT, checks to see if there is a more recent version available, and then exits.

Property	Description
property type	string
Syntax	`-U | --updatecheck`
Default value	None

Description

Checks to see if an updated version of Cloud Premigration Advisor Tool (CPAT) is available. If here is a newer version, it prints yes. If there is not a newer version, it prints no. After completing the checc, CPAT exits. Network access is required for a successful check.

The Premigration Advisor Tool `updatecheck` property checks Oracle Support to determine if an updated version of Cloud Premigration Advisor Tool (CPAT) is available.

Usage Notes

To use this property, you must have a network connection. If you do not have a network connection, then you receive the error `CPAT-4001: Error checking for latest available version of the Cloud Premigration Advisor Tool`. If your network is behind a firewall, then this switch must be used with an appropriate HTTPS proxy defined.

Example

```
export _JAVA_OPTIONS='-Dhttps.proxyHost=www-proxy.us.oracle.com -
Dhttps.proxyPort=80'
./premigration.sh --updatecheck
```

If you already have the latest version of CPAT, then you should see the following output:

```
Picked up _JAVA_OPTIONS: -Dhttps.proxyHost=www-proxy.us.oracle.com -
Dhttps.proxyPort=80There is no newer version available of the Cloud Premigration
Advisor Tool
```

19.10 List of Checks Performed By the Premigration Advisor Tool

Review information about the checks you find in a Premigration Advisor Tool report.

> **Note:**
>
> When you specify the source database and your migration target, the Premigration Advisor Tool performs the checks required for that migration scenario. Only the checks required for that scenario are performed. Your report provides responses to the migration scenario you specify when you start CPAT.

- dp_has_low_streams_pool_size
  The Premigration Advisor Tool check `dp_has_low_streams_pool_size` verifies the `STREAMS_POOL_SIZE` amount is large enough for Data Pump migrations to start and work efficiently.
- gg_enabled_replication
  The Premigration Advisor Tool check `gg_enabled_replication` notifies you that the initialization parameter `ENABLE_GOLDENGATE_REPLICATION` is not set on the source database.
- gg_force_logging
  The Premigration Advisor Tool check `gg_force_logging` indicates that forced logging of all transactions and loads during the migration is not set.
- gg_has_low_streams_pool_size
  The Premigration Advisor Tool check `gg_has_low_streams_pool_size` verifies that the `STREAMS_POOL_SIZE` amount is large enough for Oracle GoldenGate.
- gg_not_unique
  The Premigration Advisor Tool check `gg_not_unique` indicates that forced logging of all transactions and loads during the migration is not set.
- gg_not_unique_bad_col_no
  The Premigration Advisor Tool check `gg_not_unique_bad_col_no` finds tables that have no primary key and no non-nullable unique index.
- gg_not_unique_bad_col_yes
  The Premigration Advisor Tool check `gg_not_unique_bad_col_yes` finds tables that have no primary key, unique index, or key columns, including table columns defined with unbounded data types.
- gg_objects_not_supported
  The Premigration Advisor Tool check `gg_objects_not_supported` indicates that there are unsupported objects on the source database.
- gg_supplemental_log_data_min
  The Premigration Advisor Tool check `gg_supplemental_log_data_min` indicates that minimal supplemental logging is not enabled on the source database.
- gg_tables_not_supported
  The Premigration Advisor Tool check `gg_tables_not_supported_adb` indicates that some objects in the database cannot be replicated using Oracle GoldenGate.

- gg_tables_not_supported
  The Premigration Advisor Tool check `gg_tables_not_supported` indicates that some objects in the non-ADB database cannot be replicated using Oracle GoldenGate.
- gg_user_objects_in_ggadmin_schemas
  The Premigration Advisor Tool check `gg_user_objects_in_ggadmin_schemas` indicates the presence of user objects in schemas that have Oracle GoldenGate administrator privileges.
- has_absent_default_tablespace
  The Premigration Advisor Tool check `has_absent_default_tablespace` indicates that schema Owner default tablespaces are missing.
- has_absent_temp_tablespace
  The Premigration Advisor Tool check `has_absent_temp_tablespace` indicates that schema Owner temporary tablespaces are missing.
- has_active_data_guard_dedicated
  The Premigration Advisor Tool check `has_active_data_guard_dedicated` detects whether Active Data Guard is being used on the source instance.
- has_active_data_guard_serverless
  The Premigration Advisor Tool check `has_active_data_guard_serverless` detects whether Active Data Guard is being used on the source instance.
- has_basic_file_lobs
  The Premigration Advisor Tool check `has_basic_file_lobs` indicates BASICFILE LOBs are present in the schema, which are not supported with Oracle Autonomous Database.
- has_clustered_tables
  The Premigration Advisor Tool check `has_clustered_tables` indicates table clusters are present in the schema, which are not supported with Oracle Autonomous Database.
- has_columns_of_rowid_type
  The Premigration Advisor Tool check `has_columns_of_rowid_type` indicates tables with columns with `ROWID` data type that cannot be migrated.
- has_columns_with_local_timezone
  The Premigration Advisor Tool check `has_columns_with_local_timezone` indicates tables have local DBTIMEZONE columns that do not match the target instance DBTIMEZONE.
- has_columns_with_media_data_types_adb
  The Premigration Advisor Tool check `has_columns_with_media_data_types_adb` indicates tables with multimedia data type that cannot be migrated.
- has_columns_with_media_data_types_default
  The Premigration Advisor Tool check `has_columns_with_media_data_types_default` indicates tables with multimedia columns.
- has_columns_with_spatial_data_types
  The Premigration Advisor Tool check `has_columns_with_spatial_data_types` indicates there are spatial objects that are not fully supported.
- has_common_objects
  The Premigration Advisor Tool check `has_common_objects` indicates there are common objects in the database instance.
- has_compression_disabled_for_objects
  The Premigration Advisor Tool check `has_compression_disabled_for_objects` indicates there are tables or partitions lacking a `COMPRESSION` clause.

- has_csmig_schema
  The Premigration Advisor Tool check `has_csmig_schema` indicates the CSSCAN utility is installed and configured on the source database..
- has_data_in_other_tablespaces_dedicated
  The Premigration Advisor Tool check `has_data_in_other_tablespaces_dedicated` identifies data subject to tablespace restrictions when migrating to Oracle Autonomous Databases on Dedicated Infrastructure..
- has_data_in_other_tablespaces_serverless
  The Premigration Advisor Tool check `has_data_in_other_tablespaces_serverless` identifies data subject to tablespace restrictions when migrating to Oracle Autonomous Databases on Shared Infrastructure.
- has_db_link_synonyms
  The Premigration Advisor Tool check `has_db_link_synonyms` indicates the schema contains synonyms with database links.
- has_db_links
  The Premigration Advisor Tool check `has_db_links` indicates the schema contains synonyms with database links.
- has_dbms_credentials
  The Premigration Advisor Tool check `has_dbms_credentials` indicates the schema contains credentials that were not created with `DBMS_CLOUD.CREATE_CREDENTIAL`.
- has_dbms_credentials
  The Premigration Advisor Tool check `has_dbms_credentials` indicates the schema contains credentials that were not created with `DBMS_CLOUD.CREATE_CREDENTIAL`.
- has_directories
  The Premigration Advisor Tool check `has_directories` indicates that there are directories objects in the source database.
- has_enabled_scheduler_jobs
  The Premigration Advisor Tool check `has_enabled_scheduler_jobs` indicates that there are List scheduler jobs that may interfere with Oracle Data Pump export.
- has_external_tables_dedicated
  The Premigration Advisor Tool check `has_external_tables_dedicated` indicates that Non-Cloud Objects Storage External tables exist in the source database.
- has_external_tables_default
  The Premigration Advisor Tool check `has_external_tables_default` indicates that external tables cannot be migrated unless the `DIRECTORY` objects the tables rely on have been created.
- has_external_tables_serverless
  The Premigration Advisor Tool check `has_external_tables_serverless` indicates that there are non-cloud Objects Storage external tables in the source database.
- has_fmw_registry_in_system
  The Premigration Advisor Tool check `has_fmw_registry_in_system` indicates that the Fusion Middleware Schema Version Registry must be moved out of the `SYSTEM` schema before migration.
- has_illegal_characters_in_comments
  The Premigration Advisor Tool check `has_illegal_characters_in_comments` indicates that there are characters in table comments that are not legal in the databases character set.

- has_ilm_ado_policies
  The Premigration Advisor Tool check `has_ilm_ado_policies` indicates that Information Lifestyle Management (ILM) Automatic Data Optimization Policies (ADO) will not migrate.
- has_incompatible_jobs
  The Premigration Advisor Tool check `has_incompatible_jobs` indicates that Information Lifestyle Management (ILM) Automatic Data Optimization Policies (ADO) will not migrate.
- has_index_organized_tables
  The Premigration Advisor Tool check `has_index_organized_tables` indicates that Index Organized tables are present in the source database.
- has_java_objects
  The Premigration Advisor Tool check `has_java_objects` indicates that there are Java objects present in the source database.
- has_java_source
  The Premigration Advisor Tool check `has_java_source` indicates that there are Java sources present in the source database.
- has_libraries
  The Premigration Advisor Tool check `has_libraries` indicates that there are applications that require the `CREATE LIBRARY` statement.
- has_logging_off_for_partitions
  The Premigration Advisor Tool check `has_logging_off_for_partitions` indicates that there are Partitions using the `NOLOGGING` storage attribute.
- has_logging_off_for_subpartitions
  The Premigration Advisor Tool check `has_logging_off_for_subpartitions` indicates that there are Partitions using the `NOLOGGING` storage attribute.
- has_logging_off_for_tables
  The Premigration Advisor Tool check `has_logging_off_for_tables` indicates that there are tables using the `NOLOGGING` storage attribute.
- has_low_streams_pool_size
  The Premigration Advisor Tool check `has_low_streams_pool_size` indicates that Mining Models with unexpected or incorrect attributes are detected.
- has_noexport_object_grants
  The Premigration Advisor Tool check `has_noexport_object_grants` indicates that Oracle Data Pump is unable to export all object grants.
- has_oracle_streams
  The Premigration Advisor Tool check `has_oracle_streams` indicates that Oracle Streams is configured in the database.
- has_parallel_indexes_enabled
  The Premigration Advisor Tool check `has_parallel_indexes_enabled` indicates that `PARALLEL` clause indexes exist.
- has_profile_not_default
  The Premigration Advisor Tool check `has_profile_not_default` indicates that schemas exist whose `PROFILE` is not available on the target system.
- has_public_synonyms
  The Premigration Advisor Tool check `has_public_synonyms` indicates that Public Synonyms exist in source system schemas.

- has_refs_to_restricted_packages_dedicated
  The Premigration Advisor Tool check `has_refs_to_restricted_packages_dedicated` indicates that there are references to partially or completely unsupported packages.
- has_refs_to_restricted_packages_serverless
  The Premigration Advisor Tool check `has_refs_to_restricted_packages_serverless` indicates that there are references to partially or completely unsupported packages.
- has_refs_to_user_objects_in_sys
  The Premigration Advisor Tool check `has_refs_to_user_objects_in_sys` indicates that there are user schema objects dependent on `SYS` or `SYSTEM`.
- has_role_privileges
  The Premigration Advisor Tool check `has_role_privileges` indicates that some role privileges used in the source database are not available in the target database
- has_sqlt_objects_adb
  The Premigration Advisor Tool check `has_sqlt_objects_adb` indicates that `SQLTXPLAIN` objects are detected.
- has_sqlt_objects_default
  The Premigration Advisor Tool check `has_sqlt_objects_default` indicates that `SQLTXPLAIN` objects are detected that Oracle Data Pump does not export.
- has_sys_privileges
  The Premigration Advisor Tool check `has_sys_privileges` indicates that some system privileges in the source database are not available in the target database.
- has_tables_that_fail_with_dblink
  The Premigration Advisor Tool check `has_tables_that_fail_with_dblink` indicates that there are tables with `LONG` or `LONG RAW` data types
- has_tables_with_long_raw_datatype
  The Premigration Advisor Tool check `has_tables_with_long_raw_datatype` indicates that there are tables with `LONG` or `LONG RAW` data types
- has_tables_with_xmltype_column
  The Premigration Advisor Tool check `has_tables_with_xmltype_column` indicates that there are tables with `XMLTYPE` columns.
- has_trusted_server_entries
  The Premigration Advisor Tool check `has_trusted_server_entries` indicates that there are`TRUSTED_SERVER` entries that cannot be recreated on Oracle Autonomous Database.
- has_unstructured_xml_indexes Check
  The Premigration Advisor Tool check `has_unstructured_xml_indexes` indicates that there are Unstructured XML Indexes.
- has_user_defined_objects_in_sys
  The Premigration Advisor Tool check `has_user_defined_objects_in_sys` indicates that there are User-defined objects in the `SYS` schema.
- has_user_defined_objects_in_system
  The Premigration Advisor Tool check `has_user_defined_objects_in_system` indicates that there are User-defined objects in the `SYSTEM` schema.
- has_user_defined_objects_no_quota
  The Premigration Advisor Tool check `has_user_defined_objects_no_quota` indicates that there are are objects in the source database that belong to users without quota.

- has_user_defined_pvfs
  The Premigration Advisor Tool check `has_user_defined_pvfs` indicates that there are User-defined Password Verification Functions.
- has_users_with_10g_password_version
  The Premigration Advisor Tool check `has_users_with_10g_password_version` indicates that there are user accounts using `10G` password version.
- has_xmlschema_objects
  The Premigration Advisor Tool check `has_xmlschema_objects` indicates that there are XML Schema Objects in the source database.
- has_xmltype_tables
  The Premigration Advisor Tool check `has_xmltype_tables` indicates that there are `XMLType` tables in the source database.
- modified_db_parameters_dedicated
  The Premigration Advisor Tool check `modified_db_parameters_dedicated` indicates that restricted initialization parameters are modified.
- modified_db_parameters_serverless
  The Premigration Advisor Tool check `modified_db_parameters_serverless` indicates that restricted initialization parameters are modified.
- nls_character_set_conversion
  The Premigration Advisor Tool check `nls_character_set_conversion` indicates that there are character codes on the source database that are invalid in Oracle Autonomous Database.
- nls_national_character_set
  The Premigration Advisor Tool check `nls_national_character_set` indicates that the `NCHAR` and `NVARCHAR2` lengths are different on the source and target databases.
- nls_nchar_ora_910
  The Premigration Advisor Tool check `nls_nchar_ora_910` indicates that the `NCHAR` and `NVARCHAR2` lengths are greater than the maximum length on the target databases.
- options_in_use_not_available_dedicated
  The Premigration Advisor Tool check `options_in_use_not_available_dedicated` indicates that unavailable database options are present on the source database.
- options_in_use_not_available_serverless
  The Premigration Advisor Tool check `options_in_use_not_available_serverless` indicates that unavailable database options are present on the source database.
- standard_traditional_audit_adb
  The Premigration Advisor Tool check `standard_traditional_audit_adb` indicates that Traditional Audit configurations are detected in the database.
- standard_traditional_audit_default
  The Premigration Advisor Tool check `standard_traditional_audit_default` indicates that Traditional Audit configurations are detected in the database.
- timezone_table_compatibility_higher_dedicated
  The Premigration Advisor Tool check `timezone_table_compatibility_higher_dedicated` indicates that the timezone setting is a more recent version on the source than on the target database.
- timezone_table_compatibility_higher_default
  The Premigration Advisor Tool check `timezone_table_compatibility_higher_default` indicates that the timezone setting is a more recent version on the source than on the target database.

- timezone_table_compatibility_higher_serverless
  The Premigration Advisor Tool check `timezone_table_compatibility_higher_serverless` indicates that the timezone setting is a more recent version on the source than on the target database.
- unified_and_standard_traditional_audit_adb
  The Premigration Advisor Tool check `unified_and_standard_traditional_audit_adb` indicates that Traditional Audit configurations are detected in the database.
- unified_and_standard_traditional_audit_default
  The Premigration Advisor Tool check `unified_and_standard_traditional_audit_default` indicates that Traditional Audit configurations are detected in the database.
- xdb_resource_view_has_entries Check
  The Premigration Advisor Tool check `xdb_resource_view_has_entries Check` indicates that there is an XDB Repository that is not supported in Oracle Autonomous Database. Entries in `RESOURCE_VIEW` will not migrate.

19.10.1 dp_has_low_streams_pool_size

The Premigration Advisor Tool check `dp_has_low_streams_pool_size` verifies the `STREAMS_POOL_SIZE` amount is large enough for Data Pump migrations to start and work efficiently.

Result Criticality

Runtime

Has Fixup

Yes

Scope

`UNIVERSAL`

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ADWS Autonomous Data Warehouse Shared
- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared
- Default (an Oracle Database instance that is not Oracle Autonomous Database)

Description

The Premigration Advisor Tool check `dp_has_low_streams_pool_size` verifies the `STREAMS_POOL_SIZE` has been preallocated to an amount is large enough for Oracle Data Pump migrations to start and work efficiently.

Effect

The database initialization parameter `STREAMS_POOL_SIZE` value helps determine the size of the Streams pool. You should allocate sufficient memory to `STREAMS_POOL_SIZE` for the export. Failure to do this can reduce Oracle Data Pump export performance, or cause the export to

fail. Oracle recommends that you define a minimum value for `STREAMS_POOL_SIZE` in the source database before export.

Action

Run SQL to set `STREAMS_POOL_SIZE` to allocate memory for the export. For example:

```
ALTER SYSTEM SET streams_pool_size=256M SCOPE=BOTH
```

After allocating memory, restart your instance if necessary.

19.10.2 gg_enabled_replication

The Premigration Advisor Tool check `gg_enabled_replication` notifies you that the initialization parameter `ENABLE_GOLDENGATE_REPLICATION` is not set on the source database.

Result Criticality

Action required

Has Fixup

Yes

Scope

`UNIVERSAL`

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ADWS Autonomous Data Warehouse Shared
- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared
- Default (an Oracle Database instance that is not Oracle Autonomous Database)

Description

The Premigration Advisor Tool `gg_enabled_replication` check indicates that you have selected Oracle GoldenGate as your migration method, but the initialization parameter `ENABLE_GOLDENGATE_REPLICATION` is not set to `TRUE`.

Effect

For Oracle GoldenGate to perform data migration, the source database initialization parameter `ENABLE_GOLDENGATE_REPLICATION` must be set to `TRUE`. If it is not set, then the migration fails.

Action

Set `ENABLE_GOLDENGATE_REPLICATION` to `TRUE` in the database initialization file.

19.10.3 gg_force_logging

The Premigration Advisor Tool check `gg_force_logging` indicates that forced logging of all transactions and loads during the migration is not set.

Result Criticality

Review required

Has Fixup

Yes

Target Cloud

This is a default check. It applies to the following:

- ADWD Autonomous Data Warehouse Dedicated
- ADWS Autonomous Data Warehouse Shared
- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared
- Default (an Oracle Database instance that is not Oracle Autonomous Database)

Scope

`UNIVERSAL`

Description

Forced logging mode is not set on the source database. When force logging mode is set, this forces the logging of all transactions and loads, overriding any user or storage settings that indicate these transactions and loads should not be logged.

> **Note:**
>
> When the source instance is Oracle Autonomous Database, the gg_force_logging check is skipped..

Effect

If forced logging is not set, then source data in the Oracle GoldenGate Extract configuration may be missed during the migration.

Action

To enable forced logging at tablespace and database level, log in as `SYSDBA`, and turn on forced logging. For example:

```
SQL> alter database force logging;
Database altered.
```

19.10.4 gg_has_low_streams_pool_size

The Premigration Advisor Tool check `gg_has_low_streams_pool_size` verifies that the `STREAMS_POOL_SIZE` amount is large enough for Oracle GoldenGate.

Result Criticality

Runtime

Has Fixup

Yes

Scope

`UNIVERSAL`

Target Cloud

This is a default check. It applies to the following:

- ADWD Autonomous Data Warehouse Dedicated
- ADWS Autonomous Data Warehouse Shared
- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared
- Default (an Oracle Database instance that is not Oracle Autonomous Database)

Description

The Premigration Advisor Tool check `gg_has_low_streams_pool_size` verifies the `STREAMS_POOL_SIZE` has been preallocated to an amount is large enough for Oracle GoldenGate migrations to start and work efficiently.

Oracle GoldenGate Extract interacts with an underlying logmining server in the source database, and Replicat interacts with an inbound server in the target database.

The shared memory that is used by the servers comes from the Streams pool portion of the System Global Area (SGA) in the database.Therefore, you must set the database initialization parameter `STREAMS_POOL_SIZE` high enough to keep enough memory available for the number of Extract and Replicat processes that you expect to run in integrated mode. Note that Streams pool is also used by other components of the database (including Oracle Streams, Advanced Queuing, and Oracle Data Pump export/import), so take other components into account when sizing the Streams pool for Oracle GoldenGate.

By default, one Extract requests the logmining server to run with of 1GB. As a best practice, keep 25 percent of the Streams pool available. Therefor, for a single process the minimum STREAMS_POOL_SIZE would be 1.25 GB. For more information see Oracle Support Document ID 2078459.1 and the Oracle GoldenGate documentation.

Effect

Allocate sufficient memory to `STREAMS_POOL_SIZE` for Oracle GoldenGate processes. Failure to do this can reduce Oracle GoldenGate performance, or cause the Extract or Replicat to fail. Oracle recommends that you define a minimum value for `STREAMS_POOL_SIZE` in the source database before running Oracle GoldenGate

Action

Run SQL to set `STREAMS_POOL_SIZE` to allocate memory for Extract and Replicat, depending on the number of Oracle GoldenGate processes that will run. For example:

```
ALTER SYSTEM SET streams_pool_size=1250M SCOPE=BOTH;
```

After allocating memory, restart your instance if necessary.

19.10.5 gg_not_unique

The Premigration Advisor Tool check `gg_not_unique` indicates that forced logging of all transactions and loads during the migration is not set.

Criticality

Action required

Target Cloud

This is a default check. It applies to the following:

- ADWD Autonomous Data Warehouse Dedicated
- ADWS Autonomous Data Warehouse Shared
- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared
- Default (an Oracle Database instance that is not Oracle Autonomous Database)

Scope

`SCHEMA`

Description

This check applies to schemas for Oracle GoldenGate migrations on Oracle Database 19c. It identifies tables that have no primary key and no non-nullable unique index.

Effect

If there are tables without any uniqueness, then significant changes on these tables may cause GoldenGate to increasingly fall behind and not recover.

Action

To address this issue, do one of the following:

- Add a primary key to the listed tables.
- Quiesce the database as much as possible during migration.
- Migrate changes to the tables using other means, such as Oracle Data Pump.

19.10.6 gg_not_unique_bad_col_no

The Premigration Advisor Tool check `gg_not_unique_bad_col_no` finds tables that have no primary key and no non-nullable unique index.

Result Criticality

Review required

Has Fixup

No

Scope

`SCHEMA`

Target Cloud

This is a default check. It applies to the following:

- ADWD Autonomous Data Warehouse Dedicated
- ADWS Autonomous Data Warehouse Shared
- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared
- Default (an Oracle Database instance that is not Oracle Autonomous Database)

Migration Method

`GOLDENGATE`

Description

The Premigration Advisor Tool check `gg_not_unique_bad_col_no` finds tables that have no primary key and no non-nullable unique index.

High amounts of mutations on the tables identified in this check can cause GoldenGate replication to fall behind and never catch up. A full table scan is needed to replicate every `INSERT`, `UPDATE`, or `DELETE` operation.

Effect

If Oracle GoldenGate has to perform significant changes on these tables, then it can fall behind progressively as the replication continues, and not recover.

Action

To address this issue, do one of the following:

- Add a primary key to the listed tables
- Quiesce the database as much as possible during migration
- Migrate changes to the tables using another method, such as Oracle Data Pump

19.10.7 gg_not_unique_bad_col_yes

The Premigration Advisor Tool check `gg_not_unique_bad_col_yes` finds tables that have no primary key, unique index, or key columns, including table columns defined with unbounded data types.

Result Criticality

Action required

Has Fixup

No

Scope

`SCHEMA`

Target Cloud

This is a default check. It applies to the following:

- ADWD Autonomous Data Warehouse Dedicated
- ADWS Autonomous Data Warehouse Shared
- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared
- Default (an Oracle Database instance that is not Oracle Autonomous Database)

Description

The Premigration Advisor Tool check `gg_not_unique_bad_col_yes` finds tables that have no Primary Key, Unique Index or Key Columns. A **Problematic Column** indicates that the table has a column not useful in the predicate (`where` clause). The table column is defined using an unbounded data type, such as `LONG` or `BLOB`.

Effect

If there are tables without any uniqueness, and with unbounded data_types, then the table records cannot be uniquely identified and cannot be used for logical replication. These tables are not supported in the Oracle GoldenGate Guide for Oracle Databases, and cannot be migrated using Oracle GoldenGate

Action

To address this issue, if possible add a primary or unique key to the tables. If you cannot add a primary or uniquen key, then you must use some other method of migrating the tables, such as Oracle Data Pump.

19.10.8 gg_objects_not_supported

The Premigration Advisor Tool check `gg_objects_not_supported` indicates that there are unsupported objects on the source database.

Result Criticality

Action required

Has Fixup

No

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ADWS Autonomous Data Warehouse Shared
- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared
- Default (an Oracle Database instance that is not Oracle Autonomous Database)

Scope

`SCHEMA`

Description

This check applies to schemas for Oracle GoldenGate migrations. Objects exist on the source database that are not supported for migration with Oracle GoldenGate.

Effect

Typically, the objects listed under this check are not replicated successfully in the migration without additional configuration.

Action

Consult the Oracle GoldenGate documentation to see how objects with the listed `SUPPORT_MODE` values can be replicated successfully.

19.10.9 gg_supplemental_log_data_min

The Premigration Advisor Tool check `gg_supplemental_log_data_min` indicates that minimal supplemental logging is not enabled on the source database.

Result Criticality

Action required

Has Fixup

Yes

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated

- ADWS Autonomous Data Warehouse Shared
- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared
- Default (an Oracle Database instance that is not Oracle Autonomous Database)

Scope

`UNIVERSAL`

Description

This check applies to schemas for Oracle GoldenGate migrations. Minimal supplemental logging, a database-level option, is required for an Oracle source database when using Oracle GoldenGate. This configuration adds row chaining information, if any exists, to the redo log for update operations.

Effect

If minimal supplemental log data is not enabled, then Oracle GoldenGate cannot function.

Action

Log in as `SYSDBA`, and enable minimal supplemental logging on the source database. For example:

```
SQL> ALTER DATABASE ADD SUPPLEMENTAL LOG DATA;
```

19.10.10 gg_tables_not_supported

The Premigration Advisor Tool check `gg_tables_not_supported_adb` indicates that some objects in the database cannot be replicated using Oracle GoldenGate.

Result Criticality

Action required

Has Fixup

No

Target Cloud

This is a default check. It applies to the following:

- ADWD Autonomous Data Warehouse Dedicated
- ADWS Autonomous Data Warehouse Shared
- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared

Scope

`SCHEMA`

Description

This check applies to schemas for Oracle GoldenGate migrations. When objects in the source database cannot be replicated by Oracle GoldenGate, the report provides a list of those objects with this check message.

Effect

The listed objects will not be migrated with Oracle GoldenGate.

Action

At the time of the switchover, you must move the listed relevant objects to the target database using another migration method, such as Oracle Data Pump.

19.10.11 gg_tables_not_supported

The Premigration Advisor Tool check `gg_tables_not_supported` indicates that some objects in the non-ADB database cannot be replicated using Oracle GoldenGate.

Result Criticality

Action required

Has Fixup

No

Target Cloud

- Default (an Oracle Database instance that is not Oracle Autonomous Database, or ADB)

Scope

`SCHEMA`

Description

This check applies to schemas for Oracle GoldenGate migrations. When objects in the source database cannot be replicated by Oracle GoldenGate, the report provides a list of those objects with this check message.

Effect

The listed objects will not be migrated with Oracle GoldenGate.

Action

At the time of the switchover, you must move the listed relevant objects to the target database using another migration method, such as Oracle Data Pump.

19.10.12 gg_user_objects_in_ggadmin_schemas

The Premigration Advisor Tool check `gg_user_objects_in_ggadmin_schemas` indicates the presence of user objects in schemas that have Oracle GoldenGate administrator privileges.

Result Criticality

Action required

Has Fixup

No

Target Cloud

This is a default check. It applies to the following:

- ADWD Autonomous Data Warehouse Dedicated
- ADWS Autonomous Data Warehouse Shared
- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared
- Default (an Oracle Database instance that is not Oracle Autonomous Database)

Scope

`SCHEMA`

Description

This check applies to schemas for Oracle GoldenGate migrations. When user objects in schemas have Oracle GoldenGate administrator privileges, those schemas are listed in CPAT report. Oracle GoldenGate cannot migrate them.

Effect

The listed objects will not be migrated with Oracle GoldenGate.

Action

Exclude these schemas from the Oracle GoldenGate data migration. You must move the listed relevant objects to the target database using another migration method, such as Oracle Data Pump.

19.10.13 has_absent_default_tablespace

The Premigration Advisor Tool check `has_absent_default_tablespace` indicates that schema Owner default tablespaces are missing.

Result Criticality

Review required.

Has Fixup

No

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ATPD Autonomous Transaction Processing Dedicated
- Default (an Oracle Database instance that is not Oracle Autonomous Database)

Scope

`SCHEMA`

Description

This check applies to schemas for Oracle Data Pump and Oracle GoldenGate migrations. When CPAT detects that one or more schema Owner's default tablespace are missing, the schemas are listed in the report.

Effect

Schemas without a valid `DEFAULT TABLESPACE` cannot be created on the target instance due to ORA-00959 errors..

Action

If the schemas are no longer being used, then drop those schemas. However, if the schemas are being used, then either create a valid default tablespace for the schema, or define default tablespace by running a query on `DBA_TABLESPACE` to list all valid tablespace names, and select one as a valid default tablespace.

Related Topics

- DBA_TABLESPACES

19.10.14 has_absent_temp_tablespace

The Premigration Advisor Tool check `has_absent_temp_tablespace` indicates that schema Owner temporary tablespaces are missing.

Result Criticality

Review required.

Has Fixup

No

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ATPD Autonomous Transaction Processing Dedicated
- Default (an Oracle Database instance that is not Oracle Autonomous Database)

Scope

`SCHEMA`

Description

This check applies to schemas for Oracle Data Pump and Oracle GoldenGate migrations. When CPAT detects that one or more schema Owner's temporary tablespace are missing, the schemas are listed in the report.

Effect

For Oracle Autonomous Database Dedicated Infrastructure for Transaction Processing (ATPD) and Oracle Autonomous Database Dedicated Infrastructure for Data Warehouse (ADWD), unless the needed temporary tablespaces have been created before migration on the target the source database schemas without a valid `TEMPORARY TABLESPACE` cannot be created on the target instance due to ORA-00959 errors.

Action

Create the needed temporary tablespaces on the Oracle Autonomous Database Dedicated infrastructure before you start the migration, or use tablespace remapping parameters to map other tablespaces into the `TEMP` tablespace when you start migration tools. Oracle Zero Downtime Migration and Database Migration Service can perform tablespace precreation and mapping automatically as part of the migration.

Related Topics

- DBA_TABLESPACES

19.10.15 has_active_data_guard_dedicated

The Premigration Advisor Tool check `has_active_data_guard_dedicated` detects whether Active Data Guard is being used on the source instance.

Result Criticality

Review suggested.

Has Fixup

No

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ATPD Autonomous Transaction Processing Dedicated

Scope

`INSTANCE`

Description

This check detects whether Active Data Guard is being used on the source instance.

Effect

If applications or schemas that are being migrated depend on certain capabilities of Active Data Guard, then those applications may no longer work after migration.

Action

Consider using Autonomous Data Guard on your target Oracle Autonomous Database instance. For more information, and to evaluate the capabilities of Autonomous Data Guard, see "Protect Critical Databases from Failures and Disasters Using Autonomous Data Guard" in *Oracle Cloud Oracle Autonomous Database on Dedicated Exadata Infrastructure*.

Related Topics

- Protect Critical Databases from Failures and Disasters Using Autonomous Data Guard

19.10.16 has_active_data_guard_serverless

The Premigration Advisor Tool check `has_active_data_guard_serverless` detects whether Active Data Guard is being used on the source instance.

Result Criticality

Review suggested.

Has Fixup

No

Target Cloud

- ADWS Autonomous Data Warehouse Shared
- ATPS Autonomous Transaction Processing Shared

Scope

`INSTANCE`

Description

This check detects whether Active Data Guard is being used on the source instance.

Effect

If applications or schemas that are being migrated depend on certain capabilities of Active Data Guard, then those applications may no longer work after migration.

Action

Consider using Autonomous Data Guard on your target Oracle Autonomous Database instance. For more information, and to evaluate the capabilities of Autonomous Data Guard, see "Using Standby Databases with Autonomous Database for Disaster Recovery " in *Oracle Cloud Using Oracle Autonomous Database on Shared Exadata Infrastructure*.

Related Topics

- Using Standby Databases with Autonomous Database for Disaster Recovery

19.10.17 has_basic_file_lobs

The Premigration Advisor Tool check `has_basic_file_lobs` indicates BASICFILE LOBs are present in the schema, which are not supported with Oracle Autonomous Database.

Result Criticality

Review required.

Has Fixup

No

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ADWS Autonomous Data Warehouse Shared
- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared

Scope

`SCHEMA`

Description

This check applies to schemas for Oracle Data Pump and Oracle GoldenGate migrations. When CPAT detects that one or more schema Owner's temporary tablespace contain BASICFILE LOBs, the schemas are listed in the report. .

Effect

During migration, all `BASICFILE` LOBs are converted automatically to `SECUREFILE` LOBs at the time of the import.

Action

No action is required.

19.10.18 has_clustered_tables

The Premigration Advisor Tool check `has_clustered_tables` indicates table clusters are present in the schema, which are not supported with Oracle Autonomous Database.

Result Criticality

Review suggested.

Has Fixup

No

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ADWS Autonomous Data Warehouse Shared

- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared

Scope

`SCHEMA`

Description

This check applies to schemas for Oracle Data Pump and Oracle GoldenGate migrations. When CPAT detects that one or more schema s contain table clusters, the schemas are listed in the report. .

Effect

When tables are created with a `CLUSTER` clause on the Oracle Autonomous Database, the table is created as a regular table.

Action

No action is required. Consider doing some performance testing to ensure that there are no adverse effects.

19.10.19 has_columns_of_rowid_type

The Premigration Advisor Tool check `has_columns_of_rowid_type` indicates tables with columns with `ROWID` data type that cannot be migrated.

Result Criticality

Action required.

Has Fixup

Yes

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ATPD Autonomous Transaction Processing Dedicated

Scope

`SCHEMA`

Description

This check applies to schemas for Oracle Data Pump and Oracle GoldenGate migrations. The `ROWID` data type is not enabled by default in Oracle Autonomous Database on Dedicated Exadata Infrastructure deployments.

Effect

By default, columns with `ROWID` data type cannot be migrated to ATPD or ADWD.

Action

You can choose to enable the `ROWID` data type by setting the initialization parameter `ALLOW_ROWID_COLUMN_TYPE` to `true` on the target ADBD instance. However, if you do enable it, then be aware that `ROWID` columns are incompatible with rolling upgrade operations, and other internal operations that physically move a row. At a minimum, during upgrades, Oracle recommends that you suspend database activities involving `ROWID`. Applications using `ROWID` columns should introduce correctness validation to check for logical errors in the application if a row relocates.

19.10.20 has_columns_with_local_timezone

The Premigration Advisor Tool check `has_columns_with_local_timezone` indicates tables have local DBTIMEZONE columns that do not match the target instance DBTIMEZONE.

Result Criticality

Review required.

Has Fixup

Y

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ADWS Autonomous Data Warehouse Shared
- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared

Scope

`SCHEMA`

Description

This check applies to schemas for Oracle Data Pump and Oracle GoldenGate migrations. It identifies tables being migrated that have columns using `TIMESTAMP WITH LOCAL TIMEZONE` data types when the source instance `DBTIMEZONE` does not match the target instance `DBTIMEZONE`.

Effect

Migrated data will appear to be corrupted, as it will be interpreted with an incorrect time zone. This issue can cause unexpected data and other application issues.

Action

Set the `DBTIMEZONE` on the target instance to match the source instance. For example: `ALTER DATABASE SET TIME_ZONE = 'America/New_York';`

CPAT generates a fixup script for this action, called `alter_time_zone.sql`. After applying this fixup on the target instance, you must restart the instance.

19.10.21 has_columns_with_media_data_types_adb

The Premigration Advisor Tool check `has_columns_with_media_data_types_adb` indicates tables with multimedia data type that cannot be migrated.

Result Criticality

Action required.

Has Fixup

No

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ADWS Autonomous Data Warehouse Shared
- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared

Scope

`SCHEMA`

Description

This check applies to schemas for Oracle Data Pump and Oracle GoldenGate migrations. Multimedia object types such as those from `ORDSYS` cannot be used in Oracle Autonomous Database.

Effect

Migration of tables with multimedia columns will fail.

Action

Columns with media data types are not allowed in Oracle Autonomous Database. As an alternative, Oracle recommends that you consider using SecureFiles LOBs for media type storage.

Follow the instructions in the Oracle Multimedia README.txt file in *`Oracle_home`*`/ord/im/admin/README.txt`, or Oracle Support Document ID 2555923.1 to determine if Oracle Multimedia methods and packages are being used. If Oracle Multimedia is being used, then refer to Oracle Support Document ID 2347372.1 for suggestions on replacing Oracle Multimedia. Refer to Oracle Support Document ID 2375644.1 "How To Migrate Data From Oracle Multimedia Data Types to BLOB columns" for information on how to move data stored in Oracle Multimedia object types to SecureFiles LOBs.

Related Topics

- Desupport of Oracle Multimedia Component in Oracle 19c (Doc ID 2555923.1)
- Oracle Multimedia Statement of Direction (Doc ID 2347372.1)
- How To Migrate Data From Oracle Multimedia Data Types to BLOB columns (Doc ID 2375644.1)

19.10.22 has_columns_with_media_data_types_default

The Premigration Advisor Tool check `has_columns_with_media_data_types_default` indicates tables with multimedia columns.

Result Criticality

Action required.

Has Fixup

No

Scope

`SCHEMA`

Target Cloud

- Default (an Oracle Database instance that is not Oracle Autonomous Database)

Description

This check applies to schemas for Oracle Data Pump and Oracle GoldenGate migrations. Multimedia object types such as those from `ORDSYS` are desupported in Oracle Database 19c and later releases.

Effect

Migration of tables with multimedia columns can fail.

Action

Oracle Multimedia was desupported in Oracle Database 19c. Oracle recommends that you consider using SecureFiles LOBs for media type storage.

Follow the instructions in the Oracle Multimedia README.txt file in *`Oracle_home`*`/ord/im/admin/README.txt`, or Oracle Support Document ID 2555923.1 to determine if Oracle Multimedia methods and packages are being used. If Oracle Multimedia is being used, then refer to Oracle Support Document ID 2347372.1 for suggestions on replacing Oracle Multimedia. Refer to Oracle Support Document ID 2375644.1 "How To Migrate Data From Oracle Multimedia Data Types to BLOB columns" for information on how to move data stored in Oracle Multimedia object types to SecureFiles LOBs.

Related Topics

- Desupport of Oracle Multimedia Component in Oracle 19c (Doc ID 2555923.1)
- Oracle Multimedia Statement of Direction (Doc ID 2347372.1)
- How To Migrate Data From Oracle Multimedia Data Types to BLOB columns (Doc ID 2375644.1)

19.10.23 has_columns_with_spatial_data_types

The Premigration Advisor Tool check `has_columns_with_spatial_data_types` indicates there are spatial objects that are not fully supported.

Result Criticality

Review required.

Has Fixup

Yes

Target Cloud

- ADWS Autonomous Data Warehouse Shared
- ATPS Autonomous Transaction Processing Shared

Scope

`SCHEMA`

Description

This check applies to schemas for Oracle Data Pump and Oracle GoldenGate migrations. It indicates the presence of spatial data type objects.

Effect

Because some of the functionality of spatial objects are dependent on the Oracle Java (`JAVAVM`) feature, there can be objects not fully supported with Oracle Autonomous Databases on Shared Infrastructure until `JAVAVM` is enabled.

Action

Enable the JAVAVM feature on the target system by running this SQL, and then restart your instance:

```
BEGIN
   DBMS_CLOUD_ADMIN.ENABLE_FEATURE(
       feature_name => 'JAVAVM' );
END;
/
```

For more information on enabling the JAVAVM feature see "Using Oracle Java on Autonomous Database" in *Oracle Cloud Using Oracle Autonomous Database Serverless* For more information on using Spatial on ADB, see "Use Oracle Spatial with Autonomous Database" in *Oracle Cloud Using Oracle Autonomous Database Serverless*.

Related Topics

- Using Oracle Java on Autonomous Database
- Use Oracle Spatial with Autonomous Database

19.10.24 has_common_objects

The Premigration Advisor Tool check `has_common_objects` indicates there are common objects in the database instance.

Result Criticality

Action required.

Has Fixup

Yes

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ADWS Autonomous Data Warehouse Shared
- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared
- Default (an Oracle Database instance that is not Oracle Autonomous Database)

Scope

`INSTANCE`

Description

This is a default check. This check applies to source instances for Oracle Data Pump and Oracle GoldenGate migrations. It indicates the presence of common objects.

Effect

Oracle Data Pump does not migrate common objects to Oracle Autonomous Database in Oracle Cloud, and these objects are not supported on Oracle Autonomous Database (ADB). Anything dependent on the common objects will fail to be migrated properly.

Action

Those common objects needed by applications must be recreated on the target system before you start the migration. When targeting ADB, the common objects that you require must be recreated as local objects. This can be done using `DBMS_METADATA.GET_DDL`, as shown in Oracle Support Document ID 2739952.1

Related Topics

- How to Extract DDL for User including Privileges and Roles Using dbms_metadata.get_ddl (Doc ID 2739952.1)

19.10.25 has_compression_disabled_for_objects

The Premigration Advisor Tool check `has_compression_disabled_for_objects` indicates there are tables or partitions lacking a `COMPRESSION` clause.

Result Criticality

Review suggested.

Has Fixup

No

Target Cloud

- ADWS Autonomous Data Warehouse Shared
- ATPS Autonomous Transaction Processing Shared

Scope

`SCHEMA`

Description

This check applies to source schema for Oracle Data Pump and Oracle GoldenGate migrations. It indicates the presence of tables or partitions that do not have a `COMPRESSION` clause. Tables and Partitions must be compressed to `QUERY HIGH` in Oracle Autonomous Data Warehouse (ADW).

Effect

When migrating to ADW, if a table or partition SQL data definition language (DDL) statement does not contain a `COMPRESSION` clause, then it is created during the migration with a default compression of `QUERY HIGH`.

Action

No action required. To modify this behavior, either add a compression clause of your choice (or even `NOCOMPRESS`) before starting the export, or alter the compression clause after the import..

19.10.26 has_csmig_schema

The Premigration Advisor Tool check `has_csmig_schema` indicates the CSSCAN utility is installed and configured on the source database..

Result Criticality

Review suggested.

Has Fixup

Yes

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ADWS Autonomous Data Warehouse Shared
- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared
- Default (an Oracle Database instance that is not Oracle Autonomous Database)

Scope

`UNIVERSAL`

Description

This is a default check. The CSSCAN utility is no longer supported, and has been replaced by the Database Migration Assistant for Unicode (DMU) Tool..

Effect

Migration tools can ignore any objects, users, or roles related with CSSCAN utility.

Action

Remove the CSMIG user and any objects created by the CSSCAN utility: For example:

```
BEGIN FOR REC IN (SELECT SYNONYM_NAME FROM DBA_SYNONYMS WHERE TABLE_OWNER =
'CSMIG') LOOP
      EXECUTE IMMEDIATE 'DROP PUBLIC SYNONYM ' || REC.SYNONYM_NAME; END LOOP;
END; / DROP VIEW
      SYS.CSMV$KTFBUE; DROP USER CSMIG CASCADE;
```

Use The Database Migration Assistant for Unicode (DMU) Tool to scan for character set migration issues. For more information on DMU see Oracle Support Document ID 1272374.1

Related Topics

- The Database Migration Assistant for Unicode (DMU) Tool (Doc ID 1272374.1)

19.10.27 has_data_in_other_tablespaces_dedicated

The Premigration Advisor Tool check `has_data_in_other_tablespaces_dedicated` identifies data subject to tablespace restrictions when migrating to Oracle Autonomous Databases on Dedicated Infrastructure..

Result Criticality

Action required.

Has Fixup

No

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ATPD Autonomous Transaction Processing Dedicated

Scope

`SCHEMA`

Description

This check applies to source schema for Oracle Data Pump and Oracle GoldenGate migrations. It indicates the presence of data that is subject to tablespace restrictions when migrating to Autonomous Databases on Dedicated Infrastructure.

Effect

For ATPD and ADWD (Dedicated Infrastructure), errors are reported for tablespaces that have not been precreated on the target. If tablespace mapping is not employed, then errors can occur during migration.

Action

If you are migrating the database using either Zero Downtime Migration (ZDM) or Database Migration Service (DMS) then they precreate and map tablespaces automatically, so the issue does not occur.

If you are migrating using Oracle Data Pump manually, then specify `IGNORE=TABLESPACE` and `REMAP_TABLESPACE='%:DATA'` in your Data Pump `impdp` parameter file, so that other tablespaces into the `DATA` tablespace when starting migration tooling.

In all cases, you should assess your application for any dependencies on specific tablespace names.

19.10.28 has_data_in_other_tablespaces_serverless

The Premigration Advisor Tool check `has_data_in_other_tablespaces_serverless` identifies data subject to tablespace restrictions when migrating to Oracle Autonomous Databases on Shared Infrastructure.

Result Criticality

Action required.

Has Fixup

No

Target Cloud

- ADWS Autonomous Data Warehouse Shared
- ATPS Autonomous Transaction Processing Shared

Scope

`SCHEMA`

Description

This check applies to source schema for Oracle Data Pump and Oracle GoldenGate migrations. It indicates the presence of tables that have other tablespaces mapped into their table.

Effect

User-defined tablespaces are not allowed in ATPS and ADWS (Serverless Infrastructure). Each database in this cloud environment has a single 'DATA' tablespace. If tablespace mapping is not employed, and the user performing migration does not have privileges on the `DATA` tablespace, then errors can occur during migration.

Action

If you are migrating the database using either Zero Downtime Migration (ZDM) or Database Migration Service (DMS) then they precreate and map tablespaces automatically, so the issue does not occur.

If you are migrating using Oracle Data Pump manually, then specify `IGNORE=TABLESPACE` and `REMAP_TABLESPACE='%:DATA'` in your Data Pump `impdp` parameter file, so that other tablespaces into the `DATA` tablespace when starting migration tooling.

In all cases, you should assess your application for any dependencies on specific tablespace names.

19.10.29 has_db_link_synonyms

The Premigration Advisor Tool check `has_db_link_synonyms` indicates the schema contains synonyms with database links.

Result Criticality

Review suggested.

Has Fixup

Yes

Target Cloud

- ADWS Autonomous Data Warehouse Shared
- ATPS Autonomous Transaction Processing Shared

Scope

`SCHEMA`

Description

This check applies to source schema for Oracle Data Pump and Oracle GoldenGate migrations. Database links cannot be migrated.

Effect

After migration, applications relying on the synonym will fail until the database links are recreated.

Action

After migration is complete, create database links in the target Oracle Autonomous Database in using `DBMS_CLOUD_ADMIN.CREATE_DATABASE_LINK`, and then recreate the synonyms.

19.10.30 has_db_links

The Premigration Advisor Tool check `has_db_links` indicates the schema contains synonyms with database links.

Result Criticality

Review required

Has Fixup

No

Target Cloud

- ADWS Autonomous Data Warehouse Shared
- ATPS Autonomous Transaction Processing Shared

Scope

`SCHEMA`

Description

This check applies to source schema for Oracle Data Pump and Oracle GoldenGate migrations. Database links cannot be migrated.

Effect

After migration, applications relying on database links will fail until the database links are recreated.

Action

Precreate Database Links manually in ADB using `DBMS_CLOUD_ADMIN.CREATE_DATABASE_LINK` in the respective database schemas before migrating. The proper sequence of statements is as follows:

1. Create the schemas that own the links.
2. Create the links using `DBMS_CLOUD_ADMIN.CREATE_DATABASE_LINK`.
3. Import the schemas that you are migrating.

19.10.31 has_dbms_credentials

The Premigration Advisor Tool check `has_dbms_credentials` indicates the schema contains credentials that were not created with `DBMS_CLOUD.CREATE_CREDENTIAL`.

Result Criticality

Review required

Has Fixup

No

Target Cloud

- ADWS Autonomous Data Warehouse Shared
- ATPS Autonomous Transaction Processing Shared

Scope

`SCHEMA`

Description

This check applies to source schema for Oracle Data Pump and Oracle GoldenGate migrations. Credentials originally created with `DBMS_CREDENTIAL` or `DBMS_SCHEDULER` packages cannot be automatically migrated to Oracle Autonomous Database.

Effect

After migration, users with credentials originally created with `DBMS_CREDENTIAL` or `DBMS_SCHEDULER` packages receive ORA-27486: insufficient privileges errors. These credentials cannot be migrated automatically to ADBS.

Action

After migration is complete, verify that the listed credentials are still required on the target Oracle Autonomous Database instance. If these credentials are required, then recreate the credentials using `DBMS_CLOUD.CREATE_CREDENTIAL`. For more information, see My Oracle Support Document ID 2746284.1.

Related Topics

- Autonomous Database (Shared) - dbms_credential.create_credential failed with ORA-27486 (Doc ID 2746284.1)

19.10.32 has_dbms_credentials

The Premigration Advisor Tool check `has_dbms_credentials` indicates the schema contains credentials that were not created with `DBMS_CLOUD.CREATE_CREDENTIAL`.

Result Criticality

Review required

Has Fixup

No

Target Cloud

- ADWS Autonomous Data Warehouse Shared
- ATPS Autonomous Transaction Processing Shared

Scope

`SCHEMA`

Description

This check applies to source schema for Oracle Data Pump and Oracle GoldenGate migrations. Credentials originally created with `DBMS_CREDENTIAL` or `DBMS_SCHEDULER` packages cannot be automatically migrated to Oracle Autonomous Database.

Effect

After migration, users with credentials originally created with `DBMS_CREDENTIAL` or `DBMS_SCHEDULER` packages receive ORA-27486: insufficient privileges errors. The schema Owner's default tablespace must be `'DATA'`.

Action

The schema owner's `DEFAULT TABLESPACE` will be modified in ADB to be `'DATA'`. If a user has quota on multiple tablespaces, then after migration is complete, ensure that the proper quota is set.

19.10.33 has_directories

The Premigration Advisor Tool check `has_directories` indicates that there are directories objects in the source database.

Result Criticality

Review required

Has Fixup

No

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ADWS Autonomous Data Warehouse Shared
- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared

Scope

`INSTANCE`

Description

This check indicates that there are directories objects in the source database.

Effect

After migration, applications that rely on the directories will not work until the directories on the source database are recreated on the target database.

Action

After migration is complete, recreate the directories on the Oracle Autonomous Database instance.

19.10.34 has_enabled_scheduler_jobs

The Premigration Advisor Tool check `has_enabled_scheduler_jobs` indicates that there are List scheduler jobs that may interfere with Oracle Data Pump export.

Result Criticality

Review suggested

Has Fixup

No

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ADWS Autonomous Data Warehouse Shared
- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared
- Default (an Oracle Database instance that is not Oracle Autonomous Database)

Scope

`INSTANCE`

Description

This is a default check. This check indicates that there are List scheduler jobs that may interfere with Oracle Data Pump export.

Effect

If a scheduler job runs at the same time as a `FULL` export is under way, then Oracle Data Pump Export can fail with an ORA-39127 error.

Action

Disable any non-critical scheduler jobs, or plan the export at a time when you are certain that no scheduler jobs are running. Either stop scheduler jobs before the migration, or plan the export for a time when you are certain that no scheduler jobs are running.

You can run the following SQL statement to ensure no Scheduler Jobs are running during migration:

```
ALTER SYSTEM SET JOB_QUEUE_PROCESSES=0;
```

No restart is required after you run the statement.

19.10.35 has_external_tables_dedicated

The Premigration Advisor Tool check `has_external_tables_dedicated` indicates that Non-Cloud Objects Storage External tables exist in the source database.

Result Criticality

Review required

Has Fixup

No

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ATPD Autonomous Transaction Processing Dedicated

Scope

`SCHEMA`

Description

This check indicates that Non-Cloud Objects Storage external tables exist in the source database. These tables are not allowed in Oracle Autonomous Databases.

Effect

Applications relying on user-created external tables will not function as expected.

Action

Consider using the `DBMS_CLOUD` package to create external tables that use Cloud Object Storage.

Related Topics

- Attach Network File Storage to Autonomous Database on Dedicated Exadata Infrastructure

19.10.36 has_external_tables_default

The Premigration Advisor Tool check `has_external_tables_default` indicates that external tables cannot be migrated unless the `DIRECTORY` objects the tables rely on have been created.

Result Criticality

Action required

Has Fixup

No

Target Cloud

This is a default check. It applies to the following:

- Default (an Oracle Database instance that is not Oracle Autonomous Database)

Scope

`SCHEMA`

Description

This check indicates that external tables cannot be migrated unless the `DIRECTORY` objects that the tables rely on have been created already in the target database.

Effect

The schema mode migration of external tables will fail when those tables rely on `DIRECTORY` objects that don't already exist.

Action

Before migration, create the necessary `DIRECTORY` objects on the target database, or migrate to the target database using Full mode.

19.10.37 has_external_tables_serverless

The Premigration Advisor Tool check `has_external_tables_serverless` indicates that there are non-cloud Objects Storage external tables in the source database.

Result Criticality

Review required

Has Fixup

No

Target Cloud

- ADWS Autonomous Data Warehouse Shared
- ATPS Autonomous Transaction Processing Shared

Scope

`SCHEMA`

Description

Non-Cloud Objects Storage External tables were found. These objects are not allowed in Oracle Autonomous Database.

Effect

Applications relying on user-created external tables will not function as expected. External tables in Oracle Autonomous Database (ADB) must be recreated using Object Storage Service or File Storage Service.

Attempting to create a non-Cloud Object Storage external tables as part of the migration results in those tables being created as non-external tables.

Action

Drop the empty imported table. Use the `DBMS_CLOUD` package to create External Tables using Cloud Object Storage Service or use File Storage Service. for more info see

Related Topics

- Access Network File System from Autonomous Database

19.10.38 has_fmw_registry_in_system

The Premigration Advisor Tool check `has_fmw_registry_in_system` indicates that the Fusion Middleware Schema Version Registry must be moved out of the `SYSTEM` schema before migration.

Result Criticality

Action required

Has Fixup

No

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ADWS Autonomous Data Warehouse Shared
- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared

Scope

`INSTANCE`

Description

The Fusion Middleware Schema Version Registry is in the `SYSTEM` schema. It must be moved out of the `SYSTEM` schema before you start the migration.

Effect

If the Fusion Middleware Version Registry is not moved, then after upgrade, vital information regarding what Fusion Middleware applications are installed will be lost.

Action

Before migration, run the Fusion Middleware Upgrade Assistant command `ua -moveRegistry`.

19.10.39 has_illegal_characters_in_comments

The Premigration Advisor Tool check `has_illegal_characters_in_comments` indicates that there are characters in table comments that are not legal in the databases character set.

Result Criticality

Review required

Has Fixup

No

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ADWS Autonomous Data Warehouse Shared
- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared
- Default (an Oracle Database instance that is not Oracle Autonomous Database)

Scope

`SCHEMA`

Description

This is a default check for characters in `TABLE` and `COLUMN` comments as well as PL/SQL sources for characters that are not legal in the databases character set.

Effect

Illegal characters can result in "ORA-39346: data loss in character set conversion for object" errors during import. The illegal characters will be replaced with the default replacement character.

Action

Before migration, delete any illegal characters or replace them with valid characters.

19.10.40 has_ilm_ado_policies

The Premigration Advisor Tool check `has_ilm_ado_policies` indicates that Information Lifestyle Management (ILM) Automatic Data Optimization Policies (ADO) will not migrate.

Result Criticality

Review required

Has Fixup

No

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ADWS Autonomous Data Warehouse Shared
- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared

Scope

`SCHEMA`

Description

Tables exist with ILM Automatic Data Optimization Policies. These policies will not migrate to Oracle Autonomous Database.

Effect

Tables with ILM ADO Policies (Release 12c and later) will be created without the `ILM ADO` Policy in Oracle Autonomous Transaction Processing (ATP) and Oracle Autonomous Data Warehouse (ADW).

Action

No action is required.

19.10.41 has_incompatible_jobs

The Premigration Advisor Tool check `has_incompatible_jobs` indicates that Information Lifestyle Management (ILM) Automatic Data Optimization Policies (ADO) will not migrate.

Result Criticality

Review required

Has Fixup

No

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ADWS Autonomous Data Warehouse Shared
- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared

Scope

`SCHEMA`

Description

Scheduler Jobs and Programs other than `PLSQL_BLOCK` or `STORED_PROCEDURE` are present on the source, but not supported on Oracle Autonomous Database (ADB).

Effect

Scheduler Jobs and Programs types such as `EXECUTABLE` and `EXTERNAL_SCRIPT` will not run on Oracle Autonomous Database.

Action

Databases using unsupported Job or Program types should be modified before migrating to Oracle Autonomous Database. Recreate required Job or Programs using types allowed in ADB

19.10.42 has_index_organized_tables

The Premigration Advisor Tool check `has_index_organized_tables` indicates that Index Organized tables are present in the source database.

Result Criticality

Review suggested

Has Fixup

No

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ADWS Autonomous Data Warehouse Shared
- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared

Scope

`SCHEMA`

Description

Index-organized tables are not allowed in Oracle Autonomous Database (ADB). However, attempting to create one does not generate an error. Instead, a heap-organized table with a primary key index is created.

Effect

The recreated tables can perform differently, so you should review them.

Action

Tables in the target database are created as non-index-organized tables (that is, as regular tables).

19.10.43 has_java_objects

The Premigration Advisor Tool check `has_java_objects` indicates that there are Java objects present in the source database.

Result Criticality

Action required

Has Fixup

Yes

Target Cloud

- ADWS Autonomous Data Warehouse Shared
- ATPS Autonomous Transaction Processing Shared

Scope

SCHEMA

Description

Java objects will not migrate by default.

Effect

When the Java virtual machine (JAVAVM) feature is not enabled on the target system, any applications relying on Java objects will fail after migration.

Action

Non-essential Java Objects should be excluded from the migration process. Enable the JAVAVM feature on the target system, as described in "Using Oracle Java on Autonomous Database" in *Oracle Autonomous Database Using Oracle Autonomous Database on Shared Exadata Infrastructure*.

Related Topics

- Using Oracle Java on Autonomous Database

19.10.44 has_java_source

The Premigration Advisor Tool check has_java_source indicates that there are Java sources present in the source database.

Result Criticality

Action required

Has Fixup

Yes

Scope

SCHEMA

Target Cloud

- ADWS Autonomous Data Warehouse Shared
- ATPS Autonomous Transaction Processing Shared

Description

Java sources will not migrate by default.

Effect

When the Java virtual machine (JAVAVM) feature is not enabled on the target system, any applications relying on Java objects will fail after migration.

Action

Non-essential Java Objects should be excluded from the migration process. Enable the JAVAVM feature on the target system, as described in "Using Oracle Java on Autonomous Database" in

Oracle Autonomous Database Using Oracle Autonomous Database on Shared Exadata Infrastructure

Related Topics

- Using Oracle Java on Autonomous Database

19.10.45 has_libraries

The Premigration Advisor Tool check `has_libraries` indicates that there are applications that require the `CREATE LIBRARY` statement.

Result Criticality

Action required

Has Fixup

No

Scope

`SCHEMA`

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ADWS Autonomous Data Warehouse Shared
- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared

Description

The `CREATE LIBRARY` statement is not allowed on Oracle Autonomous Database.

Effect

Applications that depend on these libraries will fail, because the libraries will not be created on the target instance.

Action

Applications must be updated to remove their dependencies on any listed libraries.

Consider using Functions for business logic previously implemented in external libraries.

Related Topics

- Functions

19.10.46 has_logging_off_for_partitions

The Premigration Advisor Tool check `has_logging_off_for_partitions` indicates that there are Partitions using the `NOLOGGING` storage attribute.

Result Criticality

Review suggested

Has Fixup

No

Scope

SCHEMA

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ADWS Autonomous Data Warehouse Shared
- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared

Description

Partitions with the NOLOGGING storage attribute are be changed to LOGGING during migration.

Effect

Partitions created with NOLOGGING will automatically be created in Oracle Autonomous Database as partitions with LOGGING. Check the LOGGING attribute in DBA_TAB_PARTITIONS.

Action

No action required.

19.10.47 has_logging_off_for_subpartitions

The Premigration Advisor Tool check has_logging_off_for_subpartitions indicates that there are Partitions using the NOLOGGING storage attribute.

Result Criticality

Review suggested

Has Fixup

No

Scope

SCHEMA

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ADWS Autonomous Data Warehouse Shared
- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared

Description

Subpartitions with the NOLOGGING storage attribute are be changed to LOGGING during migration.

Effect

Subpartitions created with `NOLOGGING` will automatically be created in Oracle Autonomous Database as subpartitions with `LOGGING`. Check the `LOGGING` attribute in `DBA_TAB_SUBPARTITIONS`.

Action

No action required.

19.10.48 has_logging_off_for_tables

The Premigration Advisor Tool check `has_logging_off_for_tables` indicates that there are tables using the `NOLOGGING` storage attribute.

Result Criticality

Review suggested

Has Fixup

No

Scope

`SCHEMA`

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ADWS Autonomous Data Warehouse Shared
- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared

Description

Tables with the `NOLOGGING` storage attribute are be changed to `LOGGING` during migration.

Effect

Tables created with `NOLOGGING` will automatically be created in Oracle Autonomous Database as tables with `LOGGING`. Check the `LOGGING` attribute in `DBA_TABLES`.

Action

No action required.

19.10.49 has_low_streams_pool_size

The Premigration Advisor Tool check `has_low_streams_pool_size` indicates that Mining Models with unexpected or incorrect attributes are detected.

Result Criticality

Action required

Has Fixup

No

Scope

`SCHEMA`

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ADWS Autonomous Data Warehouse Shared
- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared
- Default (an Oracle Database instance that is not Oracle Autonomous Database)

Description

Mining models are database schema objects that perform data mining. Mining models with unexpected or incorrect attributes are detected. These mining models will not migrate.

Effect

Mining models with issues will not be exported properly, and cause `ORA-39083` errors on import.

Action

Download and apply Patch ID 33270686

19.10.50 has_noexport_object_grants

The Premigration Advisor Tool check `has_noexport_object_grants` indicates that Oracle Data Pump is unable to export all object grants.

Result Criticality

Review required

Has Fixup

Yes

Scope

`SCHEMA`

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ADWS Autonomous Data Warehouse Shared
- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared
- Default (an Oracle Database instance that is not Oracle Autonomous Database)

Description

Oracle Data Pump is unable to export all object grants.

Effect

Object grants required for your application may be missing on the target instance, preventing Oracle Data Pump from exporting them to the target instance.

Action

Recreate any required grants on the target instance. See My Oracle Support Document ID 1911151.1 for more information. Note that any `SELECT` grants on system objects will need to be replaced with `READ` grants, because `SELECT` is no longer allowed on system objects.

