Oracle® Database

SecureFiles and Large Objects Developer's Guide

12*c* Release 1 (12.1)

E17605-11

August 2014

ORACLE®

Oracle Database SecureFiles and Large Objects Developer's Guide, 12*c* Release 1 (12.1)

E17605-11

Primary Authors: Janis Greenberg, Roza Leyderman

Contributors: Bharath Aleti, Geeta Arora, Thomas H. Chang, Maria Chien, Subramanyam Chitti, Amit Ganesh, Kevin Jernigan, Vikram Kapoor, Balaji Krishnan, Jean de Lavarene, Geoff Lee, Scott Lynn, Jack Melnick, Atrayee Mullick, Eric Paapanen, Ravi Rajamani, Kam Shergill, Ed Shirk, Srinivas Vemuri

Contents

3 Managing LOBs: Database Administration

4 Using Oracle LOB Storage

8 DBFS Content API

9 Creating Your Own DBFS Store

10 Using DBFS

12 Advanced Design Considerations

13 Overview of Supplied LOB APIs

14 Performance Guidelines

Part IV SQL Access to LOBs

15 DDL and DML Statements with LOBs

16 SQL Semantics and LOBs

21 LOB APIs for BFILE Operations

22 Using LOB APIs

A LOB Demonstration Files

Preface

This guide describes database features that support application development using SecureFiles and Large Object (LOB) data types and Database File System (DBFS). The information in this guide applies to all platforms, and does not include system-specific information.

Audience

Oracle Database SecureFiles and Large Objects Developer's Guide is intended for programmers who develop new applications using LOBs and DBFS, and those who have previously implemented this technology and now want to take advantage of new features.

Efficient and secure storage of multimedia and unstructured data is increasingly important, and this guide is a key resource for this topic within the Oracle Application Developers documentation set.

Feature Coverage and Availability

Oracle Database SecureFiles and Large Objects Developer's Guide contains information that describes the SecureFiles LOB and BasicFiles LOB features and functionality of Oracle Database 12*c* Release 1 (12.1).

Prerequisites for Using LOBs

Oracle Database includes all necessary resources for using LOBs in an application; however, there are some restrictions, described in "LOB Rules and Restrictions" on page 2-6 and "Restrictions for LOBs in Partitioned Index-Organized Tables" on page 11-17.

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 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.

Related Documents

For more information, see the following manuals:

- *Oracle Database 2 Day Developer's Guide*
- *Oracle Database Development Guide*
- *Oracle Database Utilities*
- *Oracle XML DB Developer's Guide*
- *Oracle Database PL/SQL Packages and Types Reference*
- *Oracle Database Data Cartridge Developer's Guide*
- *Oracle Call Interface Programmer's Guide*
- *Oracle C++ Call Interface Programmer's Guide*
- *Pro*C/C++ Programmer's Guide*
- *Pro*COBOL Programmer's Guide*
- *Programmer's Guide to the Oracle Precompilers*
- *Pro*Fortran Supplement to the Oracle Precompilers Guide*

Java

The Oracle Java documentation set includes the following:

- *Oracle Database JDBC Developer's Guide*
- *Oracle Database Java Developer's Guide*
- *Oracle Database JPublisher User's Guide*

Oracle Multimedia

To use Oracle Multimedia applications, refer to the following:

- *Oracle Multimedia Reference*
- *Oracle Multimedia User's Guide*

Basic References

To download free release notes, installation documentation, white papers, or other collateral, please visit the Oracle Technology Network (OTN)

`http://www.oracle.com/technetwork/index.html`

For the latest version of the Oracle documentation, including this guide, visit

`http://www.oracle.com/technetwork/documentation/index.html`

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.

Convention	Meaning
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

Changes in This Release for Oracle Database SecureFiles and Large Objects Developer's Guide

This preface contains:

Changes in Oracle Database 12*c* Release 1 (12.1)

The following are changes in *Oracle Database SecureFiles and Large Objects Developer's Guide* for Oracle Database 12*c* Release 1 (12.1).

New Features

- **Support for enabling parallel DML operations on SecureFiles LOBs**

  Parallel DML Support for SecureFiles LOBs is enhanced, providing improved performance.

  See "Parallel DML (PDML) Support for LOBs" on page 20-7.

- **SecureFiles LOBs is now the default LOB storage**

  SecureFiles LOB is the default storage option for LOBs, instead of BasicFiles LOB. See "About LOB Storage" on page 4-1 for details.

- Increased size limit for `VARCHAR2`, `NVARCHAR2`, and `RAW` data types

  With this release, the maximum size of SQL data types `VARCHAR2`, `NVARCHAR2`, and `RAW` change respectively from 4,000 bytes and 2,000 bytes to 32,767 bytes each. The corresponding PL/SQL data types remain at 32,767 bytes.

  See *Oracle Database SQL Language Reference* to determine how to handle this size change in your applications.

- **Data Pump uses SecureFiles as default LOB storage**

  When you import tables, you can recreate all LOB columns as SecureFiles LOBs, thus converting BasicFiles LOBs to SecureFiles LOBs as part of data pump imports.

  See "Using Oracle Data Pump to Transfer LOB Data" on page 3-4.

- **Support for HTTP, WebDAV, and FTP Access to DBFS**

SecureFiles supports components that enable HTTP, WebDAV, and FTP access to DBFS over the Internet, using various XML DB server protocols

See "HTTP, WebDAV, and FTP Access to DBFS" on page 10-14.

Other Changes

The following are additional changes in the release:

- Chapters 1 to 10 have been renumbered and the content reorganized and rewritten.

Part I

Getting Started

This part introduces Large Objects (LOBs) and discusses general concepts for using them in your applications.

This part contains these chapters:

1

Introduction to Large Objects and SecureFiles

This chapter introduces Large Objects (LOBs), SecureFiles LOBs, and Database File System (DBFS) and discusses how LOB data types are used in application development.

Large Objects are used to hold large amounts of data inside Oracle Database, SecureFiles provides performance equal to or better than file system performance when using Oracle Database to store and manage Large Objects, and DBFS provides file system access to files stored in Oracle Database.

This chapter contains these topics:

What Are Large Objects?

Large Objects (LOBs) are a set of data types that are designed to hold large amounts of data. The maximum size for a single LOB can range from 8 terabytes to 128 terabytes depending on how your database is configured. Storing data in LOBs enables you to access and manipulate the data efficiently in your application.

Why Use Large Objects?

This section describes different types of data that you encounter when developing applications and discusses which kinds of data are suitable for large objects.

In the world today, applications must deal with the following kinds of data. Note that large objects are suitable for the last two: semistructured and unstructured.

- Simple structured data

  Simple structured data can be organized into relational tables that are structured based on business rules.

- Complex structured data

  Complex structured data is more complex than simple structured data and is suited for the object-relational features of the Oracle database such as collections, references, and user-defined types.

- Semistructured data

  Semistructured data has a logical structure that is not typically interpreted by the database, for example, an XML document that your application or an external service processes. Oracle Database provides features such as Oracle XML DB, Oracle Multimedia DICOM, and Oracle Spatial and Graph to help your application work with semistructured data.

- Unstructured data

  Unstructured data is easily not broken down into smaller logical structures and is not typically interpreted by the database or your application, such as a photographic image stored as a binary file.

Large objects are suitable for semistructured and unstructured data. Large object features allow you to store these kinds of data in the database and in operating system files that are accessed from the database.

With the growth of the Internet and content-rich applications, it has become imperative for Oracle Database to provide LOB support that:

- Can store unstructured and semistructured data in an efficient manner
- Is optimized for large amounts of data
- Provides a uniform way of accessing data stored within the database or outside the database

Using LOBs for Semistructured Data

Semistructured data include document files such as XML documents or word processor files, which contain data in a logical structure that is processed or interpreted by an application, and is not broken down into smaller logical units when stored in the database.

Applications that use semistructured data often use large amounts of character data. The Character Large Object (`CLOB`) and National Character Large Object (`NCLOB`) data types are ideal for storing and manipulating this kind of data.

Binary File objects (`BFILE` data types) can also store character data. You can use `BFILE`s to load read-only data from operating system files into `CLOB` or `NCLOB` instances that you then manipulate in your application.

Using LOBs for Unstructured Data

Unstructured data is data that cannot be decomposed into standard components.

This is in contrast to structured data, such as data about an employee typically containing these components: a name, stored as a string; an identifier, such as an ID number; a salary; and so on.

Unstructured data, such as a photograph, consists of a long stream of 1s and 0s. These bits are used to switch pixels on or off so that you can see the picture on a display, but the bits are not broken down into any standard components for database storage.

Also, unstructured data such as text, graphic images, still video clips, full motion video, and sound waveforms tends to be large in size. A typical employee record may be a few hundred bytes, while even small amounts of multimedia data can be thousands of times larger.

SQL data types that are ideal for large amounts of unstructured binary data include the `BLOB` data type (Binary Large Object) and the `BFILE` data type (Binary File object).

Why Not Use LONGs?

The database supports `LONG` and LOB data types. When possible, change your existing applications to use LOBs instead of `LONG`s because of the added benefits that LOBs provide. `LONG`-to-LOB migration enables you to easily migrate your existing applications that access `LONG` columns, to use LOB columns.

See Also: Chapter 18, "Migrating Columns from LONGs to LOBs"

Applications developed for use with Oracle7 and earlier used the `LONG` or `LONG RAW` data type to store large amounts of unstructured data.

With Oracle8*i* and later versions of the database, using LOB data types is recommended for storing large amounts of structured and semistructured data. LOB data types have several advantages over `LONG` and `LONG RAW` types including:

- LOB Capacity: LOBs can store much larger amounts of data. LOBs can store 4 GB of data or more depending on your system configuration. `LONG` and `LONG RAW` types are limited to 2 GB of data.
- Number of LOB columns in a table: A table can have multiple LOB columns. LOB columns in a table can be of any LOB type. In Oracle7 Release 7.3 and higher, tables are limited to a single `LONG` or `LONG RAW` column.
- Random piece-wise access: LOBs support random access to data, but `LONG`s support only sequential access.
- LOBs can also be object attributes.

Different Kinds of LOBs

Different kinds of LOBs can be stored in the database or in external files.

Note: LOBs in the database are sometimes also referred to as *internal LOBs* or *internal persistent LOBs*.