Related Topics

- Data Pump: GRANTs On SYS Owned Objects Are Not Transferred With Data Pump And Are Missing In The Target Database (Doc ID 1911151.1)

19.10.51 has_oracle_streams

The Premigration Advisor Tool check `has_oracle_streams` indicates that Oracle Streams is configured in the database.

Result Criticality

Review REQUIRED

Has Fixup

No

Scope

`INSTANCE`

Target Cloud

This is a default check. It applies to the following:

- ADWD Autonomous Data Warehouse Dedicated
- ADWS Autonomous Data Warehouse Shared
- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared

Description

Starting with Oracle Database 19c, Oracle Streams is desupported. Oracle strongly advices you to remove any streams configuration manually.

Effect

Oracle Streams is a desupported feature. You must remove it.

Action

Remove the Oracle Streams configuration. For detailed steps, refer to the section *Removing an Oracle Streams Configuration* in the Oracle Streams Concepts and Administration Guide specific for the Oracle release from which you are removing. For Oracle Database releases earlier than 12.1 (12.1.0.2), the procedure `dbms_streams_adm.remove_streams_configuration` must not be used, because it can lead to unwanted results. Instead, use the other procedures (`dbms_capture_adm.drop_capture`, `dbms_apply_adm.drop_apply`, `dbms_streams_adm.remove_queue`, and so on). For Oracle Database releases 12.1 (12.1.0.2) and higher, procedure `dbms_streams_adm.remove_streams_configuration` can be safely used. To avoid issues on import consider using the Oracle Data Pump option `'STREAMS_CONFIGURATION=N'`.

Related Topics

- Removing an Oracle Streams Configuration in Oracle Streams Concepts and Administration

19.10.52 has_parallel_indexes_enabled

The Premigration Advisor Tool check `has_parallel_indexes_enabled` indicates that `PARALLEL` clause indexes exist.

Result Criticality

Review suggested

Has Fixup

No

Scope

`SCHEMA`

Target Cloud

- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared

Description

When Parallel `DEGREE` is specified greater than `1` on `INDEX`, this setting can result in unexpected behavior after migration.

Effect

When migrating to Oracle Autonomous Database Transaction Processing (ATP), if a `PARALLEL` clause is specified on the index in your source database, then the clause remains with the index when it is created on the target database, either by using Oracle Data Pump, or by using manual methods. When the `PARALLEL` degree is greater than `1`, this configuration can result in SQL statements running in parallel that are unknown to the end-user.

Action

To specify serial processing, either change the `INDEX` parallel clause to `NOPARALLEL`, or alter the `PARALLEL` degree to `1` before or after the migration.

Related Topics

- Data Pump: GRANTs On SYS Owned Objects Are Not Transferred With Data Pump And Are Missing In The Target Database (Doc ID 1911151.1)

19.10.53 has_profile_not_default

The Premigration Advisor Tool check `has_profile_not_default` indicates that schemas exist whose `PROFILE` is not available on the target system.

Result Criticality

Action Required

Has Fixup

Yes

Scope

`SCHEMA`

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ADWS Autonomous Data Warehouse Shared
- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared
- Default (an Oracle Database instance that is not Oracle Autonomous Database)

Description

Identifies schemas whose `PROFILE` is not available on the target system.

Effect

Creation of the schema on the target system fails due to the missing `PROFILE`. This is a runtiume issue, unless there are profiles used that aren't available on the target instance. In that case, the severity is Action Required.

Action

Either use Oracle Data Pump in `FULL` mode, or create the needed profiles before migration on the target system, and then use the `--analysisprops` option with a properties file created by using CPAT with the `--gettargetprops` option.

19.10.54 has_public_synonyms

The Premigration Advisor Tool check `has_public_synonyms` indicates that Public Synonyms exist in source system schemas.

Result Criticality

Review required

Has Fixup

No

Scope

`SCHEMA_ONLY`

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ADWS Autonomous Data Warehouse Shared
- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared
- Default (an Oracle Database instance that is not Oracle Autonomous Database)

Description

Identifies schemas whose that contain Public Synonyms. Oracle Data Pump does not migrate Public Synonyms in `SCHEMA` mode.

Effect

Applications relying on Public Synonyms will not work correctly until the Public Synonyms are recreated on the target instance.

Action

Either use Oracle Data Pump in `FULL` mode, or recreate the listed relevant objects on the target instance.

19.10.55 has_refs_to_restricted_packages_dedicated

The Premigration Advisor Tool check `has_refs_to_restricted_packages_dedicated` indicates that there are references to partially or completely unsupported packages.

Result Criticality

Review required

Has Fixup

No

Scope

`SCHEMA`

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ATPD Autonomous Transaction Processing Dedicated

Description

Checks for references to packages that are not supported, or that are only partially supported.

Effect

Applications that reference unsupported or restricted use packages can fail.

Action

Applications that reference unsupported packages must be modified before migration to Oracle Autonomous Database Dedicated. Applications referencing partially supported packages require testing and validation to ensure that they only use unrestricted functions and procedures.

19.10.56 has_refs_to_restricted_packages_serverless

The Premigration Advisor Tool check `has_refs_to_restricted_packages_serverless` indicates that there are references to partially or completely unsupported packages.

Result Criticality

Review required

Has Fixup

No

Scope

`SCHEMA`

Target Cloud

- ADWS Autonomous Data Warehouse Shared
- ATPS Autonomous Transaction Processing Shared

Description

Checks for references to packages that are not supported, or that are only partially supported.

Effect

Applications that reference unsupported or restricted use packages can fail.

Action

Applications that reference unsupported packages must be modified before migration to Oracle Autonomous Database Serverless. Applications referencing partially supported packages require testing and validation to ensure that they only use unrestricted functions and procedures.

19.10.57 has_refs_to_user_objects_in_sys

The Premigration Advisor Tool check `has_refs_to_user_objects_in_sys` indicates that there are user schema objects dependent on `SYS` or `SYSTEM`.

Result Criticality

Action required

Has Fixup

No

Scope

`SCHEMA`

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ADWS Autonomous Data Warehouse Shared
- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared

Description

Detects if objects in user schemas depend on user-defined objects in `SYS` or `SYSTEM` schemas.

Effect

Migration will fail for schemas that depend on user-defined objects in `SYS` or `SYSTEM`.

Action

Oracle recommends that you move user-defined objects in `SYS` and `SYSTEM` schemas before migration, and update the references. Consider dropping any user-defined objects that are no longer required.

19.10.58 has_role_privileges

The Premigration Advisor Tool check `has_role_privileges` indicates that some role privileges used in the source database are not available in the target database

Result Criticality

Action required

Has Fixup

No

Scope

`SCHEMA`

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ADWS Autonomous Data Warehouse Shared
- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared

Description

Detects the presence of role privileges used in the source database that are not available on the target Oracle Autonomous Database.

Effect

After migration, applications can encounter operation failures due to role privilege issues.

Action

Find alternatives for those roles granted in the source database that are not available in the target Oracle Autonomous Database instance. For example, you may want to substitute the `PDB_DBA` role for some schemas granted the DBA role on the source instance. Similarly, you may want to substitute the `DATAPUMP_CLOUD_IMP` role on the target instance for schemas granted `DATAPUMP_IMP_FULL_DATABASE` or `IMP_FULL_DATABASE` on the source instance. Whether such alternatives are appropriate can only be determined with testing, and by experts familiar with the applications where these role privileges occur.

19.10.59 has_sqlt_objects_adb

The Premigration Advisor Tool check `has_sqlt_objects_adb` indicates that `SQLTXPLAIN` objects are detected.

Result Criticality

Review suggested

Has Fixup

No

Scope

`UNIVERSAL`

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ADWS Autonomous Data Warehouse Shared
- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared

Description

Detects the presence of `SQLTXPLAIN` (SQLT) objects, which are not supported on Oracle Autonomous Database.

Effect

Objects related to `SQLTXPLAIN` will fail on import to Oracle Autonomous Database (ADB), which can cause import errors.

Action

Oracle recommends that administrators migrating a source database to Oracle Autonomous Database apply `sqdrop.sql` in the installation directory under the `SQLTXPLAIN` installation to drop all `SQLTXPLAIN` and `SQLTXADMIN` objects. See My Oracle Support Document ID 1614107.1 for more information.

Related Topics

- SQLT Usage Instructions (Doc ID 1614107.1)

19.10.60 has_sqlt_objects_default

The Premigration Advisor Tool check `has_sqlt_objects_default` indicates that `SQLTXPLAIN` objects are detected that Oracle Data Pump does not export.

Result Criticality

Review suggested

Has Fixup

No

Scope

`UNIVERSAL`

Target Cloud

- Default (an Oracle Database instance that is not Oracle Autonomous Database)

Description

Detects the presence of `SQLTXPLAIN` (SQLT) objects that are not exported by Oracle Data Pump.

Effect

Some objects related to `SQLTXPLAIN` will not be imported on the target instance, possibly causing import errors.

Action

Oracle recommends that `SQLTXPLAIN` users run `sqcreate.sql` in the target environment after the import is complete. The `sqcreate.sql` script runs `sqdrop.sql`, and then reinstalls all required objects. For more information, see My Oracle Support Document ID 1614107.1.

Related Topics

- SQLT Usage Instructions (Doc ID 1614107.1)

19.10.61 has_sys_privileges

The Premigration Advisor Tool check `has_sys_privileges` indicates that some system privileges in the source database are not available in the target database.

Result Criticality

Action required

Has Fixup

No

Scope

`SCHEMA`

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ADWS Autonomous Data Warehouse Shared
- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared

Description

Detects that there are some system privileges used in the source database that are not available on the Oracle Autonomous Database.

Effect

Operation failures can occur on the Oracle Autonomous Database, because of system privilege issues.

Action

Verify whether all system privileges will be needed on the Oracle Autonomous Database, and remove the grants for those privileges that are no longer needed. Find alternatives for the granted system privileges that are not available in the target Oracle Autonomous Database (ADB). For example, with schemas in ADB instances, replace `GRANT CREATE JOB` to *`USER-WHO-HAD-CREATE-ANY-JOB`* Whether such alternatives are appropriate can only be determined by experts familiar with the applications in question and with testing.

19.10.62 has_tables_that_fail_with_dblink

The Premigration Advisor Tool check `has_tables_that_fail_with_dblink` indicates that there are tables with `LONG` or `LONG RAW` data types

Result Criticality

Action required

Has Fixup

No

Scope

SCHEMA

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ADWS Autonomous Data Warehouse Shared
- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared
- Default (an Oracle Database instance that is not Oracle Autonomous Database)

Description

Tables with LONG or LONG RAW data types will not migrate over a DBLINK with Oracle Data Pump.

All forms of LONG data types (LONG, LONG RAW, LONG VARCHAR, LONG VARRAW) were deprecated in Oracle8i Release 8.1.6. For succeeding releases, the LONG data type was provided for backward compatibility with existing applications. In new applications developed with later releases, Oracle strongly recommends that you use CLOB and NCLOB data types for large amounts of character data.

Effect

Any applications relying on tables with LONG or LONG RAW data types will fail.

Action

Use Oracle Data Pump without DBLINK, or exclude the schemas and tables that have columns with LONG or LONG RAW data types.

19.10.63 has_tables_with_long_raw_datatype

The Premigration Advisor Tool check has_tables_with_long_raw_datatype indicates that there are tables with LONG or LONG RAW data types

Result Criticality

Action required

Has Fixup

No

Scope

SCHEMA

Target Cloud

- ADWS Autonomous Data Warehouse Shared

Description

ADWS does not support tables with LONG or LONGRAW data where the table has the Oracle Hybrid Columnar Compression (HCC) compression clause, or where compression is DISABLED.

Effect

Tables with `LONG` or `LONG RAW` data types will not migrate.

In Oracle Autonomous Data Warehouse (ADW), tables with `LONG` or `LONG RAW` data types are not created when the table has either an HCC compression clause, or compression is `DISABLED`, which would result with tables being compressed by default with HCC compressed by default on ADW.

All forms of `LONG` data types (`LONG`, `LONG RAW`, `LONG VARCHAR`, `LONG VARRAW`) were deprecated in Oracle8i Release 8.1.6. For succeeding releases, the `LONG` data type was provided for backward compatibility with existing applications. In new applications developed with later releases, Oracle strongly recommends that you use `CLOB` and `NCLOB` data types for large amounts of character data.

Action

Oracle recommends that you you create the table manually on ADW with compression enabled.

19.10.64 has_tables_with_xmltype_column

The Premigration Advisor Tool check `has_tables_with_xmltype_column` indicates that there are tables with `XMLTYPE` columns.

Result Criticality

Action required

Has Fixup

No

Scope

`SCHEMA`

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ADWS Autonomous Data Warehouse Shared
- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared

Description

Tables with `XMLType` column will not migrate unless the `STORAGE_TYPE` setting is `BINARY`.

Effect

Any applications relying on `XMLType` columns that are not stored as `BINARY` will fail.

Action

Tables with `XMLType` columns defined with CLOB or Object-Relational storage are not supported in Oracle Autonomous Database. When the relevant objects column `XMLSCHEMA` is

not empty, this indicates that your application uses XML Schema Objects, and additional work may be required. For non-schema types, the `BINARY` storage option must be used. See Oracle Support Document ID 1581065.1 for information about how to convert `CLOB` columns to `BINARY`.

Related Topics

- How to Convert Basicfile CLOB to Securfile Binary XML (Doc ID 1581065.1)

19.10.65 has_trusted_server_entries

The Premigration Advisor Tool check `has_trusted_server_entries` indicates that there are`TRUSTED_SERVER` entries that cannot be recreated on Oracle Autonomous Database.

Result Criticality

Runtime

Has Fixup

No

Scope

`INSTANCE`

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ADWS Autonomous Data Warehouse Shared
- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared

Description

Checks for `TRUSTED_SERVER` entries. These entries cannot be recreated on Oracle Autonomous Database (ADB).

Effect

The `DBMS_DISTRIBUTED_TRUST_ADMIN` package is not available on Oracle Autonomous Database (ADB). As a result, any `TRUSTED_SERVER` entries other than the default (Trusted:All) will not be recreated on the target ADB instance.

Action

To avoid any exceptions reported by Oracle Data Pump during migration from the source database to the target database, specify `exclude=trusted_db_link`. To control access to your ADB instance, use Oracle Cloud Infrastructure firewall features to control access to your ADB instance.

Related Topics

- Protect your cloud resources using a virtual firewall

19.10.66 has_unstructured_xml_indexes Check

The Premigration Advisor Tool check `has_unstructured_xml_indexes` indicates that there are Unstructured XML Indexes.

Result Criticality

Review required

Has Fixup

No

Scope

`SCHEMA`

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ADWS Autonomous Data Warehouse Shared
- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared

Description

This check indicates that unstructured XML indexes are present. Unstructured indexes are not supported on ADB

Effect

Attempting to create an unstructured XML index will fail. Import errors should be expected when migrating unstructured XML indexes.

Action

Before migration, recreate the indexes using XML Search Index, as described in *Oracle XML DB Developer's Guide*

Related Topics

- Creating and Using an XML Search Index in *Oracle XML DB Developer's Guide*

19.10.67 has_user_defined_objects_in_sys

The Premigration Advisor Tool check `has_user_defined_objects_in_sys` indicates that there are User-defined objects in the `SYS` schema.

Result Criticality

Action required

Has Fixup

No

Scope

INSTANCE

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ADWS Autonomous Data Warehouse Shared
- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared

Description

This check indicates that user-defined objects exist in the SYS schema.

Effect

User-defined objects in the SYS schema will not migrate. Any applications relying on user-defined objects in SYS will fail.

Action

Before migration, Oracle recommends that you move out of SYS any user-defined objects. Update any hardcoded references to those objects. Consider dropping any user-defined objects that are no longer required.

19.10.68 has_user_defined_objects_in_system

The Premigration Advisor Tool check has_user_defined_objects_in_system indicates that there are User-defined objects in the SYSTEM schema.

Result Criticality

Action required

Has Fixup

No

Scope

INSTANCE

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ADWS Autonomous Data Warehouse Shared
- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared

Description

This check indicates that user-defined objects exist in the SYSTEM schema.

Effect

User-defined objects in the `SYSTEM` schema will not migrate. Any applications relying on user-defined objects in `SYSTEM` will fail.

Action

Before migration, Oracle recommends that you recreate required user-defined objects in SYSTEM schemas, or use Oracle Data Pump schema mapping parameters such as `REMAP_SCHEMA=SYSTEM:xxx` where *xxx* is an existing user in ADB. In either case, any hardcoded references to the user-defined objects from `SYSTEM` must be updated. Consider dropping any user-defined objects that are no longer required.

19.10.69 has_user_defined_objects_no_quota

The Premigration Advisor Tool check `has_user_defined_objects_no_quota` indicates that there are are objects in the source database that belong to users without quota.

Result Criticality

Runtime

Has Fixup

No

Scope

`INSTANCE`

Target Cloud

This is a default check. It applies to the following:

- ADWD Autonomous Data Warehouse Dedicated
- ADWS Autonomous Data Warehouse Shared
- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared
- Default (an Oracle Database instance that is not Oracle Autonomous Database)

Description

This check indicates that there are objects in the source database that belong to users without relevant tablespace quota (or who have not been granted `UNLIMITED TABLESPACE`). These objects will not be migrated to the target environment.

Effect

The objects belonging to these users may fail to transfer due to `ORA-01536` errors, leading to incomplete migration and potential data loss in the target database.

\

Action

To complete transfer of all user data to the target environment. before you initiate the migration, assign an appropriate quota to all listed users (or grant those users `UNILIMITED TABLESPACE`).

19.10.70 has_user_defined_pvfs

The Premigration Advisor Tool check `has_user_defined_pvfs` indicates that there are User-defined Password Verification Functions.

Result Criticality

Runtime

Has Fixup

No

Scope

`INSTANCE`

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ADWS Autonomous Data Warehouse Shared
- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared

Description

This check indicates that user-defined Password Verification Functions (PVFs) exist.

Effect

User-defined objects in the `SYS` or `SYSTEM` schemas can't be imported into Oracle Autonomous Database (ADB). Attempts to import a `PROFILE` that uses user-defined Password Verification Functions will result in ORA-39460 errors.

\

Action

Use a profile with a Password Verification Function provided by Oracle.

19.10.71 has_users_with_10g_password_version

The Premigration Advisor Tool check `has_users_with_10g_password_version` indicates that there are user accounts using `10G` password version.

Result Criticality

Review required.

Has Fixup

No

Scope

`SCHEMA`

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ADWS Autonomous Data Warehouse Shared
- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared
- Default (an Oracle Database instance that is not Oracle Autonomous Database)

Description

This check indicates that there are users on the source database that are using the 10G password version. This password version is desupported. After migration, users verified by the `10G` password version will not be able to log in.

Effect

After migration, users identified by the 10G password version fail to connect to the database, and receive ORA-1017 errors. During Oracle Data Pump migration ORA-39384 warnings are generated.

Action

To avoid Oracle Data Pump migration warnings, before migration, Oracle recommends that you change the passwords for any users listed as using the `10G` password version. Alternatively, you can modify these users' passwords after migration to avoid login failures. See Oracle Support Document ID 2289453.1 for more information.

Related Topics

- ORA-39384: Warning: User <USERNAME> Has Been Locked And The Password Expired During Import (Doc ID 2289453.1)

19.10.72 has_xmlschema_objects

The Premigration Advisor Tool check `has_xmlschema_objects` indicates that there are XML Schema Objects in the source database.

Result Criticality

Action required

Has Fixup

No

Scope

`UNIVERSAL`

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ADWS Autonomous Data Warehouse Shared
- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared

Description

This check indicates that XML Schema Objects are in the source database. These objects will not migrate.

Effect

XML Schemas are not supported in Oracle Autonomous Database.

Action

Modify your application to not use XML Schema Objects.

19.10.73 has_xmltype_tables

The Premigration Advisor Tool check `has_xmltype_tables` indicates that there are `XMLType` tables in the source database.

Result Criticality

Action required

Has Fixup

No

Scope

`SCHEMA`

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ADWS Autonomous Data Warehouse Shared
- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared

Description

This check indicates that there are XMLType Tables in the source database. These tables will not migrate unless the `STORAGE_TYPE` is `BINARY`.

Effect

Any applications relying on `XMLType` tables not stored as `BINARY` will fail.

Action

`XMLType` tables with `CLOB` or Object-Relational storage are not supported in Oracle Autonomous Database. Change the `XMLType` storage option to `BINARY`.

19.10.74 modified_db_parameters_dedicated

The Premigration Advisor Tool check `modified_db_parameters_dedicated` indicates that restricted initialization parameters are modified.

Result Criticality

Review suggested

Has Fixup

No

Scope

`INSTANCE`

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ATPD Autonomous Transaction Processing Dedicated

Description

This check indicates that there are Oracle Database parameters on the source database instance whose modification is not allowed in Oracle Autonomous Database (Dedicated Infrastructure).

Effect

You are provided with a list of initialization parameters that have been modified in your database, but cannot be modified in Oracle Autonomous Database.

Action

To undersetand what parameters you are permitted to modify, refer to the Oracle Autonomous Database documentation.

Related Topics

- List of Initialization Parameters that can be Modified

19.10.75 modified_db_parameters_serverless

The Premigration Advisor Tool check `modified_db_parameters_serverless` indicates that restricted initialization parameters are modified.

Result Criticality

Review suggested

Has Fixup

No

Scope

`INSTANCE`

Target Cloud

This is a default check. It applies to the following:

- ADWS Autonomous Data Warehouse Shared
- ATPS Autonomous Transaction Processing Shared

Description

This check indicates that there are Oracle Database parameters on the source database instance whose modification is not allowed in Oracle Autonomous Database (Shared Infrastructure).

Effect

You are provided with a list of initialization parameters that have been modified in your database, but cannot be modified in Oracle Autonomous Database.

Action

To understand what parameters you are permitted to modify, refer to the Oracle Autonomous Database documentation.

Related Topics

- List of Initialization Parameters that can be Modified

19.10.76 nls_character_set_conversion

The Premigration Advisor Tool check `nls_character_set_conversion` indicates that there are character codes on the source database that are invalid in Oracle Autonomous Database.

Result Criticality

Runtime

Has Fixup

No

Scope

`SCHEMA`

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ADWS Autonomous Data Warehouse Shared
- ATPD Autonomous Transaction Processing Dedicated

- ATPS Autonomous Transaction Processing Shared
- Default (an Oracle Database instance that is not Oracle Autonomous Database)

Description

This check warns of issues caused by conversion of character data from the source to the target database character set, such as expansion of character values beyond column length or loss of invalid character codes.

Effect

During migration you can encounter ORA-1401 or loss of invalid character codes due to character set conversion from the source to the target database character set.

Action

Correct the issue as needed. Possible solutions include the following:

- Use Database Migration Assistant for Unicode (DMU) to scan the schemas that you want to migrate, and analyze all possible convertibility issues
- Create a new target instance using the same character set as the source instance. See the Oracle Cloud Infrastructure Documentation for information on choosing a character set when creating a database instance.

See the Oracle Cloud Infrastructure documentation for information on choosing a character set when creating a database instance.

> **Note:**
>
> Oracle recommends that you use the default database character set, `AL32UTF8`

Related Topics

- The Database Migration Assistant for Unicode (DMU) Tool (Doc ID 1272374.1)

19.10.77 nls_national_character_set

The Premigration Advisor Tool check `nls_national_character_set` indicates that the `NCHAR` and `NVARCHAR2` lengths are different on the source and target databases.

Result Criticality

Review required

Has Fixup

No

Scope

`UNIVERSAL`

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ADWS Autonomous Data Warehouse Shared

- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared
- Default (an Oracle Database instance that is not Oracle Autonomous Database)

Description

This check indicates that the `NCHAR` and `NVARCHAR2` lengths are different on the source and target databases.

Check for issues caused by the conversion of character data from the source to the target national character set, such as expansion of character values beyond data type limits or loss of invalid character codes.

Effect

During migration you can encounter `ORA-01401` or loss of invalid character codes due to character set conversion from the source to the target national character set.

Action

If possible, provision the target database on Oracle Cloud Infrastructure with the same national character set as the source database, and enable extended data types in the target cloud database. See the Oracle Cloud Infrastructure documentation for information on choosing a national character set when creating a database instance.

19.10.78 nls_nchar_ora_910

The Premigration Advisor Tool check `nls_nchar_ora_910` indicates that the `NCHAR` and `NVARCHAR2` lengths are greater than the maximum length on the target databases.

Result Criticality

Action required

Has Fixup

No

Scope

`SCHEMA`

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ADWS Autonomous Data Warehouse Shared
- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared
- Default (an Oracle Database instance that is not Oracle Autonomous Database)

Description

This check indicates that the `NCHAR` and `NVARCHAR2` lengths are greater than the maximum permitted length on the target database.

Determine the maximum column length for the national database character set on the target database, and check for `NCHAR` and `NVARCHAR2` columns on the source database whose character length exceeds the limit on the target database.

Effect

During migration you can encounter `ORA-00910` errors due to the difference of the maximum character length of `NCHAR` and `NVARCHAR2` columns between the source and the target database.

Action

If possible, provision the target database on Oracle Cloud Infrastructure with the same national character set as the source database, and enable extended data types in the target cloud database. See the Oracle Cloud Infrastructure documentation for information on choosing a national character set when creating a database instance.

19.10.79 options_in_use_not_available_dedicated

The Premigration Advisor Tool check `options_in_use_not_available_dedicated` indicates that unavailable database options are present on the source database.

Result Criticality

Review suggested

Has Fixup

No

Scope

`INSTANCE`

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ATPD Autonomous Transaction Processing Dedicated

Description

Generates a list of database options that are in use on the source, but not available in Oracle Autonomous Database (Dedicated Infrastructure).

Effect

If the database that you are migrating has applications or schemas in your database that use options that are not available on Oracle Autonomous Database, then it is possible that these applications will not work after migration.

Action

Verify that the applications or schemas in your source database depend on the options that are not supported on Oracle Autonomous Database (Dedicated Infrastructure), and plan accordingly.

19.10.80 options_in_use_not_available_serverless

The Premigration Advisor Tool check `options_in_use_not_available_serverless` indicates that unavailable database options are present on the source database.

Result Criticality

Review suggested

Has Fixup

No

Scope

`INSTANCE`

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ATPD Autonomous Transaction Processing Dedicated

Description

Generates a list of database options that are in use on the source, but not available in Oracle Autonomous Database (Shared Infrastructure).

Effect

If the database that you are migrating has applications or schemas in your database that use options that are not available on Oracle Autonomous Database, then it is possible that these applications will not work after migration.

Action

Verify that the applications or schemas in your source database depend on the options that are not supported on Oracle Autonomous Database (Shared Infrastructure), and plan accordingly.

19.10.81 standard_traditional_audit_adb

The Premigration Advisor Tool check `standard_traditional_audit_adb` indicates that Traditional Audit configurations are detected in the database.

Result Criticality

Review suggested

Has Fixup

No

Scope

`INSTANCE`

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ADWS Autonomous Data Warehouse Shared
- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared

Description

Traditional audit, which was deprecated in Oracle Database 21c, is desupported starting with Oracle Database 23c. Traditional Audit configurations have been detected in this database.

Effect

Traditional Auditing is desupported in Oracle Database 23c. Oracle strongly recommends that you start using Unified Auditing.

Action

Delete the Traditional Auditing configurations. To assist you, use the instructions in Oracle Support Document ID 2909718.1.

Related Topics

- Traditional to Unified Audit Syntax Converter - Generate Unified Audit Policies from Current Traditional Audit Configuration (Doc ID 2909718.1)

19.10.82 standard_traditional_audit_default

The Premigration Advisor Tool check `standard_traditional_audit_default` indicates that Traditional Audit configurations are detected in the database.

Result Criticality

Review suggested

Has Fixup

No

Scope

`INSTANCE`

Target Cloud

- Default (an Oracle Database instance that is not Oracle Autonomous Database)

Description

Traditional audit, which was deprecated in Oracle Database 21c, is desupported starting with Oracle Database 23c. Traditional Audit configurations have been detected in this database.

Effect

Traditional Auditing is desupported in Oracle Database 23c. Oracle strongly recommends that you start using Unified Auditing.

Action

Delete the traditional auditing configurations using the instructions found in Oracle Support Document ID 2909718.1. Ensure that the following `init.ora` parameter values are set in `CDB$ROOT`, and restart the database:

```
AUDIT_TRAIL=none
AUDIT_SYS_OPERATIONS=false
```

Related Topics

- Traditional to Unified Audit Syntax Converter - Generate Unified Audit Policies from Current Traditional Audit Configuration (Doc ID 2909718.1)

19.10.83 timezone_table_compatibility_higher_dedicated

The Premigration Advisor Tool check `timezone_table_compatibility_higher_dedicated` indicates that the timezone setting is a more recent version on the source than on the target database.

Result Criticality

Runtime

Has Fixup

No

Scope

`UNIVERSAL`

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ATPD Autonomous Transaction Processing Dedicated

Description

The source database `TZ_VERSION` cannot be higher than the target `TZ_VERSION`.

Effect

Migration is not possible until the target `TZ_VERSION` is the same or higher than the source database `TZ_VERSION`.

Action

Use the "Enable time-zone update" option of the Schedule maintenance dialog for the Quarterly Maintenance Update to update the time zone version on your target instance.

Related Topics

- Schedule a Quarterly Maintenance Update

19.10.84 timezone_table_compatibility_higher_default

The Premigration Advisor Tool check `timezone_table_compatibility_higher_default` indicates that the timezone setting is a more recent version on the source than on the target database.

Result Criticality

Runtime

Has Fixup

No

Scope

`UNIVERSAL`

Target Cloud

- Default (an Oracle Database instance that is not Oracle Autonomous Database)

Description

The source database `TZ_VERSION` cannot be higher than the target `TZ_VERSION`.

Effect

Migration is not possible until the target `TZ_VERSION` is the same or higher than the source database `TZ_VERSION`.

Action

Ensure the target instance has a time zone version equal or greater than the source instance by downloading and installing an appropriate patch from Oracle Support Document ID 412160.1

Related Topics

- Primary Note DST FAQ : Updated DST Transitions and New Time Zones in Oracle RDBMS and OJVM Time Zone File Patches (Doc ID 412160.1)

19.10.85 timezone_table_compatibility_higher_serverless

The Premigration Advisor Tool check `timezone_table_compatibility_higher_serverless` indicates that the timezone setting is a more recent version on the source than on the target database.

Result Criticality

Runtime

Has Fixup

No

Scope

`UNIVERSAL`

Target Cloud

- ADWS Autonomous Data Warehouse Shared
- ATPS Autonomous Transaction Processing Shared

Description

The source database `TZ_VERSION` cannot be higher than the target `TZ_VERSION`.

Effect

Migration is not possible until the target `TZ_VERSION` is the same or higher than the source database `TZ_VERSION`.

Action

Update the Time Zone File Version. Refer to "Manage Time Zone File Version on Autonomous Database"

Related Topics

- Manage Time Zone File Version on Autonomous Database

19.10.86 unified_and_standard_traditional_audit_adb

The Premigration Advisor Tool check `unified_and_standard_traditional_audit_adb` indicates that Traditional Audit configurations are detected in the database.

Result Criticality

Runtime

Has Fixup

No

Scope

`INSTANCE`

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ADWS Autonomous Data Warehouse Shared
- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared

Description

Traditional audit, which was deprecated in Oracle Database 21c, is desupported starting with Oracle Database 23c. Traditional Audit configurations have been detected in this database, which is configured to use only Unified Auditing.

Effect

Performance can degrade unless the traditional audit configurations in the database are deleted.

Action

Oracle strongly recommends that you delete the Traditional Auditing configurations

19.10.87 unified_and_standard_traditional_audit_default

The Premigration Advisor Tool check `unified_and_standard_traditional_audit_default` indicates that Traditional Audit configurations are detected in the database.

Result Criticality

Runtime

Has Fixup

No

Scope

`INSTANCE`

Target Cloud

- Default (an Oracle Database instance that is not Oracle Autonomous Database)

Description

Traditional audit, which was deprecated in Oracle Database 21c, is desupported starting with Oracle Database 23c. Traditional Audit configurations have been detected in this database, which is configured to use only Unified Auditing..

Effect

Performance can degrade unless the traditional audit configurations in the database are deleted.

Action

Delete the traditional auditing configurations using the instructions found in Oracle Support Document ID 2909718.1. Ensure that the following `init.ora` parameter values are set in `CDB$ROOT`, and restart the database:

```
AUDIT_TRAIL=none
AUDIT_SYS_OPERATIONS=false
```

19.10.88 xdb_resource_view_has_entries Check

The Premigration Advisor Tool check `xdb_resource_view_has_entries Check` indicates that there is an XDB Repository that is not supported in Oracle Autonomous Database. Entries in `RESOURCE_VIEW` will not migrate.

Result Criticality

Review required

Has Fixup

No

Target Cloud

- ADWD Autonomous Data Warehouse Dedicated
- ADWS Autonomous Data Warehouse Shared
- ATPD Autonomous Transaction Processing Dedicated
- ATPS Autonomous Transaction Processing Shared
- Default (an Oracle Database instance that is not Oracle Autonomous Database)

Description

This check applies to source schema for Oracle Data Pump and Oracle GoldenGate migrations, and Oracle Data Pump database links. When there is an Oracle XML DB repository (XDB Repository) that is not supported in Oracle Autonomous Database (ADB), entries in `RESOURCE_VIEW` will not migrate.

Effect

Applications relying on entries in the XDB Repository `RESOURCE_VIEW` may not function as expected.

Action

Applications must be updated to remove their dependencies on the XDB Repository. For more information on determining if XDB is being used in your database see Oracle Support Document ID 733667.1

Related Topics

- How to Determine if XDB is Being Used in the Database? (Doc ID 733667.1)

19.11 Best Practices for Using the Premigration Advisor Tool

These Cloud Premigration Advisor Tool (CPAT) tips can help you use CPAT more effectively.

- Generate Properties File on the Target Database Instance
  Oracle recommends that you generate a Premigration Advisor Tool (CPAT) properties file on the target database instance.
- Focus the CPAT Analysis
  Oracle recommends that you focus the Premigration Advisor Tool (CPAT) analysis to restrict what schemas CPAT will examine.

- Reduce the Amount of Data in Reports
  Some Cloud Premigration Advisor tool checks can return thousands of objects with the same concern. Here's how you can reduce the report size.
- Generate the JSON Report and Save Logs
  Even if you only plan to use the text report, Oracle suggests you also generate a JSON output file with the Cloud Premigration Advisor tool (CPAT), and save the log files for diagnosis.
- Use Output Prefixes to Record Different Migration Scenarios
  To keep track of reports for different migration options, use the `--outfileprefix` and `--outdir` properties on the CPAT command line.

19.11.1 Generate Properties File on the Target Database Instance

Oracle recommends that you generate a Premigration Advisor Tool (CPAT) properties file on the target database instance.

To perform the most complete and targeted analysis of the source database instance, certain properties of the target database instance are required. For this reason, you should generate your CPAT properties file on the database instance that you want to migrate. To perform this function, the `--gettargetprops` property is intended to be used with the other connection-related properties.

In the following example, the CPAT script is run by the user `ADMIN` on the target database instance:

```
./premigration.sh --gettargetprops -username ADMIN --connectstring
'jdbc:oracle:thin:@service-name?TNS_ADMIN=path-to-wallet'
```

The command generates a properties file, `premigration_advisor_analysis.properties`, which you can use to analyze a source instance.

If necessary, you can copy the properties file generated on the target to the host where the source database analysis will be performed, and provide the file to CPAT using the `--analysisprops` property.

For example:

```
./premigration.sh --connectstring jdbc:oracle:oci:@ --targetcloud ATPD --
sysdba --analysisprops premigration_advisor_analysis.properties
```

If you know that you (or Oracle Zero Downtime Migration (ZDM) or Oracle Database Migration Service (DMS) will be mapping (or precreating) all needed tablespaces, then append the property `MigrationMethodProp.ALL_METHODS.TABLESPACE_MAPPING=ALL` to the properties file you provide to CPAT. This property setting causes CPAT to PASS most (if not all) of its tablespace-related checks. However, if you choose this option, then be aware that there can still be migration issues related to quotas with tablespace mapping.

19.11.2 Focus the CPAT Analysis

Oracle recommends that you focus the Premigration Advisor Tool (CPAT) analysis to restrict what schemas CPAT will examine.

Consider using the `--schema` switch property to restrict what schemas you want CPAT to examine during its analysis. When you start CPAT using `--schemas list`, where *list* is a

space-delimited list of schemas, CPAT performs checks only on those schemas. Without the `--schemas` switch, CPAT will analyze all schemas in the source instance (excluding Oracle-maintained schemas), which can result in problems being found in schemas that you do not intend to migrate. Using the `--schemas` property to restrict scope can be particularly useful if the source instance is hosting multiple applications, each of which could potentially be migrated to different Oracle Autonomous Database instances.

In the following example, the CPAT script is run by the user `ADMIN` on the target database instance to perform analysis on the schemas `schema1` and `schema2`:

```
./premigration.sh -username SYSTEM --connectstring 'jdbc:oracle:thin:@service-name?TNS_ADMIN=path-to-wallet' --schemas schema1 schema2
```

The `--schemas` switch property provides a space-separated list of schemas (schema1 and schema2) to CPAT, so that the checks it performs are restricted only to those two schemas.

19.11.3 Reduce the Amount of Data in Reports

Some Cloud Premigration Advisor tool checks can return thousands of objects with the same concern. Here's how you can reduce the report size.

Depending on the checks you run, some CPAT checks can return results for the same issue in multiple objects in the text report. To reduce the number of results, you can use the `--maxtextdatarows n` function, where n is an integer that specifies the number of rows that you want to view.

The `--maxrelevantobjects n` property performs the same function for reports, but limiting the size of JSON reports is typically not necessary.

In the following example, the CPAT script is run by the user `SYSTEM` on the target database instance, with the output set to return a maximum of 10 rows of text file data:

```
./premigration.sh --username SYSTEM --connectstring
'jdbc:oracle:thin:@service-name?TNS_ADMIN=path-to-wallet --maxtextdatarows 10"
```

19.11.4 Generate the JSON Report and Save Logs

Even if you only plan to use the text report, Oracle suggests you also generate a JSON output file with the Cloud Premigration Advisor tool (CPAT), and save the log files for diagnosis.

Oracle recommends generating the JSON report as well as the text report, and always save your log report files. Why? If you encounter an issue while using CPAT, and need to contact Oracle Support, then you can provide all possible information to assist Oracle Support with resolving your issue. You can assist Oracle Support by being prepared to submit both the text and JSON reports, as well as the `.log` reports generated by CPAT. The `--reportformat` property accepts one or more space-delimited report formats. The permitted values for the `--reportformat` switch are `json` and `text`.

For example:

```
./premigration.sh -username SYSTEM --connectstring 'jdbc:oracle:thin:@service-name --reportformat json text
```

19.11.5 Use Output Prefixes to Record Different Migration Scenarios

To keep track of reports for different migration options, use the `--outfileprefix` and `--outdir` properties on the CPAT command line.

To generate reports for different Cloud migration options, you can use the Cloud Premigration Advisor Tool (CPAT) with the `--outfileprefix`, so that you place a prefix on reports and log files that can organize the report options that you have generated. You can also use the `--outdir` property to organize reports for different instances, or to organize reports for different scenarios.

> **Note:**
>
> The `--outdir` property accepts either an absolute or a relative folder path. Using this property specifies a particular location where CPAT creates the log files, report files, and any properties files that you generate. If `--outdir` is omitted from the command line, then the log file and other generated files are created in the user's current folder, which can lead to files being overwritten when multiple analyses are performed.

For example:

```
./premigration.sh --outfileprefix ATPS_RUN_01 --outdir /path/CPAT_output --
reportformat TEXT JSON ...
```

20

ADRCI: ADR Command Interpreter

The Automatic Diagnostic Repository Command Interpreter (ADRCI) utility is a command-line tool that you use to manage Oracle Database diagnostic data.

> **Note:**
>
> Do not use `UIDRVCI.exe` file as it is used to access diagnostic data.

20.1 About the ADR Command Interpreter (ADRCI) Utility

The Automatic Diagnostic Repository Command Interpreter (ADRCI) is a command-line tool that is part of the Oracle Database fault diagnosability infrastructure.

The ADRCI utility assists you with diagnosing the cause of problems in your database (incidents). It can assist you with collecting data in an incident package that Oracle Support may need to help you to address the root cause of issues.

ADRCI assists you to do the following:

- View diagnostic data within the Automatic Diagnostic Repository (ADR).
- View Health Monitor reports.
- Package incident and problem information into a zip file for transmission to Oracle Support.

Diagnostic data includes incident and problem descriptions, trace files, dumps, health monitor reports, alert log entries, and more.

ADR data is secured by operating system permissions on the ADR directories, so there is no need to log in to `ADRCI`.

`ADRCI` has a rich command set. You can use these commands either in interactive mode, or within scripts.

> **Note:**
>
> The easier and recommended way to manage diagnostic data is with the Oracle Enterprise Manager Support Workbench (Support Workbench). ADRCI provides a command-line alternative to most of the functionality of the Support Workbench, and adds capabilities, such as listing and querying trace files.
>
> See *Oracle Database Administrator's Guide* for more information about the Oracle Database fault diagnosability infrastructure.

Related Topics

- *Oracle Database Administrator's Guide* Diagnosing and Resolving Problems

20.2 Definitions for Oracle Database ADRC

To understand how to diagnose Oracle Database problems, learn the definitions of terms that Oracle uses for the ADRCI, and the Oracle Database fault diagnosability infrastructure.

The following terms are associated with the Oracle Database automatic diagnostic repository incident fault diagnosability infrastructure (ADRCI), and the Oracle Database fault diagnosability infrastructure:

Automatic Diagnostic Repository (ADR)

The **Automatic Diagnostic Repository (ADR**) is a file-based repository for database diagnostic data such as traces, dumps, the alert log, health monitor reports, and more. It has a unified directory structure across multiple instances and multiple products. Beginning with Oracle Database 11g and later releases, Oracle Automatic Storage Management (Oracle

ASM), and other Oracle Database products or components store all diagnostic data in the ADR. Each instance of each product stores diagnostic data underneath its own ADR home directory. For example, in an Oracle Real Application Clusters (Oracle RAC) environment with shared storage and Oracle ASM, each database instance and each Oracle ASM instance has a home directory within the ADR. The ADR's unified directory structure enables customers and Oracle Support to correlate and analyze diagnostic data across multiple instances and multiple products.

Problem

A **problem** is a critical error in the database. Critical errors include internal errors, such as `ORA-00600` and other severe errors, such as `ORA-07445` (operating system exception) or `ORA-04031` (out of memory in the shared pool). Problems are tracked in the ADR. Each problem has a **problem key** and a unique **problem ID**.

Incident

An **incident** is a single occurrence of a problem. When a problem occurs multiple times, an incident is created for each occurrence. Incidents are tracked in the ADR. Each incident is identified by a numeric incident ID, which is unique within the ADR. When an incident occurs, the database makes an entry in the alert log, sends an **incident alert** to Oracle Enterprise Manager, gathers diagnostic data about the incident in the form of dump files (incident dumps), tags the incident dumps with the **incident ID**, and stores the incident dumps in an ADR subdirectory created for that incident.

Diagnosis and resolution of a critical error usually starts with an incident alert. You can obtain a list of all incidents in the ADR with an ADRCI command. Each incident is mapped to a single problem only.

Incidents are **flood-controlled**, so that a single problem does not generate too many incidents and incident dumps.

Problem Key

Every problem has a **problem key**, which is a text string that includes an error code (such as `ORA-600`) and in some cases, one or more error parameters. Two incidents are considered to have the same root cause if their problem keys match.

Incident Package

An **incident package (Package)** is a collection of data about incidents for one or more problems. Before sending incident data to Oracle Support, you must collect the date into a package, using the **Incident Packaging Service (IPS)**. After a package is created, you can add external files to the package, remove selected files from the package, or **scrub** (edit) selected files in the package to remove sensitive data.

A package is a logical construct only, until you create a physical file from the package contents. That is, an incident package starts out as a collection of metadata in the ADR. As you add and remove package contents, only the metadata is modified. When you are ready to upload the data to Oracle Support, you create a physical package by using ADRCI, which saves the data into a zip file. You can then upload the zip file to Oracle Support.

Finalizing

Before ADRCI can generate a physical package from a logical package, the package must be finalized. This means that other components are called to add any correlated diagnostic data files to the incidents already in this package. Finalizing also adds recent trace files, alert log entries, Health Monitor reports, SQL test cases, and configuration information. This step is run automatically when a physical package is generated, and can also be run manually using the

ADRCI utility. After manually finalizing a package, you can review the files that were added and then remove or edit any that contain sensitive information.

ADR Home

An **ADR home** is the root directory for all diagnostic data—traces, dumps, alert log, and so on—for a particular instance of a particular Oracle product or component. For example, in an Oracle RAC environment with Oracle ASM, each database instance and each Oracle ASM instance has an ADR home. All ADR homes share the same hierarchical directory structure. Some of the standard subdirectories in each ADR home include alert (for the alert log), trace (for trace files), and incident (for incident information). All ADR homes are located within the ADR base directory.

Some ADRCI commands can work with multiple ADR homes simultaneously. The current ADRCI **homepath** determines the ADR homes that are searched for diagnostic data when an ADRCI command is issued.

ADR Base

To permit correlation of diagnostic data across multiple ADR homes, ADR homes are grouped together under the same root directory called the **ADR base**. For example, in an Oracle RAC environment, the ADR base could be on a shared disk, and the ADR home for each Oracle RAC instance could be located under this ADR base.

The location of the ADR base for a database instance is set by the `DIAGNOSTIC_DEST` initialization parameter. If this parameter is omitted or is null, the database sets it to a default value.

When multiple database instances share an Oracle home, whether they are multiple single instances or the instances of an Oracle RAC database, and when one or more of these instances set ADR base in different locations, the last instance to start up determines the default ADR base for ADRCI.

Homepath

All ADRCI commands operate on diagnostic data in the **current** ADR homes. More than one ADR home can be current at any one time. Some ADRCI commands (such as `SHOW INCIDENT`) search for and display diagnostic data from all current ADR homes, while other commands require that only one ADR home be current, and display an error message if more than one are current.

The ADRCI **homepath** determines the ADR homes that are current. It does so by pointing to a directory within the ADR base hierarchy. If it points to a single ADR home directory, that ADR home is the only current ADR home. If the homepath points to a directory that is above the ADR home directory level in the hierarchy, all ADR homes that are below the directory that is pointed to become current.

The homepath is null by default when ADRCI starts. This means that all ADR homes under ADR base are current.

The `SHOW HOME` and `SHOW HOMEPATH` commands display a list of the ADR homes that are current, and the `SET HOMEPATH` command sets the homepath.

Related Topics

- *Oracle Database Administrator's Guide* About Incidents and Problems
- *Oracle Database Administrator's Guide*About Correlated Diagnostic Data in Incident Packages

20.3 Starting ADRCI and Getting Help

You can use ADRCI in interactive mode or batch mode.

Details are provided in the following sections:

- Using ADRCI in Interactive Mode
  When you use ADRCI in interactive mode to diagnose Oracle Database incidents, it prompts you to enter individual commands one at a time.
- Getting Help
  Learn how to obtain help when using the ADR Command Interpreter (ADRCI) Utility..
- Using ADRCI in Batch Mode
  Batch mode enables you to run a series of ADRCI commands using script or batch files, without being prompted for input.

20.3.1 Using ADRCI in Interactive Mode

When you use ADRCI in interactive mode to diagnose Oracle Database incidents, it prompts you to enter individual commands one at a time.

1. Ensure that the `ORACLE_HOME` and `PATH` environment variables are set properly.

   On Microsoft Windows platforms, these environment variables are set in the Windows registry automatically during installation. On other platforms, you must set and check environment variables with operating system commands.

   The `PATH` environment variable must include *`Oracle_home`*`/bin`

   .
2. Enter the following command at the operating system command prompt:

   ```
   ADRCI
   ```

   The utility starts and displays the following prompt:

   ```
   adrci>
   ```
3. Enter ADRCI commands, following each with the Enter key.
4. To exit ADRCI, Enter one of the following commands:

   ```
   EXIT
   QUIT
   ```

20.3.2 Getting Help

Learn how to obtain help when using the ADR Command Interpreter (ADRCI) Utility..

With the ADRCI help system, you can:

- View a list of ADR commands.
- View help for an individual command.
- View a list of ADRCI command line options.

To view a list of ADRCI commands

1. Start ADRCI in interactive mode.

2. At the ADRCI prompt, enter the following command:

```
HELP
```

To get help for a specific ADRCI command

1. Start ADRCI in interactive mode.
2. At the ADRCI prompt, enter the following command, where *command* is the ADRCI command about which you want more information:

```
HELP command
```

For example, to obtain help on the SHOW TRACEFILE command, enter the following:

```
HELP SHOW TRACEFILE
```

To view a list of command line options

- Enter the following command at the operating system command prompt:

```
ADRCI -HELP
```

The utility displays output similar to the following:

```
Syntax:
   adrci [-help] [script=script_filename] [exec="command [;command;...]"]

Options      Description                    (Default)
-----------------------------------------------------------------
script       script file name               (None)
help         help on the command options    (None)
exec         exec a set of commands         (None)
-----------------------------------------------------------------
```

Related Topics

- Using ADRCI in Interactive Mode
  When you use ADRCI in interactive mode to diagnose Oracle Database incidents, it prompts you to enter individual commands one at a time.

20.3.3 Using ADRCI in Batch Mode

Batch mode enables you to run a series of ADRCI commands using script or batch files, without being prompted for input.

To use batch mode, you add a command line parameter to the ADRCI command when you start ADRCI. Batch mode enables you to include ADRCI commands in shell scripts or Microsoft Windows batch files. As with interactive mode, the ORACLE_HOME and PATH environment variables must be set before starting ADRCI.

ADRCI Command Line Parameters for Batch Operation

The following command line parameters are available for batch operation:

Table 20-1 ADRCI Batch Operation Parameters

Parameter	Description
EXEC	Enables you to submit one or more ADRCI commands on the operating system command line that starts ADRCI. Commands are separated by semicolons (;).
SCRIPT	Enables you to run a script containing ADRCI commands.

How to Submit ADRCI Commands on the Command Line

- Enter the following command at the operating system command prompt:

  ```
  ADRCI EXEC="COMMAND[; COMMAND]..."
  ```

  For example, to run the SHOW HOMES command in batch mode, enter the following command at the operating system command prompt:

  ```
  ADRCI EXEC="SHOW HOMES"
  ```

  To run the SHOW HOMES command followed by the SHOW INCIDENT command, enter the following:

  ```
  ADRCI EXEC="SHOW HOMES; SHOW INCIDENT"
  ```

How to Run ADRCI Scripts:

Enter the following command at the operating system command prompt:

```
ADRCI SCRIPT=SCRIPT_FILE_NAME
```

For example, to run a script file named adrci_script.txt, enter the following command at the operating system command prompt:

```
ADRCI SCRIPT=adrci_script.txt
```

A script file contains a series of commands separated by semicolons (;) or line breaks. For example:

```
SET HOMEPATH diag/rdbms/orcl/orcl; SHOW ALERT -term
```

20.4 Setting the ADRCI Homepath Before Using ADRCI Commands

When diagnosing a problem, you may want to work with diagnostic data from multiple database instances or components, or you may want to focus on diagnostic data from one instance or component.

To work with diagnostic data from multiple instances or components, you must ensure that the ADR homes for all of these instances or components are *current*. To work with diagnostic data from only one instance or component, you must ensure that only the ADR home for that instance or component is current. You control the ADR homes that are current by setting the ADRCI homepath.

If multiple homes are current, this means that the homepath points to a directory in the ADR directory structure that contains multiple ADR home directories underneath it. To focus on a single ADR home, you must set the homepath to point lower in the directory hierarchy, to a single ADR home directory.

For example, if the Oracle RAC database with database name `orclbi` has two instances, where the instances have SIDs `orclbi1` and `orclbi2`, and Oracle RAC is using a shared Oracle home, the following two ADR homes exist:

```
/diag/rdbms/orclbi/orclbi1/
/diag/rdbms/orclbi/orclbi2/
```

In all ADRCI commands and output, ADR home directory paths (ADR homes) are always expressed relative to ADR base. So if ADR base is currently /u01/app/oracle, the absolute paths of these two ADR homes are the following:

```
/u01/app/oracle/diag/rdbms/orclbi/orclbi1/
/u01/app/oracle/diag/rdbms/orclbi/orclbi2/
```

You use the `SET HOMEPATH` command to set one or more ADR homes to be current. If ADR base is /u01/app/oracle and you want to set the homepath to /u01/app/oracle/diag/rdbms/orclbi/orclbi2/, you use this command:

```
adrci> set homepath diag/rdbms/orclbi/orclbi2
```

When ADRCI starts, the homepath is null by default, which means that all ADR homes under ADR base are current. In the previously cited example, therefore, the ADR homes for both Oracle RAC instances would be current.

```
adrci> show homes
ADR Homes:
diag/rdbms/orclbi/orclbi1
diag/rdbms/orclbi/orclbi2
```

In this case, any ADRCI command that you run, assuming that the command supports more than one current ADR home, works with diagnostic data from both ADR homes. If you were to set the homepath to /diag/rdbms/orclbi/orclbi2, only the ADR home for the instance with SID `orclbi2` would be current.

```
adrci> set homepath diag/rdbms/orclbi/orclbi2
adrci> show homes
ADR Homes:
diag/rdbms/orclbi/orclbi2
```

In this case, any ADRCI command that you run would work with diagnostic data from this single ADR home only.

See Also:

- *Oracle Database Administrator's Guide* for more information about the structure of ADR homes
- ADR Base
- ADR Home
- Homepath
- SET HOMEPATH
- SHOW HOMES

20.5 Viewing the Alert Log

To view the ACR Command Interpreter alert log (ADRCI), use this procedure to view the alert log in your default editor.

The alert log is written as both an XML-formatted file and as a text file. You can view either format of the file with any text editor, or you can run an ADRCI command to view the XML-formatted alert log with the XML tags omitted.

By default, ADRCI displays the alert log in your default editor. You can use the `SET EDITER` command to change your default editor.

To view the alert log with ADRCI:

1. Start ADRCI in interactive mode.
2. (Optional) Use the `SET HOMEPATH` command to select (make current) a single ADR home.

   You can use the `SHOW HOMES` command first to see a list of current ADR homes. See Homepath and Setting the ADRCI Homepath Before Using ADRCI Commands for more information.
3. At the ADRCI prompt, enter the following command:

   ```
   SHOW ALERT
   ```

   If more than one ADR home is current, you are prompted to select a single ADR home from a list. The alert log is displayed, with XML tags omitted, in your default editor.
4. Exit the editor to return to the ADRCI command prompt.

The following are variations on the `SHOW ALERT` command:

```
SHOW ALERT -TAIL
```

This displays the last portion of the alert log (the last 10 entries) in your terminal session.

```
SHOW ALERT -TAIL 50
```

This displays the last 50 entries in the alert log in your terminal session.

```
SHOW ALERT -TAIL -F
```

This displays the last 10 entries in the alert log, and then waits for more messages to arrive in the alert log. As each message arrives, it is appended to the display. This command enables you to perform *live monitoring* of the alert log. Press CTRL+C to stop waiting and return to the ADRCI prompt.

```
SPOOL /home/steve/MYALERT.LOG
SHOW ALERT -TERM
SPOOL OFF
```

This outputs the alert log, without XML tags, to the file `/home/steve/MYALERT.LOG`.

```
SHOW ALERT -P "MESSAGE_TEXT LIKE '%ORA-600%'"
```

This displays only alert log messages that contain the string 'ORA-600'. The output looks something like this:

```
ADR Home = /u01/app/oracle/product/11.1.0/db_1/log/diag/rdbms/orclbi/orclbi:
*************************************************************************
01-SEP-06 09.17.44.849000000 PM -07:00
AlertMsg1: ORA-600 dbgris01, addr=0xa9876541
```

Related Topics

- SHOW ALERT
  The ADRCI `SHOW ALERT` command shows the contents of the alert log in the default editor.

> **See Also:**
>
> - SHOW ALERT
> - SET EDITOR
> - *Oracle Database Administrator's Guide* for instructions for viewing the alert log with Oracle Enterprise Manager or with a text editor

20.6 Finding Trace Files

ADRCI enables you to view the names of trace files that are currently in the automatic diagnostic repository (ADR).

You can view the names of all trace files in the ADR, or you can apply filters to view a subset of names. For example, ADRCI has commands that enable you to:

- Obtain a list of trace files whose file name matches a search string.
- Obtain a list of trace files in a particular directory.
- Obtain a list of trace files that pertain to a particular incident.

You can combine filtering functions by using the proper command line parameters.

The `SHOW TRACEFILE` command displays a list of the trace files that are present in the trace directory and in all incident directories under the current ADR home. When multiple ADR homes are current, the traces file lists from all ADR homes are output one after another.

The following statement lists the names of all trace files in the current ADR homes, without any filtering:

```
SHOW TRACEFILE
```

The following statement lists the name of every trace file that has the string `mmon` in its file name. The percent sign (%) is used as a wildcard character, and the search string is case sensitive.

```
SHOW TRACEFILE %mmon%
```

This statement lists the name of every trace file that is located in the /home/steve/temp directory and that has the string `mmon` in its file name:

```
SHOW TRACEFILE %mmon% -PATH /home/steve/temp
```

This statement lists the names of trace files in reverse order of last modified time. That is, the most recently modified trace files are listed first.

```
SHOW TRACEFILE -RT
```

This statement lists the names of all trace files related to incident number 1681:

```
SHOW TRACEFILE -I 1681
```

> **See Also:**
>
> - SHOW TRACEFILE
> - *Oracle Database Administrator's Guide* for information about the directory structure of the ADR

20.7 Viewing Incidents

The ADRCI `SHOW INCIDENT` command displays information about open Oracle Database incidents.

When you submit a `SHOW INCIDENT` command, the ADRCI report shows the incident ID, problem key, and incident creation time for each incident. If you set the **homepath** (a directory within the ADR base hierarchy) so that there are multiple current ADR homes within that hierarchy location, then the report includes incidents from all of the ADR homes. See "Definitions for Oracle Database ADRC" for more information about homepath and other ADRCI terms.

1. Start ADRCI in interactive mode, and ensure that the homepath points to the correct directory within the ADR base directory hierarchy.
2. At the ADRCI prompt, enter the following command:

   ```
   SHOW INCIDENT
   ```

   ADRCI generates output similar to the following:

   ```
   ADR Home = /u01/app/oracle/product/11.1.0/db_1/log/diag/rdbms/orclbi/orclbi:
   ********************************************************************************
   INCIDENT_ID       PROBLEM_KEY               CREATE_TIME
   ----------------- ------------------------- ---------------------------------
   3808              ORA 603                   2010-06-18 21:35:49.322161 -07:00
   3807              ORA 600 [4137]            2010-06-18 21:35:47.862114 -07:00
   3805              ORA 600 [4136]            2010-06-18 21:35:25.012579 -07:00
   3804              ORA 1578                  2010-06-18 21:35:08.483156 -07:00
   4 rows fetched
   ```

The following are variations on the `SHOW INCIDENT` command:

```
SHOW INCIDENT -MODE BRIEF
SHOW INCIDENT -MODE DETAIL
```

These commands produce more detailed versions of the incident report. For example, to see a detailed incident report for incident 1681, enter the following command:

```
SHOW INCIDENT -MODE DETAIL -P "INCIDENT_ID=1681"
```

Related Topics

- ADRCI Command Reference
  Learn about the commands you can use with the Automatic Diagnostic Repository Command Interpreter (ADRCI).
- Definitions for Oracle Database ADRC
  To understand how to diagnose Oracle Database problems, learn the definitions of terms that Oracle uses for the ADRCI, and the Oracle Database fault diagnosability infrastructure.

20.8 Packaging Incidents

You can use ADRCI commands to *package* one or more incidents for transmission to Oracle Support for analysis.

Background information and instructions are presented in the following topics:

- About Packaging Incidents
  Packaging ADR Command Interpreter (ADRCI) incidents is a three-step process.
- Creating Incident Packages
  The following topics describe creating incident packages.

20.8.1 About Packaging Incidents

Packaging ADR Command Interpreter (ADRCI) incidents is a three-step process.

Step 1: Create a logical incident package.

The incident package (package) is denoted as logical, because it exists only as metadata in the automatic diagnostic repository (ADR). It has no content until you generate a physical package from the logical package. The logical package is assigned a package number, and you refer to it by that number in subsequent commands.

You can create the logical package as an empty package, or as a package based on an incident number, a problem number, a problem key, or a time interval. If you create the package as an empty package, then you can add diagnostic information to it in step 2.

Creating a package based on an incident means including diagnostic data—dumps, health monitor reports, and so on—for that incident. Creating a package based on a problem number or problem key means including in the package diagnostic data for incidents that reference that problem number or problem key. Creating a package based on a time interval means including diagnostic data on incidents that occurred in the time interval.

Step 2: Add diagnostic information to the incident package

If you created a logical package based on an incident number, a problem number, a problem key, or a time interval, this step is optional. You can add additional incidents to the package or you can add any file within the ADR to the package. If you created an empty package, you must use ADRCI commands to add incidents or files to the package.

Step 3: Generate the physical incident package

When you submit the command to generate the physical package, ADRCI gathers all required diagnostic files and adds them to a zip file in a designated directory. You can generate a complete zip file or an incremental zip file. An incremental file contains all the diagnostic files that were added or changed since the last zip file was created for the same logical package. You can create incremental files only after you create a complete file, and you can create as

many incremental files as you want. Each zip file is assigned a sequence number so that the files can be analyzed in the correct order.

Zip files are named according to the following scheme:

packageName_mode_sequence.zip

where:

- *packageName* consists of a portion of the problem key followed by a timestamp
- *mode* is either COM or INC, for complete or incremental
- *sequence* is an integer

For example, if you generate a complete zip file for a logical package that was created on September 6, 2006 at 4:53 p.m., and then generate an incremental zip file for the same logical package, you would create files with names similar to the following:

```
ORA603_20060906165316_COM_1.zip
ORA603_20060906165316_INC_2.zip
```

20.8.2 Creating Incident Packages

The following topics describe creating incident packages.

The ADRCI commands that you use to create a logical incident package (package) and generate a physical package are:

- Creating a Logical Incident Package
  You use variants of the IPS CREATE PACKAGE command to create a logical package (package).
- Adding Diagnostic Information to a Logical Incident Package
  After you have an existing logical package (**package**) configured for packaging incidents, you can add diagnostic information to that package.
- Generating a Physical Incident Package
  When you generate a package, you create a physical package (a zip file) for an existing logical package.

> **See Also:**
>
> About Packaging Incidents

20.8.2.1 Creating a Logical Incident Package

You use variants of the IPS CREATE PACKAGE command to create a logical package (package).

1. Start ADRCI in interactive mode, and ensure that the **homepath** (a directory within the ADR base hierarchy) points to the correct directory within the ADR base directory hierarchy for the database for which you want to create a logical package.

   See "Definitions for Oracle Database ADRC" for more information about homepath and other ADRCI terms.
2. At the ADRCI prompt, enter the following command:

```
IPS CREATE PACKAGE INCIDENT incident_number
```

For example, the following command creates a package based on incident 3:

```
IPS CREATE PACKAGE INCIDENT 3
```

ADRCI generates output similar to the following:

```
Created package 10 based on incident id 3, correlation level typical
```

The package number assigned to this logical package is 10.

The following are variations on the `IPS CREATE PACKAGE` command:

```
IPS CREATE PACKAGE
```

Entering the command without specifications creates an empty package. To add diagnostic data to the package before generating it, you then must use the `IPS ADD INCIDENT` or `IPS ADD FILE` commands.

```
IPS CREATE PACKAGE PROBLEM problem_ID
```

This command creates a package, and includes diagnostic information for incidents that reference the specified problem ID. (Problem IDs are integers.) You can obtain the problem ID for an incident from the report displayed by the `SHOW INCIDENT -MODE BRIEF` command. Because there can be many incidents with the same problem ID, ADRCI adds to the package the diagnostic information for the first three incidents (**early incidents**) that occurred and last three incidents (**late incidents**) that occurred with this problem ID, excluding any incidents that are older than 90 days.

> **Note:**
>
> The number of early and late incidents, and the 90-day age limit are defaults, which you can change. See IPS SET CONFIGURATION.

ADRCI may also add other incidents that correlate closely in time or in other criteria with the already added incidents.

```
IPS CREATE PACKAGE PROBLEMKEY "problem_key"
```

This command creates a package, and includes diagnostic information for incidents that reference the specified problem key. You can obtain problem keys from the report displayed by the `SHOW INCIDENT` command. Because there can be many incidents with the same problem key, ADRCI adds to the package only the diagnostic information for the first three early incidents, and the last three late incidents with this problem key, excluding incidents that are older than 90 days.

> **Note:**
>
> The number of early and late incidents, and the 90-day age limit are defaults, which you can change. See IPS SET CONFIGURATION.

ADRCI may also add other incidents that correlate closely in time or in other criteria with the already added incidents.

The problem key must be enclosed in single quotation marks (') or double quotation marks (") if it contains spaces or quotation marks.

```
IPS CREATE PACKAGE SECONDS sec
```

This creates a package and includes diagnostic information for all incidents that occurred from *sec* seconds ago until now. *sec* must be an integer.

```
IPS CREATE PACKAGE TIME 'start_time' TO 'end_time'
```

This command creates a package and includes diagnostic information for all incidents that occurred within the specified time range. *start_time* and *end_time* must be in the format 'YYYY-MM-DD HH24:MI:SS.FF TZR'. This string is a valid format string for the NLS_TIMESTAMP_TZ_FORMAT initialization parameter. The fraction (FF) portion of the time is optional, and the HH24:MI:SS delimiters can be either colons or periods.

For example, the following command creates a package with incidents that occurred between July 24th and July 30th of 2010:

```
IPS CREATE PACKAGE TIME '2010-07-24 00:00:00 -07:00' to '2010-07-30 23.59.59 -07:00'
```

Related Topics

- ADRCI Command Reference
  Learn about the commands you can use with the Automatic Diagnostic Repository Command Interpreter (ADRCI).
- Definitions for Oracle Database ADRC
  To understand how to diagnose Oracle Database problems, learn the definitions of terms that Oracle uses for the ADRCI, and the Oracle Database fault diagnosability infrastructure.
- IPS CREATE PACKAGE
  The ADRCI IPS CREATE PACKAGE command creates a new package. ADRCI automatically assigns the package number for the new package.

20.8.2.2 Adding Diagnostic Information to a Logical Incident Package

After you have an existing logical package (**package**) configured for packaging incidents, you can add diagnostic information to that package.

Adding diagnostic information to a logical package enables you to add incident information after the package is created, such the following:

- All diagnostic information for a particular incident
- A named file within the Automatic Diagnostic Repository (ADR).

1. Start ADRCI in interactive mode, and ensure that the **homepath** (a directory within the ADR base hierarchy) points to the correct directory within the ADR base directory hierarchy for the diagnostic information that you want to add.

   See "Definitions for Oracle Database ADRC" for more information about homepath and other ADRCI terms.

2. At the ADRCI prompt, enter the command for the diagnostic information that you want to add:

   ```
   To add all diagnostic information:

   IPS ADD INCIDENT incident_number PACKAGE package_number

   To add a file in the ADR to an existing package:
   ```

- At the ADRCI prompt, enter the following command:

  ```
  IPS ADD FILE filespec PACKAGE package_number
  ```

  filespec must be a fully qualified file name (with path). Only files that are within the ADR base directory hierarchy may be added.

Related Topics

- ADRCI Command Reference
  Learn about the commands you can use with the Automatic Diagnostic Repository Command Interpreter (ADRCI).
- Definitions for Oracle Database ADRC
  To understand how to diagnose Oracle Database problems, learn the definitions of terms that Oracle uses for the ADRCI, and the Oracle Database fault diagnosability infrastructure.

20.8.2.3 Generating a Physical Incident Package

When you generate a package, you create a physical package (a zip file) for an existing logical package.

1. Start ADRCI in interactive mode, and ensure that the homepath (a directory within the ADR base hierarchy) points to the correct directory within the ADR base directory hierarchy.

   See "Definitions for Oracle Database ADRC" for more information about homepath and other ADRCI terms.

2. At the ADRCI prompt, enter the command for the package information that you want to generate (complete or incremental):

   `To generate a complete physical package:`
   The following command generates a complete physical package (zip file) in the path you designate:

   ```
   IPS GENERATE PACKAGE package_number IN path
   ```

   For example, the following command creates a complete physical package in the directory `/home/steve/diagnostics` from logical package number 2:

   ```
   IPS GENERATE PACKAGE 2 IN /home/steve/diagnostics
   ```

   `To generate an incremental physical package`
   You can also generate an incremental package containing only the incidents that have occurred since the last package generation. At the ADRCI prompt, enter the following command:

   ```
   IPS GENERATE PACKAGE package_number IN path INCREMENTAL
   ```

Related Topics

- About Packaging Incidents
  Packaging ADR Command Interpreter (ADRCI) incidents is a three-step process.
- ADRCI Command Reference
  Learn about the commands you can use with the Automatic Diagnostic Repository Command Interpreter (ADRCI).
- Definitions for Oracle Database ADRC
  To understand how to diagnose Oracle Database problems, learn the definitions of terms that Oracle uses for the ADRCI, and the Oracle Database fault diagnosability infrastructure.

20.9 ADRCI Command Reference

Learn about the commands you can use with the Automatic Diagnostic Repository Command Interpreter (ADRCI).

There are four command types in ADRCI:

- Commands that work with one or more current ADR homes
- Commands that work with only one current ADR home, and that issue an error message if there is more than one current ADR home
- Commands that prompt you to select an ADR home when there are multiple current ADR homes
- Commands that do not need a current ADR home

All ADRCI commands support the case where there is a single current ADR home.

> **Note:**
>
> Unless otherwise specified, all commands work with multiple current ADR homes.

- CREATE REPORT
  The ADRCI `CREATE REPORT` command creates a report for the specified report type and run ID, and stores the report in the ADR.
- ECHO
  The ADRCI `ECHO` command prints the input string.
- EXIT
  The ADRCI `EXIT` command exits the ADRCI utility.
- HOST
  The ADRCI `HOST` command runs operating system commands without leaving ADRCI.
- IPS
  The ADRCI `IPS` command calls the Incident Packaging Service (IPS).
- PURGE
  The ADRCI `PURGE` command purges diagnostic data in the current ADR home, according to current purging policies.
- QUIT
  The ADRCI `QUIT` command is a synonym for the `EXIT` command.
- RUN
  The ADRCI `RUN` command runs an ADR Command Interpreter (ADRCI) script.
- SELECT
  The ADRCI `SELECT` command and its functions retrieve qualified diagnostic records for the specified incident or problem.
- SET BASE
  The ADRCI `SET BASE` command sets the ADR base to use in the current ADRCI session.
- SET BROWSER
  The ADRCI `SET BROWSER` command sets the default browser for displaying reports.

- SET CONTROL
  The ADRCI `SET CONTROL` command sets purging policies for Automatic Diagnostic Repository (ADR) contents.
- SET ECHO
  The ADRCI `SET ECHO` command turns command output on or off. This command only affects output being displayed in a script or using the spool mode.
- SET EDITOR
  The ADRCI `SET EDITOR` command sets the editor for displaying the alert log and the contents of trace files.
- SET HOMEPATH
  The ADRCI `SET HOMEPATH` command makes one or more ADR homes current. Many ADR commands work with the current ADR homes only.
- SET TERMOUT
  The ADRCI `SET TERMOUT` command turns output to the terminal on or off.
- SHOW ALERT
  The ADRCI `SHOW ALERT` command shows the contents of the alert log in the default editor.
- SHOW BASE
  The ADRCI `SET EDITOR` command shows the current ADR base.
- SHOW CONTROL
  The ADRCI `SHOW CONTROL` command displays information about the Automatic Diagnostic Repository (ADR), including the purging policy.
- SHOW HM_RUN
  The ADRCI `SHOW HM_RUN` command shows all information for Health Monitor runs.
- SHOW HOMEPATH
  The ADRCI `SHOW HOMEPATH` command is identical to the `SHOW HOMES` command.
- SHOW HOMES
  The ADRCI `SHOW HOMES` command shows the ADR homes in the current ADRCI session.
- SHOW INCDIR
  The ADRCI `SHOW INCDIR` command shows trace files for the specified incident.
- SHOW INCIDENT
  The ADRCI `SHOW INCIDENT` command lists all of the incidents associated with the current ADR home. Includes both open and closed incidents.
- SHOW LOG
  The ADRCI `SHOW LOG` command shows diagnostic log messages.
- SHOW PROBLEM
  The ADRCI `SHOW PROBLEM` command shows problem information for the current ADR home.
- SHOW REPORT
  The ADRCI `SET EDITOR` command shows a report for the specified report type and run name.
- SHOW TRACEFILE
  The ADRCI `SHOW TRACEFILE` command lists trace files.
- SPOOL
  The ADRCI `SET EDITOR` command directs ADRCI output to a file.

20.9.1 CREATE REPORT

The ADRCI `CREATE REPORT` command creates a report for the specified report type and run ID, and stores the report in the ADR.

Purpose

Creates a report for the specified report type and run ID, and stores the report in the ADR. Currently, only the `hm_run` (Health Monitor) report type is supported.

> **Note:**
>
> Results of Health Monitor runs are stored in the ADR in an internal format. To view these results, you must create a Health Monitor report from them and then view the report. You need create the report only once. You can then view it multiple times.

Syntax and Description

```
create report report_type run_name
```

The variable *report_type* must be `hm_run`. *run_name* is a Health Monitor run name. Obtain run names by using the command `SHOW HM_RUN`.

If the report already exists, then it is overwritten. To view the report, use the command `SHOW REPORT`.

This command does not support multiple ADR homes.

Example

This example creates a report for the Health Monitor run with run name `hm_run_1421`:

```
create report hm_run hm_run_1421
```

> **Note:**
>
> `CREATE REPORT REPORT` does not work when multiple ADR homes are set. To set a single ADR home as the target of the command, set the ADRCI home path before using the command.

Related Topics

- SHOW HM_RUN
  The ADRCI `SHOW HM_RUN` command shows all information for Health Monitor runs.
- SHOW REPORT
  The ADRCI `SET EDITOR` command shows a report for the specified report type and run name.

- Setting the ADRCI Homepath Before Using ADRCI Commands
  When diagnosing a problem, you may want to work with diagnostic data from multiple database instances or components, or you may want to focus on diagnostic data from one instance or component.

20.9.2 ECHO

The ADRCI `ECHO` command prints the input string.

Purpose

Prints the input string. You can use this command to print custom text from ADRCI scripts.

Syntax and Description

`ECHO quoted_string`

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

This command does not require an ADR home to be set before you can use it.

Example

These examples print the string "Hello, world!":

```
ECHO "Hello, world!"


ECHO 'Hello, world!'
```

20.9.3 EXIT

The ADRCI `EXIT` command exits the ADRCI utility.

Purpose

Exits the ADRCI utility.

Syntax and Description

```
exit
```

`EXIT` is a synonym for the `QUIT` command.

This command does not require an ADR home to be set before you can use it.

20.9.4 HOST

The ADRCI `HOST` command runs operating system commands without leaving ADRCI.

Purpose

Runs operating system commands without leaving ADRCI.

Syntax and Description

`host ["host_command_string"]`

Use `host` by itself to enter an operating system shell, which allows you to enter multiple operating system commands. Enter `EXIT` to leave the shell and return to ADRCI.

You can also specify the command on the same line (*`host_command_string`*) enclosed in double quotation marks.

This command does not require an ADR home to be set before you can use it.

Examples

```
host


host "ls -l *.pl"
```

20.9.5 IPS

The ADRCI `IPS` command calls the Incident Packaging Service (IPS).

Purpose

Calls the Incident Packaging Service (IPS). The IPS command provides options for creating logical incident packages (packages), adding diagnostic data to packages, and generating physical packages for transmission to Oracle Support.

> **Note:**
>
> IPS commands do not work when multiple ADR homes are set. For information about setting a single ADR home, see Setting the ADRCI Homepath Before Using ADRCI Commands.

- Using the <ADR_HOME> and <ADR_BASE> Variables in IPS Commands
  The ADRCI `IPS` command set provides shortcuts for referencing the current ADR home and ADR base directories.
- IPS ADD
  The ADRCI `IPS ADD` command adds incidents to a package.
- IPS ADD FILE
  The ADRCI `IPS ADD FILE` command adds a file to an existing package.
- IPS ADD NEW INCIDENTS
  The ADRCI `IPS ADD NEW INCIDENTS` command finds and adds new incidents for all of the problems in the specified package.
- IPS COPY IN FILE
  The ADRCI `IPS COPY IN FILE` command copies a file into the ADR from the external file system.
- IPS COPY OUT FILE
  The ADRCI `IPS COPY OUT FILE` command copies a file from the ADR to the external file system.

- IPS CREATE PACKAGE
  The ADRCI `IPS CREATE PACKAGE` command creates a new package. ADRCI automatically assigns the package number for the new package.
- IPS DELETE PACKAGE
  The ADRCI `IPS DELETE PACKAGE` command drops a package and its contents from the ADR.
- IPS FINALIZE
  The ADRCI `IPS FINALIZE` command finalizes a package before uploading.
- IPS GENERATE PACKAGE
  The ADRCI `IPS GENERATE PACKAGE` command creates a physical package (a zip file) in a target directory.
- IPS GET MANIFEST
  The ADRCI `IPS GET MANIFEST` command extracts the manifest from a package zip file and displays it.
- IPS GET METADATA
  The ADRCI `IPS GET METADATA` command extracts ADR-related metadata from a package file and displays it.
- IPS PACK
  The ADRCI `IPS PACK` command creates a package, and generates the physical package immediately
- IPS REMOVE
  The ADRCI `IPS REMOVE` command removes incidents from an existing package.
- IPS REMOVE FILE
  The ADRCI `IPS REMOVE FILE` command removes a file from an existing package.
- IPS SET CONFIGURATION
  The ADRCI `IPS SET CONFIGURATION` command changes the value of an IPS configuration parameter.
- IPS SHOW CONFIGURATION
  The ADRCI `IPS SHOW CONFIGURATION` command displays a list of IPS configuration parameters and their values.
- IPS SHOW FILES
  The ADRCI `IPS SHOW FILES` command lists files included in the specified package.
- IPS SHOW INCIDENTS
  The ADRCI `IPS SHOW INCIDENTS` command lists incidents included in the specified package.
- IPS SHOW PACKAGE
  The ADRCI `IPS SHOW PACKAGE` command displays information about the specified package.
- IPS UNPACK FILE
  The ADRCI `IPS UNPACK FILE` command unpacks a physical package file into the specified path.

> **See Also:**
>
> Packaging Incidents for more information about packaging

20.9.5.1 Using the <ADR_HOME> and <ADR_BASE> Variables in IPS Commands

The ADRCI `IPS` command set provides shortcuts for referencing the current ADR home and ADR base directories.

To access the current ADR home directory, use the `<ADR_HOME>` variable. For example:

```
ips add file <ADR_HOME>/trace/orcl_ora_13579.trc package 12
```

Use the `<ADR_BASE>` variable to access the ADR base directory. For example:

```
ips add file <ADR_BASE>/diag/rdbms/orcl/orcl/trace/orcl_ora_13579.trc package
12
```

> **Note:**
>
> Type the angle brackets (`< >`) as shown.

20.9.5.2 IPS ADD

The ADRCI `IPS ADD` command adds incidents to a package.

Purpose

Adds incidents to a package.

Syntax and Description

```
ips add {incident first [n] | incident inc_id | incident last [n] |
     problem first [n] | problem prob_id | problem last [n] |
     problemkey pr_key | seconds secs | time start_time to end_time}
     package package_id
```

The following table describes the arguments of `IPS ADD`.

Table 20-2 Arguments of IPS ADD command

Argument	Description
`incident first [n]`	Adds the first *n* incidents to the package, where *n* is a positive integer. For example, if *n* is set to 5, then the first five incidents are added. If *n* is omitted, then the default is 1, and the first incident is added.
`incident inc_id`	Adds an incident with ID *inc_id* to the package.
`incident last [n]`	Adds the last *n* incidents to the package, where *n* is a positive integer. For example, if *n* is set to 5, then the last five incidents are added. If *n* is omitted, then the default is 1, and the last incident is added.

Table 20-2 (Cont.) Arguments of IPS ADD command

Argument	Description
`problem first [n]`	Adds the incidents for the first *n* problems to the package, where *n* is a positive integer. For example, if *n* is set to 5, then the incidents for the first five problems are added. If *n* is omitted, then the default is 1, and the incidents for the first problem is added. Adds only the first three early incidents and last three late incidents for each problem, excluding any older than 90 days. (Note: These limits are defaults and can be changed. See "IPS SET CONFIGURATION".)
`problem prob_id`	Adds all incidents with problem ID *prob_id* to the package. Adds only the first three early incidents and last three late incidents for the problem, excluding any older than 90 days. (Note: These limits are defaults and can be changed. See "IPS SET CONFIGURATION".)
`problem last [n]`	Adds the incidents for the last *n* problems to the package, where *n* is a positive integer. For example, if *n* is set to 5, then the incidents for the last five problems are added. If *n* is omitted, then the default is 1, and the incidents for the last problem is added. Adds only the first three early incidents and last three late incidents for each problem, excluding any older than 90 days. (Note: These limits are defaults and can be changed. See "IPS SET CONFIGURATION".)
`problemkey pr_key`	Adds incidents with problem key *pr_key* to the package. Adds only the first three early incidents and last three late incidents for the problem key, excluding any older than 90 days. (Note: These limits are defaults and can be changed.)
`seconds secs`	Adds all incidents that have occurred within *secs*- seconds of the present time.
`time start_time to end_time`	Adds all incidents between *start_time* and *end_time* to the package. Time format is 'YYYY-MM-YY HH24:MI:SS.FF TZR'. Fractional part (FF) is optional.
`package package_id`	Specifies the package to which to add incidents.

Examples

This example adds incident 22 to package 12:

```
ips add incident 22 package 12
```

This example adds the first three early incidents and the last three late incidents with problem ID 6 to package 2, exuding any incidents older than 90 days:

```
ips add problem 6 package 2
```

This example adds all incidents taking place during the last minute to package 5:

```
ips add seconds 60 package 5
```

This example adds all incidents taking place between 10:00 A.M. and 11:00 P.M. on May 1, 2020:

```
ips add  time '2020-05-01 10:00:00.00 -07:00' to '2020-05-01 23:00:00.00 -07:00'
```

20.9.5.3 IPS ADD FILE

The ADRCI `IPS ADD FILE` command adds a file to an existing package.

Syntax and Description

```
ips add file file_name package package_id
```

file_name is the full path name of the file. You can use the `<ADR_HOME>` and `<ADR_BASE>` variables if desired. The file must be under the same ADR base as the package.

package_id is the package ID.

Example

This example adds a trace file to package 12:

```
ips add file <ADR_HOME>/trace/orcl_ora_13579.trc package 12
```

Related Topics

- Using the <ADR_HOME> and <ADR_BASE> Variables in IPS Commands
  The ADRCI `IPS` command set provides shortcuts for referencing the current ADR home and ADR base directories.

20.9.5.4 IPS ADD NEW INCIDENTS

The ADRCI `IPS ADD NEW INCIDENTS` command finds and adds new incidents for all of the problems in the specified package.

Syntax and Description

```
ips add new incidents package package_id
```

package_id is the ID of the package to update. Only new incidents of the problems in the package are added.

Example

This example adds up to three of the new late incidents for the problems in package 12:

```
ips add new incidents package 12
```

> **Note:**
>
> The number of late incidents added is a default that can be changed.

Related Topics

- IPS SET CONFIGURATION
  The ADRCI `IPS SET CONFIGURATION` command changes the value of an IPS configuration parameter.

20.9.5.5 IPS COPY IN FILE

The ADRCI `IPS COPY IN FILE` command copies a file into the ADR from the external file system.

Purpose

To edit a file in a package, you must copy the file out to a designated directory, edit the file, and copy it back into the package. For example, you can use this command to delete sensitive data in the file before sending the package to Oracle Support.

Syntax and Description

```
ips copy in file filename [to new_name][overwrite] package package_id
     [incident incid]
```

Copies an external file, *filename* (specified with full path name) into the ADR, associating it with an existing package, *package_id*, and optionally an incident, *incid*. Use the `to new_name` option to give the copied file a new file name within the ADR. Use the `overwrite` option to overwrite a file that exists already.

Example

This example copies a trace file from the file system into the ADR, associating it with package 2 and incident 4:

```
ips copy in file /home/nick/trace/orcl_ora_13579.trc to <ADR_HOME>/trace/
orcl_ora_13579.trc package 2 incident 4
```

Related Topics

- Using the <ADR_HOME> and <ADR_BASE> Variables in IPS Commands
  The ADRCI `IPS` command set provides shortcuts for referencing the current ADR home and ADR base directories.
- IPS SHOW FILES
  The ADRCI `IPS SHOW FILES` command lists files included in the specified package.

20.9.5.6 IPS COPY OUT FILE

The ADRCI `IPS COPY OUT FILE` command copies a file from the ADR to the external file system.

Purpose

To edit a file in a package, you must copy the file out to a designated directory, edit the file, and copy it back into the package. You may want to do this to delete sensitive data in the file before sending the package to Oracle Support.

Syntax and Description

```
ips copy out file source to target [overwrite]
```

Copies a file, *source*, to a location outside the ADR, *target* (specified with full path name). Use the `overwrite` option to overwrite the file that exists already.

Example

This example copies the file `orcl_ora_13579.trc`, in the trace subdirectory of the current ADR home, to a local folder.

```
ips copy out file <ADR_HOME>/trace/orcl_ora_13579.trc to /home/nick/trace/
orcl_ora_13579.trc
```

Related Topics

- Using the <ADR_HOME> and <ADR_BASE> Variables in IPS Commands
  The ADRCI `IPS` command set provides shortcuts for referencing the current ADR home and ADR base directories.
- IPS SHOW FILES
  The ADRCI `IPS SHOW FILES` command lists files included in the specified package.

20.9.5.7 IPS CREATE PACKAGE

The ADRCI `IPS CREATE PACKAGE` command creates a new package. ADRCI automatically assigns the package number for the new package.

Purpose

Creates a new package. ADRCI automatically assigns the package number for the new package.

Syntax and Description

```
ips create package {incident first [n] | incident inc_id |
     incident last [n] | problem first [n] | problem prob_id |
     problem last [n] | problemkey prob_key | seconds secs |
     time start_time to end_time} [correlate {basic |typical | all}]
```

(Optional) You can add incidents to the new package using the provided options.

Table 20-3 describes the arguments for `IPS CREATE PACKAGE`.

Table 20-3 Arguments of IPS CREATE PACKAGE command

Argument	Description
`incident first [n]`	Adds the first *n* incidents to the package, where *n* is a positive integer. For example, if *n* is set to 5, then the first five incidents are added. If *n* is omitted, then the default is 1, and the first incident is added.
`incident inc_id`	Adds an incident with ID *inc_id* to the package.

Table 20-3 (Cont.) Arguments of IPS CREATE PACKAGE command

Argument	Description
`incident last [n]`	Adds the last *n* incidents to the package, where *n* is a positive integer. For example, if *n* is set to 5, then the last five incidents are added. If *n* is omitted, then the default is 1, and the last incident is added.
`problem first [n]`	Adds the incidents for the first *n* problems to the package, where *n* is a positive integer. For example, if *n* is set to 5, then the incidents for the first five problems are added. If *n* is omitted, then the default is 1, and the incidents for the first problem is added. Adds only the first three early incidents and last three late incidents for each problem, excluding any older than 90 days. (Note: These limits are defaults and can be changed. See "IPS SET CONFIGURATION".)
`problem prob_id`	Adds all incidents with problem ID *prob_id* to the package. Adds only the first three early incidents and last three late incidents for the problem, excluding any older than 90 days. (Note: These limits are defaults and can be changed. See "IPS SET CONFIGURATION".)
`problem last [n]`	Adds the incidents for the last *n* problems to the package, where *n* is a positive integer. For example, if *n* is set to 5, then the incidents for the last five problems are added. If *n* is omitted, then the default is 1, and the incidents for the last problem is added. Adds only the first three early incidents and last three late incidents for each problem, excluding any older than 90 days. (Note: These limits are defaults and can be changed. See "IPS SET CONFIGURATION".)
`problemkey pr_key`	Adds all incidents with problem key *pr_key* to the package. Adds only the first three early incidents and last three late incidents for the problem key, excluding any older than 90 days. (Note: These limits are defaults and can be changed.)
`seconds secs`	Adds all incidents that have occurred within *secs* seconds of the present time.
`time start_time to end_time`	Adds all incidents taking place between *start_time* and *end_time* to the package. Time format is 'YYYY-MM-YY HH24:MI:SS.FF TZR'. Fractional part (FF) is optional.
`correlate {basic |typical | all}`	Selects a method of including correlated incidents in the package. There are three options for this argument: • `correlate basic` includes incident dumps and incident process trace files. • `correlate typical` includes incident dumps and any trace files that were modified within five minutes of each incident. You can alter the time interval by modifying the `INCIDENT_TIME_WINDOW` configuration parameter. • `correlate all` includes the incident dumps, and all trace files that were modified between the time of the first selected incident and the last selected incident. The default value is `correlate typical`.

Examples

This example creates a package with no incidents:

```
ips create package
```

Output:

```
Created package 5 without any contents, correlation level typical
```

This example creates a package containing all incidents between 10 AM and 11 PM on the given day:

```
ips create package time '2010-05-01 10:00:00.00 -07:00' to '2010-05-01 23:00:00.00
-07:00'
```

Output:

```
Created package 6 based on time range 2010-05-01 10:00:00.00 -07:00 to 2010-05-01
23:00:00.00 -07:00, correlation level typical
```

This example creates a package and adds the first three early incidents and the last three late incidents with problem ID 3, excluding incidents that are older than 90 days:

```
ips create package problem 3
```

Output:

```
Created package 7 based on problem id 3, correlation level typical
```

> **Note:**
>
> The number of early and late incidents added, and the 90-day age limit are defaults that can be changed.

Related Topics

- IPS SET CONFIGURATION
  The ADRCI `IPS SET CONFIGURATION` command changes the value of an IPS configuration parameter.
- Creating Incident Packages
  The following topics describe creating incident packages.

20.9.5.8 IPS DELETE PACKAGE

The ADRCI `IPS DELETE PACKAGE` command drops a package and its contents from the ADR.

Syntax and Description

```
ips delete package package_id
```

package_id is the package to delete.

Example

```
ips delete package 12
```

20.9.5.9 IPS FINALIZE

The ADRCI `IPS FINALIZE` command finalizes a package before uploading.

Syntax and Description

```
ips finalize package package_id
```

package_id is the package ID to finalize.

Example

```
ips finalize package 12
```

> **See Also:**
>
> *Oracle Database Administrator's Guide* for more information about finalizing packages

20.9.5.10 IPS GENERATE PACKAGE

The ADRCI `IPS GENERATE PACKAGE` command creates a physical package (a zip file) in a target directory.

Syntax and Description

```
ips generate package package_id [in path] [complete | incremental]
```

package_id is the ID of the package to generate. Optionally, you can save the file in the directory *path*. Otherwise, the package is generated in the current working directory.

The `complete` option means the package forces ADRCI to include all package files. This is the default behavior.

The `incremental` option includes only files that have been added or changed since the last time that this package was generated. With the `incremental` option, the command finishes more quickly.

Example

This example generates a physical package file in path `/home/steve`:

```
ips generate package 12 in /home/steve
```

This example generates a physical package from files added or changed since the last generation:

```
ips generate package 14 incremental
```

> **See Also:**
>
> Generating a Physical Incident Package

20.9.5.11 IPS GET MANIFEST

The ADRCI `IPS GET MANIFEST` command extracts the manifest from a package zip file and displays it.

Syntax and Description

```
ips get manifest from file filename
```

filename is a package zip file. The manifest is an XML-formatted set of metadata for the package file, including information about ADR configuration, correlated files, incidents, and how the package was generated.

This command does not require an ADR home to be set before you can use it.

Example

```
ips get manifest from file /home/steve/ORA603_20060906165316_COM_1.zip
```

20.9.5.12 IPS GET METADATA

The ADRCI `IPS GET METADATA` command extracts ADR-related metadata from a package file and displays it.

Syntax and Description

```
ips get metadata {from file filename | from adr}
```

filename is a package zip file. The metadata in a package file (stored in the file `metadata.xml`) contains information about the ADR home, ADR base, and product.

Use the `from adr` option to get the metadata from a package zip file that has been unpacked into an ADR home using `IPS UNPACK`.

The `from adr` option requires an ADR home to be set.

Example

This example displays metadata from a package file:

```
ips get metadata from file /home/steve/ORA603_20060906165316_COM_1.zip
```

This next example displays metadata from a package file that was unpacked into the directory /scratch/oracle/package1:

```
set base /scratch/oracle/package1
ips get metadata from adr
```

In this previous example, upon receiving the `SET BASE` command, ADRCI automatically adds to the homepath the ADR home that was created in `/scratch/oracle/package1` by the `IPS UNPACK FILE` command.

> **See Also:**
>
> IPS UNPACK FILE for more information about unpacking package files

20.9.5.13 IPS PACK

The ADRCI `IPS PACK` command creates a package, and generates the physical package immediately

Purpose

Creates a package, and generates the physical package immediately.

Syntax and Description

```
ips pack [incident first [n] | incident inc_id | incident last [n] |
     problem first [n] | problem prob_id | problem last [n] |
     problemkey prob_key | seconds secs | time start_time to end_time]
     [correlate {basic |typical | all}] [in path]
```

ADRCI automatically generates the package number for the new package. `IPS PACK` creates an empty package if no package contents are specified.

Table 20-4 describes the arguments for `IPS PACK`.

Table 20-4 Arguments of IPS PACK command

Argument	Description
`incident first [n]`	Adds the first *n* incidents to the package, where *n* is a positive integer. For example, if *n* is set to 5, then the first five incidents are added. If *n* is omitted, then the default is 1, and the first incident is added.
`incident inc_id`	Adds an incident with ID *inc_id* to the package.
`incident last [n]`	Adds the last *n* incidents to the package, where *n* is a positive integer. For example, if *n* is set to 5, then the last five incidents are added. If *n* is omitted, then the default is 1, and the last incident is added.

Table 20-4 (Cont.) Arguments of IPS PACK command

Argument	Description
`problem first [n]`	Adds the incidents for the first *n* problems to the package, where *n* is a positive integer. For example, if *n* is set to 5, then the incidents for the first five problems are added. If *n* is omitted, then the default is 1, and the incidents for the first problem is added. Adds only the first three early incidents and last three late incidents for each problem, excluding any older than 90 days. (Note: These limits are defaults and can be changed. See "IPS SET CONFIGURATION".)
`problem prob_id`	Adds all incidents with problem ID *prob_id* to the package. Adds only the first three early incidents and last three late incidents for the problem, excluding any older than 90 days. (Note: These limits are defaults and can be changed. See "IPS SET CONFIGURATION".)
`problem last [n]`	Adds the incidents for the last *n* problems to the package, where *n* is a positive integer. For example, if *n* is set to 5, then the incidents for the last five problems are added. If *n* is omitted, then the default is 1, and the incidents for the last problem is added. Adds only the first three early incidents and last three late incidents for each problem, excluding any older than 90 days. (Note: These limits are defaults and can be changed. See "IPS SET CONFIGURATION".)
`problemkey pr_key`	Adds incidents with problem key *pr_key* to the package. Adds only the first three early incidents and last three late incidents for the problem key, excluding any older than 90 days. (Note: These limits are defaults and can be changed.)
`seconds secs`	Adds all incidents that have occurred within *secs* seconds of the present time.
`time start_time to end_time`	Adds all incidents taking place between *start_time* and *end_time* to the package. Time format is 'YYYY-MM-YY HH24:MI:SS.FF TZR'. Fractional part (FF) is optional.
`correlate {basic |typical | all}`	Selects a method of including correlated incidents in the package. There are three options for this argument: • `correlate basic` includes incident dumps and incident process trace files. • `correlate typical` includes incident dumps and any trace files that were modified within five minutes of each incident. You can alter the time interval by modifying the `INCIDENT_TIME_WINDOW` configuration parameter. • `correlate all` includes the incident dumps, and all trace files that were modified between the time of the first selected incident and the last selected incident. The default value is `correlate typical`.
`in path`	Saves the physical package to directory *path*.

Example

This example creates an empty package:

```
ips pack
```

This example creates a physical package containing all information for incident 861:

```
ips pack incident 861
```

This example creates a physical package for all incidents in the last minute, fully correlated:

```
ips pack seconds 60 correlate all
```

Related Topics

- IPS SET CONFIGURATION
  The ADRCI `IPS SET CONFIGURATION` command changes the value of an IPS configuration parameter.

20.9.5.14 IPS REMOVE

The ADRCI `IPS REMOVE` command removes incidents from an existing package.

Purpose

Removes incidents from an existing package.

Syntax and Description

```
ips remove {incident inc_id | problem prob_id | problemkey prob_key}
     package package_id
```

After removing incidents from a package, the incidents continue to be tracked within the package metadata to prevent ADRCI from automatically including them later (such as with `ADD NEW INCIDENTS`).

The following table describes the arguments of `IPS REMOVE`.

Table 20-5 Arguments of IPS REMOVE command

Argument	Description
`incident inc_id`	Removes the incident with ID *inc_id* from the package
`problem prob_id`	Removes all incidents with problem ID *prob_id* from the package
`problemkey pr_key`	Removes all incidents with problem key *pr_key* from the package
`package package_id`	Removes incidents from the package with ID *package_id*.

Example

This example removes incident 22 from package 12:

```
ips remove incident 22 package 12
```

Related Topics

- IPS GET MANIFEST
  The ADRCI `IPS GET MANIFEST` command extracts the manifest from a package zip file and displays it.

20.9.5.15 IPS REMOVE FILE

The ADRCI `IPS REMOVE FILE` command removes a file from an existing package.

Syntax and Description

```
ips remove file file_name package package_id
```

file_name is the file to remove from package *package_id*. The complete path of the file must be specified. (You can use the <ADR_HOME> and <ADR_BASE> variables if desired.)

After removal, the file continues to be tracked within the package metadata to prevent ADRCI from automatically including it later (such as with `ADD NEW INCIDENTS`). Removing a file, therefore, only sets the `EXCLUDE` flag for the file to `Explicitly excluded`.

Example

This example removes a trace file from package 12:

```
ips remove file <ADR_HOME>/trace/orcl_ora_13579.trc package 12
Removed file <ADR_HOME>/trace/orcl_ora_13579.trc from package 12
ips show files package 12

.
.
.
FILE_ID                 4
FILE_LOCATION           <ADR_HOME>/trace
FILE_NAME               orcl_ora_13579.trc
LAST_SEQUENCE           0
EXCLUDE                 Explicitly excluded
.
.
.
```

See Also:

- IPS GET MANIFEST for information about package metadata
- Using the <ADR_HOME> and <ADR_BASE> Variables in IPS Commands for information about the <ADR_BASE> directory syntax
- IPS SHOW FILES

20.9.5.16 IPS SET CONFIGURATION

The ADRCI `IPS SET CONFIGURATION` command changes the value of an IPS configuration parameter.

Syntax and Description

```
ips set configuration {parameter_id | parameter_name} value
```

parameter_id is the ID of the parameter to change, and *parameter_name* is the name of the parameter to change. *value* is the new value. For a list of the configuration parameters and their IDs, use `IPS SHOW CONFIGURATION`.

Example

```
ips set configuration 3 10
```

Related Topics

- IPS SHOW CONFIGURATION
  The ADRCI `IPS SHOW CONFIGURATION` command displays a list of IPS configuration parameters and their values.

20.9.5.17 IPS SHOW CONFIGURATION

The ADRCI `IPS SHOW CONFIGURATION` command displays a list of IPS configuration parameters and their values.

Purpose

These parameters control various thresholds for IPS data, such as timeouts and incident inclusion intervals.

Syntax and Description

```
ips show configuration {parameter_id | parameter_name}]
```

`IPS SHOW CONFIGURATION` lists the following information for each configuration parameter:

- Parameter ID
- Name
- Description
- Unit used by parameter (such as days or hours)
- Value
- Default value
- Minimum Value
- Maximum Value
- Flags

Optionally, you can get information about a specific parameter by supplying a *parameter_id* or a *parameter_name*.

Example

This command describes all IPS configuration parameters:

```
ips show configuration
```

Output:

```
PARAMETER INFORMATION:
   PARAMETER_ID          1
   NAME                  CUTOFF_TIME
   DESCRIPTION           Maximum age for an incident to be considered for
                         inclusion
   UNIT                  Days
   VALUE                 90
   DEFAULT_VALUE         90
   MINIMUM               1
   MAXIMUM               4294967295
   FLAGS                 0

PARAMETER INFORMATION:
   PARAMETER_ID          2
   NAME                  NUM_EARLY_INCIDENTS
   DESCRIPTION           How many incidents to get in the early part of the
range
   UNIT                  Number
   VALUE                 3
   DEFAULT_VALUE         3
   MINIMUM               1
   MAXIMUM               4294967295
   FLAGS                 0

PARAMETER INFORMATION:
   PARAMETER_ID          3
   NAME                  NUM_LATE_INCIDENTS
   DESCRIPTION           How many incidents to get in the late part of the
range
   UNIT                  Number
   VALUE                 3
   DEFAULT_VALUE         3
   MINIMUM               1
   MAXIMUM               4294967295
   FLAGS                 0

PARAMETER INFORMATION:
   PARAMETER_ID          4
   NAME                  INCIDENT_TIME_WINDOW
   DESCRIPTION           Incidents this close to each other are considered
                         correlated
   UNIT                  Minutes
   VALUE                 5
   DEFAULT_VALUE         5
```

```
   MINIMUM                  1
   MAXIMUM                  4294967295
   FLAGS                    0

PARAMETER INFORMATION:
   PARAMETER_ID             5
   NAME                     PACKAGE_TIME_WINDOW
   DESCRIPTION              Time window for content inclusion is from x hours
                            before first included incident to x hours after
last
                            incident
   UNIT                     Hours
   VALUE                    24
   DEFAULT_VALUE            24
   MINIMUM                  1
   MAXIMUM                  4294967295
   FLAGS                    0

PARAMETER INFORMATION:
   PARAMETER_ID             6
   NAME                     DEFAULT_CORRELATION_LEVEL
   DESCRIPTION              Default correlation level for packages
   UNIT                     Number
   VALUE                    2
   DEFAULT_VALUE            2
   MINIMUM                  1
   MAXIMUM                  4
   FLAGS                    0
```

Examples

This command describes configuration parameter `NUM_EARLY_INCIDENTS`:

```
ips show configuration num_early_incidents
```

This command describes configuration parameter 3:

```
ips show configuration 3
```

Configuration Parameter Descriptions

The following table describes the IPS configuration parameters in detail.

Table 20-6 IPS Configuration Parameters

Parameter	ID	Description
CUTOFF_TIME	1	Maximum age, in days, for an incident to be considered for inclusion.
NUM_EARLY_INCIDENTS	2	Number of incidents to include in the early part of the range when creating a package based on a problem. By default, ADRCI adds the three earliest incidents and three most recent incidents to the package.

Table 20-6 (Cont.) IPS Configuration Parameters

Parameter	ID	Description
NUM_LATE_INCIDENTS	3	Number of incidents to include in the late part of the range when creating a package based on a problem. By default, ADRCI adds the three earliest incidents and three most recent incidents to the package.
INCIDENT_TIME_WINDOW	4	Number of minutes between two incidents in order for them to be considered correlated.
PACKAGE_TIME_WINDOW	5	Number of hours to use as a time window for including incidents in a package. For example, a value of 5 includes incidents five hours before the earliest incident in the package, and five hours after the most recent incident in the package.
DEFAULT_CORRELATION_LEVEL	6	The default correlation level to use for correlating incidents in a package. The correlation levels are: • 1 (basic): includes incident dumps and incident process trace files. • 2 (typical): includes incident dumps and any trace files that were modified within the time window specified by INCIDENT_TIME_WINDOW (see above). • 4 (all): includes the incident dumps, and all trace files that were modified between the first selected incident and the last selected incident. Additional incidents can be included automatically if they occurred in the same time range.

Related Topics

- IPS SET CONFIGURATION
  The ADRCI `IPS SET CONFIGURATION` command changes the value of an IPS configuration parameter.

20.9.5.18 IPS SHOW FILES

The ADRCI `IPS SHOW FILES` command lists files included in the specified package.

Purpose

Lists files included in the specified package.

Syntax and Description

```
ips show files package package_id
```

package_id is the package ID to display.

Example

This example shows all files associated with package 1:

```
ips show files package 1
```

Output:

```
   FILE_ID                 1
   FILE_LOCATION           <ADR_HOME>/alert
   FILE_NAME               log.xml
   LAST_SEQUENCE           1
   EXCLUDE                 Included

   FILE_ID                 2
   FILE_LOCATION           <ADR_HOME>/trace
   FILE_NAME               alert_adcdb.log
   LAST_SEQUENCE           1
   EXCLUDE                 Included

   FILE_ID                 27
   FILE_LOCATION           <ADR_HOME>/incident/incdir_4937
   FILE_NAME               adcdb_ora_692_i4937.trm
   LAST_SEQUENCE           1
   EXCLUDE                 Included

   FILE_ID                 28
   FILE_LOCATION           <ADR_HOME>/incident/incdir_4937
   FILE_NAME               adcdb_ora_692_i4937.trc
   LAST_SEQUENCE           1
   EXCLUDE                 Included

   FILE_ID                 29
   FILE_LOCATION           <ADR_HOME>/trace
   FILE_NAME               adcdb_ora_692.trc
   LAST_SEQUENCE           1
   EXCLUDE                 Included

   FILE_ID                 30
   FILE_LOCATION           <ADR_HOME>/trace
   FILE_NAME               adcdb_ora_692.trm
   LAST_SEQUENCE           1
   EXCLUDE                 Included
.
.
.
```

20.9.5.19 IPS SHOW INCIDENTS

The ADRCI `IPS SHOW INCIDENTS` command lists incidents included in the specified package.

Syntax and Description

`ips show incidents package` *`package_id`*

`package_id` is the package ID to display.

Example

This example lists the incidents in package 1:

```
ips show incidents package 1
```

Output:

```
MAIN INCIDENTS FOR PACKAGE 1:
   INCIDENT_ID            4985
   PROBLEM_ID             1
   EXCLUDE                Included

CORRELATED INCIDENTS FOR PACKAGE 1:
```

20.9.5.20 IPS SHOW PACKAGE

The ADRCI `IPS SHOW PACKAGE` command displays information about the specified package.

Syntax and Description

```
ips show package package_id {basic | brief | detail}
```

package_id is the ID of the package to display.

Use the `basic` option to display a minimal amount of information. It is the default when no *package_id* is specified.

Use the `brief` option to display more information about the package than the `basic` option. It is the default when a *package_id* is specified.

Use the `detail` option to show the information displayed by the `brief` option, as well as some package history and information about the included incidents and files.

Example

```
ips show package 12

ips show package 12 brief
```

20.9.5.21 IPS UNPACK FILE

The ADRCI `IPS UNPACK FILE` command unpacks a physical package file into the specified path.

Syntax and Description

```
ips unpack file file_name [into path]
```

file_name is the full path name of the physical package (zip file) to unpack. Optionally, you can unpack the file into directory *path*, which must exist, and muste be writable. If you omit the

path, then the current working directory is used. The destination directory is treated as an ADR base, and the entire ADR base directory hierarchy is created, including a valid ADR home.

This command does not require an ADR home to be set before you can use it.

Example

```
ips unpack file /tmp/ORA603_20060906165316_COM_1.zip into /tmp/newadr
```

20.9.6 PURGE

The ADRCI `PURGE` command purges diagnostic data in the current ADR home, according to current purging policies.

Purpose

Purges diagnostic data in the current ADR home, according to current purging policies. Only ADR contents that are due to be purged are purged.

Diagnostic data in the ADR has a default lifecycle. For example, information about incidents and problems is subject to purging after one year, whereas the associated dump files (dumps) are subject to purging after only 30 days.

Some Oracle products, such as Oracle Database, automatically purge diagnostic data at the end of its life cycle. Other products and components require you to purge diagnostic data manually with this command. You can also use this command to purge data that is due to be automatically purged.

The SHOW CONTROL command displays the default purging policies for short-lived ADR contents and long-lived ADR contents.

Syntax and Description

```
purge [-i {id | start_id end_id} |
  -age mins [-type {ALERT|INCIDENT|TRACE|CDUMP|HM|UTSCDMP}]]
```

The following table describes the flags for `PURGE`.

Table 20-7 Flags for the PURGE command

Flag	Description
`-i {id1 | start_id end_id}`	Purges either a specific incident ID (*id*) or a range of incident IDs (*start_id* and *end_id*)
`-age mins`	Purges only data older than *mins* minutes.

Table 20-7 (Cont.) Flags for the PURGE command

Flag	Description
`-type {ALERT|INCIDENT|TRACE|CDUMP|HM|UTSCDMP}`	Specifies the type of diagnostic data to purge. Used with the `-age` clause. The following types can be specified: • `ALERT` - Alert logs • `INCIDENT` - Incident data • `TRACE` - Trace files (including dumps) • `CDUMP` - Core dump files • `HM` - Health Monitor run data and reports • `UTSCDMP` - Dumps of in-memory traces for each session The `UTSCDMP` data is stored in directories under the trace directory. Each of these directories is named cdmp_*timestamp*. In response to a critical error (such as an ORA-600 or ORA-7445 error), a background process creates such a directory and writes each session's in-memory tracing data into a trace file. This data might be useful in determining what the instance was doing in the seconds leading up to the failure.

Examples

This example purges all diagnostic data in the current ADR home based on the default purging policies:

```
purge
```

This example purges all diagnostic data for all incidents between 123 and 456:

```
purge -i 123 456
```

This example purges all incident data from before the last hour:

```
purge -age 60 -type incident
```

> **Note:**
>
> `PURGE` does not work when multiple ADR homes are set. For information about setting a single ADR home, see "Setting the ADRCI Homepath Before Using ADRCI Commands".

20.9.7 QUIT

The ADRCI `QUIT` command is a synonym for the `EXIT` command.

Related Topics

- EXIT
  The ADRCI `EXIT` command exits the ADRCI utility.

20.9.8 RUN

The ADRCI `RUN` command runs an ADR Command Interpreter (ADRCI) script.

Syntax and Description

```
run script_name

@ script_name

@@ script_name
```

The variable *`script_name`* is the file containing the ADRCI commands that you want to run. ADRCI looks for the script in the current directory, unless a full path name is supplied. If the file name is given without a file extension, then ADRCI uses the default extension `.adi`.

The `run` and `@` commands are synonyms. The `@@` command is similar to `run` and `@`. However, when used inside a script, `@@` uses the path of the calling script to locate *`script_name`*, rather than the current directory.

You are not required to have an ADR home set before you can use the `run` command.

Example

```
run my_script

@my_script
```

20.9.9 SELECT

The ADRCI `SELECT` command and its functions retrieve qualified diagnostic records for the specified incident or problem.

Purpose

Retrieves qualified records for the specified incident or problem, to assist with diagnosing the issue.

Syntax and Description

```
select {*|[field1, [field2, ...]} FROM {incident|problem}
  [WHERE predicate_string]
  [ORDER BY field1 [, field2, ...] [ASC|DSC|DESC]]
  [GROUP BY field1 [, field2, ...]]
  [HAVING having_predicate_string]
```

Table 20-8 Flags for the SELECT command

Flag	Description
`field1, field2, ...`	Lists the fields to retrieve. If `*` is specified, then all fields are retrieved.
`incident|problem`	Indicates whether to query incidents or problems.

Table 20-8 (Cont.) Flags for the SELECT command

Flag	Description
`WHERE "predicate_string"`	Uses a SQL-like predicate string to show only the incident or problem for which the predicate is true. The predicate string must be enclosed in double quotation marks. SHOW INCIDENT lists the fields that can be used in the predicate string incidents. SHOW PROBLEM lists the fields that can be used in the predicate string for problems.
`ORDER BY field1, field2, ... [ASC|DSC|DESC]`	Show results sorted by field in the given order, as well as in ascending (`ASC`) and descending order (`DSC` or `DESC`). When the `ORDER BY` clause is specified, results are shown in ascending order by default.
`GROUP BY field1, field2, ...`	Show results grouped by the specified fields. The `GROUP BY` flag groups rows but does not guarantee the order of the result set. To order the groupings, use the `ORDER BY` flag.
`HAVING "having_predicate_string"`	Restrict the groups of returned rows to those groups for which the having predicate is true. The `HAVING` flag must be used in combination with the `GROUP BY` flag.

Note:

The `WHERE`, `ORDER BY`, `GROUP BY`, and `HAVING` flags are similar to the clauses with the same names in a `SELECT` SQL statement.

See *Oracle Database SQL Language Reference* for more information about the clauses in a `SELECT` SQL statement.

Restrictions

The following restrictions apply when you use the `SELECT` command:

- The command cannot join more than two tables.
- The command cannot use table aliases.
- The command can use only a limited set of functions, which are listed in this section.
- The command cannot use column wildcard (`"*"`) when joining tables or when using the `GROUP BY` clause.
- Statements must be on a single line.
- Statement cannot have subqueries.
- Statement cannot have a `WITH` clause.
- A limited set of pseudocolumns are allowed. For example, `ROWNUM` is allowed, but `ROWID` is not allowed.

Examples

This example retrieves the `incident_id` and `create_time` for incidents with an `incident_id` greater than `1`:

```
select incident_id, create_time from incident where incident_id > 1
```

The following is an example of output for this query:

```
INCIDENT_ID          CREATE_TIME
-------------------- ---------------------------------------
4801                 2011-05-27 10:10:26.541656 -07:00
4802                 2011-05-27 10:11:02.456066 -07:00
4803                 2011-05-27 10:11:04.759654 -07:00
```

This example retrieves the `problem_id` and `first_incident` for each problem with a `problem_key` that includes `600`:

```
select problem_id, first_incident from problem where problem_key like '%600%'
```

The following is an example of output for this query:

```
PROBLEM_ID           FIRST_INCIDENT
-------------------- --------------------
1                    4801
2                    4802
3                    4803
```

Functions

This section describes functions that you can use with the `SELECT` command.

The purpose and syntax of these functions are similar to the corresponding SQL functions, but there are some differences between SQL functions and the functions used with the ADRCI utility.

The following restrictions apply to all of the ADRCI functions:

- The expressions must be simple expressions.

  See *Oracle Database SQL Language Reference* for information about simple expressions.
- You cannot combine function calls. For example, the following combination of function calls is not supported:

  ```
  sum(length(column_name))
  ```
- No functions are overloaded.
- All function arguments are mandatory.
- The functions cannot be used with other ADRCI Utility commands.
- AVG
  The `AVG` function of the `ADRC SELECT` command returns the average value of an expression.
- CONCAT
  The `CONCAT` function of the `ADRC SELECT` command returns a concatenation of two character strings.
- COUNT
  The `COUNT` function of the ADRC `SELECT` command returns the number of rows returned by a query.
- DECODE
  The `DECODE` function of the `ADRC SELECT` command compares an expression to each search value one by one.

- LENGTH
  The `LENGTH` function of the `ADRC SELECT` command returns the length of a character string using as defined by the input character set.
- MAX
  The `MAX` function of the `ADRC SELECT` command returns the maximum value of an expression.
- MIN
  The `MIN` function of the `ADRC SELECT` command returns the minimum value of an expression.
- NVL
  The `NVL` function of the `ADRC SELECT` command replaces null (returned as a blank) with character data in the results of a query.
- REGEXP_LIKE
  The `REGEXP_LIKE` function of the `ADRC SELECT` command returns rows that match a specified pattern in a specified regular expression.
- SUBSTR
  The `SUBSTR` function of the `ADRC SELECT` command returns a portion of character data.
- SUM
  The `SUM` function of the `ADRC SELECT` command returns the sum of values of an expression.
- TIMESTAMP_TO_CHAR
  The `TIMESTAMP_TO_CHAR` function of the `ADRC SELECT` command converts a value of `TIMESTAMP` data type to a value of `VARCHAR2` data type in a specified format.
- TOLOWER
  The `TOLOWER` function of the `ADRC SELECT` command returns character data, with all letters lowercase.
- TOUPPER
  The `TOUPPER` function of the `ADRC SELECT` command returns character data, with all letters uppercase.

20.9.9.1 AVG

The `AVG` function of the `ADRC SELECT` command returns the average value of an expression.

Purpose

Returns the average value of an expression.

Syntax

See the description of `AVG` in *Oracle Database SQL Language Reference*.

Restrictions

The following restrictions apply when you use the `AVG` function in the `SELECT` command:

- The expression must be a numeric column or a positive numeric constant.
- The function does not support the `DISTINCT` or `ALL` keywords.
- The function does not support the `OVER` clause.

Related Topics

- *Oracle Database SQL Language Reference* AVG

20.9.9.2 CONCAT

The `CONCAT` function of the `ADRC SELECT` command returns a concatenation of two character strings.

Purpose

Returns a concatenation of two character strings. The character data can be of the data types `CHAR` and `VARCHAR2`. The return value is the same data type as the character data.

Syntax

See the description of `CONCAT` in *Oracle Database SQL Language Reference*.

Restrictions

The following restrictions apply when you use the `CONCAT` function in the `SELECT` command:

- The function does not support LOB data types, including `BLOB`, `CLOB`, `NCLOB`, and `BFILE` data types.
- The function does not support national character set data types, including `NCHAR`, `NVARCHAR2`, and `NCLOB` data types.

Related Topics

- *Oracle Database SQL Language Reference* CONCAT

20.9.9.3 COUNT

The `COUNT` function of the ADRC `SELECT` command returns the number of rows returned by a query.

Purpose

Returns the number of rows returned by the query.

Syntax

See the description of `COUNT` in *Oracle Database SQL Language Reference*.

Restrictions

The following restrictions apply when you use the `COUNT` function in the `SELECT` command:

- The expression must be a column, a numeric constant, or a string constant.
- The function does not support the `DISTINCT` or `ALL` keywords.
- The function does not support the `OVER` clause.
- The function always counts all rows for the query, including duplicates and nulls.

Examples

This example returns the number of incidents for which `flood_controlled` is `0` (zero):

```
select count(*) from incident where flood_controlled = 0;
```

This example returns the number of problems for which `problem_key` includes `ORA-600`:

```
select count(*) from problem where problem_key like '%ORA-600%';
```

Related Topics

- *Oracle Database SQL Language Reference* COUNT

20.9.9.4 DECODE

The `DECODE` function of the `ADRC SELECT` command compares an expression to each search value one by one.

Purpose

Compares an expression to each search value one by one. If the expression is equal to a search, then Oracle Database returns the corresponding result. If no match is found, then the database returns the specified default value.

Syntax

See the description of `DECODE` in *Oracle Database SQL Language Reference*.

Restrictions

The following restrictions apply when you use the `DECODE` function in the `SELECT` command:

- The search arguments must be character data.
- A default value must be specified.

Example

This example shows each `incident_id` and whether or not the incident is flood-controlled. The example uses the `DECODE` function to display text instead of numbers for the `flood_controlled` field.

```
select incident_id, decode(flood_controlled, 0, \
  "Not flood-controlled", "Flood-controlled") from incident;
```

Related Topics

- *Oracle Database SQL Language Reference* DECODE

20.9.9.5 LENGTH

The `LENGTH` function of the `ADRC SELECT` command returns the length of a character string using as defined by the input character set.

Purpose

Returns the length of a character string using as defined by the input character set. The character string can be any of the data types `CHAR`, `VARCHAR2`, `NCHAR`, `NVARCHAR2`, `CLOB`, or `NCLOB`. The return value is of data type `NUMBER`. If the character sting has data type `CHAR`, then the length includes all trailing blanks. If the character string is null, then this function returns 0 (zero).

> **Note:**
>
> The SQL function returns null if the character string is null.

Syntax

See the description of `LENGTH` in *Oracle Database SQL Language Reference*.

Restrictions

The `ADRC SELECT` command does not support the following functions: `LENGTHB`, `LENGTHC`, `LENGTH2`, and `LENGTH4`.

Example

This example shows the `problem_id` and the length of the `problem_key` for each problem.

```
select problem_id, length(problem_key) from problem;
```

Related Topics

- *Oracle Database SQL Language Reference* LENGTH

20.9.9.6 MAX

The `MAX` function of the `ADRC SELECT` command returns the maximum value of an expression.

Syntax

See MAX in *Oracle Database SQL Language Reference*

Restrictions

The following restrictions apply when you use the `MAX` function in the `SELECT` command:

- The function does not support the `DISTINCT` or `ALL` keywords.
- The function does not support the `OVER` clause.

Example

This example shows the maximum `last_incident` value for all of the recorded problems.

```
select max(last_incident) from problem;
```

20.9.9.7 MIN

The `MIN` function of the `ADRC SELECT` command returns the minimum value of an expression.

Syntax

See MIN in *Oracle Database SQL Language Reference*

Restrictions

The following restrictions apply when you use the `MIN` function in the `SELECT` command:

- The function does not support the `DISTINCT` or `ALL` keywords.
- The function does not support the `OVER` clause.

Example

This example shows the minimum `first_incident` value for all of the recorded problems.

```
select min(first_incident) from problem;
```

20.9.9.8 NVL

The `NVL` function of the `ADRC SELECT` command replaces null (returned as a blank) with character data in the results of a query.

Purpose

If the first expression specified is null, then `NVL` returns second expression specified. If first expression specified is not null, then `NVL` returns the value of the first expression.

Syntax

See NVL in *Oracle Database SQL Language Reference*

Restrictions

The following restrictions apply when you use the `NVL` function in the `SELECT` command:

- The replacement value (second expression) must be specified as character data.
- The function does not support data conversions.

Example

This example replaces `NULL` in the output for `signalling_component` with the text "No component."

```
select nvl(signalling_component, 'No component') from incident;
```

20.9.9.9 REGEXP_LIKE

The `REGEXP_LIKE` function of the `ADRC SELECT` command returns rows that match a specified pattern in a specified regular expression.

Purpose

In SQL, `REGEXP_LIKE` is a condition instead of a function.

Syntax

See REGEXP_LIKE Condition in *Oracle Database SQL Language Reference*

Restrictions

The following restrictions apply when you use the `REGEXP_LIKE` function in the `SELECT` command:

- The pattern match is always case-sensitive.

- The function does not support the `match_param` argument.

Example

This example shows the `problem_id` and `problem_key` for all problems where the `problem_key` ends with a number.

```
select problem_id, problem_key from problem \
  where regexp_like(problem_key, '[0-9]$') = true
```

20.9.9.10 SUBSTR

The `SUBSTR` function of the `ADRC SELECT` command returns a portion of character data.

Purpose

The portion of data returned begins at the specified position and is the specified substring length characters long. `SUBSTR` calculates lengths using characters as defined by the input character set.

Syntax

See SUBSTR in *Oracle Database SQL Language Reference*

Restrictions

The following restrictions apply when you use the `SUBSTR` function in the `SELECT` command:

- The function supports only positive integers. It does not support negative values or floating-point numbers.
- The `SELECT` command does not support the following functions: `SUBSTRB`, `SUBSTRC`, `SUBSTR2`, and `SUBSTR4`.

Example

This example shows each `problem_key` starting with the fifth character in the key.

```
select substr(problem_key, 5) from problem;
```

20.9.9.11 SUM

The `SUM` function of the `ADRC SELECT` command returns the sum of values of an expression.

Syntax

See SUM in *Oracle Database SQL Language Reference*

Restrictions

The following restrictions apply when you use the `SUM` function in the `SELECT` command:

- The expression must be a numeric column or a numeric constant.
- The function does not support the `DISTINCT` or `ALL` keywords.
- The function does not support the `OVER` clause.

20.9.9.12 TIMESTAMP_TO_CHAR

The `TIMESTAMP_TO_CHAR` function of the `ADRC SELECT` command converts a value of `TIMESTAMP` data type to a value of `VARCHAR2` data type in a specified format.

Purpose

If you do not specify a format, then the function converts values to the default timestamp format.

Syntax

See the syntax of the `TO_CHAR` function (TO_CHAR (datetime)) in *Oracle Database SQL Language Reference*

Restrictions

The following restrictions apply when you use the `TIMESTAMP_TO_CHAR` function in the `SELECT` command:

- The function converts only `TIMESTAMP` data type. `TIMESTAMP WITH TIME ZONE`, `TIMESTAMP WITH LOCAL TIME ZONE`, and other data types are not supported.
- The function does not support the `nlsparm` argument. The function uses the default language for your session.

Example

This example converts the create_time for each incident from a `TIMESTAMP` data type to a `VARCHAR2` data type in the `DD-MON-YYYY` format.

```
select timestamp_to_char(create_time, 'DD-MON-YYYY') from incident;
```

20.9.9.13 TOLOWER

The `TOLOWER` function of the `ADRC SELECT` command returns character data, with all letters lowercase.

Purpose

The character data can be of the data types `CHAR` and `VARCHAR2`. The return value is the same data type as the character data. The database sets the case of the characters based on the binary mapping defined for the underlying character set.

Syntax

See the syntax of the `LOWER` function (LOWER) in *Oracle Database SQL Language Reference*

Restrictions

The following restrictions apply when you use the `TOLOWER` function in the `SELECT` command:

- The function does not support LOB data types, including `BLOB`, `CLOB`, `NCLOB`, and `BFILE` data types.
- The function does not support national character set data types, including `NCHAR`, `NVARCHAR2`, and `NCLOB` data types.

Example

This example shows each `problem_key` in all lowercase letters.

```
select tolower(problem_key) from problem;
```

20.9.9.14 TOUPPER

The `TOUPPER` function of the `ADRC SELECT` command returns character data, with all letters uppercase.

Purpose

The character data can be of the data types `CHAR` and `VARCHAR2`. The return value is the same data type as the character data. The database sets the case of the characters based on the binary mapping defined for the underlying character set.

Syntax

See the syntax of the `UPPER` function (UPPER) in *Oracle Database SQL Language Reference*

Restrictions

The following restrictions apply when you use the `TOUPPER` function in the `SELECT` command:

- The function does not support LOB data types, including `BLOB`, `CLOB`, `NCLOB`, and `BFILE` data types.
- The function does not support national character set data types, including `NCHAR`, `NVARCHAR2`, and `NCLOB` data types.

Example

This example shows each `problem_key` in all uppercase letters.

```
select toupper(problem_key) from problem;
```

20.9.10 SET BASE

The ADRCI `SET BASE` command sets the ADR base to use in the current ADRCI session.

Syntax and Description

```
set base base_str
```

`base_str` is a full path to a directory. The format for *`base_str`* depends on the operating system. If there are valid ADR homes under the base directory, these homes are added to the home path of the current ADRCI session.

This command does not require an ADR home to be set before you can use it.

Example

```
set base /u01/app/oracle
```

Related Topics

- Definitions for Oracle Database ADRC
  To understand how to diagnose Oracle Database problems, learn the definitions of terms that Oracle uses for the ADRCI, and the Oracle Database fault diagnosability infrastructure.