Internal LOBs

LOBs in the database are stored inside database tablespaces in a way that optimizes space and provides efficient access. The following SQL data types are supported for declaring internal LOBs: `BLOB`, `CLOB`, and `NCLOB`. Details on these data types are given in "Large Object Data Types" on page 1-5.

Persistent and Temporary LOBs

Internal LOBs (LOBs in the database) can be either persistent or temporary. A persistent LOB is a LOB instance that exists in a table row in the database. A temporary LOB instance is created when you instantiate a LOB only within the scope of your local application.

A temporary instance becomes a persistent instance when you insert the instance into a table row.

Persistent LOBs use copy semantics and participate in database transactions. You can recover persistent LOBs in the event of transaction or media failure, and any changes to a persistent LOB value can be committed or rolled back. In other words, all the Atomicity, Consistency, Isolation, and Durability (ACID) properties that apply to database objects apply to persistent LOBs.

External LOBs and the BFILE Data Type

External LOBs are data objects stored in operating system files, outside the database tablespaces.

`BFILE` is the SQL data type that the database uses to access external LOBs and is the only SQL data type available for external LOBs.

`BFILE`s are read-only data types. The database allows read-only byte stream access to data stored in `BFILE`s. You cannot write to or update a `BFILE` from within your application.

The database uses reference semantics with `BFILE` columns. Data stored in a table column of type `BFILE` is physically located in an operating system file, not in the database.

You typically use `BFILE`s to hold:

- Binary data that does not change while your application is running, such as graphics
- Data that is loaded into other large object types, such as a `BLOB` or `CLOB`, where the data can then be manipulated
- Data that is appropriate for byte-stream access, such as multimedia

Any storage device accessed by your operating system can hold `BFILE` data, including hard disk drives, CD-ROMs, PhotoCDs, and DVDs. The database can access `BFILE`s provided the operating system supports stream-mode access to the operating system files.

> **Note:** External LOBs do not participate in transactions. Any support for integrity and durability must be provided by the underlying file system as governed by the operating system.

LOB Locators

A LOB instance has a locator and a value. The LOB locator is a reference to where the LOB value is physically stored. The LOB value is the data stored in the LOB.

When you use a LOB in an operation such as passing a LOB as a parameter, you are actually passing a LOB locator. For the most part, you can work with a LOB instance in your application without being concerned with the semantics of LOB locators. There is

no requirement to dereference LOB locators, as is required with pointers in some programming languages.

See Also:

- "LOB Locator and LOB Value" on page 2-2
- "LOB Locators and BFILE Locators" on page 2-3
- "LOB Storage Parameters" on page 11-4

Database Semantics for Internal and External LOBs

In all programmatic environments, database semantics differ between internal LOBs and external LOBs as follows:

- Internal LOBs use *copy semantics*

  With copy semantics, both the LOB locator and LOB value are logically copied during insert, update, or assignment operations. This ensures that each table cell or each variable containing a LOB, holds a unique LOB instance.

- External LOBs use *reference semantics*

  With reference semantics, only the LOB locator is copied during insert operations. Note that update operations do not apply to external LOBs because external LOBs are read-only as described in "External LOBs and the BFILE Data Type" on page 1-4.

Large Object Data Types

The database provides a set of large object data types as SQL data types where the term *LOB* generally refers to the set. In general, the descriptions given for the data types in this table and the rest of this book also apply to the corresponding data types provided for other programmatic environments.

Table 1–1 describes each large object data type that the database supports and describes the kind of data that uses it.

Table 1–1 Large Object Data Types

SQL Data Type	Description
`BLOB`	Binary Large Object Stores any kind of data in binary format. Typically used for multimedia data such as images, audio, and video.
`CLOB`	Character Large Object Stores string data in the database character set format. Used for large strings or documents that use the database character set exclusively. Characters in the database character set are in a fixed width format.
`NCLOB`	National Character Set Large Object Stores string data in National Character Set format, typically large strings or documents. Supports characters of varying width format.

Table 1–1 (Cont.) Large Object Data Types

SQL Data Type	Description
BFILE	External Binary File
	A binary file stored outside of the database in the host operating system file system, but accessible from database tables. BFILEs can be accessed from your application on a **read-only** basis. Use BFILEs to store static data, such as image data, that is not manipulated in applications.
	Any kind of data, that is, any operating system file, can be stored in a BFILE. For example, you can store character data in a BFILE and then load the BFILE data into a CLOB, specifying the character set upon loading.

Object Data Types and LOBs

You can declare LOB data types as fields, or members, of object data types. For example, you can have an attribute of type CLOB on an object type. In general, there is no difference in the use of a LOB instance in a LOB column or as a member of an object data type. When used in this guide, the term **LOB attribute** refers to a LOB instance that is a member of an object data type. Unless otherwise specified, discussions that apply to LOB columns also apply to LOB attributes.

Storing and Creating Other Data Types with LOBs

You can use LOBs to create other user-defined data types or store other data types as LOBs. This section discusses some of the data types provided with the database as examples of data types that are stored or created with LOB types.