20.9.11 SET BROWSER

The ADRCI `SET BROWSER` command sets the default browser for displaying reports.

Syntax and Description

> **Note:**
>
> This command is reserved for future use. At this time ADRCI does not support HTML-formatted reports in a browser.

```
set browser browser_program
```

browser_program is the browser program name (it is assumed the browser can be started from the current ADR working directory). If no browser is set, then ADRCI displays reports to the terminal or spool file.

This command does not require an ADR home to be set before you can use it.

Example

```
set browser mozilla
```

> **See Also:**
>
> - SHOW REPORT for more information about showing reports
> - SPOOL for more information about spooling

20.9.12 SET CONTROL

The ADRCI `SET CONTROL` command sets purging policies for Automatic Diagnostic Repository (ADR) contents.

Purpose

Sets time limit and size limit controls that manage when ADR repository files are purged.

Syntax and Description

```
set control (purge_policy = value purge_policy = value, ...)
```

In the preceding syntax, the variable *purge_policy* can be `SHORTP_POLICY`, `LONGP_POLICY`, or `SIZEP_POLICY`.

For SHORTP_POLICY and LONGP_POLICY, *value* is the number of hours after which the ADR contents become eligible for purging. The controls SHORTP_POLICY and LONGP_POLICY are not mutually exclusive. Each policy controls different types of content.

For SIZEP_POLICY, *value* is the size limit that you want to set for the ADR home. If you do not set a value, then the ADR home is purged every 24 hours. If you set a value for SIZEP_POLICY, then a MMON task is set that checks the current status of that limit every four hours. When the ADR home size reaches that limit, the ADR home is purged.

This command works with a single ADR home only.

Use SET CONTROL to set the following purge attributes:

Attribute Name	Description
SHORTP_POLICY	Number of hours after which to purge ADR contents that have a short life. Default: 720 (30 days). A setting of 0 (zero) means that all contents that have a short life can be purged. The maximum setting is 35791394. If a value greater than 35791394 is specified, then this attribute is set to 0 (zero). The ADR contents that have a short life include the following: • Trace files, including those files stored in the cdmp_*timestamp* subdirectories • Core dump files • Packaging information
LONGP_POLICY	Number of hours after which to purge ADR contents that have a long life. Default is 8760 (365 days). A setting of 0 (zero) means that all contents that have a long life can be purged. The maximum setting is 35791394. If a value greater than 35791394 is specified, then this attribute is set to 0 (zero). The ADR contents that have a long life include the following: • Incident information • Incident dumps • Alert logs
SIZEP_POLICY	(Optional) Defines the size limit for an Automatic Diagnostic Repository (ADR) home. In Oracle Database 12c Release 2 (12.2) and later releases, you can use SIZEP_POLICY to set a size limit for the AWR. When you set SIZEP_POLICY, the MMON background process collects statistics for the AWR home. By default, the ADR home is purged every 24 hours. If this purge time frame is inadequate, then you can set the SIZEP_POLICY to define a size limit for an ADR home to purge the ADR home when it approaches the purge size threshold. When you set a size limit using SIZEP_POLICY, MMON checks the current status of that limit every four hours. If the size limit is reached, then ADR purges the ADR repository.
PURGE_THRESHOLD	The PURGE_THRESHOLD value is a value at which the SIZEP_POLICY is triggered. If you set SIZEP_POLICY, then by default, the value of PURGE_THRESHOLD is 95 percent of the value of the SIZEP_POLICY. In a multitenant environment, the ADR home is shared, so the PURGE_THRESHOLD size policy is applied to the diagnostics storage location (diag). You can tune PURGE_THRESHOLD independently for each ADR home by setting the value for the PURGE_THRESHOLD column in the ADR_CONTROL_AUX relation . When you tune the PURGE_THRESHOLD, this can assist you with keeping the amount of ADR data below the SIZEP_POLICY limit, even if your ADR home is purged infrequently.

Example

Suppose the ADR purge policy is set to the default values of 720 for short life files (30 days), 8760 for long life files (365 days), and that you have no size-based purge policy set for the ADR repository. In the following example, the ADR short life files purge policy is changed to 360 (15 days), the short life files size limit before a purge is set to 18 gigabytes (G), and the size purge threshold is set to 12G

```
set control (SHORTP_POLICY = 360 SIZEP_POLICY = 18G PURGE_THRESHOLD =12G)
```

20.9.13 SET ECHO

The ADRCI `SET ECHO` command turns command output on or off. This command only affects output being displayed in a script or using the spool mode.

Syntax and Description

```
SET ECHO ON | OFF
```

This command does not require an ADR home to be set before you can use it.

Example

```
SET ECHO OFF
```

Related Topics

- SPOOL
  The ADRCI `SET EDITOR` command directs ADRCI output to a file.

20.9.14 SET EDITOR

The ADRCI `SET EDITOR` command sets the editor for displaying the alert log and the contents of trace files.

Syntax and Description

```
SET EDITOR editor_program
```

editor_program is the editor program name. If no editor is set, then ADRCI uses the editor specified by the operating system environment variable `EDITOR`. If `EDITOR` is not set, then ADRCI uses `vi` as the default editor.

This command does not require an ADR home to be set before you can use it.

Example

```
SET EDITOR xemacs
```

20.9.15 SET HOMEPATH

The ADRCI `SET HOMEPATH` command makes one or more ADR homes current. Many ADR commands work with the current ADR homes only.

Syntax and Description

```
SET HOMEPATH homepath_str1 homepath_str2 ...
```

When diagnosing data, to work with data from other instances or components, you must ensure that all the ADR homes for all of these instances or components are current. The *homepath_strn* strings are the paths of the ADR homes relative to the current ADR base. The `diag` directory name can be omitted from the path. If the specified path contains multiple ADR homes, then all of the homes are added to the home path.

If a desired new ADR home is not within the current ADR base, then you can use `SET BASE` to set a new ADR base, and then use `SET HOMEPATH`.

This command does not require an ADR home to be set before you can use it.

Example

```
SET HOMEPATH diag/rdbms/orcldw/orcldw1 diag/rdbms/orcldw/orcldw2
```

The following command sets the same home path as the previous example:

```
SET HOMEPATH rdbms/orcldw/orcldw1 rdbms/orcldw/orcldw2
```

Related Topics

- Definitions for Oracle Database ADRC
  To understand how to diagnose Oracle Database problems, learn the definitions of terms that Oracle uses for the ADRCI, and the Oracle Database fault diagnosability infrastructure.

20.9.16 SET TERMOUT

The ADRCI `SET TERMOUT` command turns output to the terminal on or off.

Syntax and Description

```
SET TERMOUT ON | OFF
```

This setting is independent of spooling. That is, the output can be directed to both terminal and a file at the same time.

This command does not require an ADR home to be set before you can use it.

> **See Also:**
>
> `SPOOL` for more information about spooling

Example

```
SET TERMOUT ON
```

Related Topics

- SPOOL
  The ADRCI `SET EDITOR` command directs ADRCI output to a file.

20.9.17 SHOW ALERT

The ADRCI `SHOW ALERT` command shows the contents of the alert log in the default editor.

Purpose

Shows the contents of the alert log in the default editor.

Syntax and Description

```
show alert [-p "predicate_string"] [-tail [num] [-f]] [-term]
  [-file alert_file_name]
```

Except when using the `-term` flag, this command works with only a single current ADR home. If more than one ADR home is set, ADRCI prompts you to choose the ADR home to use.

Table 20-9 Flags for the SHOW ALERT command

Flag	Description
`-p "predicate_string"`	Uses a SQL-like predicate string to show only the alert log entries for which the predicate is true. The predicate string must be enclosed in double quotation marks. The table that follows this table lists the fields that can be used in the predicate string.
`-tail [num][-f]`	Displays the most recent entries in the alert log. Use the *num* option to display the last *num* entries in the alert log. If *num* is omitted, then the last 10 entries are displayed. If the `-f` option is given, after displaying the requested messages, the command does not return. Instead, it remains active and continuously displays new alert log entries to the terminal as they arrive in the alert log. You can use this command to perform live monitoring of the alert log. To terminate the command, press CTRL+C.
`-term`	Directs results to the terminal. Outputs the entire alert logs from all current ADR homes, one after another. If this option is not given, then the results are displayed in the default editor.
`-file alert_file_name`	Enables you to specify an alert file outside the ADR. *alert_file_name* must be specified with a full path name. Note that this option cannot be used with the *-tail* option.

Table 20-10 Alert Fields for SHOW ALERT

Field	Type
ORIGINATING_TIMESTAMP	timestamp
NORMALIZED_TIMESTAMP	timestamp
ORGANIZATION_ID	text(65)
COMPONENT_ID	text(65)
HOST_ID	text(65)

Table 20-10 (Cont.) Alert Fields for SHOW ALERT

Field	Type
HOST_ADDRESS	text(17)
MESSAGE_TYPE	number
MESSAGE_LEVEL	number
MESSAGE_ID	text(65)
MESSAGE_GROUP	text(65)
CLIENT_ID	text(65)
MODULE_ID	text(65)
PROCESS_ID	text(33)
THREAD_ID	text(65)
USER_ID	text(65)
INSTANCE_ID	text(65)
DETAILED_LOCATION	text(161)
UPSTREAM_COMP_ID	text(101)
DOWNSTREAM_COMP_ID	text(101)
EXECUTION_CONTEXT_ID	text(101)
EXECUTION_CONTEXT_SEQUENCE	number
ERROR_INSTANCE_ID	number
ERROR_INSTANCE_SEQUENCE	number
MESSAGE_TEXT	text(2049)
MESSAGE_ARGUMENTS	text(129)
SUPPLEMENTAL_ATTRIBUTES	text(129)
SUPPLEMENTAL_DETAILS	text(4000)
PROBLEM_KEY	text(65)

Examples

This example shows all alert messages for the current ADR home in the default editor:

```
show alert
```

This example shows all alert messages for the current ADR home and directs the output to the terminal instead of the default editor:

```
show alert -term
```

This example shows all alert messages for the current ADR home with message text describing an incident:

```
show alert -p "message_text like '%incident%'"
```

This example shows the last twenty alert messages, and then keeps the alert log open, displaying new alert log entries as they arrive:

```
show alert -tail 20 -f
```

This example shows all alert messages for a single ADR home in the default editor when multiple ADR homes have been set:

```
show alert

Choose the alert log from the following homes to view:

1: diag/tnslsnr/dbhost1/listener
2: diag/asm/+asm/+ASM
3: diag/rdbms/orcl/orcl
4: diag/clients/user_oracle/host_9999999999_11
Q: to quit

Please select option:
3
```

Related Topics

- SET EDITOR
  The ADRCI `SET EDITOR` command sets the editor for displaying the alert log and the contents of trace files.

20.9.18 SHOW BASE

The ADRCI `SET EDITOR` command shows the current ADR base.

Syntax and Description

```
SHOW BASE [-product product_name]
```

(Optional) You can show the product's ADR base location for a specific product. The products currently supported are `CLIENT` and `ADRCI`.

This command does not require an ADR home to be set before you can use it.

Example

This example shows the current ADR base:

```
SHOW BASE
```

Output:

```
ADR base is "/u01/app/oracle"
```

This example shows the current ADR base for Oracle Database clients:

```
SHOW BASE -product client
```

20.9.19 SHOW CONTROL

The ADRCI `SHOW CONTROL` command displays information about the Automatic Diagnostic Repository (ADR), including the purging policy.

Syntax and Description

```
SHOW CONTROL
```

Displays various attributes of the ADR, including the following purging policy attributes:

Attribute Name	Description
SHORTP_POLICY	Number of hours after which to purge ADR contents that have a short life. Default is 720 (30 days). A setting of 0 (zero) means that all contents that have a short life can be purged. The maximum setting is 35791394. If a value greater than 35791394 is specified, then this attribute is set to 0 (zero). The ADR contents that have a short life include the following: • Trace files, including those files stored in the cdmp_*timestamp* subdirectories • Core dump files • Packaging information
LONGP_POLICY	Number of hours after which to purge ADR contents that have a long life. Default is 8760 (365 days). A setting of 0 (zero) means that all contents that have a long life can be purged. The maximum setting is 35791394. If a value greater than 35791394 is specified, then this attribute is set to 0 (zero). The ADR contents that have a long life include the following: • Incident information • Incident dumps • Alert logs
SIZEP_POLICY	(Optional) Defines the size limit for an ADR home. For example: In Oracle Database 12c Release 2 (12.2) and later releases, ADR supports size-based purging. The MMON background process collects statistics for the Automatic Workload Repository (AWR). By default, the ADR home is purged every 24 hours. If this purge time frame is inadequate, then use sizep_policy to define a size limit for an ADR home. By setting a sizeb_policy value to limit the size of an ADR home, MMON checks the current status of that limit every four hours. If the size limit is reached, then ADR purges the ADR repository.

> **Note:**
>
> The SHORTP_POLICY and LONGP_POLICY attributes are not mutually exclusive. Each policy controls different types of content.

20.9.20 SHOW HM_RUN

The ADRCI SHOW HM_RUN command shows all information for Health Monitor runs.

Purpose

Shows all information for Health Monitor runs.

Syntax and Description

```
show hm_run [-p "predicate_string]
```

predicate_string is a SQL-like predicate that specifies the field names that you want to select. The following table displays the list of field names you can use:

Table 20-11 Fields for Health Monitor Runs

Field	Type
RUN_ID	number
RUN_NAME	text(31)
CHECK_NAME	text(31)
NAME_ID	number
MODE	number
START_TIME	timestamp
RESUME_TIME	timestamp
END_TIME	timestamp
MODIFIED_TIME	timestamp
TIMEOUT	number
FLAGS	number
STATUS	number
SRC_INCIDENT_ID	number
NUM_INCIDENTS	number
ERR_NUMBER	number
REPORT_FILE	bfile

Examples

This example displays data for all Health Monitor runs:

```
show hm_run
```

This example displays data for the Health Monitor run with ID 123:

```
show hm_run -p "run_id=123"
```

Related Topics

- About Health Monitor

20.9.21 SHOW HOMEPATH

The ADRCI `SHOW HOMEPATH` command is identical to the `SHOW HOMES` command.

Syntax and Description

```
SHOW HOMEPATH | SHOW HOMES | SHOW HOME
```

This command does not require an ADR home to be set before you can use it.

Example

```
SHOW HOMEPATH
```

Output:

```
ADR Homes:
diag/tnslsnr/dbhost1/listener
diag/asm/+asm/+ASM
diag/rdbms/orcl/orcl
diag/clients/user_oracle/host_9999999999_11
```

Related Topics

- SET HOMEPATH
  The ADRCI `SET HOMEPATH` command makes one or more ADR homes current. Many ADR commands work with the current ADR homes only.

20.9.22 SHOW HOMES

The ADRCI `SHOW HOMES` command shows the ADR homes in the current ADRCI session.

Syntax and Description

```
SHOW HOMES | SHOW HOME | SHOW HOMEPATH
```

This command does not require an ADR home to be set before you can use it.

Example

```
SHOW HOMES
```

Output:

```
ADR Homes:
diag/tnslsnr/dbhost1/listener
diag/asm/+asm/+ASM
diag/rdbms/orcl/orcl
diag/clients/user_oracle/host_9999999999_11
```

20.9.23 SHOW INCDIR

The ADRCI `SHOW INCDIR` command shows trace files for the specified incident.

Syntax and Description

```
show incdir [id | id_low id_high]
```

You can provide a single incident ID (*id*), or a range of incidents (*id_low* to *id_high*). If no incident ID is given, then trace files for all incidents are listed.

Examples

This example shows all trace files for all incidents:

```
show incdir
```

Output:

```
ADR Home = /u01/app/oracle/log/diag/rdbms/emdb/emdb:
*********************************************************************
diag/rdbms/emdb/emdb/incident/incdir_3801/emdb_ora_23604_i3801.trc
diag/rdbms/emdb/emdb/incident/incdir_3801/emdb_m000_23649_i3801_a.trc
diag/rdbms/emdb/emdb/incident/incdir_3802/emdb_ora_23604_i3802.trc
diag/rdbms/emdb/emdb/incident/incdir_3803/emdb_ora_23604_i3803.trc
diag/rdbms/emdb/emdb/incident/incdir_3804/emdb_ora_23604_i3804.trc
diag/rdbms/emdb/emdb/incident/incdir_3805/emdb_ora_23716_i3805.trc
```

```
diag/rdbms/emdb/emdb/incident/incdir_3805/emdb_m000_23767_i3805_a.trc
diag/rdbms/emdb/emdb/incident/incdir_3806/emdb_ora_23716_i3806.trc
diag/rdbms/emdb/emdb/incident/incdir_3633/emdb_pmon_28970_i3633.trc
diag/rdbms/emdb/emdb/incident/incdir_3633/emdb_m000_23778_i3633_a.trc
diag/rdbms/emdb/emdb/incident/incdir_3713/emdb_smon_28994_i3713.trc
diag/rdbms/emdb/emdb/incident/incdir_3713/emdb_m000_23797_i3713_a.trc
diag/rdbms/emdb/emdb/incident/incdir_3807/emdb_ora_23783_i3807.trc
diag/rdbms/emdb/emdb/incident/incdir_3807/emdb_m000_23803_i3807_a.trc
diag/rdbms/emdb/emdb/incident/incdir_3808/emdb_ora_23783_i3808.trc
```

This example shows all trace files for incident 3713:

```
show incdir 3713
```

Output:

```
ADR Home = /u01/app/oracle/log/diag/rdbms/emdb/emdb:
*********************************************************************
diag/rdbms/emdb/emdb/incident/incdir_3713/emdb_smon_28994_i3713.trc
diag/rdbms/emdb/emdb/incident/incdir_3713/emdb_m000_23797_i3713_a.trc
```

This example shows all tracefiles for incidents between 3801 and 3804:

```
show incdir 3801 3804
```

Output:

```
ADR Home = /u01/app/oracle/log/diag/rdbms/emdb/emdb:
*********************************************************************
diag/rdbms/emdb/emdb/incident/incdir_3801/emdb_ora_23604_i3801.trc
diag/rdbms/emdb/emdb/incident/incdir_3801/emdb_m000_23649_i3801_a.trc
diag/rdbms/emdb/emdb/incident/incdir_3802/emdb_ora_23604_i3802.trc
diag/rdbms/emdb/emdb/incident/incdir_3803/emdb_ora_23604_i3803.trc
diag/rdbms/emdb/emdb/incident/incdir_3804/emdb_ora_23604_i3804.trc
```

20.9.24 SHOW INCIDENT

The ADRCI `SHOW INCIDENT` command lists all of the incidents associated with the current ADR home. Includes both open and closed incidents.

Syntax and Description

```
show incident [-p "predicate_string"] [-mode {BASIC|BRIEF|DETAIL}] [-orderby field1,
field2, ... [ASC|DSC]]
```

Table 20-12 Flags for SHOW INCIDENT command

Flag	Description
`-p "predicate_string"`	Use a predicate string to show only the incidents for which the predicate is true. The predicate string must be enclosed in double quotation marks. Refer to the table "Incident Fields for SHOW INCIDENT" for a list of the fields that can be used in the predicate string.

Table 20-12 (Cont.) Flags for SHOW INCIDENT command

Flag	Description
`-mode {BASIC|BRIEF|DETAIL}`	Choose an output mode for incidents. `BASIC` is the default. • `BASIC` displays only basic incident information (the `INCIDENT_ID`, `PROBLEM_ID`, and `CREATE_TIME` fields). It does not display flood-controlled incidents. • `BRIEF` displays all information related to the incidents, as described in the table "Incident Fields for SHOW INCIDENT." It includes flood-controlled incidents. • `DETAIL` displays all information for the incidents (as with `BRIEF` mode) as well as information about incident dumps. It includes flood-controlled incidents.
`-orderby` *`field1, field2, ...`* `[ASC|DSC]`	Show results sorted by field in the given order, as well as in ascending (`ASC`) and descending order (`DSC`). By default, results are shown in ascending order.

Table 20-13 Incident Fields for SHOW INCIDENT

Field	Type	Description
`INCIDENT_ID`	`number`	ID of the incident
`PROBLEM_ID`	`number`	ID of the problem to which the incident belongs
`CREATE_TIME`	`timestamp`	Time when the incident was created
`CLOSE_TIME`	`timestamp`	Time when the incident was closed
`STATUS`	`number`	Status of this incident
`FLAGS`	`number`	Flags for internal use
`FLOOD_CONTROLLED`	`number` (decoded to a text status by ADRCI)	Encodes the flood control status for the incident
`ERROR_FACILITY`	`text(10)`	Error facility for the error that caused the incident
`ERROR_NUMBER`	`number`	Error number for the error that caused the incident
`ERROR_ARG1`	`text(64)`	First argument for the error that caused the incident Error arguments provide additional information about the error, such as the code location that issued the error.
`ERROR_ARG2`	`text(64)`	Second argument for the error that caused the incident
`ERROR_ARG3`	`text(64)`	Third argument for the error that caused the incident
`ERROR_ARG4`	`text(64)`	Fourth argument for the error that caused the incident
`ERROR_ARG5`	`text(64)`	Fifth argument for the error that caused the incident

Table 20-13 (Cont.) Incident Fields for SHOW INCIDENT

Field	Type	Description
ERROR_ARG6	text(64)	Sixth argument for the error that caused the incident
ERROR_ARG7	text(64)	Seventh argument for the error that caused the incident
ERROR_ARG8	text(64)	Eighth argument for the error that caused the incident
SIGNALLING_COMPONENT	text(64)	Component that signaled the error that caused the incident
SIGNALLING_SUBCOMPONENT	text(64)	Subcomponent that signaled the error that caused the incident
SUSPECT_COMPONENT	text(64)	Component that has been automatically identified as possibly causing the incident
SUSPECT_SUBCOMPONENT	text(64)	Subcomponent that has been automatically identified as possibly causing the incident
ECID	text(64)	Execution Context ID
IMPACT	number	Encodes the impact of the incident
ERROR_ARG9	text(64)	Ninth argument for the error that caused the incident
ERROR_ARG10	text(64)	Tenth argument for the error that caused the incident
ERROR_ARG11	text(64)	Eleventh argument for the error that caused the incident
ERROR_ARG12	text(64)	Twelfth argument for the error that caused the incident

Examples

This example shows all incidents for this ADR home:

```
show incident
```

Output:

```
ADR Home = /u01/app/oracle/log/diag/rdbms/emdb/emdb:
*************************************************************************
INCIDENT_ID          PROBLEM_KEY                                       CREATE_TIME
-------------------- ------------------------------------------------- ----------------------------
3808                 ORA 603                                           2010-06-18 21:35:49.322161 -07:00
3807                 ORA 600 [4137]                                    2010-06-18 21:35:47.862114 -07:00
3806                 ORA 603                                           2010-06-18 21:35:26.666485 -07:00
3805                 ORA 600 [4136]                                    2010-06-18 21:35:25.012579 -07:00
3804                 ORA 1578                                          2010-06-18 21:35:08.483156 -07:00
3713                 ORA 600 [4136]                                    2010-06-18 21:35:44.754442 -07:00
3633                 ORA 600 [4136]                                    2010-06-18 21:35:35.776151 -07:00
7 rows fetched
```

This example shows the detail view for incident 3805:

```
adrci> show incident -mode DETAIL -p "incident_id=3805"
```

Output:

```
ADR Home = /u01/app/oracle/log/diag/rdbms/emdb/emdb:
*********************************************************************

*************************************************************
INCIDENT INFO RECORD 1
*************************************************************
   INCIDENT_ID                   3805
   STATUS                        closed
   CREATE_TIME                   2010-06-18 21:35:25.012579 -07:00
   PROBLEM_ID                    2
   CLOSE_TIME                    2010-06-18 22:26:54.143537 -07:00
   FLOOD_CONTROLLED              none
   ERROR_FACILITY                ORA
   ERROR_NUMBER                  600
   ERROR_ARG1                    4136
   ERROR_ARG2                    2
   ERROR_ARG3                    18.0.628
   ERROR_ARG4                    <NULL>
   ERROR_ARG5                    <NULL>
   ERROR_ARG6                    <NULL>
   ERROR_ARG7                    <NULL>
   ERROR_ARG8                    <NULL>
   SIGNALLING_COMPONENT          <NULL>
   SIGNALLING_SUBCOMPONENT       <NULL>
   SUSPECT_COMPONENT             <NULL>
   SUSPECT_SUBCOMPONENT          <NULL>
   ECID                          <NULL>
   IMPACTS                       0
   PROBLEM_KEY                   ORA 600 [4136]
   FIRST_INCIDENT                3805
   FIRSTINC_TIME                 2010-06-18 21:35:25.012579 -07:00
   LAST_INCIDENT                 3713
   LASTINC_TIME                  2010-06-18 21:35:44.754442 -07:00
   IMPACT1                       0
   IMPACT2                       0
   IMPACT3                       0
   IMPACT4                       0
   KEY_NAME                      Client ProcId
   KEY_VALUE                     oracle@dbhost1 (TNS V1-V3).23716_3083142848
   KEY_NAME                      SID
   KEY_VALUE                     127.52237
   KEY_NAME                      ProcId
   KEY_VALUE                     23.90
   KEY_NAME                      PQ
   KEY_VALUE                     (0, 1182227717)
   OWNER_ID                      1
   INCIDENT_FILE                 /.../emdb/emdb/incident/incdir_3805/emdb_ora_23716_i3805.trc
   OWNER_ID                      1
   INCIDENT_FILE                 /.../emdb/emdb/trace/emdb_ora_23716.trc
   OWNER_ID                      1
   INCIDENT_FILE                 /.../emdb/emdb/incident/incdir_3805/emdb_m000_23767_i3805_a.trc
1 rows fetched
```

Related Topics

- SHOW INCIDENT
  The ADRCI `SHOW INCIDENT` command lists all of the incidents associated with the current ADR home. Includes both open and closed incidents.

20.9.25 SHOW LOG

The ADRCI `SHOW LOG` command shows diagnostic log messages.

Syntax and Description

```
SHOW LOG [-l log_name] [-p "predicate_string"] [-term] [ [-tail [num] [-f]] ]
```

The following table describes the flags for `SHOW LOG`.

Table 20-14 Flags for SHOW LOG command

Flag	Description
`-l log_name`	Name of the log to show. If no log name is specified, then this command displays all messages from all diagnostic logs under the current ADR Home.
`-p "predicate_string"`	Use a SQL-like predicate string to show only the log entries for which the predicate is true. The predicate string must be enclosed in double quotation marks. The table "Log Fields for SHOW LOG" lists the fields that can be used in the predicate string.
`-term`	Direct results to the terminal. If this option is not specified, then this command opens the results in an editor. By default, it opens the results in the `emacs` editor, but you can use the `SET EDITOR` command to open the results in other editors.
`-tail [num] [-f]`	Displays the most recent entries in the log. Use the *num* option to display the last *num* entries in the log. If *num* is omitted, then the last 10 entries are displayed. If the `-f` option is given, then after displaying the requested messages, the command does not return. Instead, it remains active, and continuously displays new log entries to the terminal as they arrive in the log. You can use this command to perform live monitoring of the log. To terminate the command, press `CTRL+C`.

Table 20-15 Log Fields for SHOW LOG

Field	Type
ORIGINATING_TIMESTAMP	timestamp
NORMALIZED_TIMESTAMP	timestamp
ORGANIZATION_ID	text(65)
COMPONENT_ID	text(65)
HOST_ID	text(65)
HOST_ADDRESS	text(17)
MESSAGE_TYPE	number
MESSAGE_LEVEL	number
MESSAGE_ID	text(65)
MESSAGE_GROUP	text(65)
CLIENT_ID	text(65)

Table 20-15 (Cont.) Log Fields for SHOW LOG

Field	Type
MODULE_ID	text(65)
PROCESS_ID	text(33)
THREAD_ID	text(65)
USER_ID	text(65)
INSTANCE_ID	text(65)
DETAILED_LOCATION	text(161)
UPSTREAM_COMP_ID	text(101)
DOWNSTREAM_COMP_ID	text(101)
EXECUTION_CONTEXT_ID	text(101)
EXECUTION_CONTEXT_SEQUENCE	number
ERROR_INSTANCE_ID	number
ERROR_INSTANCE_SEQUENCE	number
MESSAGE_TEXT	text(2049)
MESSAGE_ARGUMENTS	text(129)
SUPPLEMENTAL_ATTRIBUTES	text(129)
SUPPLEMENTAL_DETAILS	text(4000)
PROBLEM_KEY	text(65)

20.9.26 SHOW PROBLEM

The ADRCI `SHOW PROBLEM` command shows problem information for the current ADR home.

Syntax and Description

```
show problem [-p "predicate_string"] [-last num | -all]
    [-orderby field1, field2, ... [ASC|DSC]]
```

The following table describes the flags for `SHOW PROBLEM`.

Table 20-16 Flags for SHOW PROBLEM command

Flag	Description
`-p "predicate_string"`	Use a SQL-like predicate string to show only the incidents for which the predicate is true. The predicate string must be enclosed in double quotation marks. The table "Problem Fields for SHOW PROBLEM" lists the fields that can be used in the predicate string.
`-last num | -all`	Shows the last *num* problems, or lists all the problems. By default, `SHOW PROBLEM` lists the most recent 50 problems.
`-orderby field1, field2, ... [ASC|DSC]`	Show results sorted by field in the given order (*field1*, *field2*, ...), as well as in ascending (`ASC`) and descending order (`DSC`). By default, results are shown in ascending order.

Table 20-17 Problem Fields for SHOW PROBLEM

Field	Type	Description
PROBLEM_ID	number	ID of the problem
PROBLEM_KEY	text(550)	Problem key for the problem
FIRST_INCIDENT	number	Incident ID of the first incident for the problem
FIRSTINC_TIME	timestamp	Creation time of the first incident for the problem
LAST_INCIDENT	number	Incident ID of the last incident for the problem
LASTINC_TIME	timestamp	Creation time of the last incident for the problem
IMPACT1	number	Encodes an impact of this problem
IMPACT2	number	Encodes an impact of this problem
IMPACT3	number	Encodes an impact of this problem
IMPACT4	number	Encodes an impact of this problem
SERVICE_REQUEST	text(64)	Service request for the problem (entered through Support Workbench)
BUG_NUMBER	text(64)	Bug number for the problem (entered through Support Workbench)

Example

This example lists all the problems in the current ADR home:

```
show problem -all
```

This example shows the problem with ID 4:

```
show problem -p "problem_id=4"
```

20.9.27 SHOW REPORT

The ADRCI `SET EDITOR` command shows a report for the specified report type and run name.

Purpose

Currently, only the `hm_run` (Health Monitor) report type is supported, and only in XML formatting. To view HTML-formatted Health Monitor reports, use Oracle Enterprise Manager or the `DBMS_HM` PL/SQL package.

See *Oracle Database Administrator's Guide* for more information.

Syntax and Description

```
SHOW REPORT report_type run_name
```

report_type must be `hm_run`. *run_name* is the Health Monitor run name from which you created the report. You must first create the report using the `CREATE REPORT` command.

This command does not require an ADR home to be set before you can use it.

Example

```
SHOW REPORT hm_run hm_run_1421
```

Related Topics

- CREATE REPORT
  The ADRCI `CREATE REPORT` command creates a report for the specified report type and run ID, and stores the report in the ADR.
- SHOW HM_RUN
  The ADRCI `SHOW HM_RUN` command shows all information for Health Monitor runs.

20.9.28 SHOW TRACEFILE

The ADRCI `SHOW TRACEFILE` command lists trace files.

Syntax and Description

```
show tracefile [file1 file2 ...] [-rt | -t]
  [-i inc1 inc2 ...] [-path path1 path2 ...]
```

This command searches for one or more files under the trace directory, and all incident directories of the current ADR homes, unless the `-i` or `-path` flags are given.

This command does not require an ADR home to be set unless using the `-i` option.

The following table describes the arguments of `SHOW TRACEFILE`.

Table 20-18 Arguments for SHOW TRACEFILE Command

Argument	Description
file1 file2 ...	Filter results by file name. The `%` symbol is a wildcard character.

Table 20-19 Flags for SHOW TRACEFILE Command

Flag	Description
`-rt | -t`	Order the trace file names by timestamp. `-t` sorts the file names in ascending order by timestamp, and `-rt` sorts them in reverse order. Note that file names are only ordered relative to their directory. Listing multiple directories of trace files applies a separate ordering to each directory. Timestamps are listed next to each file name when using this option.
`-i` *inc1 inc2 ...*	Select only the trace files produced for the given incident IDs.
`-path` *path1 path2 ...*	Query only the trace files under the given path names.

Examples

This example shows all the trace files under the current ADR home:

```
show tracefile
```

This example shows all the `mmon` trace files, sorted by timestamp in reverse order:

```
show tracefile %mmon% -rt
```

This example shows all trace files for incidents 1 and 4, under the path `/home/steve/temp`:

```
show tracefile -i 1 4 -path /home/steve/temp
```

20.9.29 SPOOL

The ADRCI `SET EDITOR` command directs ADRCI output to a file.

Syntax and Description

```
SPOOL filename [[APPEND] | [OFF]]
```

filename is the file name where you want the output to be directed. If a full path name is not given, then the file is created in the current ADRCI working directory. If no file extension is given, then the default extension `.ado` is used. `APPEND` causes the output to be appended to the end of the file. Otherwise, the file is overwritten. Use `OFF` to turn off spooling.

This command does not require an ADR home to be set before you can use it.

Examples

```
SPOOL myfile

SPOOL myfile.ado APPEND

SPOOL OFF

SPOOL
```

20.10 Troubleshooting ADRCI

To assist troubleshooting, review some of the common ADRCI error messages, and their possible causes and remedies.

No ADR base is set
Cause: You may have started ADRCI with a null or invalid value for the `ORACLE_HOME` environment variable.

Action: Exit ADRCI, set the `ORACLE_HOME` environment variable, and restart ADRCI. For more information, see "ADR BASE" in Definitions for Oracle Database ADRC

DIA-48323: Specified pathname *string* must be inside current ADR home
Cause: A file outside of the ADR home is not allowed as an incident file for this command.

Action: Retry using an incident file inside the ADR home.

DIA-48400: ADRCI initialization failed
Cause: The ADR Base directory does not exist.

Action: Check the value of the `DIAGNOSTIC_DEST` initialization parameter, and ensure that it points to an ADR base directory that contains at least one ADR home. If `DIAGNOSTIC_DEST` is missing or null, check for a valid ADR base directory hierarchy in *ORACLE_HOME*/log.

DIA-48431: Must specify at least one ADR home path
Cause: The command requires at least one ADR home to be current.

Action: Use the `SET HOMEPATH` command to make one or more ADR homes current.

DIA-48432: The ADR home path *string* is not valid
Cause: The supplied ADR home is not valid, possibly because the path does not exist.

Action: Check if the supplied ADR home path exists.

DIA-48447: The input path [*path*] does not contain any ADR homes
Cause: When using `SET HOMEPATH` to set an ADR home, you must supply a path relative to the current ADR base.

Action: If the new desired ADR home is not within the current ADR base, first set ADR base with `SET BASE`, and then use `SHOW HOMES` to check the ADR homes under the new ADR base. Next, use `SET HOMEPATH` to set a new ADR home if necessary.

DIA-48448: This command does not support multiple ADR homes
Cause: There are multiple current ADR homes in the current ADRCI session.

Action: Use the `SET HOMEPATH` command to make a single ADR home current.

21

DBVERIFY: Offline Database Verification Utility

DBVERIFY is an external command-line utility that performs a physical data structure integrity check.

DBVERIFY 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 data file) 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 data files, it does 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 data file 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 Data File
  In this mode, DBVERIFY scans one or more disk blocks of a single data file and performs page checks.
- Using DBVERIFY to Validate a Segment
  In this mode, DBVERIFY enables you to specify a table segment or index segment for verification.

21.1 Using DBVERIFY to Validate Disk Blocks of a Single Data File

In this mode, DBVERIFY scans one or more disk blocks of a single data file and performs page checks.

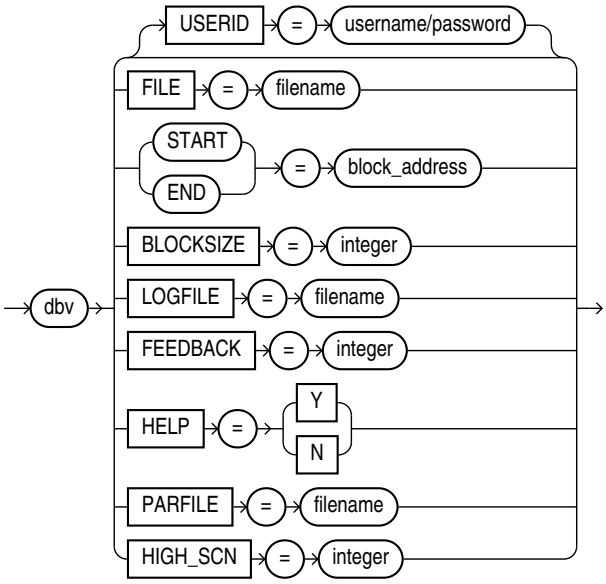
If the file you are verifying is an Oracle Automatic Storage Management (Oracle ASM) file, then you must supply a `USERID`. This is because DBVERIFY needs to connect to an Oracle instance to access Oracle ASM files.

- DBVERIFY Syntax When Validating Blocks of a Single File
  See the syntax for using `DBVERIFY` to validate blocks of a single file.
- DBVERIFY Parameters When Validating Blocks of a Single File
  See the `DBVERIFY` parameters that you can use to validate blocks of a single file.
- Example DBVERIFY Output For a Single Data File
  See an example of verification for a single data file, and how you can interpret it.

21.1.1 DBVERIFY Syntax When Validating Blocks of a Single File

See the syntax for using `DBVERIFY` to validate blocks of a single file.

The syntax for `DBVERIFY` when you want to validate disk blocks of a single data file is as follows:



21.1.2 DBVERIFY Parameters When Validating Blocks of a Single File

See the `DBVERIFY` parameters that you can use to validate blocks of a single file.

Parameter	Description
`USERID`	Specifies your username and password. This parameter is only necessary when the files being verified are Oracle ASM files. If you do specify this parameter, then you must enter both a username and a password; otherwise, a `DBV-00112: USERID incorrectly specified` error is returned.
`FILE`	The name of the database file that you want to verify.
`START`	The starting block address that you want to verify. Specify block addresses in Oracle blocks (as opposed to operating system blocks). If you do not specify `START`, then `DBVERIFY` defaults to the first block in the file.
`END`	The ending block address that you want to verify. If you do not specify `END`, then `DBVERIFY` defaults to the last block in the file.
`BLOCKSIZE`	`BLOCKSIZE` is required only if the file that you want 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`, then you will receive the error DBV-00103.

Parameter	Description
HIGH_SCN	When a value is specified for HIGH_SCN, DBVERIFY writes diagnostic messages for each block whose block-level system change number (SCN) exceeds the value specified. This parameter is optional. There is no default.
LOGFILE	Specifies the log file name and path 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, then there is no progress display.
HELP	Provides online help. For help on command line parameters in a given version of DBVERIFY, type dbv help=y at the command line.
PARFILE	Specifies the name of the parameter file to use. You can store various values for DBVERIFY parameters in flat files. Doing this enables you to customize parameter files to handle different types of data files, and to perform specific types of integrity checks on data files.

Related Topics

- DBVERIFY - Database file Verification Utility (Doc ID 35512.1)

21.1.3 Example DBVERIFY Output For a Single Data File

See an example of verification for a single data file, and how you can interpret it.

The following is an example 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.

```
% dbv FILE=t_db1.dbf FEEDBACK=100
```

Output example

The output of this command is as follows:

```
.
.
.
DBVERIFY - Verification starting : FILE = t_db1.f
DBVERIFY - Verification complete

Total Pages Examined : 120424
Total Pages Processed (Data) : 79507
Total Pages Failing (Data) : 0
Total Pages Processed (Index): 15236
Total Pages Failing (Index): 0
Total Pages Processed (Other): 5626
Total Pages Processed (Seg) : 1
Total Pages Failing (Seg) : 0
```

```
Total Pages Empty : 20055
Total Pages Marked Corrupt : 0
Total Pages Influx : 0
Total Pages Encrypted : 0
Highest block SCN : 25565681 (0.25565681)
```

Notes

- Pages = Blocks
- Total Pages Examined = number of blocks in the file.
- Total Pages Processed (Data) = number of blocks that were verified (formatted blocks).
- Total Pages Processed (Other) = metadata blocks. These blocks are not being verified, so there is no output for "Total Pages Failing (Other)."
- Total Pages Processed (Seg) = number of segment header 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, then `DBVERIFY` reads blocks multiple times to obtain a consistent image. But because the database is open, there can be blocks that are being read and written to at the same time (`INFLUX`). In that event, `DBVERIFY` cannot obtain a consistent image of pages that are in flux.
- Total Pages Encrypted = all blocks (Data, Index, Other, Seg), not only Data or Index. When "Total Pages Encrypted" is different than zero, DBVERIFY outputs the message "DBVerify cannot perform logical check against encrypted blocks, RMAN should be used."

21.2 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 ensure 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.

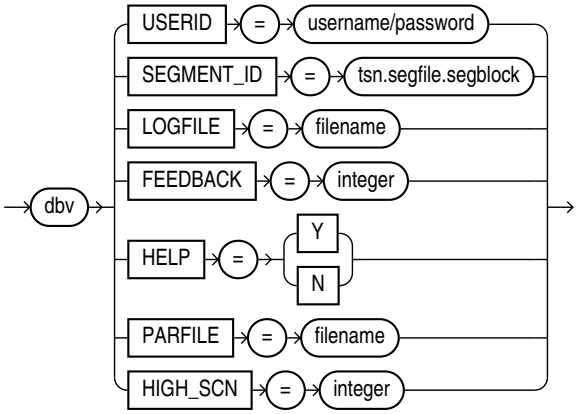
During this mode, the segment is locked. If the specified segment is an index, then the parent table is locked. Note that some indexes, such as IOTs, do not have parent tables.

- DBVERIFY Syntax When Validating a Segment
  See the syntax for using `DBVERIFY` to validate a segment.
- DBVERIFY Parameters When Validating a Single Segment
  See the `DBVERIFY` parameters that you can use to validate a single segment.
- Example DBVERIFY Output For a Validated Segment
  See an example of a verification for a validated segment.

21.2.1 DBVERIFY Syntax When Validating a Segment

See the syntax for using `DBVERIFY` to validate a segment.

The syntax for `DBVERIFY` when you want to validate a segment is as follows:



21.2.2 DBVERIFY Parameters When Validating a Single Segment

See the `DBVERIFY` parameters that you can use to validate a single segment.

Parameter	Description
`USERID`	Specifies your username and password. If you do not enter both a username and a password, then the error `DBV-00112: USERID incorrectly specified` is returned. If you are connecting to a container database (CDB), then you would enter *username@cdbname/password*.
`SEGMENT_ID`	Specifies the segment that you want to verify. A segment identifier is composed of the tablespace ID number (`tsn`), segment header file number (`segfile`), and segment header block number (`segblock`). You can obtain this information from `SYS_USER_SEGS`. The relevant columns are `TABLESPACE_ID`, `HEADER_FILE`, and `HEADER_BLOCK`. To query `SYS_USER_SEGS`, you must have `SYSDBA` privileges. For example, if the tablespace number (`TS#)` is 2, the segment header file number (`HEADER_FILE)` is 5, and the segment header block number (`HEADER_BLOCK)` is 37767, then check that segment using `SEGMENT_ID=2.5.37767`
`HIGH_SCN`	When a value is specified for `HIGH_SCN`, `DBVERIFY` writes diagnostic messages for each block whose block-level SCN exceeds the value specified. This parameter is optional. There is no default.
`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, then there is no progress display.

Parameter	Description
HELP	Provides online help.
PARFILE	Specifies the name of the parameter file that you want to use. You can store various values for DBVERIFY parameters in flat files. Doing this enables you to customize parameter files to handle different types of data files, and to perform specific types of integrity checks on data files.

21.2.3 Example DBVERIFY Output For a Validated Segment

See an example of a verification for a validated segment.

The following is an example of using the DBVERIFY command with a tablespace segment, and the output produced by a DBVERIFY operation.

```
% dbv userid=system/ SEGMENT_ID=2.5.37767
```

The output of this command is as follows:

```
DBVERIFY - Verification starting : SEGMENT_ID = 2.5.37767

DBVERIFY - Verification complete

Total Pages Examined : 640
Total Pages Processed (Data) : 0
Total Pages Failing (Data) : 0
Total Pages Processed (Index): 0
Total Pages Failing (Index): 0
Total Pages Processed (Other): 591
Total Pages Processed (Seg) : 8
Total Pages Failing (Seg) : 0
Total Pages Empty : 13
Total Pages Marked Corrupt : 0
Total Pages Influx : 0
Total Pages Encrypted : 28
Highest block SCN : 7877587 (0.7877587)
```

Related Topics

- DBVERIFY enhancement - How to scan an object/segment (Doc ID 139962.1)

22
DBNEWID Utility

DBNEWID is a database utility that can change the internal database identifier (DBID) and the database name (DBNAME) for an operational database.

See the following topics:

- What Is the DBNEWID Utility?
  The `DBNEWID` utility enables you to change only the `DBID`, `DBNAME`, or both the `DBID` and `DBNAME` of a database.
- Ramifications of Changing the DBID and DBNAME
  Before you change the `DBID` and `DBNAME` of a database with the DBNEWID utility, review these guidelines.
- 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.
- Changing Both CDB and PDB DBIDs Using DBNEWID
  The DBNEWID parameter `PDB` enables you to change the `DBID` on pluggable databases (PDBs).
- Changing the DBID and DBNAME of a Database
  To change either `DBID` or `DBNAME`, or both the `DBID` and `DBNAME` of your database, select the DBNEWID procedure that you need.
- DBNEWID Syntax
  Describes DBNEWID syntax.

22.1 What Is the DBNEWID Utility?

The `DBNEWID` utility enables you to change only the `DBID`, `DBNAME`, or both the `DBID` and `DBNAME` of a database.

Before the introduction of the DBNEWID utility, you could manually create a copy of a database and give it a new database name (`DBNAME`) by recreating 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 enabling 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

22.2 Ramifications of Changing the DBID and DBNAME

Before you change the `DBID` and `DBNAME` of a database with the DBNEWID utility, review these guidelines.

When you change the DBID, you should make a backup of the whole database immediately.

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. Changing the DBID is similar to creating a database, except that the data is already in the data files. After you change the DBID, backups and archive logs that were created before the DBID 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 recreates the online redo logs, and resets the redo log sequence to 1. Consequently,

When you change DBNAME and do not change DBID, you must change the DBNAME initialization parameter, and follow additional guidelines.

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 recreate 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.

> **Caution:**
>
> If you are using a capture process to capture changes to the database, then do not change the DBID or DBNAME of a database .

For Oracle RAC environments only, you must first detach the database from the cluster before you can run the DBNEWID utility. Use SQL*Plus to enter the following commands to set the initialization parameter value for CLUSTER_DATABASE to FALSE

1. `ALTER SYSTEM SET CLUSTER_DATABASE=FALSE SCOPE=SPFILE;`

   Restart the database after changing the CLUSTER_DATABASE parameter.
2. Shut down the database.

   ```
   SHUTDOWN IMMEDIATE
   ```

You can then run STARTUP MOUNT EXCLUSIVE, and change the global database name. If you attempt to use the DBNEWID utility while CLUSTER_DATABASE=TRUE, then the command fails with NID-00120: Database should be mounted exclusively.

Related Topics

- How to Change the DBID, DBNAME Using NID Utility (Doc ID 863800.1)

22.3 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.

You can only change a global database name 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.

For example, suppose you use DBNEWID to change a database name to `sales`. To ensure that you also change the global database name to `sales` in the domain `example.com`, you should use `ALTER DATABASE RENAME` as follows:

```
ALTER DATABASE RENAME GLOBAL_NAME TO sales.example.com
```

Related Topics

- Changing a Global Database Name: Scenario in *Oracle Database Administrator's Guide*

> **See Also:**
>
> *Oracle Database Administrator's Guide* for more information about global database names, and My Oracle Support "How to Change the DBID, DBNAME Using NID Utility (Doc ID 863800.1)"

22.4 Changing Both CDB and PDB DBIDs Using DBNEWID

The DBNEWID parameter `PDB` enables you to change the `DBID` on pluggable databases (PDBs).

By default, when you run the DBNEWID utility on a container database (CDB), the utility only changes the `DBID` of the CDB. The `DBID` values for each of the pluggable databases (PDBs) plugged into the CDB remain the same. In some cases, you can find that this default behavior causes problems with duplicate DBID values for PDBs. For example, you can encounter this issue when a CDB is cloned.

With Oracle Database 12c Release 2 (12.2) and later releases, you can use the DBNEWID utility PDB parameter in multitenant databases to change the DBID values for PDBs. You cannot specify a particular PDB; either all of them or none of them are assigned new `DBID` values. The `PDB` parameter has the following format:

```
PDB=[ALL | NONE]
```

- If you specify `ALL`, then in addition to the `DBID` for the CDB changing, the `DBID` values for all PDBs plugged into the CDB are also changed.
- Specifying `NONE` (the default) leaves the PDB `DBID`s the same, even if the CDB `DBID` is changed.

Oracle recommends that you use `PDB=ALL`. For backward compatibility, the default is `PDB=NONE`.

22.5 Changing the DBID and DBNAME of a Database

To change either DBID or DBNAME, or both the DBID and DBNAME of your database, select the DBNEWID procedure that you need.

- Changing the DBID and Database Name
  To change the DBID of a database, or both the DBID and DBNAME of a database with DBNEWID, use this procedure.
- Changing Only the Database ID
  To change the database ID (DBID) without changing the database name, use this DBNEWID procedure.
- Changing Only the Database Name
  To change the database name (DBNAME) without changing the DBID, use this DBNEWID procedure.
- Troubleshooting DBNEWID
  If you encounter an error when using DBNEWID to change a database ID, then refer to these troubleshooting hints.

22.5.1 Changing the DBID and Database Name

To change the DBID of a database, or both the DBID and DBNAME of a database with DBNEWID, use this procedure.

The following steps describe how to change the DBID of a database. You also have the option to 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 before mounting. For example:

   ```
   SHUTDOWN IMMEDIATE
   STARTUP MOUNT
   ```

3. Start the DBNEWID utility on the command line, specifying a valid user (TARGET) that has the SYSDBA privilege (you will be prompted for a password):

   ```
   % nid TARGET=SYS
   ```

   To change the database name in addition to the DBID, also specify the DBNAME parameter on the command line (you will be prompted for a password). The following example changes the database name to test_db:

   ```
   % nid TARGET=SYS DBNAME=test_db
   ```

   The DBNEWID utility performs validations in the headers of the data files and control files before attempting to modify 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 data file, including offline normal and read-only data files, 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)
   .
   ```

```
.
.
Control Files in database:
    /oracle/TEST_DB/data/cf1.dbf
    /oracle/TEST_DB/data/cf2.dbf

The following datafiles are offline clean:
    /oracle/TEST_DB/data/tbs_61.dbf (23)
    /oracle/TEST_DB/data/tbs_62.dbf (24)
    /oracle/TEST_DB/data/temp3.dbf (3)
These files must be writable by this utility.

The following datafiles are read-only:
    /oracle/TEST_DB/data/tbs_51.dbf (15)
    /oracle/TEST_DB/data/tbs_52.dbf (16)
    /oracle/TEST_DB/data/tbs_53.dbf (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.dbf - modified
    Control File /oracle/TEST_DB/data/cf2.dbf - modified
    Datafile /oracle/TEST_DB/data/tbs_01.dbf - dbid changed, wrote new name
    Datafile /oracle/TEST_DB/data/tbs_ax1.dbf - dbid changed, wrote new name
    Datafile /oracle/TEST_DB/data/tbs_02.dbf - dbid changed, wrote new name
    Datafile /oracle/TEST_DB/data/tbs_11.dbf - dbid changed, wrote new name
    Datafile /oracle/TEST_DB/data/tbs_12.dbf - dbid changed, wrote new name
    Datafile /oracle/TEST_DB/data/temp1.dbf - dbid changed, wrote new name
    Control File /oracle/TEST_DB/data/cf1.dbf - dbid changed, wrote new name
    Control File /oracle/TEST_DB/data/cf2.dbf - 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 example 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)
.
.
.
Control Files in database:
    /oracle/TEST_DB/data/cf1.dbf
    /oracle/TEST_DB/data/cf2.dbf

The following datafiles are offline clean:
    /oracle/TEST_DB/data/tbs_61.dbf (23)
    /oracle/TEST_DB/data/tbs_62.dbf (24)
    /oracle/TEST_DB/data/temp3.dbf (3)
These files must be writable by this utility.
```

```
    The following datafiles are read-only:
        /oracle/TEST_DB/data/tbs_51.dbf (15)
        /oracle/TEST_DB/data/tbs_52.dbf (16)
        /oracle/TEST_DB/data/tbs_53.dbf (22)
    These files must be writable by this utility.

    The following datafiles are offline immediate:
        /oracle/TEST_DB/data/tbs_71.dbf (25)
        /oracle/TEST_DB/data/tbs_72.dbf (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;
```

After you reset the logs, create 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.

22.5.2 Changing Only the Database ID

To change the database ID (`DBID`) without changing the database name, use this DBNEWID procedure.

Follow the steps in Changing the DBID and Database Name, 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)
.
.
.
Control Files in database:
    /oracle/TEST_DB/data/cf1.dbf
    /oracle/TEST_DB/data/cf2.dbf

The following datafiles are offline clean:
    /oracle/TEST_DB/data/tbs_61.dbf (23)
    /oracle/TEST_DB/data/tbs_62.dbf (24)
    /oracle/TEST_DB/data/temp3.dbf (3)
These files must be writable by this utility.

The following datafiles are read-only:
    /oracle/TEST_DB/data/tbs_51.dbf (15)
    /oracle/TEST_DB/data/tbs_52.dbf (16)
    /oracle/TEST_DB/data/tbs_53.dbf (22)
These files must be writable by this utility.

Changing database ID from 86997811 to 4004383693
    Control File /oracle/TEST_DB/data/cf1.dbf - modified
    Control File /oracle/TEST_DB/data/cf2.dbf - modified
    Datafile /oracle/TEST_DB/data/tbs_01.dbf - dbid changed
```

```
    Datafile /oracle/TEST_DB/data/tbs_ax1.dbf - dbid changed
    Datafile /oracle/TEST_DB/data/tbs_02.dbf - dbid changed
    Datafile /oracle/TEST_DB/data/tbs_11.dbf - dbid changed
    Datafile /oracle/TEST_DB/data/tbs_12.dbf - dbid changed
    Datafile /oracle/TEST_DB/data/temp1.dbf - dbid changed
    Control File /oracle/TEST_DB/data/cf1.dbf - dbid changed
    Control File /oracle/TEST_DB/data/cf2.dbf - 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.
```

22.5.3 Changing Only the Database Name

To change the database name (`DBNAME`) without changing the `DBID`, use this DBNEWID procedure.

Complete the following steps:

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 before mounting. For example:

   ```
   SHUTDOWN IMMEDIATE
   STARTUP MOUNT
   ```

3. Start the utility on the command line, specifying a valid user with the `SYSDBA` privilege (you will be prompted for a password). You must specify both the `DBNAME` and `SETNAME` parameters. This example changes the name to `test_db`:

   ```
   % nid TARGET=SYS DBNAME=test_db SETNAME=YES
   ```

   DBNEWID performs validations in the headers of the control files (not the data files) 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:

   ```
   .
   .
   .
   Control Files in database:
       /oracle/TEST_DB/data/cf1.dbf
       /oracle/TEST_DB/data/cf2.dbf

   The following datafiles are offline clean:
       /oracle/TEST_DB/data/tbs_61.dbf (23)
       /oracle/TEST_DB/data/tbs_62.dbf (24)
       /oracle/TEST_DB/data/temp3.dbf (3)
   These files must be writable by this utility.

   The following datafiles are read-only:
       /oracle/TEST_DB/data/tbs_51.dbf (15)
       /oracle/TEST_DB/data/tbs_52.dbf (16)
       /oracle/TEST_DB/data/tbs_53.dbf (22)
   These files must be writable by this utility.

   Changing database name from PROD to TEST_DB
       Control File /oracle/TEST_DB/data/cf1.dbf - modified
   ```

```
    Control File /oracle/TEST_DB/data/cf2.dbf - modified
    Datafile /oracle/TEST_DB/data/tbs_01.dbf - wrote new name
    Datafile /oracle/TEST_DB/data/tbs_ax1.dbf - wrote new name
    Datafile /oracle/TEST_DB/data/tbs_02.dbf - wrote new name
    Datafile /oracle/TEST_DB/data/tbs_11.dbf - wrote new name
    Datafile /oracle/TEST_DB/data/tbs_12.dbf - wrote new name
    Datafile /oracle/TEST_DB/data/temp1.dbf - wrote new name
    Control File /oracle/TEST_DB/data/cf1.dbf - wrote new name
    Control File /oracle/TEST_DB/data/cf2.dbf - 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.)

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, then 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. All previous backups are still usable.

22.5.4 Troubleshooting DBNEWID

If you encounter an error when using DBNEWID to change a database ID, then refer to these troubleshooting hints.

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 it is 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 scenario underscores the importance of having a recent backup available before you DBNEWID.

If you choose to continue with the change, then rerun your original command. The DBNEWID utility resumes, and attempts to continue the change until all data files and control files have the new value or values. At this point, the database is shut down. You should mount it before 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 Oracle Database 10g Release 1 (10.1) or a later release Oracle Database, then a summary of the operation is written to the alert file.

Example 22-1 Alert Files for a Database Name and Database ID Change

Suppose you changed a database name and database ID. In the alert file, you 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 ***
```

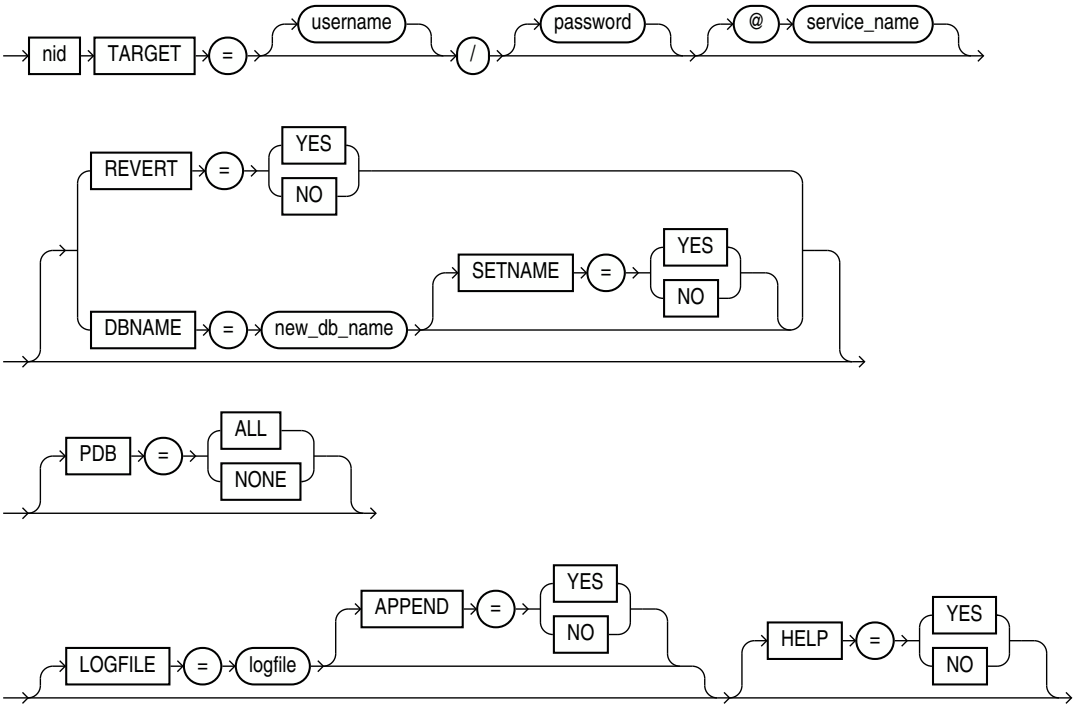
For a change of just the database name, the alert file might show something similar to the following:

```
*** 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 ***
```

22.6 DBNEWID Syntax

Describes DBNEWID syntax.

The following diagrams show the syntax for the DBNEWID utility.



- DBNEWID Parameters
  Describes the parameters for DBNEWID.
- Restrictions and Usage Notes
  Describes restrictions for the DBNEWID utility.
- Additional Restrictions for Releases Earlier Than Oracle Database 10g
  Describes additional restrictions if the DBNEWID utility is run against an Oracle Database release earlier than 10.1.

22.6.1 DBNEWID Parameters

Describes the parameters for DBNEWID.

The following table describes the parameters in the DBNEWID syntax.

Table 22-1 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.

Table 22-1 (Cont.) Parameters for the DBNEWID Utility

Parameter	Description
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.
PDB	Changes the DBID on either all or none of the pluggable databases (PDBs) in a multitenant container database (CDB). (By default, when you run the DBNEWID utility on a container database (CDB) it changes the DBID of only the CDB; the DBIDs of the pluggable databases (PDBs) comprising the CDB remain the same.) The PDB parameter is applicable only in a CDB environment.
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).

22.6.2 Restrictions and Usage Notes

Describes restrictions for the DBNEWID utility.

For example:

- To change the DBID of a database, the database must be mounted and must have been shut down consistently before 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 data files should be consistent without needing recovery.
- Normal offline data files should be accessible and writable. If this is not the case, then 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 before 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.
- The DBNEWID utility does not change global database names. See Considerations for Global Database Names.

22.6.3 Additional Restrictions for Releases Earlier Than Oracle Database 10*g*

Describes additional restrictions if the DBNEWID utility is run against an Oracle Database release earlier than 10.1.

For example:

- The `nid` executable file should be owned and run by the Oracle owner because it needs direct access to the data files and control files. If another user runs the utility, then set the user ID to the owner of the data files and control files.
- The DBNEWID utility must access the data files 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.

23

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.

See the following topics:

You can use LogMiner from a command line or you can access it 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.

> **Note:**
>
> The `continuous_mine` option for the `dbms_logmnr.start_logmnr` package is desupported in Oracle Database 19c (19.1), and is no longer available.

- 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.
- Introduction to LogMiner
  These topics provide a brief introduction to LogMiner, including the following topics.
- Using LogMiner in a CDB
  You can use LogMiner in a multitenant container database (CDB).
- 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.
- Starting LogMiner
  Call the `DBMS_LOGMNR.START_LOGMNR` procedure to start LogMiner.
- Querying V$LOGMNR_CONTENTS for Redo Data of Interest
  You access the redo data of interest by querying the `V$LOGMNR_CONTENTS` view.
- 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.
- Reapplying DDL Statements Returned to V$LOGMNR_CONTENTS
  Some DDL statements that you issue cause Oracle to internally execute one or more other DDL statements.
- Calling DBMS_LOGMNR.START_LOGMNR Multiple Times
  Even after you have successfully called `DBMS_LOGMNR.START_LOGMNR` and selected from the `V$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.

- Supplemental Logging
  Describes supplemental logging.
- Accessing LogMiner Operational Information in Views
  LogMiner operational information (as opposed to redo data) is contained in views.
- Steps in a Typical LogMiner Session
  Learn about the typical ways you can use LogMiner to extract and mine data.
- Examples Using LogMiner
  To see how you can use LogMiner for data mining, review the provided examples.
- Supported Data Types, Storage Attributes, and Database and Redo Log File Versions
  Describes information about data type and storage attribute support and the releases of the database and redo log files that are supported.

23.1 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 auditing tool, and also as a sophisticated data analysis tool.

> **Note:**
>
> LogMiner is intended for use as a debugging tool, to extract information from the redo logs to solve problems. It is not intended to be used for any third party replication of data in a production environment.

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 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, then 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 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 for an example.

- Performing postauditing. LogMiner can be used to track any data manipulation language (DML) and data definition language (DDL) statements run on the database, the order in which they were run, and who ran 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 Auditing Database Activity in *Oracle Database Administrator's Guide*.

23.2 Introduction to LogMiner

These topics provide a brief introduction to LogMiner, including the following topics.

- LogMiner Configuration
  Learn about the objects that LogMiner analyzes, and see examples of configuration files.
- 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.

23.2.1 LogMiner Configuration

Learn about the objects that LogMiner analyzes, and see examples of configuration files.

- Objects in LogMiner Configuration Files
  DataMiner Configuration files have four objects: the source database, the mining database, the LogMiner dictionary, and the redo log files containing the data of interest.
- LogMiner Configuration Example
  This example shows how you can generate redo logs on one Oracle Database release in one location, and send them to another Oracle Database of a different release in another location.
- LogMiner Requirements
  Learn about the requirements for the source and mining database, the data dictionary, the redo log files, and table and column name limits for databases that you want LogMiner to mine.

23.2.1.1 Objects in LogMiner Configuration Files

DataMiner Configuration files have four objects: 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** enables 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 data types 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 SQL statement:

```
 INSERT INTO HR.JOBS(JOB_ID, JOB_TITLE, MIN_SALARY, MAX_SALARY)
VALUES('IT_WT','Technical Writer', 4000, 11000);
```

When LogMiner delivers results without the LogMiner dictionary, LogMiner displays the following output:

```
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 to the database dictionary.

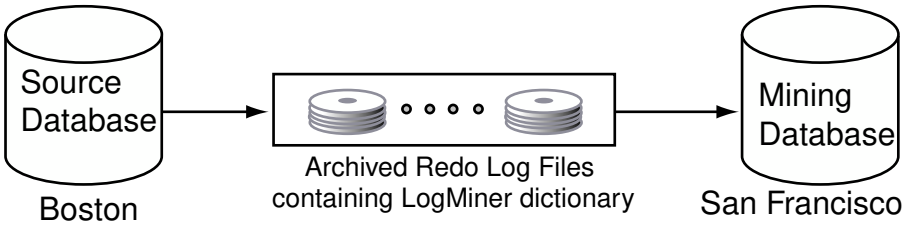
23.2.1.2 LogMiner Configuration Example

This example shows how you can generate redo logs on one Oracle Database release in one location, and send them to another Oracle Database of a different release in another location.

In the following figure, you can see an example of a LogMiner configuration, in which the Source database is in Boston, and the Target database is in San Francisco.

The Source database in Boston generates redo log files that are archived and shipped to the database in San Francisco. A LogMiner dictionary has been extracted to these redo log files. The mining database, where LogMiner actually analyzes the redo log files, is in San Francisco. The Boston database is running Oracle Database 12g and the San Francisco database is running Oracle Database 19c.

Figure 23-1 Example LogMiner Database Configuration



This example 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.

Related Topics

- LogMiner Dictionary Options
  LogMiner requires a dictionary to translate object IDs into object names when it returns redo data to you.

23.2.1.3 LogMiner Requirements

Learn about the requirements for the source and mining database, the data dictionary, the redo log files, and table and column name limits for databases that you want LogMiner to mine.

LogMiner requires the following objects:

- A Source database and a Mining database, with the following characteristics:
  - 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 using either the same release or a later release 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) that is used by the source database.
- LogMiner dictionary
  - The dictionary must be produced by the same Source database that generates the redo log files that you want LogMiner to analyze.
- All redo log files, with the following characteristics:
  - The redo log files must be produced by the same source database.
  - The redo log files must be associated with the same database `RESETLOGS SCN`.
  - The redo log files 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 Oracle9i or later database.
  - The tables or column names selected for mining must not exceed 30 characters.

> **Note:**
>
> Datatypes and features added after Oracle Database 12c Release 2 (12.2) that use extended column formats greater than 30 characters, including JSON-formatted extended varchar2 columns and extended varchar column names, are only supported from the `DBMS_ROLLING` PL/SQL package, Oracle GoldenGate, and XStream. Virtual column names that exceed 30 characters are `UNSUPPORTED` in `v$logmnr_contents` (`dba_logstdby_unsupported` and `dba_rolling_unsupported` views).

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 that you want to analyze. LogMiner requires table or column names that are 30 characters or less.

> **Note:**
>
> You must enable supplemental logging before generating log files that will be analyzed by LogMiner.
>
> 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. Therefore, at the very least, you must enable minimal supplemental logging, as the following SQL statement shows:
>
> ```
> ALTER DATABASE ADD SUPPLEMENTAL LOG DATA;
> ```
>
> To determine whether supplemental logging is enabled, query the `V$DATABASE` view, as the following SQL statement shows:
>
> ```
> SELECT SUPPLEMENTAL_LOG_DATA_MIN FROM V$DATABASE;
> ```
>
> If the query returns a value of `YES` or `IMPLICIT`, then minimal supplemental logging is enabled.

Be aware that the LogMiner utility (`DBMS_LOGMNR`) does not support long table or column names when supplemental logging is enabled. When using an online dictionary, and without any supplement logging enabled, `v$logmnr_contents` shows all names, and `sql_undo` or `sql_redo` for the relevant objects. However, using the LogMiner utility requires that you enable at least minimal supplemental logging. When mining tables with table names or column names exceeding 30 characters, entries in `v$logmnr_contents` such as the following appear:

```
select sql_redo , operation, seg_name, info
         from v$logmnr_contents  where seg_name =
        upper('my_table_with_a_very_very_long_name_for_test') or    seg_name =
        upper('table_with_long_col_name') ;
SQL_REDO --- OPERATION -- SEG_NAME ---------------------- INFO
-------------------------
Unsupported  UNSUPPORTED  MY_TABLE_W_A_VERY_VERY_LONG_NAME Object or Data type
Unsupported
Unsupported  UNSUPPORTED  TABLE_WITH_LONG_COL_NAME         Object or Data type
Unsupported
```

Accordingly, use LogMiner with tables and columns with names that are 30 characters or less.

Related Topics

- Supported Databases and Redo Log File Versions
  Describes supported database releases and redo log file versions.
- Supplemental Logging
  Describes supplemental logging.

23.2.2 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.

For example:

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 the `EXECUTE_CATALOG_ROLE` role and the `LOGMINING` privilege to query the `V$LOGMNR_CONTENTS` view and to use the LogMiner PL/SQL packages.

> **Note:**
>
> When mining a specified time or SCN range of interest within archived logs generated by an Oracle RAC database, you must ensure that you have specified all archived logs from all redo threads that were active during that time or SCN range. If you fail to do this, then any queries of `V$LOGMNR_CONTENTS` return only partial results (based on the archived logs specified to LogMiner through the `DBMS_LOGMNR.ADD_LOGFILE` procedure).
>
> The `CONTINUOUS_MINE` option for the `dbms_logmnr.start_logmnr` package is desupported in Oracle Database 19c (19.1), and is no longer available.

> **See Also:**
>
> Steps in a Typical LogMiner Session for an example of using LogMiner

23.3 Using LogMiner in a CDB

You can use LogMiner in a multitenant container database (CDB).

The following sections discuss some differences to be aware of when using LogMiner in a CDB versus a non-CDB:

LogMiner supports CDBs that have PDBs of different character sets provided the root container has a character set that is a superset of all the PDBs.

To administer a multitenant environment you must have the `CDB_DBA` role.

- LogMiner V$ Views and DBA Views in a CDB
  In a CDB, views used by LogMiner to show information about LogMiner sessions running in the system contain an additional column named CON_ID.
- The V$LOGMNR_CONTENTS View in a CDB
  In a CDB, the V$LOGMNR_CONTENTS view and its associated functions are restricted to the root database. Several new columns exist in V$LOGMNR_CONTENTS in support of CDBs.
- Enabling Supplemental Logging in a CDB
  In a CDB, the syntax for enabling and disabling database-wide supplemental logging is the same as in a non-CDB database
- Using a Flat File Dictionary in a CDB
  You cannot take a dictionary snapshot for an entire CDB in a single flat file. You must be connected to a distinct PDB, and can take a snapshot of only that PDB in a flat file.

Related Topics

- LogMiner V$ Views and DBA Views in a CDB
  In a CDB, views used by LogMiner to show information about LogMiner sessions running in the system contain an additional column named CON_ID.
- The V$LOGMNR_CONTENTS View in a CDB
  In a CDB, the V$LOGMNR_CONTENTS view and its associated functions are restricted to the root database. Several new columns exist in V$LOGMNR_CONTENTS in support of CDBs.
- Enabling Supplemental Logging in a CDB
  In a CDB, the syntax for enabling and disabling database-wide supplemental logging is the same as in a non-CDB database
- Using a Flat File Dictionary in a CDB
  You cannot take a dictionary snapshot for an entire CDB in a single flat file. You must be connected to a distinct PDB, and can take a snapshot of only that PDB in a flat file.

23.3.1 LogMiner V$ Views and DBA Views in a CDB

In a CDB, views used by LogMiner to show information about LogMiner sessions running in the system contain an additional column named CON_ID.

The CON_ID column identifies the container ID associated with the session for which information is being displayed. When you query the view from a pluggable database (PDB), only information associated with the database is displayed. The following views are affected by this new behavior:

- V$LOGMNR_DICTIONARY_LOAD
- V$LOGMNR_LATCH
- V$LOGMNR_PROCESS
- V$LOGMNR_SESSION
- V$LOGMNR_STATS

> **Note:**
>
> To support CDBs, the V$LOGMNR_CONTENTS view has several other new columns in addition to CON_ID.

The following DBA views have analogous CDB views whose names begin with CDB.

Type of Log View	DBA View	CDB View
LogMiner Log Views	DBA_LOGMNR_LOG	CDB_LOGMNR_LOG
LogMiner Purged Log Views	DBA_LOGMNR_PURGED_LOG	CDB_LOGMNR_PURGED_LOG
LogMiner Session Log Views	DBA_LOGMNR_SESSION	CDB_LOGMNR_SESSION

The DBA views show only information related to sessions defined in the container in which they are queried.

The CDB views contain an additional CON_ID column, which identifies the container whose data a given row represents. When CDB views are queried from the root, they can be used to see information about all containers.

23.3.2 The V$LOGMNR_CONTENTS View in a CDB

In a CDB, the V$LOGMNR_CONTENTS view and its associated functions are restricted to the root database. Several new columns exist in V$LOGMNR_CONTENTS in support of CDBs.

- CON_ID — contains the ID associated with the container from which the query is executed. Because V$LOGMNR_CONTENTS is restricted to the root database, this column returns a value of 1 when a query is done on a CDB.
- SRC_CON_NAME — the PDB name. This information is available only when mining is performed with a LogMiner dictionary.
- SRC_CON_ID — the container ID of the PDB that generated the redo record. This information is available only when mining is performed with a LogMiner dictionary.
- SRC_CON_DBID — the PDB identifier. This information is available only when mining is performed with a current LogMiner dictionary.
- SRC_CON_GUID — contains the GUID associated with the PDB. This information is available only when mining is performed with a current LogMiner dictionary.

23.3.3 Enabling Supplemental Logging in a CDB

In a CDB, the syntax for enabling and disabling database-wide supplemental logging is the same as in a non-CDB database

For example, use the following syntax when adding or dropping supplemental log data:

```
ALTER DATABASE [ADD|DROP] SUPPLEMENTAL LOG DATA ...
```

However, note the following:

- In a CDB, supplemental logging levels that are enabled from CDB$ROOT are enabled across the CDB.
- If at least minimal supplemental logging is enabled in CDB$ROOT, then additional supplemental logging levels can be enabled at the PDB level.
- Supplemental logging levels enabled at the CDB level from CDB$ROOT cannot be disabled at the PDB level.
- Dropping all supplemental logging from CDB$ROOT disables all supplemental logging across the CDB regardless of previous PDB level settings.

Supplemental logging operations started with `CREATE TABLE` and `ALTER TABLE` statements can be executed from either the CDB root or a PDB. They affect only the table to which they are applied.

23.3.4 Using a Flat File Dictionary in a CDB

You cannot take a dictionary snapshot for an entire CDB in a single flat file. You must be connected to a distinct PDB, and can take a snapshot of only that PDB in a flat file.

Thus, when using a flat file dictionary, you can only mine the redo logs for the changes associated with the PDB whose data dictionary is contained within the flat file.

23.4 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
  LogMiner requires a dictionary to translate object IDs into object names when it returns redo data to you.
- Redo Log File Options
  To mine data in the redo log files, LogMiner needs information about which redo log files to mine.

23.4.1 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

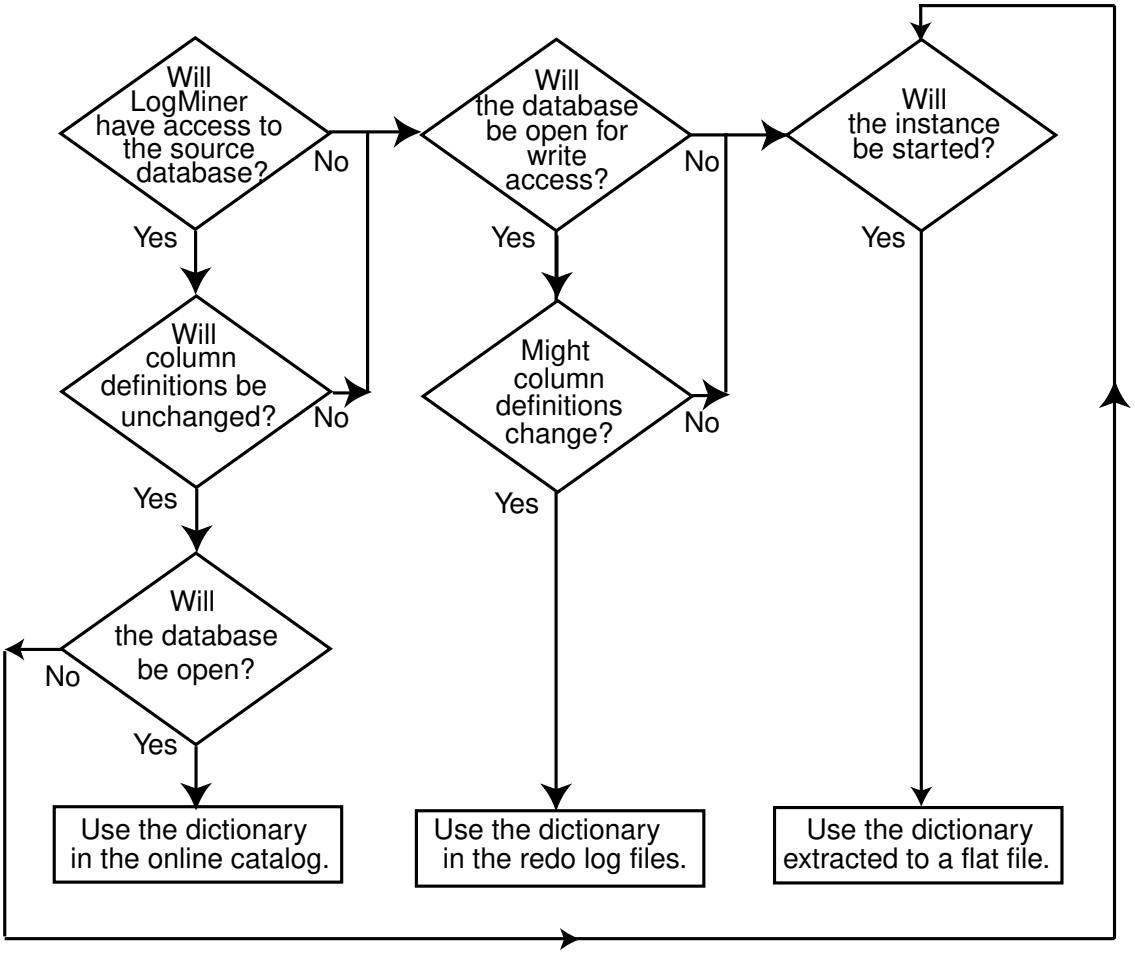
  Oracle recommends that you use this option when you will have access to the source database from which the redo log files were created and when no changes to the column definitions in the tables of interest are anticipated. This is the most efficient and easy-to-use option.
- Extracting a LogMiner Dictionary 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 to redo log files instead.

Figure 23-2 shows a decision tree to help you select a LogMiner dictionary, depending on your situation.

Figure 23-2 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.
- 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.
- 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.

23.4.1.1 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.

For example:

```
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`.

23.4.1.2 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, execute the PL/SQL `DBMS_LOGMNR_D.BUILD` procedure with the `STORE_IN_REDO_LOGS` option. Do not specify a file name or location.

```
EXECUTE DBMS_LOGMNR_D.BUILD( -
   OPTIONS=> DBMS_LOGMNR_D.STORE_IN_REDO_LOGS);
```

See Also:

- *Oracle Database Administrator's Guide* for more information about `ARCHIVELOG` mode
- *Oracle Database PL/SQL Packages and Types Reference* for a complete description of `DBMS_LOGMNR_D.BUILD`

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, then 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, then 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:

```
SELECT NAME FROM V$ARCHIVED_LOG WHERE DICTIONARY_BEGIN='YES';
SELECT NAME FROM V$ARCHIVED_LOG WHERE DICTIONARY_END='YES';
```

Specify the names of the start and end redo log files, and other redo 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, then 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.

23.4.1.3 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.

> **Note:**
>
> The ability to create flat file dictionary dumps of pluggable databases (PDBs) is desupported in Oracle Database 19c, and desupported entirely in later releases.
>
> In previous releases, using a flat file dictionary was one means of mining the redo logs for the changes associated with a specific PDB whose data dictionary was contained within the flat file. Oracle recommends that you call DBMS_LOGMNR.START_LOGMNR, and supply the system change number (SCN) or time range that you want to mine. The SCN or time range options of START_LOGMNR are enhanced to support mining of individual PDBs.

To extract database dictionary information to a flat file, use the `DBMS_LOGMNR_D.BUILD` procedure with the `STORE_IN_FLAT_FILE` option. Oracle recommends that you regularly back up the dictionary extract to ensure correct analysis of older redo log files.

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 want to.

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 location, or the procedure will fail. The directory location must be a directory object. The following is an example of using the SQL `CREATE DIRECTORY` statement to create a directory object named `my_dictionary_dir` for the path `/oracle/database`.

   ```
   SQL> CREATE DIRECTORY "my_dictionary_dir" AS '/oracle/database';
   ```

> **Note:**
>
> Prior to Oracle Database 12c Release 2 (12.2), you used the `UTL_FILE_DIR` initialization parameter to specify a directory location. However, as of Oracle Database 18c, the `UTL_FILE_DIR` initialization parameter is desupported. It is still supported for backward compatibility, but Oracle strongly recommends that you instead use directory objects.

2. If the database is closed, then use SQL*Plus to mount and open the database whose redo log files you want to analyze. For example, entering the SQL `STARTUP` command mounts and opens the database:

   ```
   SQL> STARTUP
   ```

3. Execute the PL/SQL procedure `DBMS_LOGMNR_D.BUILD`. The following example extracts the LogMiner dictionary file to a flat file named `dictionary.ora` in the directory object `my_dictionary_dir` that was created in step 1.

   ```
   SQL> EXECUTE dbms_logmnr_d.build(dictionary_location=>'my_dictionary_dir', -
                                    dictionary_filename=>'dictionary.ora', -
                                    options => dbms_logmnr_d.store_in_flat_file);
   ```

   You could also specify a file name and location without specifying the `STORE_IN_FLAT_FILE` option. The result would be the same.

Related Topics

- Start LogMiner
  See how to start LogMiner, and what options you can use to analyze redo log files, filter criteria, and other session characteristics.
- Filtering Data by SCN
  To filter data by SCN (system change number), use the `STARTSCN` and `ENDSCN` parameters to the PL/SQL `DBMS_LOGMNR.START_LOGMNR` procedure.

23.4.2 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. Although this example specifies the dictionary from the online catalog, any LogMiner dictionary can be used.

> **Note:**
>
> The `continuous_mine` option for the `dbms_logmnr.start_logmnr` package is desupported in Oracle Database 19c (19.1), and is no longer available.

LogMiner uses the database control file to find and adds redo log files that satisfy your specified time or SCN range to the LogMiner redo log file list. For example:

```
ALTER SESSION SET NLS_DATE_FORMAT = 'DD-MON-YYYY HH24:MI:SS';
EXECUTE DBMS_LOGMNR.START_LOGMNR( -
   STARTTIME => '01-Jan-2012 08:30:00', -
   ENDTIME => '01-Jan-2012 08:45:00', -
   OPTIONS => DBMS_LOGMNR.DICT_FROM_ONLINE_CATALOG + -
   );
```

(To avoid the need to specify the date format in the PL/SQL call to the `DBMS_LOGMNR.START_LOGMNR` procedure, this example uses the SQL `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`.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` PL/SQL procedure to signal that this is the beginning of a new list. For example, enter the following to specify `/oracle/logs/log1.f`:

  ```
  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 `PL/SQL DBMS_LOGMNR.ADD_LOGFILE` procedure. For example, enter the following to add `/oracle/logs/log2.f`:

  ```
  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.

23.5 Starting LogMiner

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
- DICT_FROM_REDO_LOGS
- COMMITTED_DATA_ONLY
- SKIP_CORRUPTION
- NO_SQL_DELIMITER
- PRINT_PRETTY_SQL
- NO_ROWID_IN_STMT
- DDL_DICT_TRACKING

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.

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.

23.6 Querying V$LOGMNR_CONTENTS for Redo Data of Interest

You access the redo data of interest by querying the V$LOGMNR_CONTENTS view.

(Note that you must have either the SYSDBA or LOGMINING privilege to query V$LOGMNR_CONTENTS.) 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, then 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 data types and for rolled back operations.

> **Note:**
>
> LogMiner supports Transparent Data Encryption (TDE), in that V$LOGMNR_CONTENTS shows DML operations performed on tables with encrypted columns (including the encrypted columns being updated), provided the LogMiner data dictionary contains the metadata for the object in question and provided the appropriate master key is in the Oracle wallet. The wallet must be open or V$LOGMNR_CONTENTS cannot interpret the associated redo records. TDE support is not available if the database is not open (either read-only or read-write).

> **See Also:**
>
> *Oracle Database Advanced Security Guide* for more information about TDE

Example of Querying V$LOGMNR_CONTENTS

To find any delete operations that a user named Ron performed on the oe.orders table, issue a SQL query similar to the following:

```
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 is produced by the query. 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 difference 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 be responsible for the deletion of both rows, the output in V$LOGMNR_CONTENTS does not reflect that fact. The actual DELETE statement may have been DELETE FROM OE.ORDERS WHERE CUSTOMER_ID ='101' or it may 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.
- Querying V$LOGMNR_CONTENTS Based on Column Values
  LogMiner lets you make queries based on column values.
- Querying V$LOGMNR_CONTENTS Based on XMLType Columns and Tables
  LogMiner supports redo generated for XMLType columns. XMLType data stored as CLOB is supported when redo is generated at a compatibility setting of 11.0.0.0 or higher.

23.6.1 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.

In some cases, certain columns in V$LOGMNR_CONTENTS may not be populated. For example:

- The TABLE_SPACE column is not populated for rows where the value of the OPERATION column is DDL. This is because a DDL may operate on more than one tablespace. For example, a table can be created with multiple partitions spanning multiple table spaces; hence it would not be accurate to populate the column.
- LogMiner does not generate SQL redo or SQL undo for temporary tables. The SQL_REDO column will contain the string "/* No SQL_REDO for temporary tables */" and the SQL_UNDO column will contain the string "/* No SQL_UNDO for temporary tables */".

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 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.

23.6.2 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:

```
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 data type.
- The Meaning of NULL Values Returned by the MINE_VALUE Function
  Describes the meaning of NULL values returned by the MINE_VALUE function.

- Usage Rules for the MINE_VALUE and COLUMN_PRESENT Functions
  Describes the usage rules that apply to the MINE_VALUE and COLUMN_PRESENT functions.
- Restrictions When Using the MINE_VALUE Function To Get an NCHAR Value
  Describes restrictions when using the MINE_VALUE function.

23.6.2.1 The Meaning of NULL Values Returned by the MINE_VALUE Function

Describes the meaning of NULL values returned by the MINE_VALUE function.

If the MINE_VALUE function returns a NULL value, then 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 SQL query:

```
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;
```

23.6.2.2 Usage Rules for the MINE_VALUE and COLUMN_PRESENT Functions

Describes the usage rules that apply to the MINE_VALUE and COLUMN_PRESENT functions.

Specifically:

- They can only be used within a LogMiner session.
- They must be started 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 data types.

23.6.2.3 Restrictions When Using the MINE_VALUE Function To Get an NCHAR Value

Describes restrictions when using the MINE_VALUE function.

If the DBMS_LOGMNR.MINE_VALUE function is used to get an NCHAR value that includes characters not found in the database character set, then those characters are returned as the replacement character (for example, an inverted question mark) of the database character set.

23.6.3 Querying V$LOGMNR_CONTENTS Based on XMLType Columns and Tables

LogMiner supports redo generated for `XMLType` columns. `XMLType` data stored as `CLOB` is supported when redo is generated at a compatibility setting of 11.0.0.0 or higher.

- How V$LOGMNR_CONTENTS Based on XMLType Columns and Tables are Queried
  Depending on what XMLType storage you use, LogMiner presents the `SQL_REDO` in `V$LOGMNR_CONTENTS` in different ways.
- Restrictions When Using LogMiner With XMLType Data
  Describes restrictions when using LogMiner with XMLType data.
- Example of a PL/SQL Procedure for Assembling XMLType Data
  Example showing a procedure that can be used to mine and assemble XML redo for tables that contain out of line XML data.

23.6.3.1 How V$LOGMNR_CONTENTS Based on XMLType Columns and Tables are Queried

Depending on what XMLType storage you use, LogMiner presents the `SQL_REDO` in `V$LOGMNR_CONTENTS` in different ways.

`XMLType` data stored as object-relational and binary XML is supported for redo generated at a compatibility setting of 11.2.0.3 and higher.

LogMiner presents the `SQL_REDO` in `V$LOGMNR_CONTENTS` in different ways depending on the `XMLType` storage. In all cases, the contents of the `SQL_REDO` column, in combination with the `STATUS` column, require careful scrutiny, and usually require reassembly before a SQL or PL/SQL statement can be generated to redo the change. There can be cases when it is not possible to use the `SQL_REDO` data to construct such a change. The examples in the following subsections are based on `XMLType` stored as `CLOB` which is generally the simplest to use for reconstruction of the complete row change.

> **Note:**
>
> `XMLType` data stored as `CLOB` was deprecated in Oracle Database 12c Release 1 (12.1), and can be desupported. For any existing applications that you plan to use on ADB, be aware that many XML schema-related features are not supported

Querying V$LOGMNR_CONTENTS For Changes to Tables With XMLType Columns

The example in this section is for a table named `XML_CLOB_COL_TAB` that has the following columns:

- f1 `NUMBER`
- f2 `VARCHAR2(100)`
- f3 `XMLTYPE`
- f4 `XMLTYPE`
- f5 `VARCHAR2(10)`

Assume that a LogMiner session has been started with the logs and with the COMMITED_DATA_ONLY option. The following query is executed against V$LOGMNR_CONTENTS for changes to the XML_CLOB_COL_TAB table.

```
SELECT OPERATION, STATUS, SQL_REDO FROM V$LOGMNR_CONTENTS
  WHERE SEG_OWNER = 'SCOTT' AND TABLE_NAME = 'XML_CLOB_COL_TAB';
```

The query output looks similar to the following:

```
OPERATION         STATUS  SQL_REDO

INSERT            0       insert into "SCOTT"."XML_CLOB_COL_TAB"("F1","F2","F5") values
                             ('5010','Aho40431','PETER')

XML DOC BEGIN     5       update "SCOTT"."XML_CLOB_COL_TAB" a set a."F3" = XMLType(:1)
                             where a."F1" = '5010' and a."F2" = 'Aho40431' and a."F5" = 'PETER'

XML DOC WRITE     5       XML Data

XML DOC WRITE     5       XML Data

XML DOC WRITE     5       XML Data

XML DOC END       5
```

In the SQL_REDO columns for the XML DOC WRITE operations there will be actual data for the XML document. It will not be the string 'XML Data'.

This output shows that the general model for an insert into a table with an XMLType column is the following:

1. An initial insert with all of the scalar columns.
2. An XML DOC BEGIN operation with an update statement that sets the value for one XMLType column using a bind variable.
3. One or more XML DOC WRITE operations with the data for the XML document.
4. An XML DOC END operation to indicate that all of the data for that XML document has been seen.
5. If there is more than one XMLType column in the table, then steps 2 through 4 will be repeated for each XMLType column that is modified by the original DML.

If the XML document is not stored as an out-of-line column, then there will be no XML DOC BEGIN, XML DOC WRITE, or XML DOC END operations for that column. The document will be included in an update statement similar to the following:

```
OPERATION   STATUS        SQL_REDO

UPDATE      0             update "SCOTT"."XML_CLOB_COL_TAB" a
                          set a."F3" = XMLType('<?xml version="1.0"?>
                          <PO pono="1">
                          <PNAME>Po_99</PNAME>
                          <CUSTNAME>Dave Davids</CUSTNAME>
                          </PO>')
                          where a."F1" = '5006' and a."F2" = 'Janosik' and a."F5" = 'MMM'
```

Querying V$LOGMNR_CONTENTS For Changes to XMLType Tables

DMLs to XMLType tables are slightly different from DMLs to XMLType columns. The XML document represents the value for the row in the XMLType table. Unlike the XMLType column

case, an initial insert cannot be done which is then followed by an update containing the XML document. Rather, the whole document must be assembled before anything can be inserted into the table.

Another difference for XMLType tables is the presence of the OBJECT_ID column. An object identifier is used to uniquely identify every object in an object table. For XMLType tables, this value is generated by Oracle Database when the row is inserted into the table. The OBJECT_ID value cannot be directly inserted into the table using SQL. Therefore, LogMiner cannot generate SQL_REDO which is executable that includes this value.

The V$LOGMNR_CONTENTS view has a new OBJECT_ID column which is populated for changes to XMLType tables. This value is the object identifier from the original table. However, even if this same XML document is inserted into the same XMLType table, a new object identifier will be generated. The SQL_REDO for subsequent DMLs, such as updates and deletes, on the XMLType table will include the object identifier in the WHERE clause to uniquely identify the row from the original table.

23.6.3.2 Restrictions When Using LogMiner With XMLType Data

Describes restrictions when using LogMiner with XMLType data.

Mining XMLType data should only be done when using the DBMS_LOGMNR.COMMITTED_DATA_ONLY option. Otherwise, incomplete changes could be displayed or changes which should be displayed as XML might be displayed as CLOB changes due to missing parts of the row change. This can lead to incomplete and invalid SQL_REDO for these SQL DML statements.

The SQL_UNDO column is not populated for changes to XMLType data.

23.6.3.3 Example of a PL/SQL Procedure for Assembling XMLType Data

Example showing a procedure that can be used to mine and assemble XML redo for tables that contain out of line XML data.

This shows how to assemble the XML data using a temporary LOB. Once the XML document is assembled, it can be used in a meaningful way. This example queries the assembled document for the EmployeeName element and then stores the returned name, the XML document and the SQL_REDO for the original DML in the EMPLOYEE_XML_DOCS table.

> **Note:**
>
> This procedure is an example only and is simplified. It is only intended to illustrate that DMLs to tables with XMLType data can be mined and assembled using LogMiner.

Before calling this procedure, all of the relevant logs must be added to a LogMiner session and DBMS_LOGMNR.START_LOGMNR() must be called with the COMMITTED_DATA_ONLY option. The MINE_AND_ASSEMBLE() procedure can then be called with the schema and table name of the table that has XML data to be mined.

```
-- table to store assembled XML documents
create table employee_xml_docs  (
  employee_name         varchar2(100),
  sql_stmt                    varchar2(4000),
  xml_doc                   SYS.XMLType);
```

```
-- procedure to assemble the XML documents
create or replace procedure mine_and_assemble(
  schemaname        in varchar2,
  tablename         in varchar2)
AS
  loc_c      CLOB;
  row_op     VARCHAR2(100);
  row_status NUMBER;
  stmt       VARCHAR2(4000);
  row_redo   VARCHAR2(4000);
  xml_data   VARCHAR2(32767 CHAR);
  data_len   NUMBER;
  xml_lob    clob;
  xml_doc    XMLType;
BEGIN

-- Look for the rows in V$LOGMNR_CONTENTS that are for the appropriate schema
-- and table name but limit it to those that are valid sql or that need assembly
-- because they are XML documents.

 For item in ( SELECT operation, status, sql_redo  FROM v$logmnr_contents
 where seg_owner = schemaname and table_name = tablename
 and status IN (DBMS_LOGMNR.VALID_SQL, DBMS_LOGMNR.ASSEMBLY_REQUIRED_SQL))
 LOOP
    row_op := item.operation;
    row_status := item.status;
    row_redo := item.sql_redo;

     CASE row_op

          WHEN 'XML DOC BEGIN' THEN
             BEGIN
               -- save statement and begin assembling XML data
               stmt := row_redo;
               xml_data := '';
               data_len := 0;
               DBMS_LOB.CreateTemporary(xml_lob, TRUE);
             END;

          WHEN 'XML DOC WRITE' THEN
             BEGIN
               -- Continue to assemble XML data
               xml_data := xml_data || row_redo;
               data_len := data_len + length(row_redo);
               DBMS_LOB.WriteAppend(xml_lob, length(row_redo), row_redo);
             END;

          WHEN 'XML DOC END' THEN
             BEGIN
               -- Now that assembly is complete, we can use the XML document
              xml_doc := XMLType.createXML(xml_lob);
              insert into employee_xml_docs values
                        (extractvalue(xml_doc, '/EMPLOYEE/NAME'), stmt, xml_doc);
              commit;

              -- reset
              xml_data := '';
              data_len := 0;
              xml_lob := NULL;
             END;

          WHEN 'INSERT' THEN
```

```
                        BEGIN
                           stmt := row_redo;
                        END;

                     WHEN 'UPDATE' THEN
                        BEGIN
                           stmt := row_redo;
                        END;

                     WHEN 'INTERNAL' THEN
                        DBMS_OUTPUT.PUT_LINE('Skip rows marked INTERNAL');

                     ELSE
                        BEGIN
                           stmt := row_redo;
                           DBMS_OUTPUT.PUT_LINE('Other - ' || stmt);
                           IF row_status != DBMS_LOGMNR.VALID_SQL then
                              DBMS_OUTPUT.PUT_LINE('Skip rows marked non-executable');
                           ELSE
                              dbms_output.put_line('Status : ' || row_status);
                           END IF;
                        END;

                  END CASE;

               End LOOP;

            End;
            /

            show errors;
```

This procedure can then be called to mine the changes to the `SCOTT.XML_DATA_TAB` and apply the DMLs.

```
EXECUTE MINE_AND_ASSEMBLE ('SCOTT', 'XML_DATA_TAB');
```

As a result of this procedure, the `EMPLOYEE_XML_DOCS` table will have a row for each out-of-line XML column that was changed. The `EMPLOYEE_NAME` column will have the value extracted from the XML document and the `SQL_STMT` column and the `XML_DOC` column reflect the original row change.

The following is an example query to the resulting table that displays only the employee name and SQL statement:

```
SELECT EMPLOYEE_NAME, SQL_STMT FROM EMPLOYEE_XML_DOCS;

EMPLOYEE_NAME
SQL_STMT

Scott Davis          update "SCOTT"."XML_DATA_TAB" a set a."F3" = XMLType(:1)
                         where a."F1" = '5000' and a."F2" = 'Chen' and a."F5" = 'JJJ'

Richard Harry        update "SCOTT"."XML_DATA_TAB" a set a."F4" = XMLType(:1)
                         where a."F1" = '5000' and a."F2" = 'Chen' and a."F5" = 'JJJ'

Margaret Sally       update "SCOTT"."XML_DATA_TAB" a set a."F4" = XMLType(:1)
                         where a."F1" = '5006' and a."F2" = 'Janosik' and a."F5" = 'MMM'
```

23.7 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.

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 Re-execution
- 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 using the COMMITTED_DATA_ONLY option to DBMS_LOGMNR.START_LOGMNR, only rows belonging to committed transactions are shown in the V$LOGMNR_CONTENTS view.
- Skipping Redo Corruptions
  When you use the SKIP_CORRUPTION option to DBMS_LOGMNR.START_LOGMNR, any corruptions in the redo log files are skipped during select operations from the V$LOGMNR_CONTENTS view.
- Filtering Data by Time
  To filter data by time, set the STARTTIME and ENDTIME parameters in the DBMS_LOGMNR.START_LOGMNR procedure.
- Filtering Data by SCN
  To filter data by SCN (system change number), use the STARTSCN and ENDSCN parameters to the PL/SQL DBMS_LOGMNR.START_LOGMNR procedure.
- Formatting Reconstructed SQL Statements for Re-execution
  By default, a ROWID clause is included in the reconstructed SQL_REDO and SQL_UNDO statements and the statements are ended with a semicolon.
- Formatting the Appearance of Returned Data for Readability
  LogMiner provides the PRINT_PRETTY_SQL option that formats the appearance of returned data for readability.

23.7.1 Showing Only Committed Transactions

When using 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:

```
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, then 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, then 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:

```
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, then 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 for a complete example that uses the `COMMITTED_DATA_ONLY` option

23.7.2 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 SQL example shows how this option works:

```
-- Add redo log files of interest.
--
EXECUTE DBMS_LOGMNR.ADD_LOGFILE(-
   logfilename => '/usr/oracle/data/db1arch_1_16_482701534.log' -
   options => DBMS_LOGMNR.NEW);

-- Start LogMiner
--
EXECUTE DBMS_LOGMNR.START_LOGMNR();

-- Select from the V$LOGMNR_CONTENTS view. This example shows corruptions are -- in the
redo log files.
--
SELECT rbasqn, rbablk, rbabyte, operation, status, info
   FROM V$LOGMNR_CONTENTS;

ERROR at line 3:
ORA-00368: checksum error in redo log block
ORA-00353: log corruption near block 6 change 73528 time 11/06/2011 11:30:23
ORA-00334: archived log: /usr/oracle/data/dbarch1_16_482701534.log

-- Restart LogMiner. This time, specify the SKIP_CORRUPTION option.
--
EXECUTE DBMS_LOGMNR.START_LOGMNR(-
   options => DBMS_LOGMNR.SKIP_CORRUPTION);

-- Select from the V$LOGMNR_CONTENTS view again. The output indicates that
-- corrupted blocks were skipped: CORRUPTED_BLOCKS is in the OPERATION
-- column, 1343 is in the STATUS column, and the number of corrupt blocks
-- skipped is in the INFO column.
--
SELECT rbasqn, rbablk, rbabyte, operation, status, info
   FROM V$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
```

23.7.3 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 PL/SQL `DBMS_LOGMNR.START_LOGMNR` procedure, you can use the SQL `ALTER SESSION SET NLS_DATE_FORMAT` statement first, as shown in the following example.

```
ALTER SESSION SET NLS_DATE_FORMAT = 'DD-MON-YYYY HH24:MI:SS';
EXECUTE DBMS_LOGMNR.START_LOGMNR( -
   DICTFILENAME => '/oracle/database/dictionary.ora', -
   STARTTIME => '01-Jan-2019 08:30:00', -
   ENDTIME => '01-Jan-2019 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.

> **Note:**
>
> You must add log files before filtering. Continuous logging is no longer supported. If logfiles have not been added that match the time or the SCN that you provide, then `DBMS_LOGMNR.START_LOGMNR` fails with the error `1291 ORA-01291: missing logfile`.

23.7.4 Filtering Data by SCN

To filter data by SCN (system change number), use the `STARTSCN` and `ENDSCN` parameters to the PL/SQL `DBMS_LOGMNR.START_LOGMNR` procedure.

For example:

```
EXECUTE DBMS_LOGMNR.START_LOGMNR(-
   STARTSCN => 621047, -
   ENDSCN   => 625695, -
   OPTIONS  => DBMS_LOGMNR.DICT_FROM_ONLINE_CATALOG + -
               );
```

The `STARTSCN` and `ENDSCN` parameters override the `STARTTIME` and `ENDTIME` parameters in situations where all are specified.

> **Note:**
>
> You must add log files before filtering. Continuous logging is no longer supported. If logfiles have not been added that match the time or the SCN that you provide, then `DBMS_LOGMNR.START_LOGMNR` fails with the error `1291 ORA-01291: missing logfile`.

23.7.5 Formatting Reconstructed SQL Statements for Re-execution

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 re-execute the SQL_REDO or SQL_UNDO statements against a different database than the one against which they were originally executed, then 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.

23.7.6 Formatting the Appearance of Returned Data for Readability

LogMiner provides the PRINT_PRETTY_SQL option that formats 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 for a complete example of using the `PRINT_PRETTY_SQL` option

23.8 Reapplying DDL Statements Returned to V$LOGMNR_CONTENTS

Some DDL statements that you issue cause Oracle to internally execute one or more other DDL statements.

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, do not execute statements that were executed internally by Oracle.

> **Note:**
>
> If you execute DML statements that were executed internally by Oracle, then you may corrupt your database. See Step 5 of Example 4: Using the LogMiner Dictionary in the Redo Log Files 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.

To reapply SQL DDL as it was originally applied, re-execute the DDL SQL contained in the `SQL_REDO` or `SQL_UNDO` column of `V$LOGMNR_CONTENTS` only if the `INFO` column contains the value `USER_DDL`.

23.9 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.

Examples: Calling DBMS_LOGMNR.START_LOGMNR Multiple Times

The following are some examples of when it could 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 2: 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 3: 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

23.10 Supplemental Logging

Describes 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 by default LogMiner is not usable. Therefore, you must enable at least minimal supplemental logging before generating log files which will be analyzed by LogMiner.

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
  LogMiner provides different types of database-level supplemental logging: minimal supplemental logging, identification key logging, and procedural supplemental logging, as described in these sections.
- Disabling Database-Level Supplemental Logging
  Disable database-level supplemental logging using the SQL `ALTER DATABASE` statement with the `DROP SUPPLEMENTAL LOGGING` clause.
- Table-Level Supplemental Logging
  Table-level supplemental logging specifies, at the table level, which columns are to be supplementally logged.
- 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).
- DDL_DICT_TRACKING and Supplemental Logging Settings
  Describes interactions that occur when various settings of dictionary tracking and supplemental logging are combined.
- 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 before your requested starting time or SCN (as specified with `DBMS_LOGMNR.START_LOGMNR`) when the `DDL_DICT_TRACKING` option is enabled.

> **See Also:**
>
> Querying Views for Supplemental Logging Settings

23.10.1 Database-Level Supplemental Logging

LogMiner provides different types of database-level supplemental logging: minimal supplemental logging, identification key logging, and procedural supplemental logging, as described in these 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.
- 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.
- Procedural Supplemental Logging
  Procedural supplemental logging causes LogMiner to log certain procedural invocations to redo, so that they can be replicated by rolling upgrades or Oracle GoldenGate.

23.10.1.1 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 and index-organized tables. To enable minimal supplemental logging, execute the following SQL statement:

```
ALTER DATABASE ADD SUPPLEMENTAL LOG DATA;
```

23.10.1.2 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 unconditional supplemental log group

  This option specifies that when a row is updated, all columns of that row (except for LOBs, `LONGS`, and `ADT`s) are placed in the redo log file.

  To enable all column logging at the database level, run the following statement:

  ```
  SQL> ALTER DATABASE ADD SUPPLEMENTAL LOG DATA (ALL) COLUMNS;
  ```

- `PRIMARY KEY` system-generated unconditional 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 should be defined to have primary or unique index keys.

To enable primary key logging at the database level, run the following statement:

```
SQL> ALTER DATABASE ADD SUPPLEMENTAL LOG DATA (PRIMARY KEY) COLUMNS;
```

- `UNIQUE` system-generated conditional supplemental log group

  This option causes the database to place all columns of a row's composite unique key or bitmap index in the redo log file, if any column belonging to the composite unique key or bitmap index is modified. The unique key can be due either to a unique constraint, or to a unique index.

  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 SQL statement:

  ```
  ALTER DATABASE ADD SUPPLEMENTAL LOG DATA (FOREIGN KEY) COLUMNS;
  ```

> **Note:**
>
> Regardless of whether 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 Re-execution for details.

Keep the following in mind when you use identification key logging:

- If the database is open when you enable identification key logging, then 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.
- If you specify `ENABLE NOVALIDATE` for the primary key, then the primary key will not be considered a valid identification key. If there are no valid unique constraints, then all scalar columns are logged. Out of line columns (for example, LOBs, XML, 32k varchar, and so on) are never supplementally logged.
- Supplemental logging statements are cumulative. If you issue the following SQL statements, then both primary key and unique key supplemental logging is enabled:

  ```
  ALTER DATABASE ADD SUPPLEMENTAL LOG DATA (PRIMARY KEY) COLUMNS;
  ALTER DATABASE ADD SUPPLEMENTAL LOG DATA (UNIQUE) COLUMNS;
  ```

23.10.1.3 Procedural Supplemental Logging

Procedural supplemental logging causes LogMiner to log certain procedural invocations to redo, so that they can be replicated by rolling upgrades or Oracle GoldenGate.

Procedural supplemental logging must be enabled for rolling upgrades and Oracle GoldenGate to support replication of AQ queue tables, hierarchy-enabled tables, and tables with `SDO_TOPO_GEOMETRY` or `SDO_GEORASTER` columns. Use the following SQL statement to enable procedural supplemental logging:

```
ALTER DATABASE ADD SUPPLEMENTAL LOG DATA FOR PROCEDURAL REPLICATION END
SUBHEADING
```

If procedural supplemental logging is enabled, then minimal supplemental logging cannot be dropped unless procedural supplemental logging is dropped first.

23.10.2 Disabling Database-Level Supplemental Logging

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 SQL statements, in the following order:

```
ALTER DATABASE ADD SUPPLEMENTAL LOG DATA (PRIMARY KEY) COLUMNS;
ALTER DATABASE ADD SUPPLEMENTAL LOG DATA (UNIQUE) COLUMNS;
ALTER DATABASE DROP SUPPLEMENTAL LOG DATA (PRIMARY KEY) COLUMNS;
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 key supplemental logging are enabled.
- After the third statement, only unique 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:

```
ALTER DATABASE ADD SUPPLEMENTAL LOG DATA (PRIMARY KEY) COLUMNS;
ALTER DATABASE ADD SUPPLEMENTAL LOG DATA (UNIQUE) COLUMNS;
ALTER DATABASE DROP SUPPLEMENTAL LOG DATA (PRIMARY KEY) COLUMNS;
ALTER DATABASE DROP SUPPLEMENTAL LOG DATA (UNIQUE) COLUMNS;
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.

23.10.3 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 key.
- Table-Level User-Defined Supplemental Log Groups
  In addition to table-level identification key logging, Oracle supports user-defined supplemental log groups.
- Usage Notes for User-Defined Supplemental Log Groups
  Hints for using user-defined supplemental log groups.

23.10.3.1 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 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, `LONG`s, and `ADT`s) will be placed in the redo log file:

```
ALTER DATABASE ADD SUPPLEMENTAL LOG DATA (ALL) COLUMNS;
```

However, if you enter the following SQL 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, `LONG`s, and `ADT`s) of the table be placed in the redo log file. If a column changes in the `departments` table, then only the changed column will be placed in the redo log file.

```
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, then all DML cursors for that table in the cursor cache are invalidated. This can affect performance until the cursor cache is repopulated.
- Supplemental logging statements are cumulative. If you issue the following SQL statements, then both primary key and unique index key table-level supplemental logging is enabled:

```
ALTER TABLE HR.EMPLOYEES
  ADD SUPPLEMENTAL LOG DATA (PRIMARY KEY) COLUMNS;
ALTER TABLE HR.EMPLOYEES
  ADD SUPPLEMENTAL LOG DATA (UNIQUE) COLUMNS;
```

See Database-Level Identification Key Logging for a description of each of the identification key logging options.

23.10.3.2 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:

```
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 are logged every time an UPDATE statement is executed on the hr.employees table, regardless of whether the update affected these columns. (To have the entire row image logged any time an update is 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 the SQL ALTER TABLE statement, as shown in the following example:

  ```
  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. As in 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 are 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:

```
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, for logging 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.

23.10.3.3 Usage Notes for User-Defined Supplemental Log Groups

Hints for using 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.

- If you specify the same columns to be logged both conditionally and unconditionally, then the columns are logged unconditionally.

23.10.4 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, then you can use the `DDL_DICT_TRACKING` option to the PL/SQL `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:

```
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 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, then 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, then 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, then 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.

23.10.5 DDL_DICT_TRACKING and Supplemental Logging Settings

Describes interactions that occur when various settings of dictionary tracking and supplemental logging are combined.

Note the following:

- 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$LOGMNR_CONTENTS` will terminate with the ORA-01347 error.
  - A DML transaction is encountered in the redo log file, then LogMiner will not assume that the current version of the table (underlying the DML) in its dictionary is correct, and columns in `V$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.

23.10.6 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 before 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 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$LOGMNR_PARAMETERS view to see where LogMiner will actually start reading. Regardless of where LogMiner starts reading, only rows in your requested range will be returned from the V$LOGMNR_CONTENTS view.

23.11 Accessing LogMiner Operational Information in Views

LogMiner operational information (as opposed to redo data) is contained in 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.

- 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.

- Options for Viewing LogMiner Operational Information
  To check LogMiner operations, you can use SQL to query the LogMiner views, as you would any other view.
- Querying V$LOGMNR_LOGS
  To determine which redo log files have been manually or automatically added to the list of redo log files for LogMiner to analyze, you can query the V$LOGMNR_LOGS view.
- Querying Views for Supplemental Logging Settings
  To determine the current settings for supplemental logging, you can query several different views.
- Querying Individual PDBs Using LogMiner
  To locate a dictionary build, by time or by SCN (for example, when starting per-PDB mining), you can use the SYS.DBA_LOGMNR_DICTIONARY_BUILDLOG view on the source database.

23.11.1 Options for Viewing LogMiner Operational Information

To check LogMiner operations, you can use SQL to query the LogMiner views, as you would any other view.

In addition to V$LOGMNR_CONTENTS, the following is a list of other views and what they show.

- 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.

- 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_COLUMN

  Shows information about the current settings for supplemental logging.

- SYS.DBA_LOGMNR_DICTIONARY_BUILDLOG
  Locates a dictionary build, either by time or by SCN. This view is available in Oracle Database 19c (Release Update 10 and later) both to the CDB$ROOT log miner, and to the per-pdb log miner. For example, when you want to obtain per-PDB log mining, you may need to specify the time or the SCN when you run START_LOGMNR,

23.11.2 Querying V$LOGMNR_LOGS

To determine which redo log files have been manually or automatically added to the list of redo log files for LogMiner to analyze, you can query the V$LOGMNR_LOGS view.

V$LOGMNR_LOGS contains one row for each redo log file. This view provides valuable information about each of the redo log files, including file name, 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. The redo log file 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` option 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, then 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.

> **Note:**
>
> The `continuous_mine` option for the `dbms_logmnr.start_logmnr` package is desupported in Oracle Database 19c (19.1), and is no longer available.

- 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$LOGMNR_CONTENTS` view, then a row is returned in `V$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`

23.11.3 Querying Views for Supplemental Logging Settings

To determine the current settings for supplemental logging, you can query several different views.

You can use one of several views, depending on the information you require:

- `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 option is not enabled
    - YES - if database-level identification key logging with the UNIQUE 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_TYPE 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

23.11.4 Querying Individual PDBs Using LogMiner

To locate a dictionary build, by time or by SCN (for example, when starting per-PDB mining), you can use the `SYS.DBA_LOGMNR_DICTIONARY_BUILDLOG` view on the source database.

Starting with Oracle Database 19c (Release Update 10 and later), you can chose to connect either to the `CDB$ROOT`, or to an individual PDB.

In a traditional On Premises log mining session, you connect to `CDB$ROOT`, and your query is performed for the entire multitenant architecture, including `CDB$ROOT` and the PDBs. With Per-PDB log mining sessions, when you connect to a specific PDB, LogMiner returns rows only for the PDB to which you have connected. This method is required when you want to query redo log files for Oracle Autonomous Database on Oracle Autonomous Cloud Platform Services.

To view log history information for a PDB, you continue to use the `V$LOGMNR_CONTENTS` view. However, to start LogMiner for a PDB, you no longer add log files. Instead, you call `DBMS_LOGMNR.START_LOGMNR`, and supply a system change number (SCN) for the PDB log history that you want to view. You can use any `START_SCN` value that you find in the `DBA_LOGMNR_DICTIONARY_BUILDLOG` view for the PDB.

> **Note:**
>
> When starting LogMiner, if you know the `ENDSCN` or `ENDTIME` value for the log history that you want to view, then you can specify one of those end values.

Example 23-1 Querying SYS.DBA_LOGMNR_DICTIONARY

In the following example, after you connect to the PDB, you query `DBA_LOGMNR_DICTIONARY_BUILDLOG`, identify a `START_SCN` value, and then start LogMiner with `DBMS_LOGMNR.START_LOGMNR`, specifying the SCN value of the log that you want to query.

```
SQL> execute dbms_logmnr_d.build(options => dbms_logmnr_d.store_in_redo_logs);

PL/SQL procedure successfully completed.

SQL> select date_of_build, start_scn from dba_logmnr_dictionary_buildlog;

DATE_OF_BUILD  START_SCN
-------------------- --------------------
09/02/2020 15:58:42 2104064
09/02/2020 19:35:36 3943026
09/02/2020 19:35:54 3943543
09/02/2020 19:35:57 3944009
09/02/2020 19:36:00 3944473
09/10/2020 20:13:22 5902422
09/15/2020 10:03:16 7196131

7 rows selected.

SQL> execute dbms_logmnr.start_logmnr(Options =>
dbms_logmnr.DDL_DICT_TRACKING + dbms_logmnr.DICT_FROM_REDO_LOGS,
startscn=>5902422);
```

```
PL/SQL procedure successfully completed.

SQL> select count(sql_redo) from v$logmnr_contents;

     COUNT(SQL_REDO)
--------------------
              619958

SQL>
```

23.13 Examples Using LogMiner

To see how you can use LogMiner for data mining, review the provided examples.

> **Note:**
>
> All examples in this section assume that minimal supplemental logging has been enabled:
>
> ```
> SQL> ALTER DATABASE ADD SUPPLEMENTAL LOG DATA;
> ```
>
> All examples, except the LogMiner Use Case Scenario examples, 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
  Use examples to see how to specify redo log files.
- LogMiner Use Case Scenarios
  See typical examples of how you can perform data mining tasks with LogMiner.

Related Topics

- Supplemental Logging
  Describes supplemental logging.

23.13.1 Examples of Mining by Explicitly Specifying the Redo Log Files of Interest

Use examples to see how to specify redo log files.

These examples demonstrate how to use LogMiner when you know which redo log files contain the data of interest. These examples are best read sequentially, because each example builds on the example or examples that precede it.

The SQL output formatting can be different on your display than that shown in these examples.

> **Note:**
>
> The `continuous_mine` option for the `dbms_logmnr.start_logmnr` package is desupported in Oracle Database 19c (19.1), and is no longer available. You must specify log files manually

- Example 1: Finding All Modifications in the Last Archived Redo Log File
  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.
- Example 2: Grouping DML Statements into Committed Transactions
  Learn how to use LogMiner to group redo log transactions.
- Example 3: Formatting the Reconstructed SQL
  To make visual inspection easy, you can run LogMiner with the `PRINT_PRETTY_SQL` option.
- Example 4: Using the LogMiner Dictionary in the Redo Log Files
  Learn how to use the dictionary that has been extracted to the redo log files.
- Example 5: Tracking DDL Statements in the Internal Dictionary
  Learn how to use the `DBMS_LOGMNR.DDL_DICT_TRACKING` option to update the LogMiner internal dictionary with the DDL statements encountered in the redo log files.
- Example 6: Filtering Output by Time Range
  To filter a set of redo logs by time, learn about the different ways you can return log files by specifying a time range.

23.13.1.1 Example 1: Finding All Modifications in the Last Archived Redo Log File

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.

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 assumes that you know you want to mine the redo log file that was most recently archived. It 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 (Oracle RAC) database).

1. Determine which redo log file was most recently archived.

```
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
```

2. Specify the list of redo log files to be analyzed. In this case, it is the redo log file that was returned by the query in Step 1.

```
EXECUTE DBMS_LOGMNR.ADD_LOGFILE( -
  LOGFILENAME => '/usr/oracle/data/db1arch_1_16_482701534.dbf', -
  OPTIONS => DBMS_LOGMNR.NEW);
```

3. Start LogMiner and specify the dictionary to use.

```
EXECUTE DBMS_LOGMNR.START_LOGMNR( -
   OPTIONS => DBMS_LOGMNR.DICT_FROM_ONLINE_CATALOG);
```

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.

```
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-2012
                  ('306','Nandini','Shastry',           13:34:43', 'dd-mon-yyyy hh24:mi:ss')
                  'NSHASTRY', '1234567890',             and "JOB_ID" = 'HR_REP' and
                  TO_DATE('10-jan-2012 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-2012
                  ('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-2012 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;
```

5. End the LogMiner session.

```
SQL> EXECUTE DBMS_LOGMNR.END_LOGMNR();
```

23.13.1.2 Example 2: Grouping DML Statements into Committed Transactions

Learn how to use LogMiner to group redo log transactions.

As shown in Example 1, 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.

1. Determine which redo log file was most recently archived by the database.

```
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
```

2. Specify the redo log file that was returned by the query in Step 1. The list will consist of one redo log file.

```
EXECUTE DBMS_LOGMNR.ADD_LOGFILE( -
   LOGFILENAME => '/usr/oracle/data/db1arch_1_16_482701534.dbf', -
   OPTIONS => DBMS_LOGMNR.NEW);
```

3. Start LogMiner by specifying the dictionary to use and the `COMMITTED_DATA_ONLY` option.

```
EXECUTE DBMS_LOGMNR.START_LOGMNR( -
   OPTIONS => DBMS_LOGMNR.DICT_FROM_ONLINE_CATALOG + -
   DBMS_LOGMNR.COMMITTED_DATA_ONLY);
```

4. Query the `V$LOGMNR_CONTENTS` view.

   Although transaction 1.11.1476 was started before transaction 1.1.1484 (as revealed in Step 1), 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.

```
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-2012
                  ('306','Nandini','Shastry',         13:34:43', 'dd-mon-yyyy hh24:mi:ss')
                  'NSHASTRY', '1234567890',           and "JOB_ID" = 'HR_REP' and
                  TO_DATE('10-jan-2012 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-2012
                  ('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-2012 13:41:03',     = '50' and ROWID = 'AAAHSkAABAAAY6rAAP';
                  'dd-mon-yyyy hh24:mi:ss'),
                  'SH_CLERK','110000', '.05',
                  '105','50');

HR    1.11.1476   commit;
```

5. End the LogMiner session.

```
EXECUTE DBMS_LOGMNR.END_LOGMNR();
```

23.13.1.3 Example 3: Formatting the Reconstructed SQL

To make visual inspection easy, you can run LogMiner with the `PRINT_PRETTY_SQL` option.

As shown in Example 2, 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.

1. Determine which redo log file was most recently archived.

```
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
```

2. Specify the redo log file that was returned by the query in Step 1.

```
EXECUTE DBMS_LOGMNR.ADD_LOGFILE( -
   LOGFILENAME => '/usr/oracle/data/db1arch_1_16_482701534.dbf', -
   OPTIONS => DBMS_LOGMNR.NEW);
```

3. Start LogMiner by specifying the dictionary to use and the `COMMITTED_DATA_ONLY` and `PRINT_PRETTY_SQL` options.

```
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.

4. Query the `V$LOGMNR_CONTENTS` view for `SQL_REDO` statements.

```
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-2012 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-2012 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;
```

5. Query the `V$LOGMNR_CONTENTS` view for reconstructed `SQL_UNDO` statements.

```
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-2012 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-2012 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;
```

6. End the LogMiner session.

```
EXECUTE DBMS_LOGMNR.END_LOGMNR();
```

23.13.1.4 Example 4: Using the LogMiner Dictionary in the Redo Log Files

Learn 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.

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.

1. Determine which redo log file was most recently archived by the database.

```
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
```

2. 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:

a. 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.

```
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
```

b. 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:

```
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
```

c. 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.

```
EXECUTE DBMS_LOGMNR.ADD_LOGFILE(-
   LOGFILENAME => '/usr/oracle/data/db1arch_1_210_482701534.dbf', -
       OPTIONS => DBMS_LOGMNR.NEW);
EXECUTE DBMS_LOGMNR.ADD_LOGFILE(-
   LOGFILENAME => '/usr/oracle/data/db1arch_1_208_482701534.dbf');
EXECUTE DBMS_LOGMNR.ADD_LOGFILE(-
   LOGFILENAME => '/usr/oracle/data/db1arch_1_207_482701534.dbf');
```

d. 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.

3. Start LogMiner by specifying the dictionary to use and the `COMMITTED_DATA_ONLY` and `PRINT_PRETTY_SQL` options.

```
EXECUTE DBMS_LOGMNR.START_LOGMNR(-
   OPTIONS => DBMS_LOGMNR.DICT_FROM_REDO_LOGS + -
              DBMS_LOGMNR.COMMITTED_DATA_ONLY + -
              DBMS_LOGMNR.PRINT_PRETTY_SQL);
```

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.)

```
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-2012 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;
```

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, then 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.

```
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-2012 14:01:03', 'dd-mon-yyyy hh24:mi:ss'),
    "MTIME" = TO_DATE('13-jan-2012 14:01:03', 'dd-mon-yyyy hh24:mi:ss'),
    "STIME" = TO_DATE('13-jan-2012 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-2012 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-2012 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;
```

6. End the LogMiner session.

```
EXECUTE DBMS_LOGMNR.END_LOGMNR();
```

23.13.1.5 Example 5: Tracking DDL Statements in the Internal Dictionary

Learn how to use the `DBMS_LOGMNR.DDL_DICT_TRACKING` option to update the LogMiner internal dictionary with the DDL statements encountered in the redo log files.

1. Determine which redo log file was most recently archived by the database.

```
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
```

2. Because the dictionary can be contained in more than one redo log file, determine which redo log files contain the start and end of the data dictionary. To do this, query the `V$ARCHIVED_LOG` view, as follows:

   a. 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.

```
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
```

   b. 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:

```
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
```

3. Ensure that 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:

```
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
```

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.

```
EXECUTE DBMS_LOGMNR.ADD_LOGFILE(-
   LOGFILENAME => '/usr/oracle/data/db1arch_1_210_482701534.dbf', -


       OPTIONS => DBMS_LOGMNR.NEW);


EXECUTE DBMS_LOGMNR.ADD_LOGFILE(-
   LOGFILENAME => '/usr/oracle/data/db1arch_1_209_482701534.dbf');
EXECUTE DBMS_LOGMNR.ADD_LOGFILE(-
   LOGFILENAME => '/usr/oracle/data/db1arch_1_208_482701534.dbf');
EXECUTE DBMS_LOGMNR.ADD_LOGFILE(-
   LOGFILENAME => '/usr/oracle/data/db1arch_1_207_482701534.dbf');
```

5. Start LogMiner by specifying the dictionary to use and the `DDL_DICT_TRACKING`, `COMMITTED_DATA_ONLY`, and `PRINT_PRETTY_SQL` options.

```
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);
```

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.

```
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-2012 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-2012 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-2012 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;
```

7. End the LogMiner session.