VARRAYs Stored as LOBs

An instance of type VARRAY in the database is stored as an array of LOBs when you create a table in the following scenarios:

- If the VARRAY storage clause is not specified, and the declared size of varray data is more than 4000 bytes: `VARRAY varray_item STORE AS`
- If the VARRAY column properties are specified using the STORE AS LOB clause: `VARRAY varray_item STORE AS LOB ...`

LOBs Used in Oracle Multimedia

Oracle Multimedia uses LOB data types to create object types specialized for use in multimedia application. Multimedia data types include ORDAudio, ORDDoc, ORDImage, ORDVideo, and ORDDicom. Oracle Multimedia uses the database infrastructure to define object types, methods, and LOBs necessary to represent these specialized types of data in the database.

See Also:

- *Oracle Multimedia User's Guide* for more information about Oracle Multimedia
- *Oracle Multimedia Reference* for more information about Oracle Multimedia data types

BasicFiles and SecureFiles LOBs

SecureFiles LOB storage is one of two storage types used with Oracle Database 12c; the other type is BasicFiles LOB storage. Certain advanced features can be applied to SecureFiles LOBs, including compression and deduplication (part of the Advanced Compression Option), and encryption (part of the Advanced Security Option).

SecureFiles LOBs can only be created in a tablespace managed with Automatic Segment Space Management (ASSM).

SecureFiles is the default storage mechanism for LOBs starting with Oracle Database 12*c*, and Oracle strongly recommends SecureFiles for storing and managing LOBs, rather then BasicFiles. BasicFiles will be deprecated in a future release.

> **See Also:** "Using Oracle LOB Storage" on page 4-1 for a discussion of both storage types

Database File System (DBFS)

Database File System (DBFS) provides a file system interface to files that are stored in an Oracle Database. The files are usually stored as SecureFiles LOBs, and pathnames, directories, and other filesystem information is stored in database tables. SecureFiles LOBs is the default storage method for DBFS, but BasicFiles LOBs can be used in some situations.

> **See Also:** "What is Database File System (DBFS)?" on page 5-1

With DBFS, you can make references from SecureFiles LOB locators to files stored outside the database. These references are called DBFS Links or Database File System Links.

> **See Also:** "Database File System Links" on page 7-13

2

Working with LOBs

This chapter describes the usage and semantics of LOBs required for application development and covers various techniques for working with LOBs.

Most of the discussions in this chapter regarding persistent LOBs assume that you are dealing with existing LOBs in tables. The task of creating tables with LOB columns is typically performed by your database administrator.

See Also:

- Chapter 4, "Using Oracle LOB Storage" for creating LOBs using the SecureFiles paradigm
- Chapter 11, "LOB Storage with Applications" for storage parameters used in creating LOBs

This chapter contains these topics:

LOB Column States

The techniques you use when accessing a cell in a LOB column differ depending on the state of the given cell. A cell in a LOB Column can be in one of the following states:

- **NULL**

  The table cell is created, but the cell holds no locator or value.

- **Empty**

  A LOB instance with a locator exists in the cell, but it has no value. The length of the LOB is zero.

- **Populated**

  A LOB instance with a locator and a value exists in the cell.

Locking a Row Containing a LOB

You can lock a row containing a LOB to prevent other database users from writing to the LOB during a transaction. To do this, specify the `FOR UPDATE` clause when you select the row. While the row is locked, other users cannot lock or update the LOB until you end your transaction.

Opening and Closing LOBs

The LOB APIs include operations that enable you to explicitly open and close a LOB instance. You can open and close a persistent LOB instance of any type: `BLOB`, `CLOB`, `NCLOB`, or `BFILE`. You open a LOB to achieve one or both of the following results:

- Open the LOB in read-only mode

  This ensures that the LOB (both the LOB locator and LOB value) cannot be changed in your session until you explicitly close the LOB. For example, you can open the LOB to ensure that the LOB is not changed by some other part of your program while you are using the LOB in a critical operation. After you perform the operation, you can then close the LOB.

- Open the LOB in read write/mode, for persistent `BLOB`, `CLOB`, or `NCLOB` instances only

  Opening a LOB in read/write mode defers any index maintenance on the LOB column until you close the LOB. Opening a LOB in read/write mode is only useful if there is an extensible index on the LOB column, and you do not want the database to perform index maintenance every time you write to the LOB. This technique can increase the performance of your application if you are doing several write operations on the LOB while it is open.

If you open a LOB, then you must close the LOB at some point later in your session. This is the only requirement for an open LOB. While a LOB instance is open, you can perform as many operations as you want on the LOB—provided the operations are allowed in the given mode.

> **See Also:** "Opening Persistent LOBs with the OPEN and CLOSE Interfaces" on page 12-9 for details on usage of these APIs

LOB Locator and LOB Value

There are two techniques that you can use to access and modify LOB values:

- Using the Data Interface for LOBs
- Using the LOB Locator to Access and Modify LOB Values

Using the Data Interface for LOBs

You can perform bind and define operations on `CLOB` and `BLOB` columns in C applications using the data interface for LOBs in OCI. Doing so enables you to insert or select out data in a LOB column without using a LOB locator as follows:

- Use a bind variable associated with a LOB column to insert character data into a `CLOB`, or `RAW` data into a `BLOB`.
- Use a define operation to define an output buffer in your application that holds character data selected from a `CLOB` or `RAW` data selected from a `BLOB`.