```
EXECUTE DBMS_LOGMNR.END_LOGMNR();
```

23.13.1.6 Example 6: Filtering Output by Time Range

To filter a set of redo logs by time, learn about the different ways you can return log files by specifying a time range.

In Example 4 and Example 5, you saw how to filter rows by specifying a timestamp-based predicate (`timestamp > '10-jan-2012 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.

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-13-2012.

```
--
-- 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;
/

EXECUTE my_add_logfiles(in_start_time => '13-jan-2012 14:00:00');
```

2. To see the list of redo log files, query the `V$LOGMNR_LOGS` view.

   This example includes the size of the redo log files in the output.

```
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-2012 14:02:35  23683584
/usr/orcl/arch1_311_482932022.dbf    13-jan-2012 14:56:35  2564096
/usr/orcl/arch1_312_482932022.dbf    13-jan-2012 15:10:43  23683584
/usr/orcl/arch1_313_482932022.dbf    13-jan-2012 15:17:52  23683584
/usr/orcl/arch1_314_482932022.dbf    13-jan-2012 15:23:10  23683584
```

```
/usr/orcl/arch1_315_482932022.dbf    13-jan-2012 15:43:22  23683584
/usr/orcl/arch1_316_482932022.dbf    13-jan-2012 16:03:10  23683584
/usr/orcl/arch1_317_482932022.dbf    13-jan-2012 16:33:43  23683584
/usr/orcl/arch1_318_482932022.dbf    13-jan-2012 17:23:10  23683584
```

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 can use the query predicate (`timestamp > '13-jan-2012 15:00:00' and timestamp < '13-jan-2012 16:00:00'`) to accomplish this goal. 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.

```
EXECUTE DBMS_LOGMNR.START_LOGMNR(-
   STARTTIME => '13-jan-2012 15:00:00', -
   ENDTIME   => '13-jan-2012 16:00:00', -
   OPTIONS   => DBMS_LOGMNR.DICT_FROM_ONLINE_CATALOG + -
                DBMS_LOGMNR.COMMITTED_DATA_ONLY + -
                DBMS_LOGMNR.PRINT_PRETTY_SQL);
```

4. Query the `V$LOGMNR_CONTENTS` view.

```
SELECT TIMESTAMP, (XIDUSN || '.' || XIDSLT || '.' || XIDSQN) AS XID,


 SQL_REDO FROM V$LOGMNR_CONTENTS WHERE SEG_OWNER = 'OE';

TIMESTAMP             XID          SQL_REDO
--------------------  -----------  --------------------------------
13-jan-2012 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-2012 15:29:34   1.17.2376      insert into "OE"."PRODUCT_TRACKING"
                                        values
                                        "PRODUCT_ID" = 3399,
                                        "MODIFIED_TIME" = TO_DATE('13-jan-2012 15:29:34',
                                        'dd-mon-yyyy hh24:mi:ss'),
                                        "OLD_LIST_PRICE" = 815,
                                        "OLD_WARRANTY_PERIOD" = TO_YMINTERVAL('+02-00');

13-jan-2012 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-2012 15:52:43   1.15.1756      insert into "OE"."PRODUCT_TRACKING"
```

```
values
"PRODUCT_ID" = 1768,
"MODIFIED_TIME" = TO_DATE('13-jan-2012 16:52:43',
'dd-mon-yyyy hh24:mi:ss'),
"OLD_LIST_PRICE" = 715,
"OLD_WARRANTY_PERIOD" = TO_YMINTERVAL('+02-00');
```

5. End the LogMiner session.

```
EXECUTE DBMS_LOGMNR.END_LOGMNR();
```

23.13.2 LogMiner Use Case Scenarios

See typical examples of how you can perform data mining tasks with LogMiner.

- Using LogMiner to Track Changes Made by a Specific User
  Learn how to use LogMiner to identify all changes made to the database in a specific time range by a single user.
- Using LogMiner to Calculate Table Access Statistics
  Learn how to use LogMiner to calculate table access statistics over a given time range.

23.13.2.1 Using LogMiner to Track Changes Made by a Specific User

Learn how to use LogMiner to identify all changes made to the database in a specific time range by a single user.

Suppose you want to determine all the changes that the user `joedevo` has made to the database in a specific time range. To perform this task, you can use LogMiner:

1. Connect to the database.
2. 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. This example uses a LogMiner dictionary that has been extracted to the redo log files.
3. 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:

   ```
   EXECUTE DBMS_LOGMNR.ADD_LOGFILE( -
      LOGFILENAME => 'log1orc1.ora', -
      OPTIONS => DBMS_LOGMNR.NEW);
   ```

   If desired, add additional redo log files, as follows:

   ```
   EXECUTE DBMS_LOGMNR.ADD_LOGFILE( -
      LOGFILENAME => 'log2orc1.ora', -
      OPTIONS => DBMS_LOGMNR.ADDFILE);
   ```
4. Start LogMiner and limit the search to the specified time range:

   ```
   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'));
```

5. 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:

```
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;
```

6. End the LogMiner session.

   Use the `DBMS_LOGMNR.END_LOGMNR` procedure to finish the LogMiner session properly:

```
DBMS_LOGMNR.END_LOGMNR( );
```

23.13.2.2 Using LogMiner to Calculate Table Access Statistics

Learn how to use LogMiner to calculate table access statistics over a given time range.

In this example, assume you manage a direct marketing database, and you want to determine how productive the customer contacts have been in generating revenue for a 2-week period in January. In this case, we assume that you have already created the LogMiner dictionary, and added the redo log files that you want to search. To identify those contacts, search your logs by the time range in January, as follows:

1. Start LogMiner and specify a range of times:

```
EXECUTE DBMS_LOGMNR.START_LOGMNR( -
   STARTTIME => TO_DATE('07-Jan-2012 08:30:00','DD-MON-YYYY HH:MI:SS'), -
   ENDTIME => TO_DATE('21-Jan-2012 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.)

```
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 can 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:

```
DBMS_LOGMNR.END_LOGMNR( );
```

23.14 Supported Data Types, Storage Attributes, and Database and Redo Log File Versions

Describes information about data type and storage attribute support and the releases of the database and redo log files that are supported.

- Supported Data Types and Table Storage Attributes
  Describes supported data types and table storage attributes.
- Unsupported Data Types and Table Storage Attributes
  Describes unsupported data types and table storage attributes.
- Supported Databases and Redo Log File Versions
  Describes supported database releases and redo log file versions.
- SecureFiles LOB Considerations
  SecureFiles LOBs are supported when database compatibility is set to 11.2 or later.

23.14.1 Supported Data Types and Table Storage Attributes

Describes supported data types and table storage attributes.

Database Compatibility and Data Type Release Changes

Be aware that some data types are supported only in certain releases.

In Oracle Database 12c Release 1 (12.1) and later releases, the maximum size of the `VARCHAR2`, `NVARCHAR2`, and `RAW` data types was increased to 32 KB when the `COMPATIBLE` initialization parameter is set to 12.0 or higher, and the `MAX_STRING_SIZE` initialization parameter is set to `EXTENDED`.

For supplemental logging, LogMiner treats 32 KB columns as LOBs.

A 32 KB column cannot be part of an `ALWAYS` supplemental logging group.

Supported Data Types Using LogMiner

LogMiner supports the following data types:

- `BINARY_DOUBLE`
- `BINARY_FLOAT`
- `BLOB`
- `CHAR`
- `CLOB` and `NCLOB`
- `DATE`
- `INTERVAL YEAR TO MONTH`
- `INTERVAL DAY TO SECOND`
- LOBs stored as SecureFiles (requires that the database be run at a compatibility of 11.2 or higher.
- `LONG`
- `LONG RAW`

- NCHAR
- NUMBER
- NVARCHAR2
- Objects stored as VARRAYs
- Objects (Simple and Nested ADTs without Collections)

  Object support (including Oracle-supplied types such as SDO_GEOMETRY, ORDIMAGE, and so on) requires that the database be running Oracle Database 12*c* Release 1 (12.1) or higher with a redo compatibility setting of 12.0.0.0 or higher. The contents of the SQL_REDO column for the XML data-related operations is never valid SQL or PL/SQL.
- Oracle Text
- RAW
- TIMESTAMP
- TIMESTAMP WITH TIMEZONE
- TIMESTAMP WITH LOCAL TIMEZONE
- VARCHAR and VARCHAR2
- XDB
- XMLType data for all storage models, assuming the following primary database compatibility requirements:
  - XMLType stored in CLOB format requires that you run Oracle Database with a compatibility setting of 11.0 or higher. Using XMLType stored as CLOB is deprecated as of Oracle Database 12c Release 1 (12.1).
  - XMLType stored in object-relational format or as binary XML requires that you run Oracle Database with a compatibility setting of 11.2.0.3 or higher, and with a redo compatibility setting of 11.2.0.3 or higher. The contents of the SQL_REDO column for the XML data-related operations is never valid SQL or PL/SQL.
  - For any existing applications that you plan to use on Oracle Autonomous Database (ADB), be aware that many XML schema-related features are not supported. For example, XML storage associated with XML schemas are not available. Use Transportable Binary XML storage instead. Object-relational XML storage and Schema-based binary XML storage are also unavailable on ADB. Review *Oracle XML DB Developer's Guide* for details about XMLType restrictions.

Supported Table Storage Types Using LogMiner

LogMiner supports the following table storage attributes:

- Cluster tables (including index clusters and heap clusters).
- Index-organized tables (IOTs) (partitioned and nonpartitioned, including overflow segments).
- Heap-organized tables (partitioned and nonpartitioned).
- Advanced row compression and basic table compression. Both of these options require a database compatibility setting of 11.1.0 or higher.
- Tables containing LOB columns stored as SecureFiles, when Oracle Database compatibility is set to 11.2 or higher.

- Tables using Hybrid Columnar Compression, when Oracle Database compatibility is set to 11.2.0.2 or higher.
- Compatibility Requirements
  LogMiner support for certain data types and table storage attributes has database compatibility requirements.

Related Topics

- Hybrid Columnar Compression

23.14.1.1 Compatibility Requirements

LogMiner support for certain data types and table storage attributes has database compatibility requirements.

Specifically:

- Multibyte `CLOB` support requires the database to run at a compatibility of 10.1 or higher.
- IOT support without `LOB`s and Overflows requires the database to run at a compatibility of 10.1 or higher.
- IOT support with `LOB` and Overflow requires the database to run at a compatibility of 10.2 or higher.
- TDE and TSE support require the database to run at a compatibility of 11.1 or higher.
- Basic compression and advanced row compression require the database to run at a compatibility of 11.1 or higher.
- Hybrid Columnar Compression support is dependent on the underlying storage system and requires the database to run at a compatibility of 11.2 or higher.

> **See Also:**
>
> - *Oracle Database Concepts* for more information about Hybrid Columnar Compression

23.14.2 Unsupported Data Types and Table Storage Attributes

Describes unsupported data types and table storage attributes.

LogMiner does not support the following data types and table storage attributes. If a table contains columns having any of these unsupported data types, then the entire table is ignored by LogMiner.

- `BFILE`
- Nested tables
- Objects with nested tables
- Tables with identity columns
- Temporal validity columns
- PKREF columns
- PKOID columns

- Nested table attributes and stand-alone nested table columns

23.14.3 Supported Databases and Redo Log File Versions

Describes supported database releases 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 release 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 data types supported by LogMiner.

> **See Also:**
>
> Steps in a Typical LogMiner Session and Supplemental Logging

23.14.4 SecureFiles LOB Considerations

SecureFiles LOBs are supported when database compatibility is set to 11.2 or later.

Only `SQL_REDO` columns can be filled in for SecureFiles LOB columns; `SQL_UNDO` columns are not filled in.

Transparent Data Encryption (TDE) and data compression can be enabled on SecureFiles LOB columns at the primary database.

Deduplication of SecureFiles LOB columns is fully supported. Fragment operations are not supported.

If LogMiner encounters redo generated by unsupported operations, then it generates rows with the `OPERATION` column set to `UNSUPPORTED`. No SQL_REDO or SQL_UNDO will be generated for these redo records.

23.12 Steps in a Typical LogMiner Session

Learn about the typical ways you can use LogMiner to extract and mine data.

- Understanding How to Run LogMiner Sessions
  On Premises and Oracle Autonomous Cloud Platform Services LogMiner Sessions are similar, but require different users.
- Typical LogMiner Session Task 1: Enable Supplemental Logging
  To be able to use LogMiner with redo log files, you must enable supplemental logging.
- Typical LogMiner Session Task 2: Extract a LogMiner Dictionary
  To use LogMiner, you must select an option to supply LogMiner with a database dictionary.
- Typical LogMiner Session Task 3: Specify Redo Log Files for Analysis
  You must specify the redo log files that you want to analyze with `DBMS_LOGMNR_ADD_LOGFILE` before starting LogMiner.

- Start LogMiner
  See how to start LogMiner, and what options you can use to analyze redo log files, filter criteria, and other session characteristics.
- Query V$LOGMNR_CONTENTS
  After you start LogMiner, you can query the Oracle Database `V$LOGMNR_CONTENTS` view.
- Typical LogMiner Session Task 6: End the LogMiner Session
  Ending the LogMiner session.

23.12.1 Understanding How to Run LogMiner Sessions

On Premises and Oracle Autonomous Cloud Platform Services LogMiner Sessions are similar, but require different users.

In a traditional LogMiner session, and when you run LogMiner on `CDB$ROOT`, you run LogMiner by using a PL/SQL package that is owned by `SYS`. To use LogMiner, there are requirements for the user account that you use with LogMiner.

When you run LogMiner in an On-Premise Oracle Database, you can create one `CDB$ROOT` capture extract to capture data from multiple PDBs at the same time, or mine multiple individual PDB logs using Oracle GoldenGate, each capturing data from just one PDB. However for Oracle Autonomous Database Cloud Platform Services, where you do not have access to `CDB$ROOT`, you must use the per-PDB capture method. In this mode, you provision a local user with a predefined set of privileges to the source PDB whose logs you want to review. All LogMiner processing is restricted to this PDB only.

With On-Premise PDBs, you can start as many sessions as resources allow. But for Cloud configurations, while you can still start many concurrent sessions in `CDB$ROOT`, you can start only one session for each PDB using the LogMiner PL/SQL package.

To run LogMiner on `CDB$ROOT`, you use the PL/SQL package `DBMS_LOGMNR.ADD_LOGFILE` and add log files explicitly. Additionally, if you choose to extract a LogMiner dictionary rather than use the online catalog, then you can also use the `DBMS_LOGMNR_D` package.

To run LogMiner on individual PDBs, the procedures are slightly different. instead of using `DBMS_LOGMNR.ADD_LOGFILE`. you specify a period in which you want to review log files for the PDB. Specify the SCN value of the log that you want to query, with either `startScn` and, if you choose, `endScn`, or `startTime`, and if you choose, `endTime`. You then start LogMiner with `DBMS_LOGMNR.START_LOGMNR`. `DBMS_LOGMNR.START_LOGMNR` automatically adds the redo logs for you to analyze.

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.

Requirements for Running LogMiner for Individual PDB

To run LogMiner to query individual PDBs, you must provision a local user with the necessary privilege, using the procedure call `DBMS_GOLDENGATE_AUTH.GRANT_ADMIN_PRIVILEGE`. Also, users with the `GGADMIN` privilege can run Per-PDB capture Extracts.

Again, with individual PDBs, you do not specify the archive logs that you want to mine. Instead, connect to the PDB that you want to mine, and then run `dbms_logmnr_d.STORE_IN_REDO_LOGS`. For example:

```
SQL> execute dbms_logmnr_d.build(option=>dbms_logmnr_d.STORE_IN_REDO_LOGS);
```

You can then connect to the PDB, identify SCNs, then run `dbms_logmnr.start_logmnr` to query the log files for the starting point system change number (SCN) for the PDB log history that you want to view, and if you choose, an end point SCN. Mining proceeds at that point just as with traditional LogMiner queries to the V$LOGMNR_CONTENTS view. However, only redo generated for the PDB to which you are connected is available

> **Note:**
>
> If you shut down a PDB while Extract and any LogMiner processes are running, then these processes are terminated, as with other active sessions. When the PDB is reopened, restart of Extract mining should continue as normal. When you unplug the PDB, there are no special actions required. However, when you plug in a PDB after unplugging it, all LogMiner and Capture sessions that previously existed in the PDB are removed.

Requirements for Running Traditional LogMiner Sessions When Not Connected As SYS

With On Premises log mining, the LogMiner PL/SQL packages are owned by the `SYS` schema. Therefore, if you are not connected as user `SYS`, then:

- You must include `SYS` in your call. For example:

```
EXECUTE SYS.DBMS_LOGMNR.END_LOGMNR;
```

- You must have been granted the `EXECUTE_CATALOG_ROLE` role.

Related Topics

- Querying Individual PDBs Using LogMiner
  To locate a dictionary build, by time or by SCN (for example, when starting per-PDB mining), you can use the `SYS.DBA_LOGMNR_DICTIONARY_BUILDLOG` view on the source database.
- DBMS_LOGMNR
- Overview of PL/SQL Packages

23.12.2 Typical LogMiner Session Task 1: Enable Supplemental Logging

To be able to use LogMiner with redo log files, you must enable supplemental logging.

Redo-based applications can require that additional columns are logged in the redo log files. The process of logging these additional columns is called **supplemental logging**. By default, Oracle Database does not have supplemental logging enabled. At the very least, to use LogMiner, you must enable minimal supplemental logging.

Example 23-2 Enabling Minimal Supplemental Logging

To enable supplemental logging, enter the following statement:

```
ALTER DATABASE ADD SUPPLEMENTAL LOG DATA;
```

Related Topics

- Supplemental Logging
  Describes supplemental logging.

23.12.3 Typical LogMiner Session Task 2: Extract a LogMiner Dictionary

To use LogMiner, you must select an option to supply LogMiner with a database dictionary.

Choose one of the following options:

- Specify use of the online catalog by using the `DICT_FROM_ONLINE_CATALOG` option when you start LogMiner.
- Extract the database dictionary information to the redo log files.
- Extract database dictionary information to a flat file.

Related Topics

- 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.
- 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.
- 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.

23.12.4 Typical LogMiner Session Task 3: Specify Redo Log Files for Analysis

You must specify the redo log files that you want to analyze with `DBMS_LOGMNR_ADD_LOGFILE` before starting LogMiner.

To query logs on `CDB$ROOT` for On Premises, before you can start LogMiner, you must specify the redo log files that you want to analyze. To specify log files, run the `DBMS_LOGMNR.ADD_LOGFILE` procedure, as demonstrated in the following steps. You can add and remove redo log files in any order.

> **Note:**
>
> To query logs for an individual PDB, you use a slightly different procedure. After you connect to the PDB, you query `DBA_LOGMNR_DICTIONARY_BUILDLOG`, identify a `START_SCN` value, and then start LogMiner with `DBMS_LOGMNR.START_LOGMNR`, specifying the SCN value of the log that you want to review. `DBMS_LOGMNR.START_LOGMNR` automatically adds the redo logs for you to analyze. Refer to "Querying Individual PDBs Using LogMiner" for an example.

1. Use SQL*Plus to start an Oracle Database instance, with the database either mounted or unmounted. For example, enter the `STARTUP` statement at the SQL prompt:

```
STARTUP
```

2. Create a list of redo log files. Specify the `NEW` option of the `DBMS_LOGMNR.ADD_LOGFILE` PL/SQL 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:

```
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` PL/SQL procedure. For example, enter the following to add the `/oracle/logs/log2.f` redo log file:

```
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 can simply enter the following:

```
EXECUTE DBMS_LOGMNR.ADD_LOGFILE( -
   LOGFILENAME=>'/oracle/logs/log2.f');
```

4. If desired, remove redo log files by using the `DBMS_LOGMNR.REMOVE_LOGFILE` PL/SQL procedure. For example, enter the following to remove the `/oracle/logs/log2.f` redo log file:

```
EXECUTE DBMS_LOGMNR.REMOVE_LOGFILE( -
   LOGFILENAME => '/oracle/logs/log2.f');
```

Related Topics

- Querying Individual PDBs Using LogMiner
  To locate a dictionary build, by time or by SCN (for example, when starting per-PDB mining), you can use the `SYS.DBA_LOGMNR_DICTIONARY_BUILDLOG` view on the source database.

23.12.5 Start LogMiner

See how to start LogMiner, and what options you can use to analyze redo log files, filter criteria, and other session characteristics.

After you have created a LogMiner dictionary file and specified which redo log files to analyze, you can start LogMiner and analyze your Oracle Database transactions.

1. To start LogMiner, execute the `DBMS_LOGMNR.START_LOGMNR` procedure.

   Oracle recommends that you specify a LogMiner dictionary option. If you do not specify a dictionary option, then LogMiner cannot translate internal object identifiers and data types to object names and external data formats. As a result, LogMiner returns internal object IDs and present data as binary data. Additionally, you cannot use the `MINE_VALUE` and `COLUMN_PRESENT` functions without a dictionary.

If you are specifying the name of a flat file LogMiner dictionary, then you must supply a fully qualified file name for the dictionary file. For example, to start LogMiner using `/oracle/database/dictionary.ora`, issue the following statement:

```
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`, then 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. To see an example of this task, refer to "Extracting a LogMiner Dictionary to the Redo Log Files."

> **Note:**
>
> If you add additional redo log files after LogMiner has been started, then you must restart LogMiner. LogMiner does not retain options included in the previous call to `DBMS_LOGMNR.START_LOGMNR`; you must respecify the options that you want to use. However, if you do not specify a dictionary in the current call to `DBMS_LOGMNR.START_LOGMNR`, then LogMiner does retain the dictionary specification from the previous call.

2. Optionally, you can filter or format your query, or 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:

```
EXECUTE DBMS_LOGMNR.START_LOGMNR(OPTIONS => -
   DBMS_LOGMNR.DICT_FROM_ONLINE_CATALOG + -
   DBMS_LOGMNR.COMMITTED_DATA_ONLY);
```

You can execute the `DBMS_LOGMNR.START_LOGMNR` procedure multiple times, specifying different options each time. For example, if you did not obtain the desired results from a query of `V$LOGMNR_CONTENTS`, you can 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`.

Related Topics

- 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.
- 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.

23.12.6 Query V$LOGMNR_CONTENTS

After you start LogMiner, you can query the Oracle Database `V$LOGMNR_CONTENTS` view.

For example:

```
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');


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;
```

To see more examples, refer to "Filtering an Formatting Data Returned to `V$LOGMNR_CONTENTS`.

Related Topics

- 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.

23.12.7 Typical LogMiner Session Task 6: End the LogMiner Session

Ending the LogMiner session.

To properly end a LogMiner session, use the `DBMS_LOGMNR.END_LOGMNR` PL/SQL procedure, as follows:

```
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, then LogMiner retains all its allocated resources until the end of the Oracle session in which it was called. 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.

24
Using the Metadata APIs

Using the DBMS_METADATA APIs.

The DBMS_METADATA API enables 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 DBMS_METADATA_DIFF API lets you compare objects between databases to identify metadata changes over time in objects of the same type.

- Why Use the DBMS_METADATA API?
  The DBMS_METADATA API eliminates the need for you to write and maintain your own code for metadata extraction.
- Overview of the DBMS_METADATA API
  Overview of the DBMS_METADATA API.
- Using the DBMS_METADATA API to Retrieve an Object's Metadata
  The retrieval interface of the DBMS_METADATA API lets you specify the kind of object to be retrieved.
- Using the DBMS_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.
- Using the DBMS_METADATA API to Retrieve Collections of Different Object Types
  Description and example of retrieving collections of different object types.
- Using the DBMS_METADATA_DIFF API to Compare Object Metadata
  Description and example that uses the retrieval, comparison, and submit interfaces of DBMS_METADATA and DBMS_METADATA_DIFF to fetch metadata for two tables, compare the metadata, and generate ALTER statements which make one table like the other.
- Performance Tips for the Programmatic Interface of the DBMS_METADATA API
  Describes how to enhance performance when using the programmatic interface of the DBMS_METADATA API.
- Example Usage of the DBMS_METADATA API
  Example of how the DBMS_METADATA API could be used.
- Summary of DBMS_METADATA Procedures
  Provides brief descriptions of the procedures provided by the DBMS_METADATA API.
- Summary of DBMS_METADATA_DIFF Procedures
  Provides brief descriptions of the procedures and functions provided by the DBMS_METADATA_DIFF API.

24.1 Why Use the DBMS_METADATA API?

The DBMS_METADATA API eliminates the need for you to write and maintain your own code for metadata extraction.

If you have developed your own code for Oracle Database for extracting metadata from the dictionary, or for manipulating the metadata (adding columns, changing column data types, and so on), and converting the metadata to DDL so that you could recreate the object on the same or another database, then maintenance is an issue. Keeping that code updated to support new dictionary features has probably proven to be challenging.

Oracle Database provides a centralized facility for the extraction, manipulation, and recreation of dictionary metadata. Oracle Database also supports all dictionary objects at their most current level.

Although the `DBMS_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. You can install the `DBMS_METADATA` API in the same way as data dictionary views, by running `catproc.sql` to run a SQL script at database installation time. After you have installed `DBMS_METADATA`, it is available whenever the instance is operational, even in restricted mode.

When you change database releases using the `DBMS_METADATA` API, you are not required to make any source code changes. The DBMS_METADATA API enables the code to be upwardly compatible across different Oracle Database releases. XML documents retrieved by one release can be processed by the submit interface on the same or later releases. For example, XML documents retrieved by an Oracle Database 10g Release 2 (10.2) database can be submitted to Oracle Database 12c.

24.2 Overview of the DBMS_METADATA API

Overview of the `DBMS_METADATA` API.

For the purposes of the `DBMS_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.

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 `DBMS_METADATA` API makes use of XML (Extensible Markup Language) and XSLT (Extensible Stylesheet Language Transformation). The `DBMS_METADATA` API represents object metadata as XML because it is a universal format that can be easily parsed and transformed. The `DBMS_METADATA` API uses XSLT to transform XML documents into either other XML documents or into SQL DDL.

You can use the `DBMS_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 and examples of their implementation, see the following sections:

- Using the DBMS_METADATA API to Retrieve an Object's Metadata
- Using the DBMS_METADATA API to Re-Create a Retrieved Object
- Using the DBMS_METADATA API to Retrieve Collections of Different Object Types

Using Views to Determine Valid DBMS_METADATA Options

You can use the following views to determine which `DBMS_METADATA` transforms are allowed for each object type transformation, the parameters for each transform, and their parse items.

- DBMS_METADATA_TRANSFORMS - documents all valid Oracle-supplied transforms that are used with the `DBMS_METADATA` package.
- DBMS_METADATA_TRANSFORM_PARAMS - documents the valid transform parameters for each transform.
- DBMS_METADATA_PARSE_ITEMS- documents the valid parse items.

For example, suppose you want to know which transforms are allowed for `INDEX` objects. The following query returns the transforms that are valid for `INDEX` objects, the required input types, and the resulting output types:

```
SQL> SELECT transform, output_type, input_type, description
2 FROM dbms_metadata_transforms
3 WHERE object_type='INDEX';


TRANSFORM  OUTPUT_TYP INPUT_TYPE           DESCRIPTION
---------- ---------- --------------------
--------------------------------------------------------------------
ALTERXML   ALTER_XML  SXML difference doc  Generate ALTER_XML from SXML
difference document
SXMLDDL    DDL        SXML                 Convert SXML to DDL
MODIFY     XML        XML                  Modify XML document according to
transform parameters
SXML       SXML       XML                  Convert XML to SXML
DDL        DDL        XML                  Convert XML to SQL to create the
object
ALTERDDL   ALTER_DDL  ALTER_XML            Convert ALTER_XML to ALTER_DDL
MODIFYSXML SXML       SXML                 Modify SXML document
```

You might then want to know which transform parameters are valid for the DDL transform:

```
SQL> SELECT param, datatype, default_val, description
2 FROM dbms_metadata_transform_params
3 WHERE object_type='INDEX' and transform='DDL'
4 ORDER BY param;


PARAM                     DATATYPE   DEFAULT_VA  DESCRIPTION
------------------------- ---------- ----------
--------------------------------------------------------------------
INCLUDE_PARTITIONS        TEXT                   Include generated interval
and list partitions in DDL
                                                      transformation
INDEX_COMPRESSION_CLAUSE  TEXT       ""          Text of user-specified index
compression clause
PARTITIONING              BOOLEAN    TRUE        Include partitioning clauses
in transformation
PARTITION_NAME            TEXT       ""          Name of partition selected
for the transformation
```

```
PCTSPACE                  NUMBER     ""          Percentage by which space
allocation is to be modified
SEGMENT_ATTRIBUTES        BOOLEAN    TRUE        Include segment attribute
clauses (physical attributes, storage
                                                      attribues, tablespace,
logging) in transformation
STORAGE                   BOOLEAN    TRUE        Include storage clauses in
transformation
SUBPARTITION_NAME         TEXT       ""          Name of subpartition
selected for the transformation
TABLESPACE                BOOLEAN    TRUE        Include tablespace clauses
in transformation
```

You can also perform the following query which returns specific metadata about the `INDEX` object type::

```
SQL> SELECT parse_item, description
2 FROM dbms_metadata_parse_items
3 WHERE object_type='INDEX' and convert='Y';

PARSE_ITEM           DESCRIPTION
--------------------
-----------------------------------------------------------------
OBJECT_TYPE          Object type
TABLESPACE           Object tablespace (default tablespace for partitioned
objects)
BASE_OBJECT_SCHEMA   Schema of the base object
SCHEMA               Object schema, if any
NAME                 Object name
BASE_OBJECT_NAME     Name of the base object
BASE_OBJECT_TYPE     Object type of the base object
SYSTEM_GENERATED     Y = system-generated object; N = not system-generated
```

24.3 Using the DBMS_METADATA API to Retrieve an Object's Metadata

The retrieval interface of the `DBMS_METADATA` API 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` role. However, roles are disabled within many PL/SQL objects (stored procedures, functions, definer's rights APIs). 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 the `SELECT_CATALOG_ROLE` role), then you must put the code in an invoker's rights API.

You can use the programmatic 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 `DBMS_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 API.
- Retrieving Multiple Objects
  Description and example of retrieving multiple objects.
- Placing Conditions on Transforms
  You can use transform parameters to specify conditions on the transforms that you add.
- Accessing Specific Metadata Attributes
  Description and example of accessing specific metadata attributes.

24.3.1 Typical Steps Used for Basic Metadata Retrieval

When you retrieve metadata, you use the `DBMS_METADATA` PL/SQL API.

The following examples illustrate the programmatic and browsing interfaces.

The `DBMS_METADATA` programmatic interface example provides a basic demonstration of using the `DBMS_METADATA` programmatic interface to retrieve metadata for one table. It creates a `DBMS_METADATA` program that creates a function named `get_table_md`. This function returns metadata for one table.

The `DBMS_METADATA` browsing interface example demonstrates how you can use the browsing interface to obtain the same results.

Example 24-1 Using the DBMS_METADATA Programmatic Interface to Retrieve Data

1. Create a `DBMS_METADATA` program that creates a function named `get_table_md`, which will return the metadata for one table, `timecards`, in the `hr` schema. The content of such a program looks as follows. (For this example, 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 user `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"
```

Example 24-2 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 of this query are same as shown in step 5 in the `DBMS_METADATA` programmatic interface example.

24.3.2 Retrieving Multiple Objects

Description and example of retrieving multiple objects.

In the previous example “Using the `DBMS_METADATA` Programmatic Interface to Retrieve Data,” 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;
```

The following example demonstrates use of this construct and retrieving multiple objects. Connect as user `scott` for this example. The password is `tiger`.

Example 24-3 Retrieving Multiple Objects

1. Create a table named `my_metadata` and a procedure named `get_tables_md`, as follows. Because not all objects can be returned, they are stored in a table and queried at the end.

```
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;
/
```

2. Execute the procedure:

```
EXECUTE get_tables_md;
```

3. Query the `my_metadata` table to see what was retrieved:

```
SET LONG 9000000
SET PAGES 0
SELECT * FROM my_metadata;
```

24.3.3 Placing Conditions on Transforms

You can use transform parameters to specify conditions on the transforms that 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, then 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. The default is that segment attributes do appear in the DDL.

Example 24-4 shows use of the `SET_TRANSFORM_PARAM` procedure.

Example 24-4 Placing Conditions on Transforms

1. Create a function named `get_table_md`, as follows:

```
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.
 -- Note that this call uses the TRANSFORM handle, not the OPEN handle.
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;
/
```

2. Perform the following query:

```
SQL> SELECT get_table_md FROM dual;
```

   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 `DBMS_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 24-5 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 24-5 Modifying an XML Document

1. Create a function named remap_schema:

```
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;
/
```

2. Perform the following query:

```
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, then you can add your own user-written transforms to process the XML.

24.3.4 Accessing Specific Metadata Attributes

Description and example of 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 `DBMS_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 24-6 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 24-6 Using Parse Items to Access Specific Metadata Attributes

1. Create a table named `my_metadata` and a procedure named `get_tables_and_indexes`, as follows:

```
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$_ddl
objname VARCHAR2(30);   -- the parsed object name
idxddls sys.ku$_ddls;   -- metadata is returned in sys.ku$_ddls,
                        --  a nested table of sys.ku$_ddl objects
idxname VARCHAR2(30);   -- the parsed index 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 index name be returned as a parse item.
  DBMS_METADATA.SET_PARSE_ITEM(h2,'NAME');

  -- 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);


  LOOP
   idxddls := dbms_metadata.fetch_ddl(h2);

   -- When there are no more objects to  be retrieved, FETCH_DDL returns NULL.
   EXIT WHEN idxddls IS NULL;

     FOR i in idxddls.FIRST..idxddls.LAST LOOP
       ddl := idxddls(i).ddlText;
       pi  := idxddls(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
             idxname := pi(j).value;
           END IF;
         END LOOP;
        END IF;

        -- Store the metadata in our table.
         INSERT INTO my_metadata(object_type, name, md)
           VALUES ('INDEX',idxname,ddl);
        COMMIT;
      END LOOP;  -- for loop
  END LOOP;
  DBMS_METADATA.CLOSE(h2);
END LOOP;
```

```
 DBMS_METADATA.CLOSE(h1);
END;
/
```

2. Execute the procedure:

```
EXECUTE get_tables_and_indexes;
```

3. Perform the following query to see what was retrieved:

```
SET LONG 9000000
SET PAGES 0
SELECT * FROM my_metadata;
```

24.4 Using the DBMS_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.

Example 24-7 fetches the XML for a table in one schema, and then uses the submit interface to re-create the table in another schema.

Example 24-7 Using the Submit Interface to Re-Create a Retrieved Object

1. Connect as a privileged user:

```
CONNECT system
Enter password: password
```

2. Create an invoker's rights package to hold the procedure because access to objects in another schema requires the `SELECT_CATALOG_ROLE` 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;
/
```

3. Now create a table named `my_example` in the schema `SCOTT`:

```
CONNECT scott
Enter password:
-- The password is tiger.
```

```
DROP TABLE my_example;
CREATE TABLE my_example (a NUMBER, b VARCHAR2(30));

CONNECT system
Enter password: password

SET LONG 9000000
SET PAGESIZE 0
SET SERVEROUTPUT ON SIZE 100000
```

4. Copy the `my_example` table to the `SYSTEM` schema:

```
DROP TABLE my_example;
EXECUTE example_pkg.move_table('MY_EXAMPLE','SCOTT','SYSTEM');
```

5. Perform the following query to verify that it worked:

```
SELECT DBMS_METADATA.GET_DDL('TABLE','MY_EXAMPLE') FROM dual;
```

24.5 Using the DBMS_METADATA API to Retrieve Collections of Different Object Types

Description and example of 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 `DBMS_METADATA` API provides several heterogeneous object types. A heterogeneous object type is an ordered set of object types.

Oracle supplies the following 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 24-8 shows how to retrieve the object types in the `scott` schema. Connect as user `scott`. The password is `tiger`.

Example 24-8 Retrieving Heterogeneous Object Types

1. Create a table to store the retrieved objects:

```
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;
/
```

2. Execute the procedure:

```
EXECUTE get_schema_md;
```

3. Perform the following query to see what was retrieved:

```
SET LONG 9000000
SET PAGESIZE 0
SELECT * FROM my_metadata;
```

In this example, 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, then there will usually be no errors. (The exceptions usually involve circular references; for example, if package A contains a call to package B, and package B contains a call to package A, then one of the packages will need to be recompiled a second time.)

- Filtering the Return of Heterogeneous Object Types
  Description and example of filtering the return of heterogeneous object types.

24.5.1 Filtering the Return of Heterogeneous Object Types

Description and example of filtering the return of heterogeneous object types.

For finer control of the objects returned, 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 24-9 shows how you can use `SET_FILTER` to specify finer control on the objects returned. Connect as user `scott`. The password is `tiger`.

Example 24-9 Filtering the Return of Heterogeneous Object Types

1. Create a table, `my_metadata`, to store the retrieved objects. And create a procedure, `get_schema_md2`.

```
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;
/
```

2. Execute the procedure:

```
EXECUTE get_schema_md2;
```

3. Perform the following query to see what was retrieved:

```
SET LONG 9000000
SET PAGESIZE 0
SELECT * FROM my_metadata;
```

24.6 Using the DBMS_METADATA_DIFF API to Compare Object Metadata

Description and example that uses the retrieval, comparison, and submit interfaces of `DBMS_METADATA` and `DBMS_METADATA_DIFF` to fetch metadata for two tables, compare the metadata, and generate `ALTER` statements which make one table like the other.

For simplicity, function variants are used throughout the example.

Example 24-10 Comparing Object Metadata

1. Create two tables, `TAB1` and `TAB2`:

```
SQL> CREATE TABLE TAB1
  2     (    "EMPNO" NUMBER(4,0),
  3          "ENAME" VARCHAR2(10),
  4          "JOB" VARCHAR2(9),
  5          "DEPTNO" NUMBER(2,0)
  6     ) ;

Table created.

SQL> CREATE TABLE TAB2
  2     (    "EMPNO" NUMBER(4,0) PRIMARY KEY ENABLE,
  3          "ENAME" VARCHAR2(20),
  4          "MGR" NUMBER(4,0),
  5          "DEPTNO" NUMBER(2,0)
  6     ) ;

Table created.
```

   Note the differences between `TAB1` and `TAB2`:

   - The table names are different
   - `TAB2` has a primary key constraint; `TAB1` does not
   - The length of the `ENAME` column is different in each table
   - `TAB1` has a `JOB` column; `TAB2` does not
   - `TAB2` has a `MGR` column; `TAB1` does not

2. Create a function to return the table metadata in SXML format. The following are some key points to keep in mind about SXML when you are using the `DBMS_METADATA_DIFF` API:

   - SXML is an XML representation of object metadata.
   - The SXML returned is not the same as the XML returned by `DBMS_METADATA.GET_XML`, which is complex and opaque and contains binary values, instance-specific values, and so on.
   - SXML looks like a direct translation of SQL creation DDL into XML. The tag names and structure correspond to names in the *Oracle Database SQL Language Reference*.
   - SXML is designed to support editing and comparison.

   To keep this example simple, a transform parameter is used to suppress physical properties:

```
SQL> CREATE OR REPLACE FUNCTION get_table_sxml(name IN VARCHAR2) RETURN CLOB IS
  2   open_handle NUMBER;
  3   transform_handle NUMBER;
  4   doc CLOB;
  5  BEGIN
  6   open_handle := DBMS_METADATA.OPEN('TABLE');
  7   DBMS_METADATA.SET_FILTER(open_handle,'NAME',name);
  8   --
  9   -- Use the 'SXML' transform to convert XML to SXML
 10   --
 11   transform_handle := DBMS_METADATA.ADD_TRANSFORM(open_handle,'SXML');
 12   --
 13   -- Use this transform parameter to suppress physical properties
 14   --
 15   DBMS_METADATA.SET_TRANSFORM_PARAM(transform_handle,'PHYSICAL_PROPERTIES',
 16                                     FALSE);
 17   doc := DBMS_METADATA.FETCH_CLOB(open_handle);
 18   DBMS_METADATA.CLOSE(open_handle);
 19   RETURN doc;
 20  END;
 21  /

Function created.
```

3. Use the `get_table_sxml` function to fetch the table SXML for the two tables:

```
SQL> SELECT get_table_sxml('TAB1') FROM dual;

  <TABLE xmlns="http://xmlns.oracle.com/ku" version="1.0">
   <SCHEMA>SCOTT</SCHEMA>
   <NAME>TAB1</NAME>
   <RELATIONAL_TABLE>
      <COL_LIST>
         <COL_LIST_ITEM>
            <NAME>EMPNO</NAME>
            <DATATYPE>NUMBER</DATATYPE>
            <PRECISION>4</PRECISION>
            <SCALE>0</SCALE>
         </COL_LIST_ITEM>
         <COL_LIST_ITEM>
            <NAME>ENAME</NAME>
            <DATATYPE>VARCHAR2</DATATYPE>
            <LENGTH>10</LENGTH>
         </COL_LIST_ITEM>
         <COL_LIST_ITEM>
            <NAME>JOB</NAME>
            <DATATYPE>VARCHAR2</DATATYPE>
            <LENGTH>9</LENGTH>
         </COL_LIST_ITEM>
         <COL_LIST_ITEM>
            <NAME>DEPTNO</NAME>
            <DATATYPE>NUMBER</DATATYPE>
            <PRECISION>2</PRECISION>
            <SCALE>0</SCALE>
         </COL_LIST_ITEM>
      </COL_LIST>
   </RELATIONAL_TABLE>
</TABLE>

1 row selected.

SQL> SELECT get_table_sxml('TAB2') FROM dual;
```

```
<TABLE xmlns="http://xmlns.oracle.com/ku" version="1.0">
 <SCHEMA>SCOTT</SCHEMA>
 <NAME>TAB2</NAME>
 <RELATIONAL_TABLE>
    <COL_LIST>
       <COL_LIST_ITEM>
          <NAME>EMPNO</NAME>
          <DATATYPE>NUMBER</DATATYPE>
          <PRECISION>4</PRECISION>
          <SCALE>0</SCALE>
       </COL_LIST_ITEM>
       <COL_LIST_ITEM>
          <NAME>ENAME</NAME>
          <DATATYPE>VARCHAR2</DATATYPE>
          <LENGTH>20</LENGTH>
       </COL_LIST_ITEM>
       <COL_LIST_ITEM>
          <NAME>MGR</NAME>
          <DATATYPE>NUMBER</DATATYPE>
          <PRECISION>4</PRECISION>
          <SCALE>0</SCALE>
       </COL_LIST_ITEM>
       <COL_LIST_ITEM>
          <NAME>DEPTNO</NAME>
          <DATATYPE>NUMBER</DATATYPE>
          <PRECISION>2</PRECISION>
          <SCALE>0</SCALE>
       </COL_LIST_ITEM>
    </COL_LIST>
    <PRIMARY_KEY_CONSTRAINT_LIST>
       <PRIMARY_KEY_CONSTRAINT_LIST_ITEM>
          <COL_LIST>
             <COL_LIST_ITEM>
                <NAME>EMPNO</NAME>
             </COL_LIST_ITEM>
          </COL_LIST>
       </PRIMARY_KEY_CONSTRAINT_LIST_ITEM>
    </PRIMARY_KEY_CONSTRAINT_LIST>
 </RELATIONAL_TABLE>
</TABLE>

1 row selected.
```

4. Compare the results using the `DBMS_METADATA` browsing APIs:

```
SQL> SELECT dbms_metadata.get_sxml('TABLE','TAB1') FROM dual;
SQL> SELECT dbms_metadata.get_sxml('TABLE','TAB2') FROM dual;
```

5. Create a function using the `DBMS_METADATA_DIFF` API to compare the metadata for the two tables. In this function, the `get_table_sxml` function that was just defined in step 2 is used.

```
SQL> CREATE OR REPLACE FUNCTION compare_table_sxml(name1 IN VARCHAR2,
  2                                                name2 IN VARCHAR2) RETURN CLOB IS
  3   doc1 CLOB;
  4   doc2 CLOB;
  5   diffdoc CLOB;
  6   openc_handle NUMBER;
  7  BEGIN
  8   --
  9   -- Fetch the SXML for the two tables
 10   --
```

```
 11   doc1 := get_table_sxml(name1);
 12   doc2 := get_table_sxml(name2);
 13   --
 14   -- Specify the object type in the OPENC call
 15   --
 16   openc_handle := DBMS_METADATA_DIFF.OPENC('TABLE');
 17   --
 18   -- Add each document
 19   --
 20   DBMS_METADATA_DIFF.ADD_DOCUMENT(openc_handle,doc1);
 21   DBMS_METADATA_DIFF.ADD_DOCUMENT(openc_handle,doc2);
 22   --
 23   -- Fetch the SXML difference document
 24   --
 25   diffdoc := DBMS_METADATA_DIFF.FETCH_CLOB(openc_handle);
 26   DBMS_METADATA_DIFF.CLOSE(openc_handle);
 27   RETURN diffdoc;
 28  END;
 29  /

Function created.
```

6. Use the function to fetch the SXML difference document for the two tables:

```
SQL> SELECT compare_table_sxml('TAB1','TAB2') FROM dual;

<TABLE xmlns="http://xmlns.oracle.com/ku" version="1.0">
  <SCHEMA>SCOTT</SCHEMA>
  <NAME value1="TAB1">TAB2</NAME>
  <RELATIONAL_TABLE>
    <COL_LIST>
      <COL_LIST_ITEM>
        <NAME>EMPNO</NAME>
        <DATATYPE>NUMBER</DATATYPE>
        <PRECISION>4</PRECISION>
        <SCALE>0</SCALE>
      </COL_LIST_ITEM>
      <COL_LIST_ITEM>
        <NAME>ENAME</NAME>
        <DATATYPE>VARCHAR2</DATATYPE>
        <LENGTH value1="10">20</LENGTH>
      </COL_LIST_ITEM>
      <COL_LIST_ITEM src="1">
        <NAME>JOB</NAME>
        <DATATYPE>VARCHAR2</DATATYPE>
        <LENGTH>9</LENGTH>
      </COL_LIST_ITEM>
      <COL_LIST_ITEM>
        <NAME>DEPTNO</NAME>
        <DATATYPE>NUMBER</DATATYPE>
        <PRECISION>2</PRECISION>
        <SCALE>0</SCALE>
      </COL_LIST_ITEM>
      <COL_LIST_ITEM src="2">
        <NAME>MGR</NAME>
        <DATATYPE>NUMBER</DATATYPE>
        <PRECISION>4</PRECISION>
        <SCALE>0</SCALE>
      </COL_LIST_ITEM>
    </COL_LIST>
    <PRIMARY_KEY_CONSTRAINT_LIST src="2">
      <PRIMARY_KEY_CONSTRAINT_LIST_ITEM>
        <COL_LIST>
```

```
          <COL_LIST_ITEM>
            <NAME>EMPNO</NAME>
          </COL_LIST_ITEM>
        </COL_LIST>
      </PRIMARY_KEY_CONSTRAINT_LIST_ITEM>
    </PRIMARY_KEY_CONSTRAINT_LIST>
  </RELATIONAL_TABLE>
</TABLE>

1 row selected.
```

The SXML difference document shows the union of the two SXML documents, with the XML attributes `value1` and `src` identifying the differences. When an element exists in only one document it is marked with `src`. Thus, `<COL_LIST_ITEM src="1">` means that this element is in the first document (`TAB1`) but not in the second. When an element is present in both documents but with different values, the element's value is the value in the second document and the `value1` gives its value in the first. For example, `<LENGTH value1="10">20</LENGTH>` means that the length is 10 in `TAB1` (the first document) and 20 in `TAB2`.

7. Compare the result using the `DBMS_METADATA_DIFF` browsing APIs:

```
SQL> SELECT dbms_metadata_diff.compare_sxml('TABLE','TAB1','TAB2') FROM dual;
```

8. Create a function using the `DBMS_METADATA.CONVERT` API to generate an ALTERXML document. This is an XML document containing `ALTER` statements to make one object like another. You can also use parse items to get information about the individual `ALTER` statements. (This example uses the functions defined thus far.)

```
SQL> CREATE OR REPLACE FUNCTION get_table_alterxml(name1 IN VARCHAR2,
  2                                                name2 IN VARCHAR2) RETURN CLOB IS
  3   diffdoc CLOB;
  4   openw_handle NUMBER;
  5   transform_handle NUMBER;
  6   alterxml CLOB;
  7  BEGIN
  8   --
  9   -- Use the function just defined to get the difference document
 10   --
 11   diffdoc := compare_table_sxml(name1,name2);
 12   --
 13   -- Specify the object type in the OPENW call
 14   --
 15   openw_handle := DBMS_METADATA.OPENW('TABLE');
 16   --
 17   -- Use the ALTERXML transform to generate the ALTER_XML document
 18   --
 19   transform_handle := DBMS_METADATA.ADD_TRANSFORM(openw_handle,'ALTERXML');
 20   --
 21   -- Request parse items
 22   --
 23   DBMS_METADATA.SET_PARSE_ITEM(openw_handle,'CLAUSE_TYPE');
 24   DBMS_METADATA.SET_PARSE_ITEM(openw_handle,'NAME');
 25   DBMS_METADATA.SET_PARSE_ITEM(openw_handle,'COLUMN_ATTRIBUTE');
 26   --
 27   -- Create a temporary LOB
 28   --
 29   DBMS_LOB.CREATETEMPORARY(alterxml, TRUE );
 30   --
 31   -- Call CONVERT to do the transform
 32   --
```

```
 33   DBMS_METADATA.CONVERT(openw_handle,diffdoc,alterxml);
 34   --
 35   -- Close context and return the result
 36   --
 37   DBMS_METADATA.CLOSE(openw_handle);
 38   RETURN alterxml;
 39  END;
 40  /

Function created.
```

9. Use the function to fetch the ALTER_XML document:

```
SQL> SELECT get_table_alterxml('TAB1','TAB2') FROM dual;

<ALTER_XML xmlns="http://xmlns.oracle.com/ku" version="1.0">
   <OBJECT_TYPE>TABLE</OBJECT_TYPE>
   <OBJECT1>
      <SCHEMA>SCOTT</SCHEMA>
      <NAME>TAB1</NAME>
   </OBJECT1>
   <OBJECT2>
      <SCHEMA>SCOTT</SCHEMA>
      <NAME>TAB2</NAME>
   </OBJECT2>
   <ALTER_LIST>
      <ALTER_LIST_ITEM>
         <PARSE_LIST>
            <PARSE_LIST_ITEM>
               <ITEM>NAME</ITEM>
               <VALUE>MGR</VALUE>
            </PARSE_LIST_ITEM>
            <PARSE_LIST_ITEM>
               <ITEM>CLAUSE_TYPE</ITEM>
               <VALUE>ADD_COLUMN</VALUE>
            </PARSE_LIST_ITEM>
         </PARSE_LIST>
         <SQL_LIST>
            <SQL_LIST_ITEM>
               <TEXT>ALTER TABLE "SCOTT"."TAB1" ADD ("MGR" NUMBER(4,0))</TEXT>
            </SQL_LIST_ITEM>
         </SQL_LIST>
      </ALTER_LIST_ITEM>
      <ALTER_LIST_ITEM>
         <PARSE_LIST>
            <PARSE_LIST_ITEM>
               <ITEM>NAME</ITEM>
               <VALUE>JOB</VALUE>
            </PARSE_LIST_ITEM>
            <PARSE_LIST_ITEM>
               <ITEM>CLAUSE_TYPE</ITEM>
               <VALUE>DROP_COLUMN</VALUE>
            </PARSE_LIST_ITEM>
         </PARSE_LIST>
         <SQL_LIST>
            <SQL_LIST_ITEM>
               <TEXT>ALTER TABLE "SCOTT"."TAB1" DROP ("JOB")</TEXT>
            </SQL_LIST_ITEM>
         </SQL_LIST>
      </ALTER_LIST_ITEM>
      <ALTER_LIST_ITEM>
         <PARSE_LIST>
            <PARSE_LIST_ITEM>
```

```
                  <ITEM>NAME</ITEM>
                  <VALUE>ENAME</VALUE>
               </PARSE_LIST_ITEM>
               <PARSE_LIST_ITEM>
                  <ITEM>CLAUSE_TYPE</ITEM>
                  <VALUE>MODIFY_COLUMN</VALUE>
               </PARSE_LIST_ITEM>
               <PARSE_LIST_ITEM>
                  <ITEM>COLUMN_ATTRIBUTE</ITEM>
                  <VALUE> SIZE_INCREASE</VALUE>
               </PARSE_LIST_ITEM>
            </PARSE_LIST>
            <SQL_LIST>
               <SQL_LIST_ITEM>
                  <TEXT>ALTER TABLE "SCOTT"."TAB1" MODIFY
                       ("ENAME" VARCHAR2(20))
                  </TEXT>
               </SQL_LIST_ITEM>
            </SQL_LIST>
         </ALTER_LIST_ITEM>
         <ALTER_LIST_ITEM>
            <PARSE_LIST>
               <PARSE_LIST_ITEM>
                  <ITEM>CLAUSE_TYPE</ITEM>
                  <VALUE>ADD_CONSTRAINT</VALUE>
               </PARSE_LIST_ITEM>
            </PARSE_LIST>
            <SQL_LIST>
               <SQL_LIST_ITEM>
                  <TEXT>ALTER TABLE "SCOTT"."TAB1" ADD  PRIMARY KEY
                        ("EMPNO") ENABLE
                  </TEXT>
               </SQL_LIST_ITEM>
            </SQL_LIST>
         </ALTER_LIST_ITEM>
         <ALTER_LIST_ITEM>
            <PARSE_LIST>
               <PARSE_LIST_ITEM>
                  <ITEM>NAME</ITEM>
                  <VALUE>TAB1</VALUE>
               </PARSE_LIST_ITEM>
               <PARSE_LIST_ITEM>
                  <ITEM>CLAUSE_TYPE</ITEM>
                  <VALUE>RENAME_TABLE</VALUE>
               </PARSE_LIST_ITEM>
            </PARSE_LIST>
            <SQL_LIST>
               <SQL_LIST_ITEM>
                  <TEXT>ALTER TABLE "SCOTT"."TAB1" RENAME TO "TAB2"</TEXT>
               </SQL_LIST_ITEM>
            </SQL_LIST>
         </ALTER_LIST_ITEM>
      </ALTER_LIST>
</ALTER_XML>


1 row selected.
```

10. Compare the result using the `DBMS_METADATA_DIFF` browsing API:

```
SQL> SELECT dbms_metadata_diff.compare_alter_xml('TABLE','TAB1','TAB2') FROM dual;
```

11. The ALTER_XML document contains an ALTER_LIST of each of the alters. Each ALTER_LIST_ITEM has a PARSE_LIST containing the parse items as name-value pairs and a SQL_LIST containing the SQL for the particular alter. You can parse this document and decide which of the SQL statements to execute, using the information in the PARSE_LIST. (Note, for example, that in this case one of the alters is a DROP_COLUMN, and you might choose not to execute that.)

12. Create one last function that uses the `DBMS_METADATA.CONVERT` API and the `ALTER DDL` transform to convert the ALTER_XML document into SQL DDL:

```
SQL> CREATE OR REPLACE FUNCTION get_table_alterddl(name1 IN VARCHAR2,
  2                                                name2 IN VARCHAR2) RETURN CLOB IS
  3   alterxml CLOB;
  4   openw_handle NUMBER;
  5   transform_handle NUMBER;
  6   alterddl CLOB;
  7  BEGIN
  8   --
  9   -- Use the function just defined to get the ALTER_XML document
 10   --
 11   alterxml := get_table_alterxml(name1,name2);
 12   --
 13   -- Specify the object type in the OPENW call
 14   --
 15   openw_handle := DBMS_METADATA.OPENW('TABLE');
 16   --
 17   -- Use ALTERDDL transform to convert the ALTER_XML document to SQL DDL
 18   --
 19   transform_handle := DBMS_METADATA.ADD_TRANSFORM(openw_handle,'ALTERDDL');
 20   --
 21   -- Use the SQLTERMINATOR transform parameter to append a terminator
 22   -- to each SQL statement
 23   --
 24   DBMS_METADATA.SET_TRANSFORM_PARAM(transform_handle,'SQLTERMINATOR',true);
 25   --
 26   -- Create a temporary lob
 27   --
 28   DBMS_LOB.CREATETEMPORARY(alterddl, TRUE );
 29   --
 30   -- Call CONVERT to do the transform
 31   --
 32   DBMS_METADATA.CONVERT(openw_handle,alterxml,alterddl);
 33   --
 34   -- Close context and return the result
 35   --
 36   DBMS_METADATA.CLOSE(openw_handle);
 37   RETURN alterddl;
 38  END;
 39  /

Function created.
```

13. Use the function to fetch the SQL ALTER statements:

```
SQL> SELECT get_table_alterddl('TAB1','TAB2') FROM dual;
ALTER TABLE "SCOTT"."TAB1" ADD ("MGR" NUMBER(4,0))
/
  ALTER TABLE "SCOTT"."TAB1" DROP ("JOB")
/
  ALTER TABLE "SCOTT"."TAB1" MODIFY ("ENAME" VARCHAR2(20))
/
  ALTER TABLE "SCOTT"."TAB1" ADD  PRIMARY KEY ("EMPNO") ENABLE
```

```
/
  ALTER TABLE "SCOTT"."TAB1" RENAME TO "TAB2"
/

1 row selected.
```

14. Compare the results using the `DBMS_METADATA_DIFF` browsing API:

```
SQL> SELECT dbms_metadata_diff.compare_alter('TABLE','TAB1','TAB2') FROM dual;
ALTER TABLE "SCOTT"."TAB1" ADD ("MGR" NUMBER(4,0))
  ALTER TABLE "SCOTT"."TAB1" DROP ("JOB")
  ALTER TABLE "SCOTT"."TAB1" MODIFY ("ENAME" VARCHAR2(20))
  ALTER TABLE "SCOTT"."TAB1" ADD  PRIMARY KEY ("EMPNO") USING INDEX
  PCTFREE 10 INITRANS 2 STORAGE ( INITIAL 16384 NEXT 16384 MINEXTENTS 1
  MAXEXTENTS 505 PCTINCREASE 50 FREELISTS 1 FREELIST GROUPS 1 BUFFER_POOL
  DEFAULT)  ENABLE ALTER TABLE "SCOTT"."TAB1" RENAME TO "TAB2"

1 row selected.
```

24.7 Performance Tips for the Programmatic Interface of the DBMS_METADATA API

Describes how to enhance performance when using the programmatic interface of the `DBMS_METADATA` API.

Specifically:

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 DBMS_METADATA API 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 `DBMS_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.

24.8 Example Usage of the DBMS_METADATA API

Example of how the `DBMS_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 and any associated indexes, triggers, and grants.
- Creates a package, `PAYROLL_DEMO`, that uses the `DBMS_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 user `system`. You will be prompted for a password.

   ```
   sqlplus system
   ```

2. Install the demo, which is located in the file `mddemo.sql` in `rdbms/demo`:

   ```
   SQL> @mddemo
   ```

   For an explanation of what happens during this step, see What Does the DBMS_METADATA Example Do?.

3. Connect as user `mddemo`. You will be prompted for a password, which is also `mddemo`.

   ```
   SQL> CONNECT mddemo
   Enter password:
   ```

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 for a listing of the actual output.

- What Does the DBMS_METADATA Example Do?
  Explanation of the `DBMS_METADATA` example.
- Output Generated from the GET_PAYROLL_TABLES Procedure
  Explanation of the output generated from the `GET_PAYROLL_TABLES` procedure.

24.8.1 What Does the DBMS_METADATA Example Do?

Explanation of the `DBMS_METADATA` example.

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, then a message to that effect is displayed, no harm is done, and the demo continues to execute.

   ```
   CONNECT system
   Enter password: password
   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. Connect to SQL*Plus as `mddemo` (the password is also `mddemo`):

```
CONNECT mddemo
Enter password:
```

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 `mddemo.sql` located in your $`ORACLE_HOME/rdbms/demo` directory.

24.8.2 Output Generated from the GET_PAYROLL_TABLES Procedure

Explanation of the 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" ;
```

24.9 Summary of DBMS_METADATA Procedures

Provides brief descriptions of the procedures provided by the `DBMS_METADATA` API.

For detailed descriptions of these procedures, see *Oracle Database PL/SQL Packages and Types Reference.*

The following table provides a brief description of the procedures provided by the `DBMS_METADATA` programmatic interface for retrieving multiple objects.

Table 24-1 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. You can query the `DBMS_METADATA_PARSE_ITEMS` to see all valid parse items.
`DBMS_METADATA.ADD_TRANSFORM()`	Specifies a transform that `FETCH_xxx` applies to the XML representation of the retrieved objects. You can query the `DBMS_METADATA_TRANSFORMS` view to see all valid Oracle-supplied transforms.
`DBMS_METADATA.SET_TRANSFORM_PARAM()`	Specifies parameters to the XSLT stylesheet identified by `transform_handle`. You can query the `DBMS_METADATA_TRANSFORM_PARAMS` view to see all the valid transform parameters for each transform.
`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.

The following table 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 24-2 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, then 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.

The following table provides a brief description of the `DBMS_METADATA` procedures and functions used for XML submission.

Table 24-3 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`.

24.10 Summary of DBMS_METADATA_DIFF Procedures

Provides brief descriptions of the procedures and functions provided by the `DBMS_METADATA_DIFF` API.

For detailed descriptions of these procedures, see *Oracle Database PL/SQL Packages and Types Reference.*

Table 24-4 DBMS_METADATA_DIFF Procedures and Functions

PL/SQL Procedure Name	Description
`OPENC` function	Specifies the type of objects to be compared.
`ADD_DOCUMENT` procedure	Specifies an SXML document to be compared.
`FETCH_CLOB` functions and procedures	Returns a CLOB showing the differences between the two documents specified by `ADD_DOCUMENT`.
`CLOSE` procedure	Invalidates the handle returned by `OPENC` and cleans up associated state.

25
Original Export

The original Export utility (`exp`) was used to write data from Oracle Database into an operating system file in binary format.

This file is stored outside the database, and it can be read into another Oracle Database by using the original Import utility.

> **Note:**
>
> Original Export is desupported for general use as of Oracle Database 11g. The only supported use of original Export in Oracle Database 11g and later releases is backward migration of `XMLType` data to Oracle Database 10g Release 2 (10.2) or earlier. Oracle strongly recommends that you use the new Oracle Data Pump Export and Import utilities. The only exception to this guidelines is in the following situations, which require original Export and Import:
>
> - You want to import files that were created using the original Export utility (`exp`).
> - You want to export files that must be imported using the original Import utility (`imp`). An example of this would be exporting data from Oracle Database 10g and then importing it into an earlier database release.

- Conventional Path Export Versus Direct Path Export
  Export provides two methods for exporting table data.
- Starting a Direct Path Export
  To use direct path Export, specify the `DIRECT=y` parameter on the command line or in the parameter file.
- Network Considerations for Original Oracle Data Pump Export
  When you use original Export (`exp`) across a network, review protocols and connection qualifier strings.
- Character Set and Globalization Support Considerations
  These sections describe the globalization support behavior of Export with respect to character set conversion of user data and data definition language (DDL).
- 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.
- Considerations When Exporting Database Objects
  These sections describe points that you should consider when you export particular database objects.
- Transportable Tablespaces
  The transportable tablespace feature enables you to move a set of tablespaces from one Oracle database to another.
- Exporting From a Read-Only Database
  Describes exporting from a read-only database.
- 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.
- Using Different Releases of Export and Import
  Describes compatibility issues that relate to using different releases of Export and the Oracle database.

25.1 What is the Export Utility?

The original Export utility (`exp`) provided a simple way for you to transfer data objects between Oracle Database instances, regardless of hardware or software configurations.

> **Note:**
>
> Original Export is desupported for general use as of Oracle Database 11g. The only supported use of original Export in Oracle Database 11g and later releases is backward migration of `XMLType` data to Oracle Database 10g Release 2 (10.2) or earlier. Oracle strongly recommends that you use the new Oracle Data Pump Export and Import utilities. The only exception to this guidelines is in the following situations, which require original Export and Import:
>
> - You want to import files that were created using the original Export utility (`exp`).
> - You want to export files that must be imported using the original Import utility (`imp`). An example of this would be exporting data from Oracle Database 10g and then importing it into an earlier database release.

When you run Export against an Oracle Database instance, objects (such as tables) are extracted, followed by their related objects (such as indexes, comments, and grants), if any.

An Export file is an Oracle binary-format dump file that is typically located on disk or tape. You can transfer the dump files by using FTP, or by physically transported (for example, 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, even if the databases reside on platforms with different hardware and software configurations. The files can also be used as backups, in addition to normal backup procedures.

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

You can also display the contents of an export file without actually performing an import. To display the contents, use the Import `SHOW` parameter. To load data from ASCII fixed-format or delimited files, use the SQL*Loader utility.

Related Topics

- SHOW
  Lists the contents of the export file before importing.

25.2 Before Using Export

Before you use the original Export (`exp`) utility, review the preparation checklist, and complete tasks as necessary on your server or Oracle Database instances.

- Preparation Checklist for Using Export
  Before you begin using the original Export utility (`exp`), perform these checks.
- Running catexp.sql or catalog.sql
  To use Export, you must run the script `catexp.sql` or `catalog.sql` (which runs `catexp.sql`) after the database has been created or migrated to a newer release.
- 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.
- Verifying Access Privileges for Export and Import Operations
  To use Export, you must have the `CREATE SESSION` privilege on an Oracle database.

25.2.1 Preparation Checklist for Using Export

Before you begin using the original Export utility (`exp`), perform these checks.

To avoid issues, ensure that each of the following conditions is true:

- If you created your database manually, ensure that the `catexp.sql` or `catalog.sql` script has been run. If you created your database using the Database Configuration Assistant (DBCA), it is not necessary to run these scripts.
- Ensure there is sufficient disk or tape storage to write the export file
- Verify that you have the required access privileges

25.2.2 Running catexp.sql or catalog.sql

To use Export, you must run the script `catexp.sql` or `catalog.sql` (which runs `catexp.sql`) after the database has been created or migrated to a newer release.

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` and `IMP_FULL_DATABASE` roles
- 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

The `EXP_FULL_DATABASE` and `IMP_FULL_DATABASE` roles are powerful. Database administrators should use caution when granting these roles to users.

25.2.3 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, then 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

25.2.4 Verifying Access Privileges for Export and Import Operations

To use Export, 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, then 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.

Several system schemas cannot be exported because they are not user schemas; they contain Oracle-managed data and metadata. Examples of schemas that are not exported include `SYS`, `ORDSYS`, and `MDSYS`.

25.3 Invoking Export

You can start Export and specify parameters by using one of three different methods.

Before you use one of these methods, be sure to read the descriptions of the available parameters. See Export Parameters .

- Invoking Export as SYSDBA
  `SYSDBA` is used internally and has specialized functions; its behavior is not the same as for generalized users.
- Command-Line Entries
  You can specify all valid parameters and their values from the command line.
- Parameter Files
  You can start the original Export utility and specify all valid Export parameters and their values in a parameter file.
- Interactive Mode
  If you prefer to be prompted for the value of each parameter, then specify `exp` at the command line.
- Getting Online Help
  Export provides online help. Enter `exp help=y` on the command line to display Export help.

25.3.1 Invoking Export as SYSDBA

`SYSDBA` is used internally and has specialized functions; its behavior is not the same as for generalized users.

Therefore, you should not typically need to start Export `as SYSDBA` except in the following situations:

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

25.3.2 Command-Line Entries

You can specify all valid parameters and their values from the command line.

Use the following syntax (you will be prompted for a username and password):

```
exp PARAMETER=value
```

or

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

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

25.3.3 Parameter Files

You can start the original Export utility and specify all valid Export parameters and their values in a parameter file.

Storing the parameters in a file allows them to be easily modified or reused. Oracle recommends that you use this method to start original Export. If you use different parameters for different databases, then 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:

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 started with the following line:

```
exp 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.

25.3.4 Interactive Mode

If you prefer to be prompted for the value of each parameter, then specify `exp` at the command line.

After you enter your username and password at the prompts, commonly used parameters are displayed.

You can accept the default parameter value, if one is provided, or enter a different 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, then Oracle recommends that you use the Oracle Enterprise Manager Export Wizard.

- Restrictions When Using Export's Interactive Method
  Describes restrictions when using the Export interactive method.

25.3.4.1 Restrictions When Using Export's Interactive Method

Describes restrictions when using the Export 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, then Export defaults to the exporter's schema or the schema containing the last table exported in the current session.

  For example, if `beth` is a privileged user exporting in table mode, then Export assumes that all tables are in 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," then the Export utility exits.

25.3.5 Getting Online Help

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

25.4 Export Modes

The Export utility supports four modes of operation.

Specifically:

- Full: Exports a full database. Only users with the `EXP_FULL_DATABASE` role 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 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.
- Table: Enables you to export specific tables and partitions. A privileged user can qualify the tables by specifying the schema that contains them. For any table for which a schema name is not specified, Export defaults to the exporter's schema name. Use the `TABLES` parameter to specify this mode.

See Table 25-1 for a list of objects that are exported and imported in each mode.

> **Note:**
>
> The original Export utility does not export any table that was created with deferred segment creation and has not had a segment created for it. The most common way for a segment to be created is to store a row into the table, though other operations such as `ALTER TABLE ALLOCATE EXTENTS` will also create a segment. If a segment does exist for the table and the table is exported, then the `SEGMENT CREATION DEFERRED` clause is not included in the `CREATE TABLE` statement that is executed by the original Import utility.

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.

Table 25-1 Objects Exported 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
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

Table 25-1 (Cont.) Objects Exported in Each Mode

Object	Table Mode	User Mode	Full Database Mode	Tablespace Mode
Operators	No	Yes	Yes	No
Password history	No	No	Yes	No
Postinstance actions and objects	No	No	Yes	No
Postschema procedural actions and objects	No	Yes	Yes	No
Posttable actions	Yes	Yes	Yes	Yes
Posttable procedural actions and objects	Yes	Yes	Yes	Yes
Preschema procedural objects and actions	No	Yes	Yes	No
Pretable actions	Yes	Yes	Yes	Yes
Pretable procedural actions	Yes	Yes	Yes	Yes
Private synonyms	No	Yes	Yes	No
Procedural objects	No	Yes	Yes	No
Profiles	No	No	Yes	No
Public synonyms	No	No	Yes	No
Referential integrity constraints	Yes	Yes	Yes	No
Refresh groups	No	Yes	Yes	No
Resource costs	No	No	Yes	No
Role grants	No	No	Yes	No
Roles	No	No	Yes	No
Rollback segment definitions	No	No	Yes	No
Security policies for table	Yes	Yes	Yes	Yes
Sequence numbers	No	Yes	Yes	No
Snapshot logs	No	Yes	Yes	No
Snapshots and materialized views	No	Yes	Yes	No
System privilege grants	No	No	Yes	No
Table constraints (primary, unique, check)	Yes	Yes	Yes	Yes
Table data	Yes	Yes	Yes	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

Table 25-1 (Cont.) Objects Exported in Each Mode

Object	Table Mode	User Mode	Full Database Mode	Tablespace Mode
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.

25.4.1 Table-Level and Partition-Level Export

You can export tables, partitions, and subpartitions.

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

- Table-Level Export
  Exports all data from the specified tables.
- Partition-Level Export
  Exports only data from the specified source partitions or subpartitions.

25.4.1.1 Table-Level Export

Exports all data from the specified tables.

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, then 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.

25.4.1.2 Partition-Level Export

Exports only data from the specified source partitions or subpartitions.

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.

25.5 Export Parameters

Describes the Export command-line parameters.

- BUFFER
- COMPRESS
- CONSISTENT
- CONSTRAINTS
- DIRECT
  The `DIRECT` parameter for the Export utility specifies the use of direct path Export.
- FEEDBACK
  The `FEEDBACK` Export utility parameter specifies that Export should display a progress meter in the form of a period for *n* number of rows exported.
- FILE
  The `FILE` Export utility parameter specifies the names of the export dump files.
- FILESIZE
  The `FILESIZE` Export utility parameter specifies the size of the dump file.
- FLASHBACK_SCN
  The FLASHBACK_SCN Export utility parameter specifies the system change number (SCN) that Export is going to use to enable flashback.
- FLASHBACK_TIME
  The `FLASHBACK_TIME` Export utility parameter enables you to specify a timestamp. Export finds the SCN that most closely matches the specified timestamp. This SCN is used to enable flashback.
- FULL
  The `FULL` Export parameter indicates that the export is a full database mode export (that is, it exports the entire database).
- GRANTS
  The GRANTS Export utility parameter specifies whether the Export utility exports object grants.
- HELP
  The `HELP` parameter of Export utility displays a description of the Export parameters.
- INDEXES
  `INDEXES` Export parameter specifies whether the Export utility exports indexes.
- LOG
  Specifies a file name (for example, export.log) to receive informational and error messages.
- OBJECT_CONSISTENT
  Specifies whether 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.
- OWNER
  Indicates that the export is a user-mode export and lists the users whose objects will be exported.
- PARFILE
  Specifies a file name for a file that contains a list of Export parameters.

- QUERY
  This parameter enables you to select a subset of rows from a set of tables when doing a table mode export.
- RECORDLENGTH
  Specifies the length, in bytes, of the file record.
- RESUMABLE
  The `RESUMABLE` parameter is used to enable and disable resumable space allocation.
- RESUMABLE_NAME
  The value for the `RESUMABLE_NAME` parameter identifies the statement that is resumable.
- RESUMABLE_TIMEOUT
  The value of the `RESUMABLE_TIMEOUT` parameter specifies the time period during which an error must be fixed.
- ROWS
  Specifies whether the rows of table data are exported.
- STATISTICS
  Specifies the type of database optimizer statistics to generate when the exported data is imported. Options are `ESTIMATE`, `COMPUTE`, and `NONE`.
- TABLES
  The original Export command-line mode TABLES parameter specifies that the export is a table-mode export, and lists the table names and partition and subpartition names that you specified for export.
- TABLESPACES
  The `TABLESPACES` parameter specifies that all tables in the specified tablespace be exported to the Export dump file.
- TRANSPORT_TABLESPACE
  When specified as `y`, this parameter enables the export of transportable tablespace metadata.
- TRIGGERS
  Specifies whether the Export utility exports triggers.
- TTS_FULL_CHECK
  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)
  Specifies the username, password, and optional connect string of the user performing the export.
- VOLSIZE
  Specifies the maximum number of bytes in an export file on each volume of tape.

25.5.1 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, then 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

25.5.1.1 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.

25.5.2 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), then the allocated space will be larger than the space required to hold the data.

If you specify `COMPRESS=n`, then 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.

The `COMPRESS` parameter does not work with bitmapped tablespaces.

> **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, then 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.

25.5.3 CONSISTENT

Default: `n`

Specifies whether 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`, then each table is usually exported in a single transaction. However, if a table contains nested tables, then the outer table and each inner table are exported as separate transactions. If a table is partitioned, then each partition is exported as a separate transaction.

Therefore, if nested tables and partitioned tables are being updated by other applications, then 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 25-2 shows a sequence of events by two users: `user1` exports partitions in a table and `user2` updates data in that table.

Table 25-2 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
3	Ends export of TAB:P1	No activity
4	Exports TAB:P2	No activity

If the export uses `CONSISTENT=y`, then none of the updates by `user2` are written to the export file.

If the export uses `CONSISTENT=n`, then 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, then 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, then 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"

25.5.4 CONSTRAINTS

Default: y

Specifies whether the Export utility exports table constraints.

25.5.5 DIRECT

The DIRECT parameter for the Export utility specifies the use of direct path Export.

Default

n

Syntax and Procedure

```
DIRECT=[Y|N]
```

Specifying `DIRECT=y` causes Export to extract data by reading the data directly, bypassing the SQL command-processing layer (evaluating buffer). This method can be much faster than a conventional path Export.

There are security and performance considerations for Export procedures. Review the relevant topic.

Related Topics

- Starting a Direct Path Export
  To use direct path Export, specify the `DIRECT=y` parameter on the command line or in the parameter file.

25.5.6 FEEDBACK

The `FEEDBACK` Export utility parameter specifies that Export should display a progress meter in the form of a period for *n* number of rows exported.

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`, then 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.

25.5.7 FILE

The `FILE` Export utility parameter specifies the names of the export dump files.

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 file names to be used. For example:

```
exp scott 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 file names to complete the export, then Export prompts you to provide additional file names.

25.5.8 FILESIZE

The `FILESIZE` Export utility parameter specifies the size of the dump file.

Default: Data is written to one file until the maximum size, as specified in Table 25-3, is reached.

Export supports writing to multiple export files, and Import can read from multiple export files. If you specify a value (byte limit) for the FILESIZE parameter, then 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 or, if it has used all the names specified in the FILE parameter, then it prompts you to provide a new export file name. 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, then 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 25-3 shows that the maximum size for dump files depends on the operating system you are using and on the release of the Oracle database that you are using.

Table 25-3 Maximum Size for Dump Files

Operating System	Release of Oracle Database	Maximum Size
Any	Before 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).

25.5.9 FLASHBACK_SCN

The FLASHBACK_SCN Export utility parameter specifies the system change number (SCN) that Export is going to use to enable flashback.

Default: none

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

> **See Also:**
>
> - *Oracle Database Backup and Recovery User's Guide* for more information about performing flashback recovery

The following is an example of specifying an SCN. When the export is performed, the data will be consistent as of SCN `3482971`.

```
> exp FILE=exp.dmp FLASHBACK_SCN=3482971
```

25.5.10 FLASHBACK_TIME

The `FLASHBACK_TIME` Export utility parameter enables you to specify a timestamp. Export finds the SCN that most closely matches the specified timestamp. This SCN is used to enable flashback.

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 FILE=exp.dmp FLASHBACK_TIME="TIMESTAMP '2006-05-01 11:00:00'"

> exp FILE=exp.dmp FLASHBACK_TIME="TO_TIMESTAMP('12-02-2005 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 FILE=exp.dmp FLASHBACK_TIME="'2006-05-01 11:00:00'"
```

> **See Also:**
>
> - *Oracle Database Backup and Recovery User's Guide* for more information about performing flashback recovery
> - *Oracle Database PL/SQL Packages and Types Reference* for more information about the `DBMS_FLASHBACK` PL/SQL package

25.5.11 FULL

The `FULL` Export parameter indicates that the export is a full database mode export (that is, it exports the entire database).

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.

25.5.11.1 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.
- A full export also does not export the default profile. If you have modified the default profile in the source database (for example, by adding a password verification function owned by schema `SYS`), then you must manually pre-create the function and modify the default profile in the target database after the import completes.
- 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 data files
  - 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, then 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 data file names as the exported database. This can result in problems in the following situations:
  - If the data files belong to any other database, then they will become corrupted. This is especially true if the exported database is on the same system, because its data files will be reused by the database into which you are importing.
  - If the data files have names that conflict with existing operating system files.

25.5.12 GRANTS

The GRANTS Export utility parameter specifies whether the Export utility exports object grants.

Default: `y`

Specifies whether 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.

25.5.13 HELP

The `HELP` parameter of Export utility displays a description of the Export parameters.

Default: none

Displays a description of the Export parameters. Enter `exp help=y` on the command line to display the help content.

25.5.14 INDEXES

`INDEXES` Export parameter specifies whether the Export utility exports indexes.

Default: `y`

Specifies whether the Export utility exports indexes.

25.5.15 LOG

Specifies a file name (for example, export.log) to receive informational and error messages.

Default: none

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

25.5.16 OBJECT_CONSISTENT

Specifies whether 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.

Default: `n`

If `OBJECT_CONSISTENT` is set to `y`, then 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

25.5.17 OWNER

Indicates that the export is a user-mode export and lists the users whose objects will be exported.

Default: none

If the user initiating the export is the database administrator (DBA), then 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.

25.5.18 PARFILE

Specifies a file name for a file that contains a list of Export parameters.

Default: none

For more information about using a parameter file, see Invoking Export.

25.5.19 QUERY

This parameter enables you to select a subset of rows from a set of tables when doing a table mode export.

Default

None.

Purpose

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 `TABLES` parameter.

For example, if user `scott` wants to export only those employees whose job title is `SALESMAN` and whose salary is less than 1600, then he could do the following (this example is UNIX-based):

```
exp scott 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 string `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 TABLES parameter. For example, the following statement will unload rows in both emp and bonus that match the query:

```
exp scott 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, then an error message will be produced, and no rows will be exported for the offending table.

- Restrictions When Using the QUERY Parameter
  Describes restrictions when using the QUERY parameter.

25.5.19.1 Restrictions When Using the QUERY Parameter

Describes 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.

25.5.20 RECORDLENGTH

Specifies the length, in bytes, of the file record.

Default

Operating system-dependent.

Purpose

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, then 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.

25.5.21 RESUMABLE

The `RESUMABLE` parameter is used to enable and disable resumable space allocation.

Default

`n`

Purpose

Because this parameter is disabled by default, you must set `RESUMABLE=y` to use its associated parameters, `RESUMABLE_NAME` and `RESUMABLE_TIMEOUT`.

> **See Also:**
>
> *Oracle Database Administrator's Guide* for more information about resumable space allocation.

25.5.22 RESUMABLE_NAME

The value for the `RESUMABLE_NAME` parameter identifies the statement that is resumable.

Default

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

Purpose

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.

25.5.23 RESUMABLE_TIMEOUT

The value of the `RESUMABLE_TIMEOUT` parameter specifies the time period during which an error must be fixed.

Default

`7200` seconds (2 hours)

Purpose

If the error is not fixed within the timeout period, then execution of the statement is terminated.

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

25.5.24 ROWS

Specifies whether the rows of table data are exported.

Default

`y`

25.5.25 STATISTICS

Specifies the type of database optimizer statistics to generate when the exported data is imported. Options are `ESTIMATE`, `COMPUTE`, and `NONE`.

Default: `ESTIMATE`

In some cases, Export will place the precalculated statistics in the export file, and also 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.

25.5.26 TABLES

The original Export command-line mode TABLES parameter specifies that the export is a table-mode export, and lists the table names and partition and subpartition names that you specified for export.

Default

None.

Purpose

Specifies that the export is a table-mode export and lists the table names and partition and subpartition names that you specified to export.

Syntax

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

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

- *schemaname* specifies the name of the user's schema from which to export the table or partition. If a schema name is not specified, then the exporter's schema is used as the default. System schema names such as ORDSYS, MDSYS, CTXSYS, LBACSYS, and ORDPLUGINS are reserved by Export.
- *tablename* specifies the name of the table or tables that you want to export. 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, then all of 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 of 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 enables you to export one or more specified partitions or subpartitions within a table.

Usage Notes

If you use *tablename*:*partition_name*, then 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, then the *partition_name* is ignored and the entire table is exported.

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, then 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, then 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, then 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.

- Table Name Restrictions
  This is an explanation of table name restrictions for Export utility.

Related Topics

- Example Export Session Using Partition-Level Export

25.5.26.1 Table Name Restrictions

This is an explanation of table name restrictions for Export utility.

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, then 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, then 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, then 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.

25.5.27 TABLESPACES

The `TABLESPACES` parameter specifies that all tables in the specified tablespace be exported to the Export dump file.

Default: none

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.

25.5.28 TRANSPORT_TABLESPACE

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

Default: `n`

Encrypted columns are not supported in transportable tablespace mode.

> **Note:**
>
> You cannot export transportable tablespaces and then import them into a database at a lower release level. The target database must be at the same or later release level as the source database.

> **See Also:**
>
> - Transportable Tablespaces
> - *Oracle Database Administrator's Guide* for more information about transportable tablespaces

25.5.29 TRIGGERS

Specifies whether the Export utility exports triggers.

Default: `y`

25.5.30 TTS_FULL_CHECK

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.

Default: `n`

25.5.31 USERID (username/password)

Specifies the username, password, and optional connect string of the user performing the export.

Default: none

If you omit the password, then Export will prompt you for it.

If you connect as user `SYS`, then 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:**
>
> The user's guide for your Oracle Net protocol for information about specifying a connect string for Oracle Net.

25.5.32 VOLSIZE

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

Default: none

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`).

25.6 Example Export Sessions

Examples of the types of Export sessions.

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

25.6.1 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 PARFILE=params.dat
```

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

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

Command-Line Method

```
> exp 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.

25.6.2 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 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 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.
```

25.6.3 Example Export Sessions in Table Mode

In table mode, you can export table data or the table definitions. (If no rows are exported, then 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, then Export defaults to the exporter's schema name. In the following example, Export defaults to the `SYSTEM` schema for table `a` and table `c`:

```
> exp 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
- Example 2: User Exports Tables That He Owns
- Example 3: Using Pattern Matching to Export Various Tables

25.6.3.1 Example 1: DBA Exporting Tables for Two Users

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

Parameter File Method

```
> exp 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 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.
```

25.6.3.2 Example 2: User Exports Tables That He Owns

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

Parameter File Method

```
> exp blake 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 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.
```

25.6.3.3 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 PARFILE=params.dat
```

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

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

Command-Line Method

```
> exp 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.
```

25.6.4 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
- Example 2: Exporting a Table with a Specified Partition
- Example 3: Exporting a Composite Partition

25.6.4.1 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, then all of the partitions are exported.

Parameter File Method

```
> exp scott PARFILE=params.dat
```

The `params.dat` file contains the following:

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

Command-Line Method

```
> exp scott 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.
```

25.6.4.2 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, then only the specified partition is exported.

Parameter File Method

```
 > exp scott PARFILE=params.dat
```

The `params.dat` file contains the following:

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

Command-Line Method

```
> exp scott 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.
```

25.6.4.3 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`, then all its subpartitions (`sp1` and `sp2`) will be exported. If you export the table and specify a subpartition (`sp4`), then only the specified subpartition is exported.

Parameter File Method

```
> exp scott PARFILE=params.dat
```

The `params.dat` file contains the following:

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

Command-Line Method

```
> exp scott 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.
```

25.7 Warning, Error, and Completion Messages

These sections describes the different types of messages issued by Export and how to save them in a log file.

- Log File
  You can capture all Export messages in a log file, either by using the `LOG` parameter or, for those systems that permit it, by redirecting the output to a file.
- Warning Messages
  Export does not terminate after recoverable errors. These recoverable errors are known as warnings.
- Nonrecoverable Error Messages
  Some errors are nonrecoverable and terminate the Export session.

- Completion Messages
  When an export completes without errors, a message to that effect is displayed.

25.7.1 Log File

You can capture all Export 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 any errors that may have occurred.

25.7.2 Warning Messages

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

For example, if an error occurs while exporting a table, then Export displays (or logs) an error message, skips to the next table, and continues processing.

Export also issues warnings when invalid objects are encountered.

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

25.7.3 Nonrecoverable Error Messages

Some errors are nonrecoverable and terminate the Export session.

These errors typically occur because of an internal problem or because a resource, such as memory, is not available or has been exhausted. For example, if the `catexp.sql` script is not executed, then Export issues the following nonrecoverable error message:

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

25.7.4 Completion Messages

When an export 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, then a message similar to the following is displayed:

```
Export terminated successfully with warnings
```

If a nonrecoverable error occurs, then the job terminates immediately and displays a message stating so, for example:

```
Export terminated unsuccessfully
```

25.8 Exit Codes for Inspection and Display

Export provides 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 25-4 shows the exit codes that get returned for various results.

Table 25-4 Exit Codes for Export

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

For UNIX, the exit codes are as follows:

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

25.9 Conventional Path Export Versus Direct Path Export

Export provides two methods for exporting table data.

Specifically:

- 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.

25.10 Starting 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. Review the security, performance, and restrictions for direct path Export operations:

- Security Considerations for Direct Path Exports
  Oracle Virtual Private Database (VPD) and Oracle Label Security are not enforced during direct path Exports.

- Performance Considerations for Direct Path Exports
  Improving performance by increasing the value of the `RECORDLENGTH` parameter when you start a direct path Export.
- Restrictions for Direct Path Exports
  Restrictions for using direct path mode.

25.10.1 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
- *Oracle Label Security Administrator's Guide*
- *Oracle Database Security Guide* for more information about using VPD to control data access

25.10.2 Performance Considerations for Direct Path Exports

Improving performance by increasing the value of the `RECORDLENGTH` parameter when you start 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.

25.10.3 Restrictions for Direct Path Exports

Restrictions for using direct path mode.

Specifically :

- To start a direct path Export, you must use either the command-line method or a parameter file. You cannot start 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 equals the database character set. If `NLS_LANG` is not set or if it is different than the database character set, then a warning is displayed and the export is discontinued. The default value for the `NLS_LANG` environment variable is `AMERICAN_AMERICA.US7ASCII`.

25.11 Network Considerations for Original Oracle Data Pump Export

When you use original Export (`exp`) across a network, review protocols and connection qualifier strings.

- 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.
- Exporting with Oracle Net
  With Oracle Net, you can perform exports over a network.

25.11.1 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.

25.11.2 Exporting with Oracle Net

With Oracle Net, you can perform exports over a network.

For example, if you run Export locally, then you can write data from a remote Oracle database into a local export file.

To use Export with Oracle Net, include the connection qualifier string `@connect_string` when entering the username and password in the `exp` command. For the exact syntax of this clause, see the user's guide for your Oracle Net protocol.

> **See Also:**
>
> - *Oracle Database Net Services Administrator's Guide*

25.12 Character Set and Globalization Support Considerations

These sections describe the globalization support behavior of Export 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.)
- Data Definition Language (DDL)
  Up to three character set conversions may be required for data definition language (DDL) during an export/import operation.
- 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.
- 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.)

25.12.1 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, then 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.

25.12.1.1 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* for more information about character sets

25.12.2 Data Definition Language (DDL)

Up to three character set conversions may be required for data definition language (DDL) during an export/import operation.

Specifically:

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.

25.12.3 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.

25.12.4 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.

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

25.13 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* for more information about affinity

25.14 Considerations When Exporting Database Objects

These sections describe points that you should consider when you export particular database objects.

- Exporting Sequences
  If transactions continue to access sequence numbers during an export, then sequence numbers might be skipped.
- Exporting LONG and LOB Data Types
  Describes exporting `LONG` and `LOB` data types.
- Exporting Foreign Function Libraries
  Describes exporting foreign function libraries.
- Exporting Offline Locally-Managed Tablespaces
  Describes exporting offline locally-managed tablespaces.
- Exporting Directory Aliases
  Describes exporting directory alias definitions.

- Exporting BFILE Columns and Attributes
  Describes exporting `BFILE` columns and attributes.
- Exporting External Tables
  Describes exporting external tables.
- Exporting Object Type Definitions
  Describes exporting object type definitions.
- Exporting Nested Tables
  Describes exporting nested tables.
- Exporting Advanced Queue (AQ) Tables
  Describes exporting Advanced Queue (AQ) tables.
- Exporting Synonyms
  Describes exporting synonyms.
- Possible Export Errors Related to Java Synonyms
  Describes possible export errors related to Java synonyms.
- Support for Fine-Grained Access Control
  Describes support for fine-grained access control policies.

25.14.1 Exporting Sequences

If transactions continue to access sequence numbers during an export, then 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.

25.14.2 Exporting LONG and LOB Data Types

Describes exporting `LONG` and `LOB` data types.

On export, `LONG` data types 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.

25.14.3 Exporting Foreign Function Libraries

Describes 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.

25.14.4 Exporting Offline Locally-Managed Tablespaces

Describes exporting offline locally-managed tablespaces.

If the data you are exporting contains offline locally-managed tablespaces, then 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.

25.14.5 Exporting Directory Aliases

Describes exporting directory alias definitions.

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.

25.14.6 Exporting BFILE Columns and Attributes

Describes 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, then the database administrator (DBA) must move the directories containing the specified files to a new location where they can be accessed.

25.14.7 Exporting External Tables

Describes 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.

25.14.8 Exporting Object Type Definitions

Describes 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, then 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`, then the DBA should perform a user mode export of both `blake` and `scott` to preserve the type definitions needed by the table.

25.14.9 Exporting Nested Tables

Describes 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.

25.14.10 Exporting Advanced Queue (AQ) Tables

Describes 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 and the table definition is exported, the user is responsible for maintaining application-level data integrity when queue table data is imported.

> **See Also:**
>
> *Oracle Database Advanced Queuing User's Guide*

25.14.11 Exporting Synonyms

Describes 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, then 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.

25.14.12 Possible Export Errors Related to Java Synonyms

Describes 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, then 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, then ensure that both the `DBMS_JAVA` synonym and `DBMS_JAVA` package are created and valid before rerunning the export.

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

25.14.13 Support for Fine-Grained Access Control

Describes support for fine-grained access control policies.

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

- The user who imports from an export file containing such tables must have the appropriate privileges (specifically, the `EXECUTE` privilege on the `DBMS_RLS` package so that the tables' security policies can be reinstated). If a user without the correct privileges attempts to export a table with fine-grained access policies enabled, then only those rows that the exporter is privileged to read will be exported.
- If fine-grained access control is enabled on a `SELECT` statement, then conventional path Export may not export the entire table because fine-grained access may rewrite the query.
- Only user SYS, or a user with the `EXP_FULL_DATABASE` role enabled or who has been granted `EXEMPT ACCESS POLICY`, can perform direct path Exports on tables having fine-grained access control.

25.15 Transportable Tablespaces

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

You cannot export transportable tablespaces and then import them into a database at a lower release level. The target database must be at the same or later release level as the source database.

To move or copy a set of tablespaces, you must make the tablespaces read-only, copy the data files of these tablespaces, and use Export and Import to move the database information (metadata) stored in the data dictionary. Both the data files 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 data files 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 and TRANSPORT_TABLESPACE for more information about using these parameters during an export operation.

> **See Also:**
>
> - *Oracle Database Administrator's Guide* for details about managing transportable tablespaces

25.16 Exporting From a Read-Only Database

Describes 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.

25.17 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, then be aware of the following advantages and disadvantages.

- Advantages of Partitioning a Migration
  Describes the advantages of partitioning a migration.
- Disadvantages of Partitioning a Migration
  Describes the disadvantages of partitioning a migration.
- How to Use Export and Import to Partition a Database Migration
  Describes how to partition a database migration using Export and Import.

25.17.1 Advantages of Partitioning a Migration

Describes the 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.

25.17.2 Disadvantages of Partitioning a Migration

Describes the 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, then you may not have the required parent records when you import the table into the dependent schema.

25.17.3 How to Use Export and Import to Partition a Database Migration

Describes how to partition a database migration using Export and Import.

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 FILE=full FULL=y CONSTRAINTS=n TRIGGERS=n ROWS=n INDEXES=n`
   b. `imp FILE=full FULL=y`
2. For each schema*n* in the database, issue the following commands:
   a. `exp OWNER=schema`*n* `FILE=schema`*n*
   b. `imp 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.

25.18 Using Different Releases of Export and Import

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 release. For example, if you try to use the Import utility 9.2.0.7 to import into a 9.2.0.8 database, then you may encounter errors.
- The version of the Export utility must be equal to the release of either the source or target database, whichever is earlier.

  For example, to create an export file for an import into a later release database, use a version of the Export utility that equals 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 equals the release 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.

25.18.1 Restrictions When Using Different Releases of Export and Import

Describes restrictions that apply when you are using different releases of Export and Import.

Specifically:

- 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. 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 release of the Oracle database, categories of database objects that did not exist in the earlier release 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.

25.18.2 Examples of Using Different Releases of Export and Import

Shows examples of using different releases of Export and Import.

Table 25-5 shows some examples of which Export and Import releases to use when moving data between different releases of the Oracle database.

Table 25-5 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

Table 25-5 covers moving data only between the original Export and Import utilities. For Oracle Database 10*g* release 1 (10.1) or later, Oracle recommends the Data Pump Export and Import utilities in most cases because these utilities provide greatly enhanced performance compared to the original Export and Import utilities.

See Also:

Oracle Database Upgrade Guide for more information about exporting and importing data between different releases, including releases later than 10.1

26
Original Import

The original Import utility (`imp`) imports dump files that were created using the original Export utility (`exp`).

- What Is the Import Utility?
  The original Import utility (`imp`) read object definitions and table data from dump files created by the original Export utility (`exp`).
- Before Using Import
  Describes what you should do before using Import.
- Importing into Existing Tables
  These sections describe factors to consider when you import data into existing tables.
- 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.
- Invoking Import
  To start the original Import utility and specify parameters, use one of three different methods.
- Import Modes
  The Import utility supports four modes of operation.
- Import Parameters
  These sections contain descriptions of the Import command-line parameters.
- Example Import Sessions
  These sections give some examples of import sessions that show you how to use the parameter file and command-line methods.
- Exit Codes for Inspection and Display
  Import provides 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.
- Error Handling During an Import
  These sections describe errors that can occur when you import database objects.
- Table-Level and Partition-Level Import
  You can import tables, partitions, and subpartitions.
- Controlling Index Creation and Maintenance
  These sections describe the behavior of Import with respect to index creation and maintenance.
- Network Considerations for Using Oracle Net with Original Import
  To perform imports over a network, you can use the Oracle Data Pump original Import utility (`imp`) with Oracle Net.
- Character Set and Globalization Support Considerations
  These sections describe the globalization support behavior of Import with respect to character set conversion of user data and data definition language (DDL).

- Using Instance Affinity
  You can use instance affinity to associate jobs with instances in databases you plan to export and import.
- Considerations When Importing Database Objects
  These sections describe restrictions and points you should consider when you import particular database objects.
- Support for Fine-Grained Access Control
  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.
- Snapshots and Snapshot Logs
  In certain situations, particularly those involving data warehousing, snapshots may be referred to as *materialized views*. These sections retain the term snapshot.
- Transportable Tablespaces
  The transportable tablespace feature enables you to move a set of tablespaces from one Oracle database to another.
- Storage Parameters
  By default, a table is imported into its original tablespace.
- Read-Only Tablespaces
  Read-only tablespaces can be exported. On import, if the tablespace does not already exist in the target database, then the tablespace is created as a read/write tablespace.
- 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`.
- Reorganizing Tablespaces
  If a user's quota allows it, the user's tables are imported into the same tablespace from which they were exported.
- Importing Statistics
  If statistics are requested at export time and analyzer statistics are available for a table, then Export will include the `ANALYZE` statement used to recalculate the statistics for the table into the dump file.
- 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.
- Tuning Considerations for Import Operations
  These sections discuss some ways to improve the performance of an import operation.
- Using Different Releases of Export and Import
  These sections describe compatibility issues that relate to using different releases of Export and the Oracle database.

26.1 What Is the Import Utility?

The original Import utility (`imp`) read object definitions and table data from dump files created by the original Export utility (`exp`).

> **Note:**
>
> Original Export is desupported for general use as of Oracle Database 11g. The only supported use of original Export in Oracle Database 11g and later releases is backward migration of `XMLType` data to Oracle Database 10g Release 2 (10.2) or earlier. Oracle strongly recommends that you use the new Oracle Data Pump Export and Import utilities. The only exception to this guidelines is in the following situations, which require original Export and Import:
>
> - You want to import files that were created using the original Export utility (`exp`).
> - You want to export files that must be imported using the original Import utility (`imp`). An example of this would be exporting data from Oracle Database 10g and then importing it into an earlier database release.

If you use original Import, then the following conditions must be true:

- The dump file is in an Oracle binary-format that can be read only by original Import.
- The version of the Import utility cannot be earlier than the version of the Export utility used to create the dump file.
- Table Objects: Order of Import
  Table objects are imported as they are read from the export dump file.

26.1.1 Table Objects: Order of Import

Table objects are imported as they are read from the export dump file.

The dump 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).

26.2 Before Using Import

Describes what you should do before using Import.

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

- If you created your database manually, ensure that the `catexp.sql` or `catalog.sql` script has been run. If you created your database using the Database Configuration Assistant (DBCA), it is not necessary to run these scripts.
- Verify that you have the required access privileges.
- Running catexp.sql or catalog.sql
  To use Import, you must run the script `catexp.sql` or `catalog.sql` (which runs `catexp.sql`) after the database has been created or migrated to a newer version.
- Verifying Access Privileges for Import Operations
  To use Import, you must have the `CREATE SESSION` privilege on an Oracle database. This privilege belongs to the `CONNECT` role established during database creation.
- Processing Restrictions
  Restrictions apply when you process data with the Import utility.

26.2.1 Running catexp.sql or catalog.sql

To use Import, you must run the script `catexp.sql` or `catalog.sql` (which runs `catexp.sql`) after the database has been created or migrated to a newer version.

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 import views in the data dictionary
- Creates the `EXP_FULL_DATABASE` and `IMP_FULL_DATABASE` roles
- 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

26.2.2 Verifying Access Privileges for Import Operations

To use Import, you must have the `CREATE SESSION` privilege on an Oracle database. This privilege belongs to the `CONNECT` role established during database creation.

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).

- Importing Objects Into Your Own Schema
  Privileges needed to import objects into your own schema.
- 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`.
- 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.

26.2.2.1 Importing Objects Into Your Own Schema

Privileges needed to import objects into your own schema.

Table 26-1 lists the privileges required to import objects into your own schema. All of these privileges initially belong to the RESOURCE role.

Table 26-1 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)
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)

26.2.2.2 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 26-2 shows the required conditions for the authorizations to be valid on the target system.

Table 26-2 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 and also the `WITH ADMIN OPTION`.

26.2.2.3 Importing Objects Into Other Schemas

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

26.2.2.4 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:

- 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

26.2.3 Processing Restrictions

Restrictions apply when you process data with the Import utility.

Specifically:

- When a type definition has evolved and 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.

26.3 Importing into Existing Tables

These sections describe factors to consider when you import data into existing tables.

- Manually Creating Tables Before Importing Data
  You can manually create tables before importing data.
- Disabling Referential Constraints
  Describes how to disable referential constraints.
- Manually Ordering the Import
  Describes manually ordering the import.

26.3.1 Manually Creating Tables Before Importing Data

You can manually create 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 data type of a column to an incompatible data type (`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.

26.3.2 Disabling Referential Constraints

Describes how to disable 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`, then 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 dump file, but a referential check exists from the `emp` table into the `dept` table, then 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.

26.3.3 Manually Ordering the Import

Describes 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, then 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.

26.4 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.

26.5 Invoking Import

To start the original Import utility and specify parameters, use one of three different methods.

The three methods you have to start the original Import utility are:

- Command-line entries
- Parameter files
- Interactive mode

Before you use one of these methods, be sure to read the descriptions of the available parameters.

- Command-Line Entries
  You can specify all valid parameters and their values from the command line.
- Parameter Files
  You can specify all valid parameters and their values in a parameter file.
- Interactive Mode
  If you prefer to be prompted for the value of each parameter, then you can simply specify `imp` at the command line.
- Invoking Import As SYSDBA
  Starting the original Import utility as `SYSDBA` is a specialized procedure, which should only be done under specific scenarios.
- Getting Online Help
  Import provides online help. Enter `imp help=y` to display Import help.

Related Topics

- Import Parameters
  These sections contain descriptions of the Import command-line parameters.

26.5.1 Command-Line Entries

You can specify all valid parameters and their values from the command line.

Use the following syntax (you will then be prompted for a username and password):

```
imp PARAMETER=value
```

or

```
imp PARAMETER=(value1,value2,...,valuen)
```

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

26.5.2 Parameter Files

You can specify all valid parameters and their values in a parameter file.

Storing the parameters in a file allows them to be easily modified or reused. If you use different parameters for different databases, then you can have multiple parameter files.

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

The syntax for parameter file specifications can be any 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. Import ignores all characters to the right of the pound (#) sign.

You can specify a parameter file at the same time that you are entering parameters on the command line. In fact, you can specify the same parameter in both places. The position of the `PARFILE` parameter and other parameters on the command line determines which parameters take precedence. For example, assume the parameter file `params.dat` contains the parameter `INDEXES=y` and Import is started with the following line:

```
imp PARFILE=params.dat INDEXES=n
```

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

> **See Also:**
>
> - Import Parameters
> - Network Considerations for information about how to specify an export from a remote database

26.5.3 Interactive Mode

If you prefer to be prompted for the value of each parameter, then you can simply specify `imp` at the command line.

You will be prompted for a username and password.

Commonly used parameters are then displayed. You can accept the default value, if one is provided, or enter a different value. The command-line interactive method does not provide prompts for all functionality and is provided only for backward compatibility.

26.5.4 Invoking Import As SYSDBA

Starting the original Import utility as `SYSDBA` is a specialized procedure, which should only be done under specific scenarios.

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

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

26.5.5 Getting Online Help

Import provides online help. Enter `imp help=y` to display Import help.

26.6 Import Modes

The Import utility supports four modes of operation.

Specifically:

- Full: Imports a full database. Only users with the `IMP_FULL_DATABASE` role 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 import all objects that belong to you (such as tables, grants, indexes, and procedures). A privileged user importing in user mode can import all objects in the schemas of a specified set of users. Use the `FROMUSER` parameter to specify this mode.
- Table: Enables you to import specific tables and partitions. A privileged user can qualify the tables by specifying the schema that contains them. Use the `TABLES` parameter to specify this mode.

> **Note:**
>
> 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, then a user-level Import is performed.

> **Note:**
>
> As of Oracle Database 12*c* release 2 (12.2) the import utility (imp), for security reasons, will no longer import objects as user SYS. If a dump file contains objects that need to be re-created as user SYS, then the imp utility tries to re-create them as user SYSTEM instead. If the object cannot be re-created by user SYSTEM, then you must manually re-create the object yourself after the import is completed.
>
> If the import job is run by a user with the imp_full_database role, then you receive a IMP-403 warning, and an empty file ("xx_sys.sql") is generated.
>
> If the import job is run by a user with the DBA role, and not all objects can be re-created by user SYSTEM, then the following warning message is written to the log file:
>
> ```
> IMP-00403: Warning: This import generated a separate SQL file
> "logfilename_sys" which contains DDL that failed due to a privilege
> issue.
> ```
>
> The SQL file that is generated contains the failed DDL of objects that could not be re-created by user SYSTEM. To re-create those objects, you must manually execute the failed DDL after the import finishes.
>
> The SQL file is automatically named by appending '_sys.sql' to the file name specified for the LOG parameter. For example, if the log file name was JulyImport, then the SQL file name would be JulyImport_sys.sql.
>
> If no log file was specified, then the default name of the SQL file is import_sys.sql.
>
> Note: Not all import jobs generate a SQL file; only those jobs run as user DBA.

The following table lists the objects that are imported in each mode.

Table 26-3 Objects 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

Table 26-3 (Cont.) Objects Imported in Each Mode

Object	Table Mode	User Mode	Full Database Mode	Tablespace Mode
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
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
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

Table 26-3 (Cont.) Objects Imported in Each Mode

Object	Table Mode	User Mode	Full Database Mode	Tablespace Mode
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 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.

26.7 Import Parameters

These sections contain descriptions of the Import command-line parameters.

- BUFFER
- COMMIT
- COMPILE
- CONSTRAINTS
- DATA_ONLY
  The `DATA_ONLY` import parameter imports only data from a dump file.
- DATAFILES
  The `DATAFILES` import parameter lists the data files to be transported into the database.
- DESTROY
  The `DESTROY` Import parameter specifies whether the existing data files making up the database should be reused.
- FEEDBACK
  The `FEEDBACK` Import utility parameter specifies that Import should display a progress meter in the form of a period for *n* number of rows imported.
- FILE
  The `FILE` Import utility parameter specifies the names of the export files to import.
- FILESIZE
  The `FILESIZE` Import utility parameter lets you specify the same maximum dump file size that you specified on export.
- FROMUSER
  The `FROMUSER` parameter of the Import utility enables you to import a subset of schemas from an export file containing multiple schemas.
- FULL
  The `FULL` Import utility parameter specifies whether to import the entire export dump file.
- GRANTS
  Specifies whether to import object grants.
- HELP
  The `HELP` parameter of Import utility displays a description of the Import parameters.
- IGNORE
  The IGNORE Import utility parameter specifies how object creation errors should be handled.
- INDEXES
  Indexes import parameter specifies whether to import indexes.
- INDEXFILE
  INDEXFILE parameter of Import utility specifies a file to receive index-creation statements.
- LOG
  Specifies a file (for example, import.log) to receive informational and error messages.
- PARFILE
  Specifies a file name for a file that contains a list of Import parameters.
- RECORDLENGTH
  Specifies the length, in bytes, of the file record.
- RESUMABLE
  The `RESUMABLE` parameter is used to enable and disable resumable space allocation.
- RESUMABLE_NAME
  The value for the `RESUMABLE_NAME` parameter identifies the statement that is resumable.

- RESUMABLE_TIMEOUT
  The value of the `RESUMABLE_TIMEOUT` parameter specifies the time period during which an error must be fixed.
- ROWS
  Specifies whether to import the rows of table data.
- SHOW
  Lists the contents of the export file before importing.
- SKIP_UNUSABLE_INDEXES
  Both Import and the Oracle database provide a `SKIP_UNUSABLE_INDEXES` parameter.
- STATISTICS
  Specifies what is done with the database optimizer statistics at import time.
- STREAMS_CONFIGURATION
  Specifies whether to import any general GoldenGate Replication metadata that may be present in the export dump file.
- STREAMS_INSTANTIATION
  Specifies whether to import Streams instantiation metadata that may be present in the export dump file.
- TABLES
- TABLESPACES
  The `TABLESPACES` parameter for the Import utility.
- TOID_NOVALIDATE
  Use the `TOID_NOVALIDATE` parameter to specify types to exclude from TOID comparison.
- TOUSER
  Specifies a list of user names whose schemas will be targets for Import.
- TRANSPORT_TABLESPACE
  When specified as `y`, instructs Import to import transportable tablespace metadata from an export file.
- TTS_OWNERS
  When `TRANSPORT_TABLESPACE` is specified as `y`, use this parameter to list the users who own the data in the transportable tablespace set.
- USERID (username/password)
  Specifies the username, password, and an optional connect string of the user performing the import.
- VOLSIZE
  Specifies the maximum number of bytes in a dump file on each volume of tape.

26.7.1 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, LONG, BFILE, REF, ROWID,UROWID, or TIMESTAMP 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, then Import attempts to allocate a larger buffer.

For DATE columns, two or more rows are inserted at once if the buffer is large enough.

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

26.7.2 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, then 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, then each partition and subpartition in the Export file is imported in a separate transaction.

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

26.7.3 COMPILE

Default: y

Specifies whether Import should compile packages, procedures, and functions as they are created.

If COMPILE=n, then 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 "

26.7.4 CONSTRAINTS

Default: y

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

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

26.7.5 DATA_ONLY

The `DATA_ONLY` import parameter imports only data from a dump file.

Default

`n`

To import only data (no metadata) from a dump file, specify `DATA_ONLY=y`.

When you specify `DATA_ONLY=y`, any import parameters related to metadata that are entered on the command line (or in a parameter file) become invalid. This means that no metadata from the dump file will be imported.

The metadata-related parameters are the following: `COMPILE`, `CONSTRAINTS`, `DATAFILES`, `DESTROY`, `GRANTS`, `IGNORE`, `INDEXES`, `INDEXFILE`, `ROWS=n`, `SHOW`, `SKIP_UNUSABLE_INDEXES`, `STATISTICS`, `STREAMS_CONFIGURATION`, `STREAMS_INSTANTIATION`, `TABLESPACES`, `TOID_NOVALIDATE`, `TRANSPORT_TABLESPACE`, `TTS_OWNERS`.

26.7.6 DATAFILES

The `DATAFILES` import parameter lists the data files to be transported into the database.

Default

None.

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

> **See Also:**
>
> TRANSPORT_TABLESPACE

26.7.7 DESTROY

The `DESTROY` Import parameter specifies whether the existing data files making up the database should be reused.

Default

`n`

Specifies whether the existing data files making up the database should be reused. That is, specifying `DESTROY=y` causes Import to include the `REUSE` option in the data file clause of the SQL `CREATE TABLESPACE` statement, which causes Import to reuse the original database's data files after deleting their contents.

Note that the export file contains the data file 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), then the Import utility will overwrite the first database's data files when it creates the tablespace. In this situation you should use the default, `DESTROY=n`, so that an error occurs if the data files already exist when the tablespace is created. Also, when you need to import into the original database, you will need to specify `IGNORE=y` to add to the existing data files without replacing them.

> **Note:**
>
> If data files are stored on a raw device, then `DESTROY=n` *does not prevent* files from being overwritten.

26.7.8 FEEDBACK

The `FEEDBACK` Import utility parameter specifies that Import should display a progress meter in the form of a period for *n* number of rows imported.

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`, then 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.

26.7.9 FILE

The `FILE` Import utility parameter specifies the names of the export files to import.

Default: `expdat.dmp`

Description

Specifies the names of the export files to import. The default extension is `.dmp`. Because Export supports multiple export files, it can be necessary to specify multiple file names that you want to be imported.

You do not need to be the user that exported the export files. However, you must have read access to the files. If you did not export the files under your user ID, then you must also have the `IMP_FULL_DATABASE` role granted to you.

Example

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

26.7.10 FILESIZE

The `FILESIZE` Import utility parameter lets you specify the same maximum dump file size that you specified on export.

Default: operating system-dependent

Lets you specify the same maximum dump file size that 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`).

26.7.11 FROMUSER

The `FROMUSER` parameter of the Import utility enables you to import a subset of schemas from an export file containing multiple schemas.

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. The user that you specify with `TOUSER` must exist in the target database before the import operation; otherwise an error is returned.

If you do not specify `TOUSER`, then 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.

26.7.12 FULL

The `FULL` Import utility parameter specifies whether to import the entire export dump file.

Default: `y`

Specifies whether to import the entire export dump file.

- 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.

26.7.12.1 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.
- A full export also does not export the default profile. If you have modified the default profile in the source database (for example, by adding a password verification function owned by schema `SYS`), then you must manually pre-create the function and modify the default profile in the target database after the import completes.
- 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 data files
  - 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, then 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 data file names as the exported database. This can result in problems in the following situations:
  - If the data files belong to any other database, then they will become corrupted. This is especially true if the exported database is on the same system, because its data files will be reused by the database into which you are importing.
  - If the data files have names that conflict with existing operating system files.

26.7.13 GRANTS

Specifies whether to import object grants.

Default: `y`

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

If the export was a full database mode export, then 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`, then 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, then access privileges would be changed and the importer would not be aware of this.

26.7.14 HELP

The `HELP` parameter of Import utility displays a description of the Import parameters.

Default: none

Displays a description of the Import parameters. Enter `imp HELP=y` on the command line to display the help content.

26.7.15 IGNORE

The IGNORE Import utility parameter specifies how object creation errors should be handled.

Default: `n`

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

If you specify `IGNORE=y`, then 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`, then 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.

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

26.7.16 INDEXES

Indexes import parameter specifies whether to import indexes.

Default: `y`

Specifies whether 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, then Import performs index maintenance when data is inserted into the table.

26.7.17 INDEXFILE

INDEXFILE parameter of Import utility specifies a file to receive index-creation statements.

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`, then 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.

26.7.18 LOG

Specifies a file (for example, import.log) to receive informational and error messages.

Default: none

If you specify a log file, then the Import utility writes all information to the log in addition to the terminal display.

26.7.19 PARFILE

Specifies a file name for a file that contains a list of Import parameters.

Default: none

For more information about using a parameter file, see Parameter Files.

26.7.20 RECORDLENGTH

Specifies the length, in bytes, of the file record.

Default

Operating system-dependent.

Purpose

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, then 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.

26.7.21 RESUMABLE

The RESUMABLE parameter is used to enable and disable resumable space allocation.

Default

n

Purpose

Because this parameter is disabled by default, you must set RESUMABLE=y to use its associated parameters, RESUMABLE_NAME and RESUMABLE_TIMEOUT.

> **See Also:**
>
> *Oracle Database Administrator's Guide* for more information about resumable space allocation.

26.7.22 RESUMABLE_NAME

The value for the `RESUMABLE_NAME` parameter identifies the statement that is resumable.

Default

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

Purpose

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.

26.7.23 RESUMABLE_TIMEOUT

The value of the `RESUMABLE_TIMEOUT` parameter specifies the time period during which an error must be fixed.

Default

`7200` seconds (2 hours)

Purpose

If the error is not fixed within the timeout period, then execution of the statement is terminated.

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

26.7.24 ROWS

Specifies whether to import the rows of table data.

Default

`y`

Purpose

If `ROWS=n`, then statistics for all imported tables will be locked after the import operation is finished.

26.7.25 SHOW

Lists the contents of the export file before importing.

Default

`n`

Syntax and Description

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.

26.7.26 SKIP_UNUSABLE_INDEXES

Both Import and the Oracle database provide a `SKIP_UNUSABLE_INDEXES` parameter.

Default: the value of the Oracle database configuration parameter, `SKIP_UNUSABLE_INDEXES`, as specified in the initialization parameter file.

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, then 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, then row insertions that attempt to update unusable indexes will fail.

> **See Also:**
>
> The `ALTER SESSION` statement in the *Oracle Database SQL Language Reference*

26.7.27 STATISTICS

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

Default: `ALWAYS`

The options are:

- `ALWAYS`

  Always import database optimizer statistics regardless of whether 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, then 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

26.7.28 STREAMS_CONFIGURATION

Specifies whether to import any general GoldenGate Replication metadata that may be present in the export dump file.

Default: `y`

26.7.29 STREAMS_INSTANTIATION

Specifies whether to import Streams instantiation metadata that may be present in the export dump file.

Default: `n`

Specify `y` if the import is part of an instantiation in a Streams environment.

26.7.30 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 Import Modes. 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, then 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`*, then 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, then 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 TABLES=(jones.accts, scott.emp, scott.dept)
```

The valid specification to import these tables is as follows:

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

For a more detailed example, see "Example Import Using Pattern Matching to Import Various Tables".

> **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
  This is an explanation of table name restrictions for Import utility.

26.7.30.1 Table Name Restrictions

This is an explanation of table name restrictions for Import utility.

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, then 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, then 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, then 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.

26.7.31 TABLESPACES

The `TABLESPACES` parameter for the Import utility.

Default: none

When `TRANSPORT_TABLESPACE` is specified as `y`, use this parameter to list the tablespaces to be transported into the database. If there is more than one tablespace in the export file, then you must specify all of them as part of the import operation.

See TRANSPORT_TABLESPACE for more information.

26.7.32 TOID_NOVALIDATE

Use the `TOID_NOVALIDATE` parameter to specify types to exclude from TOID comparison.

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 TABLES=jobs TOID_NOVALIDATE=typ1
imp scott TABLES=salaries TOID_NOVALIDATE=(fred.typ0,sally.typ2,typ3)
```

If you do not specify a schema name for the type, then it defaults to the schema of the importing user. For example, in the first preceding example, the type `typ1` defaults to `scott.typ1` and in the second example, the type `typ3` defaults to `scott.typ3`.

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"
[...]
```

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

26.7.33 TOUSER

Specifies a list of user names whose schemas will be targets for Import.

Default: none

The user names must exist before 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 FROMUSER=scott TOUSER=joe TABLES=emp
```

If multiple schemas are specified, then 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 FROMUSER=scott,fred TOUSER=joe,ted
```

If the `FROMUSER` list is longer than the `TOUSER` list, then 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 FROMUSER=scott,adams TOUSER=ted,ted
```

Note that user `ted` is listed twice.

> **See Also:**
>
> FROMUSER for information about restrictions when using `FROMUSER` and `TOUSER`

26.7.34 TRANSPORT_TABLESPACE

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

Default: `n`

Encrypted columns are not supported in transportable tablespace mode.

> **Note:**
>
> You cannot export transportable tablespaces and then import them into a database at a lower release level. The target database must be at the same or later release level as the source database.

26.7.35 TTS_OWNERS

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

Default: none

See TRANSPORT_TABLESPACE.

26.7.36 USERID (username/password)

Specifies the username, password, and an optional connect string of the user performing the import.

Default: none

If you connect as user `SYS`, then 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:**
>
> The user's guide for your Oracle Net protocol for information about specifying a connect string for Oracle Net.

26.7.37 VOLSIZE

Specifies the maximum number of bytes in a dump file on each volume of tape.

Default: none

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`).

26.8 Example Import Sessions

These sections give some examples of import sessions that show you how to use the parameter file and command-line methods.

- 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

26.8.1 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 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 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. Status messages are also displayed.

26.8.2 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 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 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. Status messages are also displayed.

26.8.3 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 PARFILE=params.dat
```

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

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

Command-Line Method

```
> imp 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.
```

26.8.4 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
- Example 2: A Partition-Level Import of a Composite Partitioned Table

- Example 3: Repartitioning a Table on a Different Column

26.8.4.1 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 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 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. Status messages are also displayed.

26.8.4.2 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 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 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.
```

26.8.4.3 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 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
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 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
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;
```

26.8.5 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 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 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.
```

26.9 Exit Codes for Inspection and Display

Import provides 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 26-4 shows the exit codes that get returned for various results.

Table 26-4 Exit Codes for Import

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

For UNIX, the exit codes are as follows:

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

26.10 Error Handling During an Import

These sections describe 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, then Import displays a warning message but continues processing the rest of the table.
- Errors Importing Database Objects
  Errors can occur for many reasons when you import database objects, as described in these sections.

26.10.1 Row Errors

If a row is rejected due to an integrity constraint violation or invalid data, then 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.
- 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.

26.10.1.1 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 Development Guide* for information about using integrity constraints in applications
>
> – *Oracle Database Concepts* for more information about integrity constraints

26.10.1.2 Invalid Data

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

The error is caused by data that is too long to fit into a new table's columns, by invalid data types, or by any other `INSERT` error.

26.10.2 Errors Importing Database Objects

Errors can occur for many reasons when you import database objects, as described in these sections.

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, then an object creation error occurs.
- Sequences
  If sequence numbers need to be reset to the value in an export file as part of an import, then you should drop sequences.
- Resource Errors
  Resource limitations can cause objects to be skipped. When you are importing tables, for example, resource errors can occur because of internal problems or when a resource such as memory has been exhausted.
- Domain Index Metadata
  Domain indexes can have associated application-specific metadata that is imported using anonymous PL/SQL blocks.

26.10.2.1 Object Already Exists

If a database object to be imported already exists in the database, then an object creation error occurs.

What happens next depends on the setting of the `IGNORE` parameter.

If `IGNORE=n` (the default), then 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`, then object creation errors are not reported. The database object is not replaced. If the object is a table, then 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.

> **Note:**
>
> 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.

26.10.2.2 Sequences

If sequence numbers need to be reset to the value in an export file as part of an import, then you should drop sequences.

If a sequence is not dropped before the import, then 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, then the export file's `CREATE SEQUENCE` statement fails and the sequence is not imported.

26.10.2.3 Resource Errors

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

If a resource error occurs while you are importing a row, then Import stops processing the current table and skips to the next table. If you have specified `COMMIT=y`, then Import commits the partial import of the current table. If not, then a rollback of the current table occurs before Import continues. See the description of COMMIT.

26.10.2.4 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, before the `CREATE INDEX` statement. If a PL/SQL block causes an error, then the associated index is not created because the metadata is considered an integral part of the index.

26.11 Table-Level and Partition-Level Import

You can import tables, partitions, and subpartitions.

Specifically:

- 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.
- Guidelines for Using Table-Level Import
  For each specified table, table-level Import imports all rows of the table.
- 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.
- Migrating Data Across Partitions and Tables
  If you specify a partition name for a composite partition, then all subpartitions within the composite partition are used as the source.

26.11.1 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, then table-level Import creates a partitioned table. If the table creation is successful, then 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.

26.11.2 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, then 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, then 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, then 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, then 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`, then 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, then 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.

26.11.3 Migrating Data Across Partitions and Tables

If you specify a partition name for a composite partition, then 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 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 FILE=expdat.dmp TABLES=(e:qc, e:qd) IGNORE=y
```

If table `e` does not exist in the import target database, then it is created and data is inserted into the same partitions. If table `e` existed on the target system before import, then 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`.

26.12 Controlling Index Creation and Maintenance

These sections describe 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.
- Index Creation and Maintenance Controls
  Describes index creation and maintenance controls.

26.12.1 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.

26.12.2 Index Creation and Maintenance Controls

Describes index creation and maintenance controls.

If SKIP_UNUSABLE_INDEXES=y, then 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
  Shows an example of postponing index maintenance.

26.12.2.1 Example of Postponing Index Maintenance

Shows an example of postponing index maintenance.

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 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.

26.13 Network Considerations for Using Oracle Net with Original Import

To perform imports over a network, you can use the Oracle Data Pump original Import utility (imp) with Oracle Net.

For example, if you run Import locally, then you can read data into a remote Oracle Database instance.

To use Import with Oracle Net, when you run the `imp` command and enter the username and password, include the connection qualifier string `@connect_string`. For the exact syntax of this clause, see the user's guide for your Oracle Net protocol.

Related Topics

- *Oracle Database Net Services Administrator's Guide*

26.14 Character Set and Globalization Support Considerations

These sections describe the globalization support behavior of 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.)
- Data Definition Language (DDL)
  Up to three character set conversions may be required for data definition language (DDL) during an export/import operation.
- Single-Byte Character Sets
  Some 8-bit characters can be lost (that is, converted to 7-bit equivalents) when you import an 8-bit character set export file.
- Multibyte Character Sets
  During character set conversion, any characters in the export file that have no equivalent in the target character set are replaced with a default character. (The default character is defined by the target character set.)

26.14.1 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, then 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.

26.14.1.1 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* for more information about character sets

26.14.2 Data Definition Language (DDL)

Up to three character set conversions may be required for data definition language (DDL) during an export/import operation.

Specifically:

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.

26.14.3 Single-Byte Character Sets

Some 8-bit characters can be lost (that is, converted to 7-bit equivalents) when you import an 8-bit character set export file.

This occurs if the system on which the import occurs has a native 7-bit character set, or the `NLS_LANG` operating system environment variable is set to a 7-bit character set. Most often, this is apparent when accented characters lose the accent mark.

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

26.14.4 Multibyte Character Sets

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

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

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

26.15 Using Instance Affinity

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* for more information about affinity

26.16 Considerations When Importing Database Objects

These sections describe restrictions and points you should consider when you import particular database objects.

- Importing Object Identifiers
- Importing Existing Object Tables and Tables That Contain Object Types
  Importing existing Object Tables and tables that contain Object Types is one of the considerations when importing database objects. The tables must be created with the same definitions as were previously used or a compatible format (except for storage parameters).
- Importing Nested Tables
- Importing REF Data
  Importing `REF` data is one of the considerations when importing database objects. `REF` columns and attributes may contain a hidden `ROWID` that points to the referenced type instance.
- Importing BFILE Columns and Directory Aliases
  Importing BFILE Columns and Directory Aliases is one of the considerations when importing database objects. When you import table data that contains BFILE columns, the

BFILE locator is imported with the directory alias and file name that was present at export time.

- Importing Foreign Function Libraries
  Importing Foreign Function Libraries is one of the considerations when importing database objects. Import does not verify that the location referenced by the foreign function library is correct.
- 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`.
- Importing Java Objects
  Importing Java Objects is one of the considerations when importing database objects. When you import Java objects into any schema, the Import utility leaves the resolver unchanged.
- Importing External Tables
  Importing external tables is one of the considerations when importing database objects. Import does not verify that the location referenced by the external table is correct.
- Importing Advanced Queue (AQ) Tables
  Importing Advanced Queue Tables is a one of the considerations when importing database objects. Importing a queue table also imports any underlying queues and the related dictionary information.
- Importing LONG Columns
  Importing LONG columns is one of the considerations when importing database objects. In importing and exporting, the LONG columns must fit into memory with the rest of each row's data.
- Importing LOB Columns When Triggers Are Present
  Importing LOB columns when triggers are present is one of the considerations when importing database objects. The Import utility automatically changes all LOBs that were empty at export time to be `NULL` after they are imported.
- Importing Views
  Importing views that contain references to tables in other schemas requires that the importer have the `READ ANY TABLE` or `SELECT ANY TABLE` privilege.
- Importing Partitioned Tables
  Importing partitioned tables is one of the considerations when importing database objects. 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`*.