See Also: Chapter 20, "Data Interface for Persistent LOBs" for more information on implicit assignment of LOBs to other data types

Using the LOB Locator to Access and Modify LOB Values

The value of a LOB instance stored in the database can be accessed through a LOB locator, a reference to the location of the LOB value. Database tables store only locators in `CLOB`, `BLOB`, `NCLOB` and `BFILE` columns. Note the following with respect to LOB locators and values:

- LOB locators are passed to various LOB APIs to access or manipulate a LOB value.
- A LOB locator can be assigned to any LOB instance of the same type.
- LOB instances are characterized as temporary or persistent, but the locator is not.

LOB Locators and BFILE Locators

There are differences between the semantics of locators for LOB types `BLOB`, `CLOB`, and `NCLOB` on one hand and the semantics of locators for the `BFILE` type on the other hand:

- For LOB types `BLOB`, `CLOB`, and `NCLOB`, the LOB column stores a locator to the LOB value. Each LOB instance has its own distinct LOB locator and also a distinct copy of the LOB value.
- For initialized `BFILE` columns, the row stores a locator to the external operating system file that holds the value of the `BFILE`. Each `BFILE` instance in a given row has its own distinct locator; however, two different rows can contain a `BFILE` locator that points to the same operating system file.

Regardless of where the value of a LOB is stored, a locator is stored in the table row of any initialized LOB column. Also, when you select a LOB from a table, the LOB returned is always a temporary LOB. For more information on locators for temporary LOBs, see "LOBs Returned from SQL Functions" on page 16-10.

Note: When the term locator is used without an identifying prefix term, it refers to both LOB locators and `BFILE` locators.

Table print_media

This guide uses the `print_media` table of the Oracle Database Sample Schema `PM` in many examples. It is defined as:

```
CREATE TABLE print_media
    ( product_id        NUMBER(6)
    , ad_id             NUMBER(6)
    , ad_composite      BLOB
    , ad_sourcetext     CLOB
    , ad_finaltext      CLOB
    , ad_fltextn        NCLOB
    , ad_textdocs_ntab  textdoc_tab
    , ad_photo          BLOB
    , ad_graphic        BFILE
    , ad_header         adheader_typ
    ) NESTED TABLE ad_textdocs_ntab STORE AS textdocs_nestedtab;
```

See Also: "Creating a Table Containing One or More LOB Columns" on page 15-1 for information about creating `print_media` and its associated tables and files

Initializing a LOB Column to Contain a Locator

LOB instances that are NULL do not have a locator. Before you can pass a LOB instance to any LOB API routine, the instance must contain a locator. For example, you can select a NULL LOB from a row, but you cannot pass the instance to the PL/SQL DBMS_LOB.READ procedure. You must initialize a LOB instance, which provides it with a locator, to make it non-NULL. Then you can pass the LOB instance.

The following topics describe initialization:

- Initializing a Persistent LOB Column
- Initializing BFILEs

Initializing a Persistent LOB Column

Before you can start writing data to a persistent LOB using supported programmatic environment interfaces such as PL/SQL, OCI, Visual Basic, or Java, you must make the LOB column/attribute non-NULL.

You can accomplish this by initializing the persistent LOB to empty, using an INSERT/UPDATE statement with the function EMPTY_BLOB for BLOBs or EMPTY_CLOB for CLOBs and NCLOBs.

Note: You can use SQL to populate a LOB column with data even if it contains a NULL value.

See Also:

- Chapter 11, "LOB Storage with Applications" for more information on initializing LOB columns
- "Programmatic Environments That Support LOBs" on page 13-1 for all supported interfaces

Running the EMPTY_BLOB() or EMPTY_CLOB() function in and of itself does not raise an exception. However, using a LOB locator that was set to empty to access or manipulate the LOB value in any PL/SQL DBMS_LOB or OCI function raises an exception.

Valid places where *empty* LOB locators may be used include the VALUES clause of an INSERT statement and the SET clause of an UPDATE statement.

See Also:

- "Directory Objects" on page 21-3 for details of CREATE DIRECTORY and BFILENAME usage
- *Oracle Database SQL Language Reference*, CREATE DIRECTORY statement

---

Note: Character strings are inserted using the default character set for the instance.

The INSERT statement in the next example uses the print_media table described in "Table print_media" on page 2-3 and does the following:

- Populates ad_sourcetext with the character string 'my Oracle'

- Sets ad_composite, ad_finaltext, and ad_fltextn to an empty value
- Sets ad_photo to NULL
- Initializes ad_graphic to point to the file my_picture located under the logical directory my_directory_object

```
CREATE OR REPLACE DIRECTORY my_directory_object AS 'oracle/work/tklocal';
INSERT INTO print_media VALUES (1726, 1, EMPTY_BLOB(),
    'my Oracle', EMPTY_CLOB(), EMPTY_CLOB(),
    NULL, NULL, BFILENAME('my_directory_object', 'my_picture'), NULL);
```

Similarly, the LOB attributes for the ad_header column in print_media can be initialized to NULL, empty, or a character/raw literal, which is shown in the following statement:

```
INSERT INTO print_media (product_id, ad_id, ad_header)
    VALUES (1726, 1, adheader_typ('AD FOR ORACLE', sysdate,
    'Have Grid', EMPTY_BLOB()));
```

See Also:

- "Inserting a Row by Selecting a LOB From Another Table" on page 15-4
- "Inserting a LOB Value Into a Table" on page 15-5
- "Inserting a Row by Initializing a LOB Locator Bind Variable" on page 15-5
- "OCILobLocator Pointer Assignment" on page 13-10 for details on LOB locator semantics in OCI

Initializing BFILEs

Before you can access BFILE values using LOB APIs, the BFILE column or attribute must be made non-NULL. You can initialize the BFILE column to point to an external operating system file by using the BFILENAME function.

See Also: "Accessing BFILEs" on page 21-3 for more information on initializing BFILE columns

Accessing LOBs

You can access a LOB instance using the following techniques:

- Accessing a LOB Using SQL
- Accessing a LOB Using the Data Interface
- Accessing a LOB Using the Locator Interface

Accessing a LOB Using SQL

Support for columns that use LOB data types is built into many SQL functions, enabling you to use SQL semantics to access LOB columns in SQL. In most cases, you can use the same SQL semantics on a LOB column that you would use on a VARCHAR2 column.

See Also: For details on SQL semantics support for LOBs, see Chapter 16, "SQL Semantics and LOBs"

Accessing a LOB Using the Data Interface

You can select a LOB directly into `CHAR` or `RAW` buffers using LONG-to-LOB APIs in OCI and PL/SQL interfaces. In the following PL/SQL example, `ad_finaltext` is selected into a `VARCHAR2` buffer `final_ad`.

```
DECLARE
    final_ad VARCHAR2(32767);
BEGIN
    SELECT ad_finaltext INTO final_ad FROM print_media
        WHERE product_id = 2056 and ad_id = 12001 ;
    /* PUT_LINE can only output up to 255 characters at a time  */
    ...
    DBMS_OUTPUT.PUT_LINE(final_ad);
    /* more calls to read final_ad */
    ...
END;
```

> **See Also:** For more details on accessing LOBs using the data interface, see Chapter 20, "Data Interface for Persistent LOBs"

Accessing a LOB Using the Locator Interface

You can access and manipulate a LOB instance by passing the LOB locator to the LOB APIs supplied with the database. An extensive set of LOB APIs is provided with each supported programmatic environment. In OCI, a LOB locator is mapped to a locator pointer, which is used to access the LOB value.

> **Note:** In all environments, including OCI, the LOB APIs operate on the LOB value implicitly—there is no requirement to dereference the LOB locator.

> **See Also:**
>
> - Chapter 13, "Overview of Supplied LOB APIs"
> - "OCILobLocator Pointer Assignment" on page 13-10 for details on LOB locator semantics in OCI

LOB Rules and Restrictions

This section provides details on LOB rules and restrictions.

Rules for LOB Columns

LOB columns are subject to the following rules and restrictions:

- You cannot specify a LOB as a primary key column.
- Oracle Database has limited support for remote LOBs and ORA-22992 errors can occur when remote LOBs are used in ways that are not supported.

  Remote LOBs are supported in these ways:

  – Create table as select or insert as select

    Only standalone LOB columns are allowed in the select list for statements structured in the following manner:

    ```
    CREATE TABLE t AS SELECT * FROM table1@remote_site;
    ```

```
INSERT INTO t SELECT * FROM table1@remote_site;

UPDATE t SET lobcol = (SELECT lobcol FROM table1@remote_site);

INSERT INTO table1@remote_site SELECT * FROM local_table;

UPDATE table1@remote_site SET lobcol = (SELECT lobcol FROM local_table);

DELETE FROM table1@remote_site <WHERE clause involving non_lob_columns>
```

  - Functions on remote LOBs returning scalars

    SQL and PL/SQL functions having a LOB parameter and returning a scalar data type are supported. Other SQL functions and DBMS_LOB APIs are not supported for use with remote LOB columns. For example, the following statement is supported:

    ```
    CREATE TABLE tab AS SELECT DBMS_LOB.GETLENGTH@dbs2(clob_col) len FROM
    tab@dbs2;
    CREATE TABLE tab AS SELECT LENGTH(clob_col) len FROM tab@dbs2;
    ```

    However, the following statement is not supported because DBMS_LOB.SUBSTR returns a LOB:

    ```
    CREATE TABLE tab AS SELECT DBMS_LOB.SUBSTR(clob_col) from tab@dbs2;
    ```

  - Data Interface for remote LOBs

    You can insert a character or binary buffer into a remote CLOB or BLOB, and select a remote CLOB or BLOB into a character or binary buffer, for example, using PL/SQL:

    ```
    SELECT clobcol1, type1.blobattr INTO varchar_buf1, raw_buf2 FROM
       table1@remote_site;
    INSERT INTO table1@remotesite (clobcol1, type1.blobattr) VALUES varchar_
    buf1,
       raw_buf2;
    INSERT INTO table1@remotesite (lobcol) VALUES ('test');
    UPDATE table1 SET lobcol = 'xxx';
    ```

- Clusters cannot contain LOBs, either as key or nonkey columns.
- The following data structures are supported only as temporary instances. You cannot store these instances in database tables:
  - VARRAY of any LOB type
  - VARRAY of any type containing a LOB type, such as an object type with a LOB attribute
  - ANYDATA of any LOB type
  - ANYDATA of any type containing a LOB
- You cannot specify LOB columns in the ORDER BY clause of a query, the GROUP BY clause of a query, or an aggregate function.
- You cannot specify a LOB column in a SELECT... DISTINCT or SELECT... UNIQUE statement or in a join. However, you can specify a LOB attribute of an object type column in a SELECT... DISTINCT statement, a query that uses the UNION, or a MINUS

set operator if the object type of the column has a MAP or ORDER function defined on it.