26.16.1 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, then 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 for more information.

> **Note:**
>
> 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, then 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, then 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, then 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, then any tables using the object type are not imported.
- For object tables, if IGNORE=y, then 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, then an error is reported and the table is not imported.
- For object tables, if IGNORE=n and the table already exists, then 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, then 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, then 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.

26.16.2 Importing Existing Object Tables and Tables That Contain Object Types

Importing existing Object Tables and tables that contain Object Types is one of the considerations when importing database objects. The tables must be created with the same definitions as were previously used or a compatible format (except for storage parameters).

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 before 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, then 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.

26.16.3 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, then 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, then 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, then its outer table row will already have been inserted in the outer table and data will continue to be inserted into 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, then 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.

26.16.4 Importing REF Data

Importing `REF` data is one of the considerations when importing database objects. `REF` columns and attributes may contain a hidden `ROWID` that points to the referenced type instance.

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

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

> **See Also:**
>
> *Oracle Database SQL Language Reference* for more information about the `ANALYZE` statement

26.16.5 Importing BFILE Columns and Directory Aliases

Importing BFILE Columns and Directory Aliases is one of the considerations when importing database objects. When you import table data that contains BFILE columns, the BFILE locator is imported with the directory alias and file name that was present at export time.

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

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

26.16.6 Importing Foreign Function Libraries

Importing Foreign Function Libraries is one of the considerations when importing database objects. Import does not verify that the location referenced by the foreign function library is correct.

Import does not verify that the location referenced by the foreign function library is correct. If the formats for directory and file names used in the library's specification on the export file are invalid on the import system, then 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.

26.16.7 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`.

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, then it can be accessed by remote procedures without error.

If `COMPILE=n`, then 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

26.16.8 Importing Java Objects

Importing Java Objects is one of the considerations when importing database objects. When you import Java objects into any schema, the Import utility leaves the resolver unchanged.

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.

26.16.9 Importing External Tables

Importing external tables is one of the considerations when importing database objects. Import does not verify that the location referenced by the external table is correct.

Import does not verify that the location referenced by the external table is correct. If the formats for directory and file names used in the table's specification on the export file are invalid on the import system, then 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.

26.16.10 Importing Advanced Queue (AQ) Tables

Importing Advanced Queue Tables is a one of the considerations when importing database objects. Importing a queue table also imports any underlying queues and the related dictionary information.

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 pre-table and post-table action procedures maintain the queue dictionary.

> **See Also:**
>
> *Oracle Database Advanced Queuing User's Guide*

26.16.11 Importing LONG Columns

Importing LONG columns is one of the considerations when importing database objects. In importing and exporting, the LONG columns must fit into memory with the rest of each row's data.

`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.

26.16.12 Importing LOB Columns When Triggers Are Present

Importing LOB columns when triggers are present is one of the considerations when importing database objects. The Import utility automatically changes all LOBs that were empty at export time to be `NULL` after they are imported.

As of Oracle Database 10g, 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, then be sure you know which LOB columns are NULL and which are empty before you perform the import.

26.16.13 Importing Views

Importing views that contain references to tables in other schemas requires that the importer have the READ ANY TABLE or SELECT ANY TABLE privilege.

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, then Export cannot determine the proper ordering of viewa and viewc. If viewa is exported before viewc, and procb already exists on the import system, then 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, then 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 the READ ANY TABLE or SELECT ANY TABLE privilege. If the importer has not been granted this privilege, then 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.

26.16.14 Importing Partitioned Tables

Importing partitioned tables is one of the considerations when importing database objects. 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*.

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, then 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.

26.17 Support for Fine-Grained Access Control

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 import from an export file that contains tables with fine-grained access control policies, then a warning message is issued.

26.18 Snapshots and Snapshot Logs

In certain situations, particularly those involving data warehousing, snapshots may be referred to as *materialized views*. These sections retain the term snapshot.

- 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.
- Snapshots
  A snapshot that has been restored from an export file has reverted to a previous state.

26.18.1 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.

26.18.2 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.
- Importing a Snapshot into a Different Schema
  Snapshots and related items are exported with the schema name given in the DDL statements.

26.18.2.1 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), then they are used, and the fast refresh completes successfully. Otherwise, the fast refresh fails, generating an error that says a complete refresh is required.

26.18.2.2 Importing a Snapshot into a Different Schema

Snapshots and related items are exported with the schema name 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.

> **Note:**
>
> 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.

26.19 Transportable Tablespaces

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

> **Note:**
>
> You cannot export transportable tablespaces and then import them into a database at a lower release level. The target database must be at the same or later release level as the source database.

To move or copy a set of tablespaces, you must make the tablespaces read-only, manually copy the data files of these tablespaces to the target database, and use Export and Import to move the database information (metadata) stored in the data dictionary over to the target database. The transport of the data 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 data files 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 and TRANSPORT_TABLESPACE for information about using these parameters during an import operation.

> **See Also:**
>
> - *Oracle Database Administrator's Guide* for details about managing transportable tablespaces

26.20 Storage Parameters

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

If the tablespace no longer exists, or the user does not have sufficient quota in the tablespace, then 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, then the user's tables are not imported. See Reorganizing Tablespaces 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.
- Overriding Storage Parameters
  Before using the Import utility to import data, you may want to create large tables with different storage parameters.

26.20.1 The OPTIMAL Parameter

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

26.20.2 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 before exporting, then the tables are exported using those altered storage parameters. Note, however, that storage parameters for LOB data cannot be altered before exporting (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, then 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.

26.20.3 Overriding Storage Parameters

Before using the Import utility to import data, you may want to create large tables with different storage parameters.

If so, then you must specify `IGNORE=y` on the command line or in the parameter file.

26.21 Read-Only Tablespaces

Read-only tablespaces can be exported. On import, if the tablespace does not already exist in the target database, then the tablespace is created as a read/write tablespace.

To get 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, then you must make it read/write before the import.

26.22 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 except for 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.

26.23 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, then 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, then 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.)

26.24 Importing Statistics

If statistics are requested at export time and analyzer statistics are available for a table, then 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 Import parameter STATISTICS.

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, then you should specify STATISTICS=NONE.

26.25 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, then be aware of the following advantages and disadvantages.

- Advantages of Partitioning a Migration
  Describes the advantages of partitioning a migration.
- Disadvantages of Partitioning a Migration
  Describes the disadvantages of partitioning a migration.
- How to Use Export and Import to Partition a Database Migration
  Describes how to use Export and Import to partition a migration.

26.25.1 Advantages of Partitioning a Migration

Describes the advantages of partitioning a migration.

Specifically:

- 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.

26.25.2 Disadvantages of Partitioning a Migration

Describes the disadvantages of partitioning a migration.

Specifically:

- 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, then you may not have the required parent records when you import the table into the dependent schema.

26.25.3 How to Use Export and Import to Partition a Database Migration

Describes how to use Export and Import to partition a 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 FILE=full FULL=y CONSTRAINTS=n TRIGGERS=n ROWS=n INDEXES=n`
   b. `imp FILE=full FULL=y`
2. For each schema*n* in the database, issue the following commands:
   a. `exp OWNER=schema`*n* `FILE=schema`*n*
   b. `imp 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.

26.26 Tuning Considerations for Import Operations

These sections discuss some ways to improve the performance of an import operation.

- Changing System-Level Options
  Describes system-level options that may help improve the performance of an import operation.
- Changing Initialization Parameters
  These suggestions about settings in your initialization parameter file may help improve performance of an import operation.
- Changing Import Options
  These suggestions about the usage of import options may help improve performance.
- Dealing with Large Amounts of LOB Data
  Describes importing large amounts of LOB data.
- 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.

26.26.1 Changing System-Level Options

Describes system-level options that may help improve the performance of an import operation.

Specifically :

- 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:**
>
> Oracle recommends that you use automatic undo management instead of rollback segments.

- 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.
- Ensure that 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.*

26.26.2 Changing Initialization Parameters

These suggestions about settings in your initialization parameter file may help 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, then the value is probably set too high.)
- Increase the value for `DB_BLOCK_BUFFERS` and `SHARED_POOL_SIZE`.

26.26.3 Changing Import Options

These suggestions about the usage of import options may help improve performance.

Be sure to also read the individual descriptions of all the available options in Import Parameters.

- 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, then 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`.

26.26.4 Dealing with Large Amounts of LOB Data

Describes importing large amounts of LOB data.

Specifically:

- 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.
- Ensure that there is enough space available in large contiguous chunks to complete the data load.

26.26.5 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

26.27 Using Different Releases of Export and Import

These sections describe 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. For example, if you try to use the Import utility 9.2.0.7 to import into a 9.2.0.8 database, then you may encounter errors.
- The version of the Export utility must be equal to the version of either the source or target database, whichever is earlier.

  For example, to create an export file for an import into a later release database, use a version of the Export utility that equals 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 equals 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.
- Restrictions When Using Different Releases of Export and Import
  Restrictions apply when you are using different releases of Export and Import.
- Examples of Using Different Releases of Export and Import
  Using different releases of Export and Import.

26.27.1 Restrictions When Using Different Releases of Export and Import

Restrictions apply when you are using different releases of Export and Import.

Specifically:

- 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.

26.27.2 Examples of Using Different Releases of Export and Import

Using different releases of Export and Import.

Table 26-5 shows some examples of which Export and Import releases to use when moving data between different releases of the Oracle database.

Table 26-5 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

Table 26-5 covers moving data only between the original Export and Import utilities. For Oracle Database 10*g* release 1 (10.1) or later, Oracle recommends the Data Pump Export and Import utilities in most cases because these utilities provide greatly enhanced performance compared to the original Export and Import utilities.

See Also:

Oracle Database Upgrade Guide for more information about exporting and importing data between different releases, including releases later than 10.1

Part V
Appendixes

This section contains the following topics:

A

SQL*Loader Syntax Diagrams

This appendix describes SQL*Loader syntax in graphic form (sometimes called railroad diagrams or DDL diagrams).

How to Read Graphic Syntax Diagrams

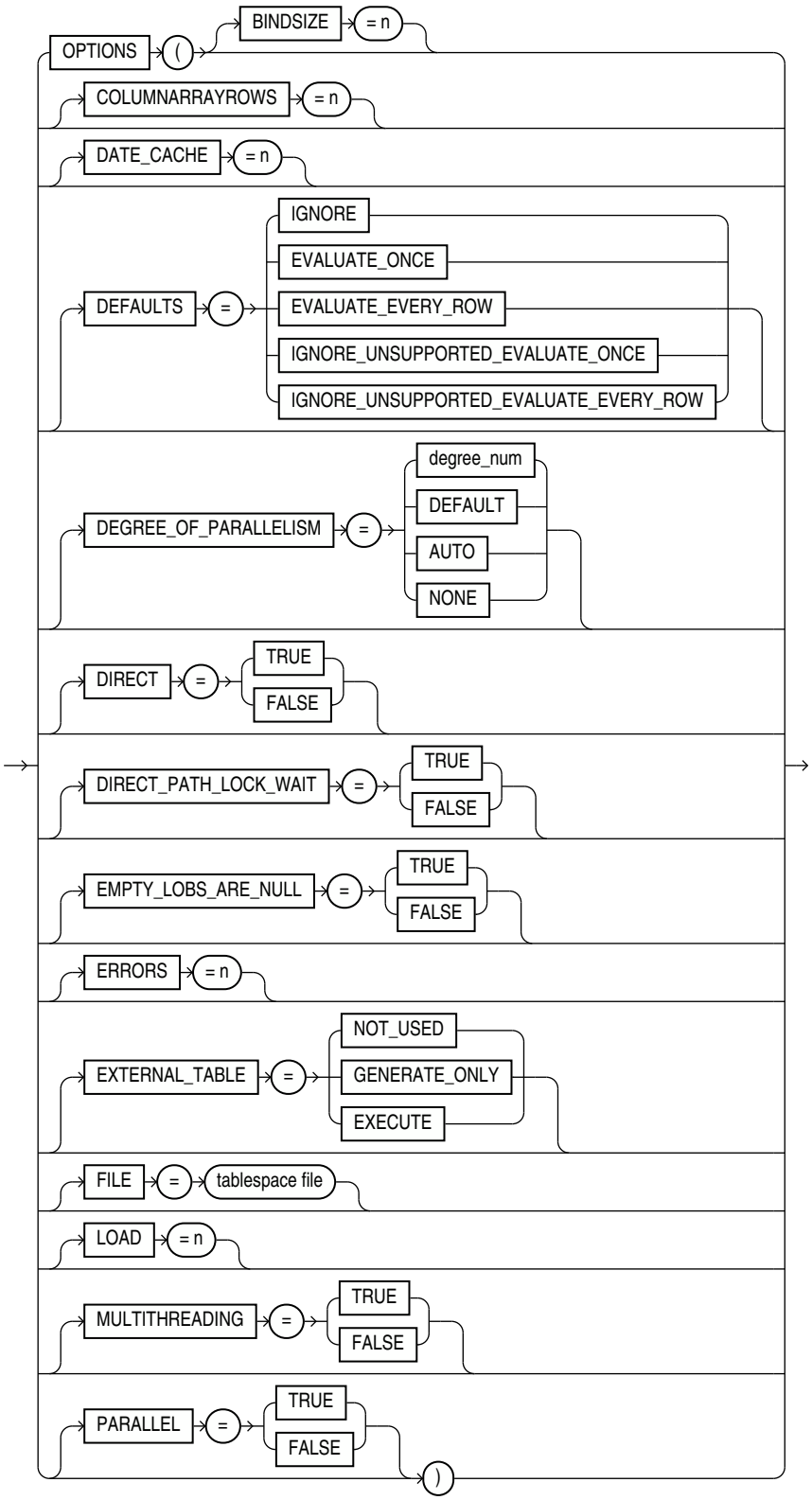
Syntax diagrams are drawings that illustrate valid SQL syntax. To read a diagram, trace it from left to right, in the direction shown by the arrows.

For more information about standard SQL syntax notation, see:

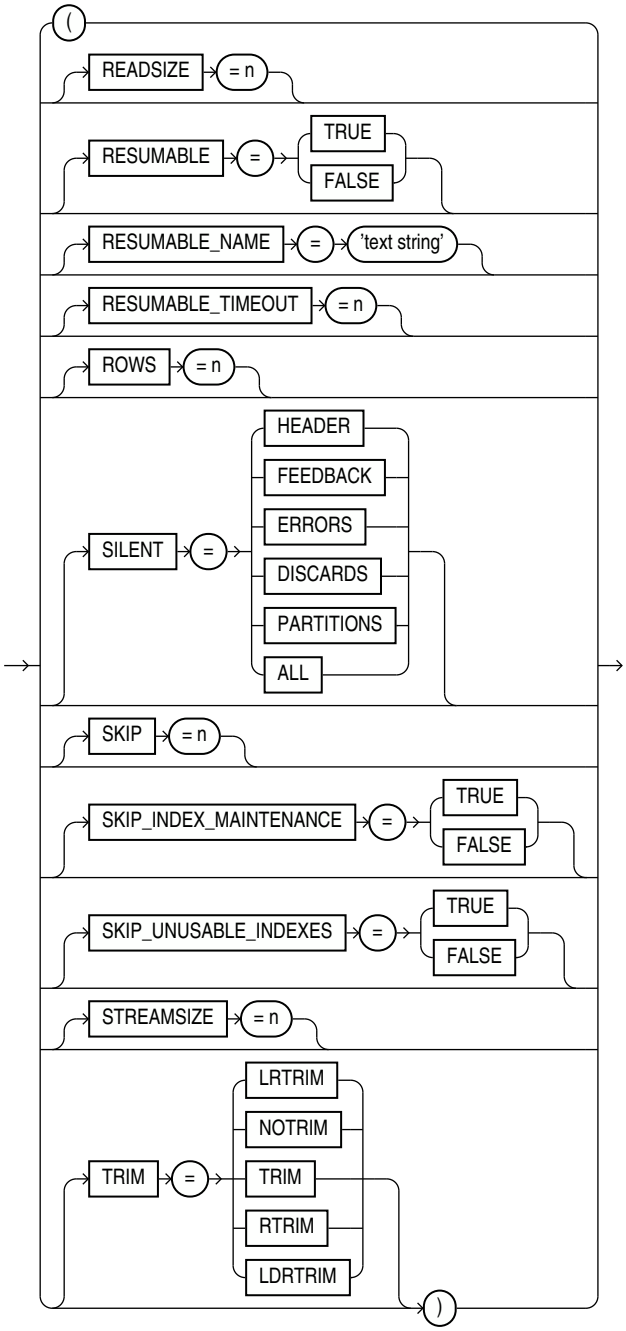
How to Read Syntax Diagrams in *Oracle Database SQL Language Reference*

The following diagrams are shown with certain clauses collapsed (such as `pos_spec`). These diagrams are expanded and explained further along in the appendix.

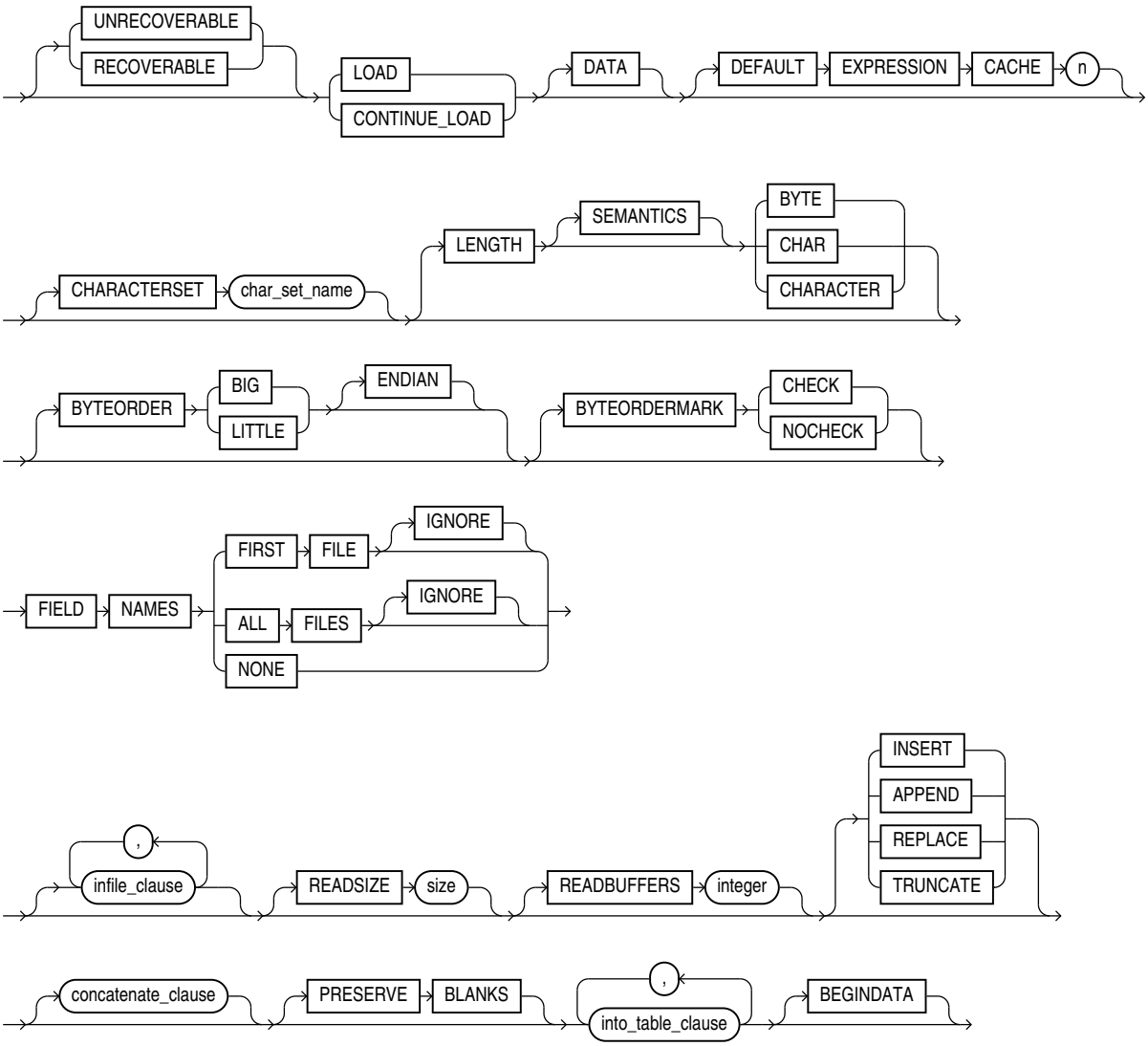
Options Clause



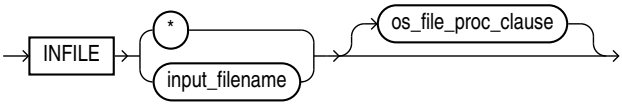
Options_Cont

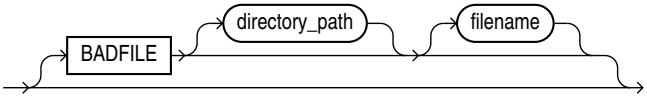


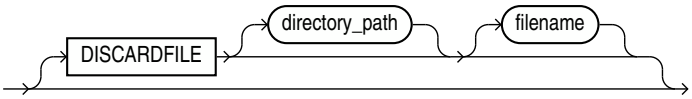
Load Statement

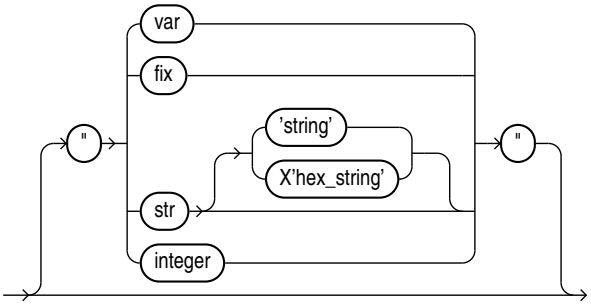


infile_clause



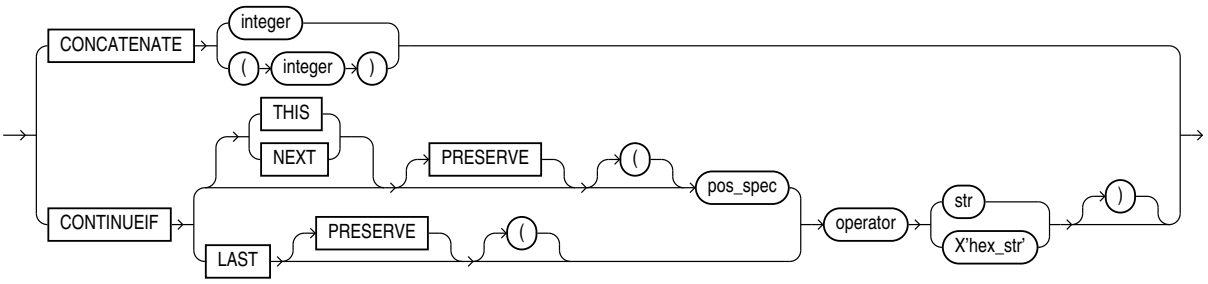




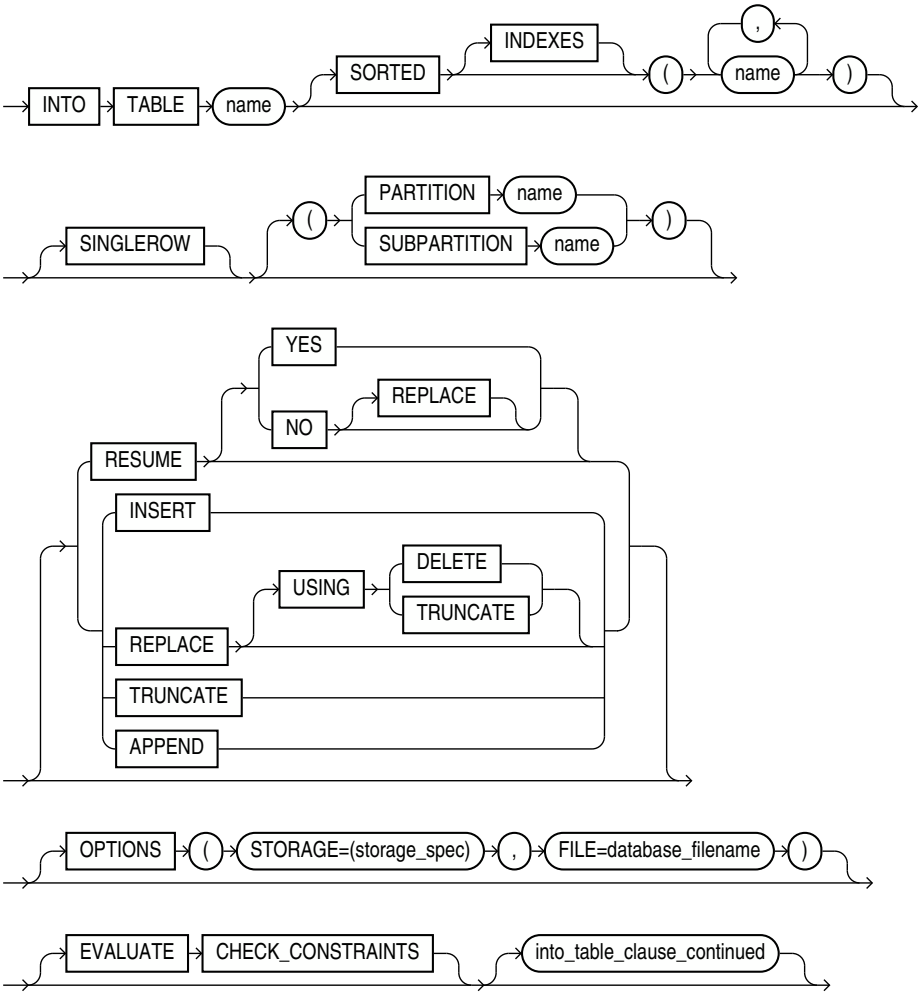


> **Note:**
>
> On the `BADFILE and DISCARDFILE` clauses, you must specify either a directory path, or a filename, or both.

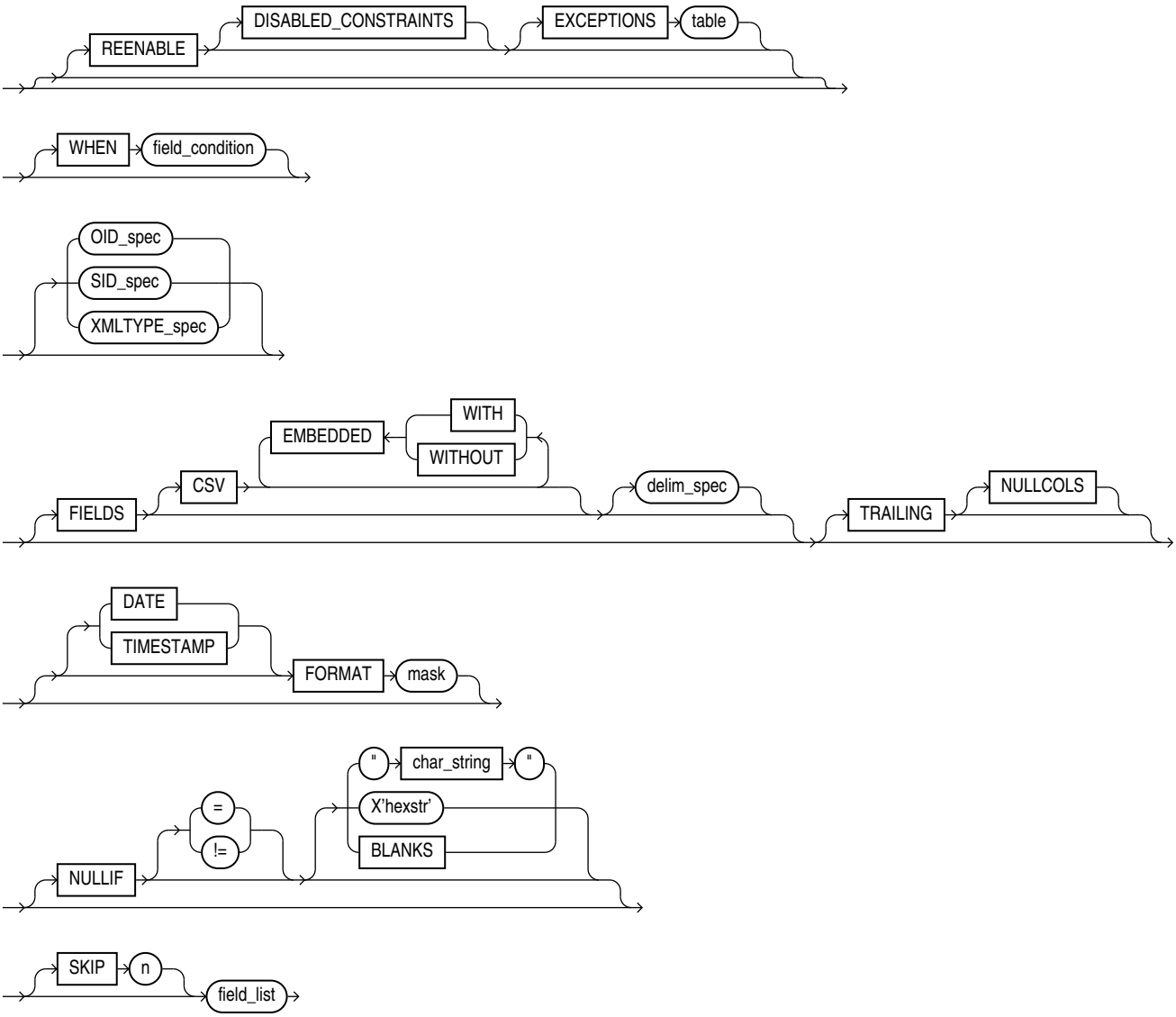
concatenate_clause



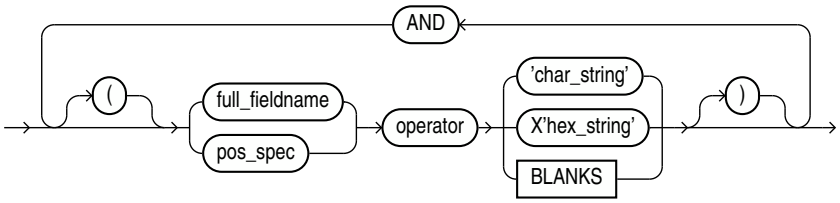
into_table_clause



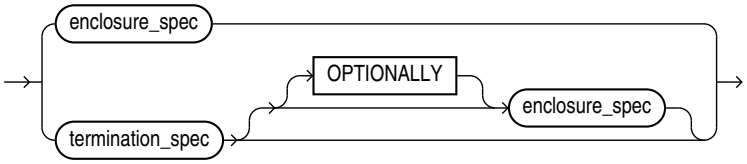
into_table_clause_continued



field_condition

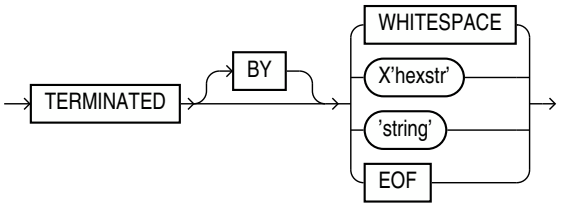


delim_spec

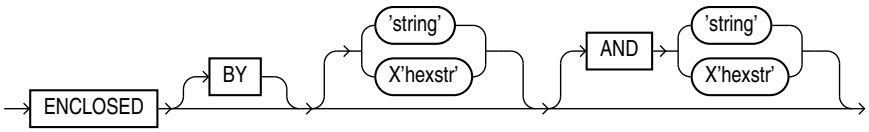


full_fieldname

termination_spec

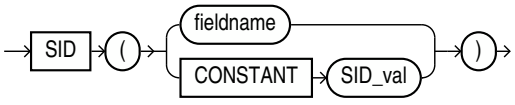


enclosure_spec

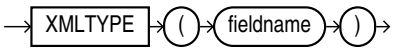


oid_spec

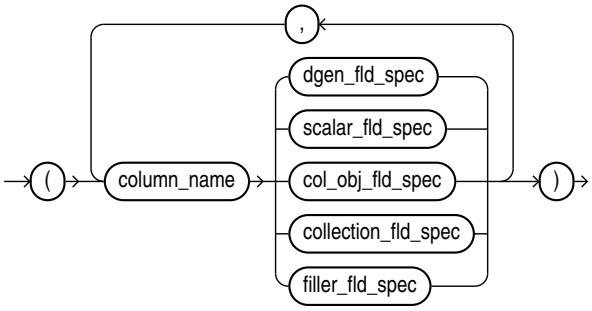
sid_spec



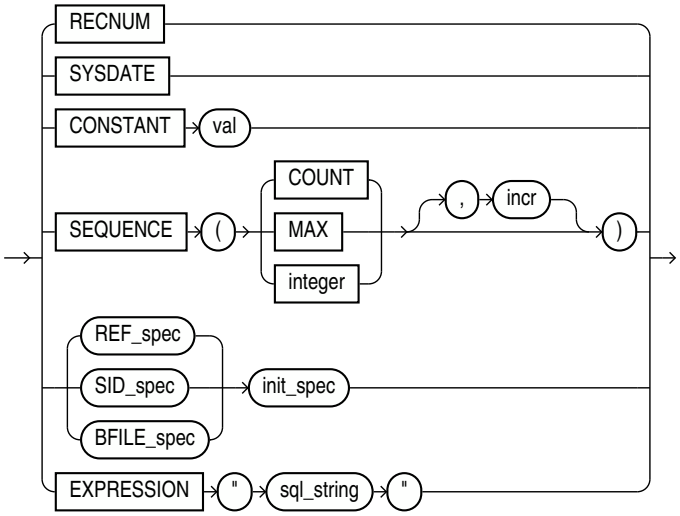
xmltype_spec



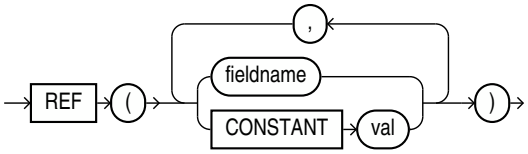
field_list



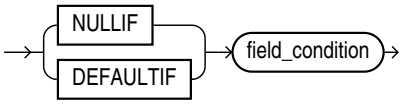
dgen_fld_spec



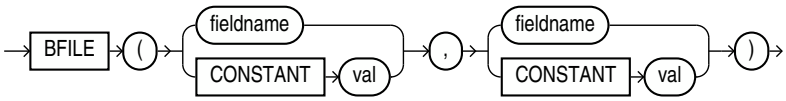
ref_spec



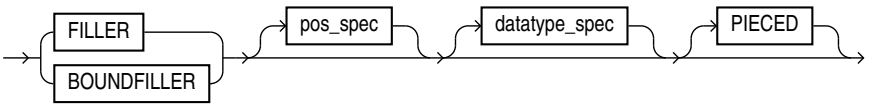
init_spec



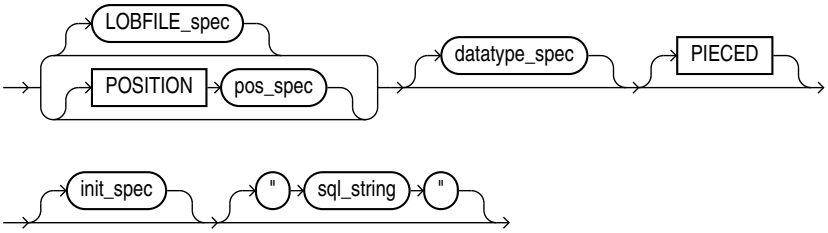
bfile_spec



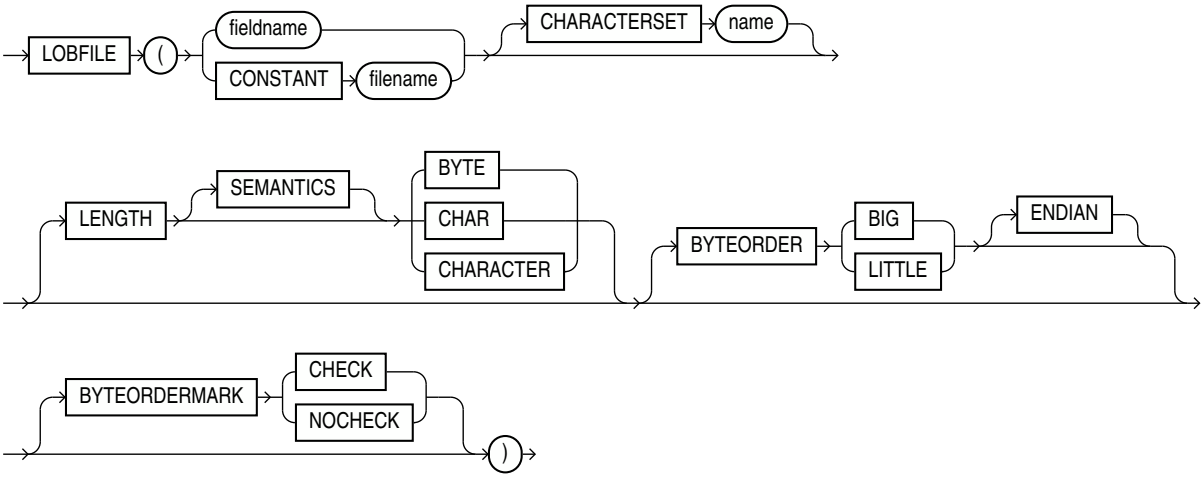
filler_fld_spec



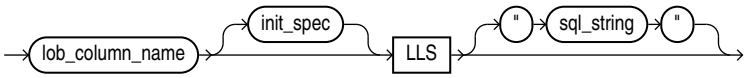
scalar_fld_spec



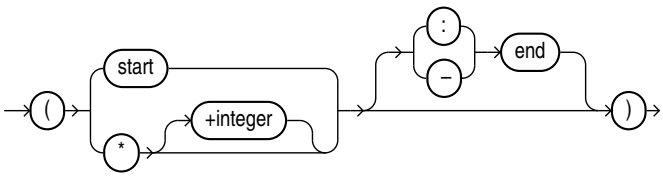
lobfile_spec



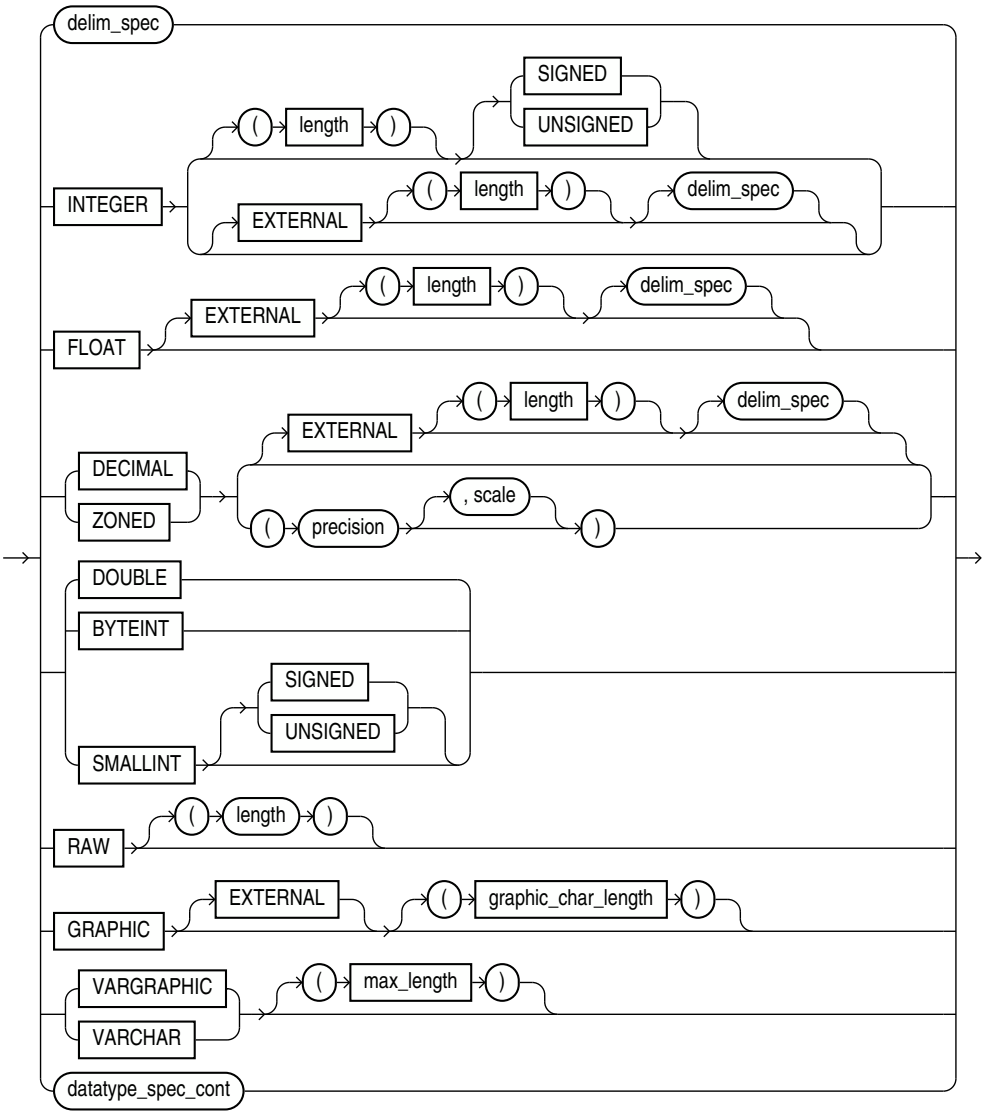
lls_field_spec



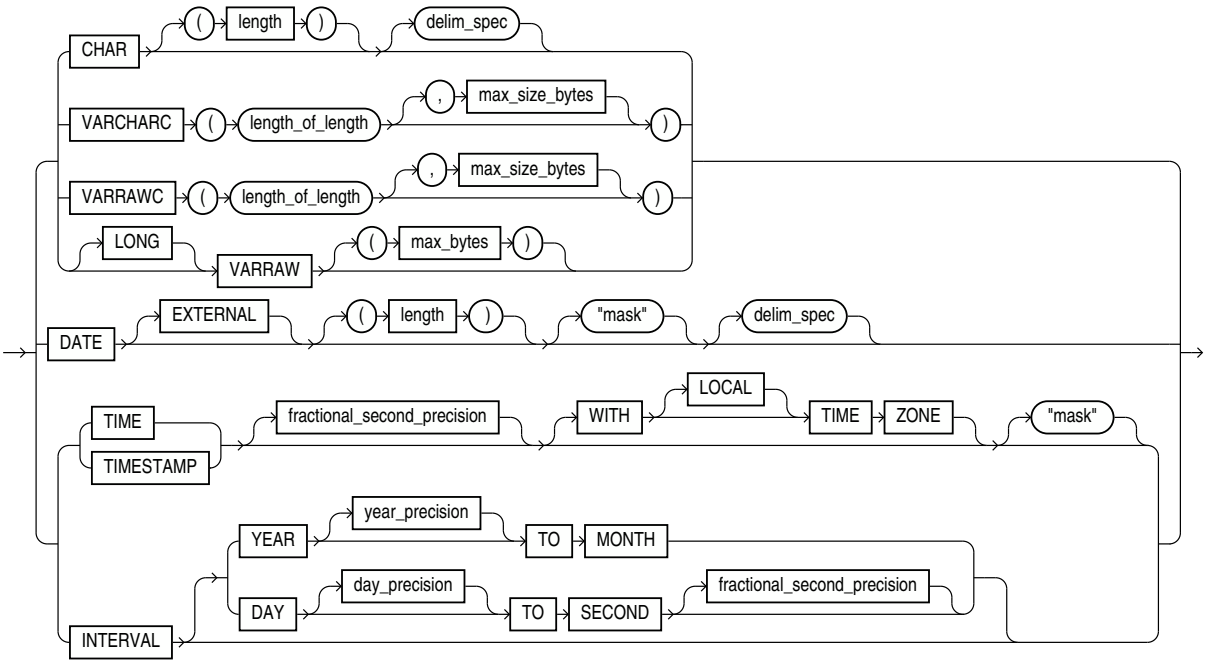
pos_spec



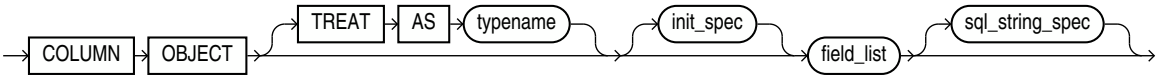
datatype_spec



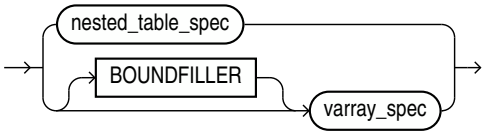
datatype_spec_cont



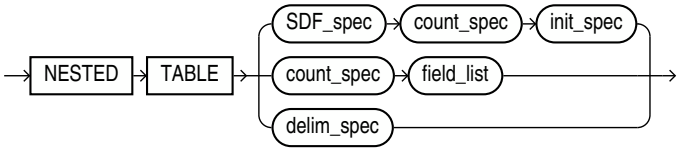
col_obj_fld_spec



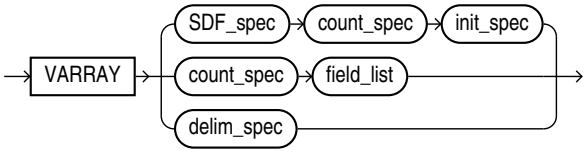
collection_fld_spec



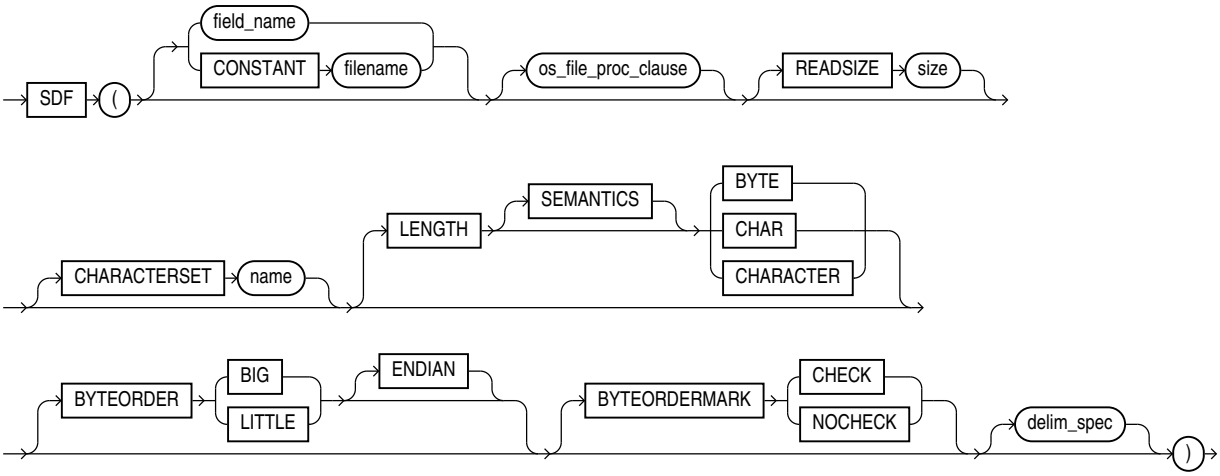
nested_table_spec



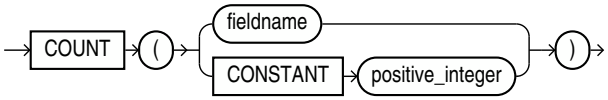
varray_spec



sdf_spec



count_spec



B

Instant Client for SQL*Loader, Export, and Import

Oracle Instant Client allows you to run your applications without installing the standard Oracle client or having an Oracle home.

The following topics are discussed:

- What is the Tools Instant Client?
  The Tools Instant Client package is available on platforms that support the Oracle Call Interface (OCI) Instant Client.
- Choosing the Instant Client to Install
  Before you install the Tools Instant Client Tools package, you must first choose either Basic Instant Client or Instant Client Light.
- Installing Tools Instant Client by Downloading from OTN
  The OTN downloads for Linux are RPM packages. The OTN downloads for UNIX and Windows are zip files.
- Installing Tools Instant Client from the 12c Client Release Media
  The Tools Instant Client package can be installed from client release media.
- Configuring Tools Instant Client Package
  The Tools Instant Client package executable should only be used with the matching version of the OCI Instant Client. No `ORACLE_HOME` or `ORACLE_SID` environment variables need to be set.
- Connecting to a Database with the Tools Instant Client Package
  After the Tools Instant Client package is installed and configured, you can connect to a database.
- Uninstalling Instant Client
  The Tools Instant Client package can be removed separately from the OCI Instant Client.

B.1 What is the Tools Instant Client?

The Tools Instant Client package is available on platforms that support the Oracle Call Interface (OCI) Instant Client.

The Tools package contains several command-line utilities, including SQL*Loader, Oracle Data Pump Export, Oracle Data Pump Import, Original (classic) Export, and Original (classic) Import. Instant Client installations are standalone, with all of the functionality of the command-line versions of the products. The Instant Client connects to existing remote Oracle Databases, but does not include its own database. It is easy to install, and uses significantly less disk space than the full Oracle Database Client installation required to use the command-line versions of products.

Overview of Steps Required to use Tools Instant Client

To use the Tools Instant Client, you need two packages:

- Tools Instant Client Package

- Either the Basic OCI Instant Client package, or the OCI Instant Client Light package.

The basic steps required to use the Tools Instant Client are as follows. Each of these steps is described in this appendix.

1. Choose which OCI Package (Basic or Light) you want to use, and also select the directory in which to install the Instant Client files.
2. Copy the Tools Instant Client Package, and the OCI Instant Client package of your choice, from an installed Oracle instance or download them from OTN.
3. Install (unpack) the Tools Instant Client package and the OCI package. A new directory `instantclient_12_2` is created as part of the installation.
4. Configure the Instant Client.
5. Connect to a remote instance with the utility you want to run.

Both the Tools package and OCI package must be from Oracle Database version 12.2.0.0.0, or higher, and the versions for both must be the same.

See *Oracle Call Interface Programmer's Guide* for more information about the OCI Instant Client.

Related Topics

- About Oracle Instant Client

B.2 Choosing the Instant Client to Install

Before you install the Tools Instant Client Tools package, you must first choose either Basic Instant Client or Instant Client Light.

Basic Instant Client

The Tools Instant Client package, when used with Basic Instant Client works with any `NLS_LANG` setting supported by the Oracle Database. It supports all character sets and language settings available in the Oracle Database.

Instant Client Light

The Instant Client Light (English) version of Instant Client further reduces the disk space requirements of the client installation. The size of the library has been reduced by removing error message files for languages other than English and leaving only a few supported character set definitions out of around 250.

Instant Client Light is geared toward applications that use either `US7ASCII`, `WE8DEC`, `WE8ISO8859P1`, `WE8MSWIN1252`, or a Unicode character set. There is no restriction on the `LANGUAGE` and the `TERRITORY` fields of the `NLS_LANG` setting, so the Instant Client Light operates with any language and territory settings. Because only English error messages are provided with the Instant Client Light, error messages generated on the client side, such as Net connection errors, are always reported in English, even if `NLS_LANG` is set to a language other than `AMERICAN`. Error messages generated by the database side, such as syntax errors in SQL statements, are in the selected language provided the appropriate translated message files are installed in the Oracle home of the database instance.

B.3 Installing Tools Instant Client by Downloading from OTN

The OTN downloads for Linux are RPM packages. The OTN downloads for UNIX and Windows are zip files.

Instant Client packages should never be installed on an Oracle home.

Installing Instant Client from Linux RPM Packages

1. Download the RPM packages containing the Tools Instant Client package, and the OCI package from the OTN Instant Client page at http://www.oracle.com/technetwork/database/database-technologies/instant-client/overview/index.html. Both packages must be version 12.2.0.0.0 or higher, and the versions of both must be the same.
2. Use `rpm -i` for the initial install of the RPM packages, or `rpm -u` to upgrade to a newer version of the packages. Install the OCI Package first.
3. Configure Instant Client. See Configuring Tools Instant Client Package.

Installing Instant Client from the UNIX or Windows Zip Files

1. Download the RPM packages containing the Tools Instant Client package, and the OCI package from the OTN Instant Client page at http://www.oracle.com/technetwork/database/database-technologies/instant-client/overview/index.html. Both packages must be version 12.2.0.0.0 or higher, and the versions of both must be the same.
2. Create a new directory, for example, `/home/instantclient12_2` on UNIX or `c:\instantclient12_2` on Windows.
3. Unzip the two packages into the new directory. Install the OCI Package first.
4. Configure Instant Client. See Configuring Tools Instant Client Package

List of Files Required for Tools Instant Client

The following table lists the required files from the Tools Instant Client package. You also need the files from one of the OCI packages (either Basic or Light). Other files installed that are not listed here can be ignored, or can be removed to save disk space.

See the *Oracle Call Interface Programmer's Guide* for more information about required files for the OCI Instant Client.

Table B-1 Instant Client Files in the Tools Package

Linux and UNIX	Windows	Description
exp	exp.exe	Original (classic) export executable
expdp	expdp.exe	Data Pump export executable
imp	imp.exe	Original (classic) import executable
impdp	impdp.exe	Data Pump import executable
libnfsodm12.so	Not applicable	A shared library used by the SQL*Loader Instant Client to use the Oracle Disk Manager (ODM).
sqlldr	sqlldr.exe	SQL*Loader executable
TOOLS_README	Not applicable	Readme for the Tools Instant Client package

Table B-1 (Cont.) Instant Client Files in the Tools Package

Linux and UNIX	Windows	Description
wrc	wrc.exe	The Tools Instant Client package contains tools other than those described in this appendix. The wrc tool is the Workload Replay Client (wrc) for the Oracle Database Replay feature. The wrc tool is listed here for completeness sake, but it is not covered by the information in this appendix.

B.4 Installing Tools Instant Client from the 12*c* Client Release Media

The Tools Instant Client package can be installed from client release media.

1. Run the installer on the Oracle Database 12*c* Client Release media and choose the Administrator option.
2. Create a new directory, for example, `/home/instantclient12_2` on UNIX and Linux, or `c:\instantclient12_2` on Windows.
3. Copy the Tools Instant Client package and the OCI Instant Client files to the new directory. All files must be copied from the same Oracle home.
4. Configure the Tools Instant Client package. See Configuring Tools Instant Client Package.

Installing Tools Instant Client on UNIX or Linux

To install the Tools Instant Client using the Basic Instant Client package on UNIX and Linux, copy the following files:

```
$ORACLE_HOME/instantclient/libociei.so
$ORACLE_HOME/lib/libnfsodm12.so
$ORACLE_HOME/bin/exp
$ORACLE_HOME/bin/imp
$ORACLE_HOME/bin/expdp
$ORACLE_HOME/bin/impdp
$ORACLE_HOME/bin/sqlldr
```

To install the Tools Instant Client using the Instant Client Light package on UNIX and Linux, copy the following files:

```
$ORACLE_HOME/instantclient/light/libociicus.so
$ORACLE_HOME/lib/libnfsodm12.so
$ORACLE_HOME/bin/exp
$ORACLE_HOME/bin/imp
$ORACLE_HOME/bin/expdp
$ORACLE_HOME/bin/impdp
$ORACLE_HOME/bin/sqlldr
```

Installing Tools Instant Client on Windows

To install the Tools Instant Client using the Basic Instant Client package on Windows, copy the following files:

```
%ORACLE_HOME%\instantclient\oraociei12.dll
%ORACLE_HOME%\bin\exp.exe
%ORACLE_HOME%\bin\imp.exe
%ORACLE_HOME%\bin\expdp.exe
%ORACLE_HOME%\bin\impdp.exe
%ORACLE_HOME%\bin\sqlldr.exe
```

To install the Tools Instant Client using the Instant Client Light package on Windows, copy the following files:

```
ORACLE_HOME%\instantclient\light\oraociicus12.dll
%ORACLE_HOME%\bin\exp.exe
%ORACLE_HOME%\bin\imp.exe
%ORACLE_HOME%\bin\expdp.exe
%ORACLE_HOME%\bin\impdp.exe
%ORACLE_HOME%\bin\sqlldr.exe
```

B.5 Configuring Tools Instant Client Package

The Tools Instant Client package executable should only be used with the matching version of the OCI Instant Client. No `ORACLE_HOME` or `ORACLE_SID` environment variables need to be set.

Configuring Tools Instant Client Package (from RPMS) on Linux

The RPMs downloaded from OTN install into Oracle specific sub-directories in the `/usr` file system. The subdirectory structure enables multiple versions of Instant Client to be available.

1. Add the name of the directory containing the Instant Client libraries to `LD_LIBRARY_PATH`. Remove any other Oracle directories.

   For example, to set `LD_LIBRARY_PATH` in the Bourne or Korn shells, use the following syntax:

   ```
   LD_LIBRARY_PATH=/usr/lib/oracle/18.1/client/lib:${LD_LIBRARY_PATH}
   export LD_LIBRARY_PATH
   ```

   Or, to set `LD_LIBRARY_PATH` in the C shell, use the following syntax:

   ```
   % setenv LD_LIBRARY_PATH /usr/lib/oracle/18.1/client/lib:$LD_LIBRARY_PATH
   ```

2. Make sure the Tools executables installed from the RPM are the first executables found in your `PATH`. For example, to test this you could enter `which sqlldr` which should return `/usr/bin/sqlldr`. If it does not, then remove any other Oracle directories from `PATH`, or put `/usr/bin` before other Tools executables in `PATH`, or use an absolute or relative path to start Tools Instant Client.

For example, to set `PATH` in the bash shell:

```
PATH=/usr/bin:${PATH}
export PATH
```

3. Set Oracle globalization variables required for your locale. A default locale will be assumed if no variables are set.

```
NLS_LANG=AMERICAN_AMERICA.UTF8
export NLS_LANG
```

Configuring Tools Instant Client Package (from Client Media or Zip File) on Linux and UNIX

1. Add the name of the directory containing the Instant Client files to the appropriate shared library path `LD_LIBRARY_PATH`, `LIBPATH` or `SHLIB_PATH`. Remove any other Oracle directories.

   For example on Solaris in the Bourne or Korn shells:

```
LD_LIBRARY_PATH=/home/instantclient12_2:${LD_LIBRARY_PATH}
export LD_LIBRARY_PATH
```

2. Add the directory containing the Instant Client files to the `PATH` environment variable. If it is not set, then an absolute or relative path must be used to start the utilities provided in the Tools package. Remove any other Oracle directories from `PATH`. For example:

```
PATH=/home/instantclient12_2:${PATH}
export PATH
```

3. Set Oracle globalization variables required for your locale. A default locale will be assumed if no variables are set. For example:

```
NLS_LANG=AMERICAN_AMERICA.UTF8
export NLS_LANG
```

Configuring Tools Instant Client Package on Windows

The environment may be configured using `SET` commands in a Windows command prompt or made permanent by setting Environment Variables in System Properties.

For example, to set environment variables in Windows 2000 using System Properties, open **System** from the Control Panel, click the **Advanced** tab and then click **Environment Variables**.

1. Add the directory containing the Instant Client files to the `PATH` system environment variable. Remove any other Oracle directories from `PATH`.

   For example, add `c:\instantclient12_2` to the beginning of `PATH`.

2. Set Oracle globalization variables required for your locale. A default locale will be assumed if no variables are set.

   For example, to set `NLS_LANG` for a Japanese environment, create a user environment variable `NLS_LANG` set to `JAPANESE_JAPAN.JA16EUC`.

B.6 Connecting to a Database with the Tools Instant Client Package

After the Tools Instant Client package is installed and configured, you can connect to a database.

The utilities supplied in the Tools Instant Client are always remote from any database server. It is assumed that the server has an Oracle instance up and running and has the TNS listener running. For the Data Pump Export and Import clients, the dump files reside on the remote server; an Oracle directory object on the server must exist and should have the appropriate permissions.

To connect to a database you must specify the database using an Oracle Net connection identifier. The following information uses the SQL*Loader (sqlldr) utility, but the information applies to other utilities supplied in the Tools Instant Client package as well.

An example using an Easy Connection identifier to connect to the `HR` schema in the `MYDB` database running on *mymachine* is:

```
sqlldr hr/your_password@\"//mymachine.mydomain:port/MYDB\"
```

Alternatively you can use a Net Service Name:

```
sqlldr hr/your_password@MYDB
```

Net Service Names can be stored in a number of places, including LDAP. The use of LDAP is recommended to take advantage of the new features of Oracle Database 12*c*

To use Net Service Names configured in a local Oracle Net `tnsnames.ora` file, set the environment variable `TNS_ADMIN` to the directory containing the `tnsnames.ora` file. For example, on UNIX, if your `tnsnames.ora` file is in `/home/user1` and it defines the Net Service Name `MYDB2`:

```
TNS_ADMIN=/home/user1
export TNS_ADMIN
sqlldr hr@MYDB2
```

If `TNS_ADMIN` is not set, then an operating system dependent set of directories is examined to find `tnsnames.ora` . This search path includes looking in the directory specified by the `ORACLE_HOME` environment variable for `network/admin/tnsnames.ora` . This is the only reason to set the `ORACLE_HOME` environment variable for SQL*Loader Instant Client. If `ORACLE_HOME` is set when running Instant Client applications, it must be set to a directory that exists.

This example assumes the `ORACLE_HOME` environment variable is set, and the `$ORACLE_HOME/network/admin/tnsnames.ora` or `ORACLE_HOME\network\admin\tnsnames.ora` file defines the Net Service Name `MYDB3`:

```
sqlldr hr@MYDB3
```

The `TWO_TASK` (on UNIX) or `LOCAL` (on Windows) environment variable can be set to a connection identifier. This removes the need to explicitly enter the connection identifier

whenever a connection is made in SQL*Loader or SQL*Loader Instant Client. This UNIX example connects to the database known as `MYDB4`:

```
TNS_ADMIN=/home/user1
export TNS_ADMIN
TWO_TASK=MYDB4
export TWO_TASK
sqlldr hr
```

On Windows, `TNS_ADMIN` and `LOCAL` may be set in the System Properties.

B.7 Uninstalling Instant Client

The Tools Instant Client package can be removed separately from the OCI Instant Client.

After uninstalling the Tools Instant Client package, the remaining OCI Instant Client libraries enable custom written OCI programs or third party database utilities to connect to a database.

Uninstalling Tools Instant Client

1. For installations on Linux from RPM packages, use `rpm -e` only on the Tools Instant Client package
   OR

   For installations on UNIX and Windows, and installations on Linux from the Client Release media, manually remove any files specific to the Tools Instant Client. The files to be deleted should be in the Instant Client directory that you specified at installation. Be sure you do not remove any Oracle home files.

   Reset environment variables and remove `tnsnames.ora` if necessary.

Uninstalling the Complete Instant Client

1. For installations on Linux from RPM packages, use `rpm -qa` to find the Tools Instant Client and Basic OCI package names and run `rpm -e`to remove them

   OR

   For installations on UNIX and Windows, and installations on Linux from the Client Release media, manually delete the directory containing the Tools executable and Oracle libraries.
2. Reset environment variables such as `PATH`, `LD_LIBRARY_PATH` and `TNS_ADMIN`.
3. Remove `tnsnames.ora` if necessary.

Index

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X