- The first (INITIAL) extent of a LOB segment must contain at least three database blocks.
- The minimum extent size is 14 blocks. For an 8K block size (the default), this is equivalent to 112K.
- When creating an AFTER UPDATE DML trigger, you cannot specify a LOB column in the UPDATE OF clause.
- You cannot specify a LOB column as part of an index key. However, you can specify a LOB column in the indextype specification of a domain index. In addition, Oracle Text lets you define an index on a CLOB column.
- In an INSERT... AS SELECT operation, you can bind up to 4000 bytes of data to LOB columns and attributes. There is no length restriction when you doINSERT... AS SELECT from one table to another table using SQL with no bind variables.
- If a table has both LONG and LOB columns, you cannot bind more than 4000 bytes of data to both the LONG and LOB columns in the same SQL statement. However, you can bind more than 4000 bytes of data to either the LONG or the LOB column.

Note: For a table on which you have defined an AFTER UPDATE DML trigger, if you use OCI functions or the DBMS_LOB package to change the value of a LOB column or the LOB attribute of an object type column, the database does not fire the DML trigger.

See Also:

- Chapter 4, "Using Oracle LOB Storage" for SecureFiles capabilities (encryption, compression, and deduplication)
- "Restrictions for LOBs in Partitioned Index-Organized Tables" on page 11-17
- Chapter 18, "Migrating Columns from LONGs to LOBs" under "Migrating Applications from LONGs to LOBs" on page 18-7, for migration limitations on clustered tables, replication, triggers, domain indexes, and function-based indexes
- "Unsupported Use of LOBs in SQL" on page 16-9 for restrictions on SQL semantics
- "Restriction on First Extent of a LOB Segment" on page 11-3
- "Using the Data Interface with Remote LOBs" on page 20-22

Restrictions for LOB Operations

Other general LOB restrictions include the following:

- In SQL Loader, a field read from a LOB cannot be used as an argument to a clause. See "Database Utilities for Loading Data into LOBs" on page 3-1.
- Session migration is not supported for BFILEs in shared server (multithreaded server) mode. This implies that operations on open BFILEs can persist beyond the end of a call to a shared server. In shared server sessions, BFILE operations are bound to one shared server, they cannot migrate from one server to another.

- Case-insensitive searches on `CLOB` columns often do not succeed. For example, to do a case-insensitive search on a `CLOB` column:

```
ALTER SESSION SET NLS_COMP=LINGUISTIC;
ALTER SESSION SET NLS_SORT=BINARY_CI;
SELECT * FROM ci_test WHERE LOWER(clob_col) LIKE 'aa%';
```

  The select fails without the `LOWER` function. You can do case-insensitive searches with Oracle Text or `DBMS_LOB.INSTR()`. See Chapter 16, "SQL Semantics and LOBs".

3

Managing LOBs: Database Administration

This chapter describes administrative tasks that must be performed to set up, maintain, and use a database that contains LOBs.

This chapter contains these topics:

- Database Utilities for Loading Data into LOBs
- Managing Temporary LOBs
- Managing BFILEs
- Changing Tablespace Storage for a LOB

Database Utilities for Loading Data into LOBs

The following utilities are recommended for bulk loading data into LOB columns as part of database setup or maintenance tasks:

- SQL*Loader
- Oracle Data Pump

> **Note:** Application Developers: If you are loading data into a LOB in your application, then using the LOB APIs is recommended. See Chapter 22, "Using LOB APIs".

Using SQL*Loader to Load LOBs

There are two general techniques for using SQL*Loader to load data into LOBs:

- Loading data from a primary data file
- Loading from a secondary data file using LOB files

Consider the following issues when loading LOBs with SQL*Loader:

- For SQL*Loader conventional path loads, failure to load a particular LOB does not result in the rejection of the record containing that LOB; instead, the record ends up containing an empty LOB.

  For SQL*Loader direct-path loads, the LOB could be *empty* or *truncated*. LOBs are sent in pieces to the server for loading. If there is an error, then the LOB piece with the error is discarded and the rest of that LOB is not loaded. In other words, if the entire LOB with the error is contained in the first piece, then that LOB column is either empty or truncated.

- When loading from LOB files, specify the maximum length of the field corresponding to a LOB-type column. If the maximum length is specified, then it is taken as a hint to help optimize memory usage. It is important that the maximum length specification does not underestimate the true maximum length.
- When using SQL*Loader direct-path load, loading LOBs can take up substantial memory. If the message "SQL*Loader 700 (out of memory)" appears when loading LOBs, then internal code is probably batching up more rows in each load call than can be supported by your operating system and process memory. A work-around is to use the ROWS option to read a smaller number of rows in each data save.
- You can also use the Direct Path API to load LOBs. See *Oracle Call Interface Programmer's Guide*.
- Using LOB files is recommended when loading columns containing XML data in CLOBs or XMLType columns. Consider the following validation criteria for XML documents in determining whether to use direct-path load or conventional path load with SQL*Loader:
  - If the XML document must be validated upon loading, then use conventional path load.
  - If it is not necessary to ensure that the XML document is valid, or if you can safely assume that the XML document is valid, then you can perform a direct-path load. Direct-path load performs better because you avoid the overhead of XML validation.

  A *conventional path load* executes SQL INSERT statements to populate tables in an Oracle database.

  A *direct-path load* eliminates much of the Oracle database overhead by formatting Oracle data blocks and writing the data blocks directly to the database files. Additionally, it does not compete with other users for database resources, so it can usually load data at near disk speed. Considerations inherent to direct path loads, such as restrictions, security, and backup implications, are discussed in *Oracle Database Utilities*.
- Tables to be loaded must already exist in the database. SQL*Loader never creates tables. It loads existing tables that either contain data or are empty.
- The following privileges are required for a load:
  - You must have INSERT privileges on the table to be loaded.
  - You must have DELETE privileges on the table to be loaded, when using the REPLACE or TRUNCATE option to empty out the old data before loading the new data in its place.

    See Also: *Oracle Database Utilities* for details on using SQL*Loader to load LOBs

Using SQL*Loader to Populate a BFILE Column

This section describes how to load data from files in the file system into a BFILE column.

See Also: "Supported Environments for BFILE APIs" on page 21-2

Note that the BFILE data type stores unstructured *binary data* in operating system files outside the database. A BFILE column or attribute stores a file *locator* that points to a server-side external file containing the data.

Note: A particular file to be loaded as a BFILE does not have to actually exist at the time of loading.

SQL*Loader assumes that the necessary DIRECTORY objects have been created.

See Also: See "Directory Objects" on page 21-3 and the sections following it for more information on creating directory objects

A control file field corresponding to a BFILE column consists of the column name followed by the BFILE directive.

The BFILE directive takes as arguments a DIRECTORY object name followed by a BFILE name. Both of these can be provided as string constants, or they can be dynamically sourced through some other field.

See Also: *Oracle Database Utilities* for details on SQL*Loader syntax

The following two examples illustrate the loading of BFILES.

Note: You may be required to set up the following data structures for certain examples to work (you are prompted for the password):

```
CONNECT system
Enter password:
Connected.
GRANT CREATE ANY DIRECTORY to samp;
CONNECT samp
Enter password:
Connected.
CREATE OR REPLACE DIRECTORY adgraphic_photo as '/tmp';
CREATE OR REPLACE DIRECTORY adgraphic_dir as '/tmp';
```

In the following example based on the "Table print_media" on page 2-3, only the file name is specified dynamically.

Control file:

```
LOAD DATA
INFILE sample9.dat
INTO TABLE Print_media
FIELDS TERMINATED BY ','
(product_id  INTEGER EXTERNAL(6),
 FileName    FILLER CHAR(30),
 ad_graphic  BFILE(CONSTANT "modem_graphic_2268_21001", FileName))
```

Data file:

```
007, modem_2268.jpg,
008, monitor_3060.jpg,
009, keyboard_2056.jpg,
```

Note: product_ID defaults to (255) if a size is not specified. It is mapped to the file names in the data file. ADGRAPHIC_PHOTO is the directory where all files are stored. ADGRAPHIC_DIR is a DIRECTORY object created previously.

In the following example, the BFILE and the DIRECTORY objects are specified dynamically.

Control file:

```
LOAD DATA
INFILE sample10.dat
INTO TABLE Print_media
FIELDS TERMINATED BY ','
(
 product_id INTEGER EXTERNAL(6),
 ad_graphic BFILE (DirName, FileName),
 FileName  FILLER CHAR(30),
 DirName   FILLER CHAR(30)
)
```

Data file:

```
007,monitor_3060.jpg,ADGRAPHIC_PHOTO,
008,modem_2268.jpg,ADGRAPHIC_PHOTO,
009,keyboard_2056.jpg,ADGRAPHIC_DIR,
```

Note: DirName FILLER CHAR (30) is mapped to the data file field containing the directory name corresponding to the file being loaded.

Using Oracle Data Pump to Transfer LOB Data

You can use Oracle Data Pump to transfer LOB data from one database to another.

Beginning with Oracle Database 12*c*, Data Pump has an option to create all LOB columns as SecureFiles LOBs.

See Also: "SecureFiles LOB Storage" on page 4-2 for an introduction to SecureFiles LOBs

When Data Pump recreates tables, however, it recreates them as they existed in the source database, by default. Therefore, if a LOB column was a BasicFiles LOB in the source database, Data Pump attempts to recreate it as a BasicFiles LOB in the imported database. You can force creation of LOBs as SecureFiles LOBs in the tables being recreated using a TRANSFORM parameter for the command line or a LOB_STORAGE parameter for the DBMS_DATAPUMP and DBMS_METADATA packages.

Note: The transform name is not valid in transportable import.

See Also:

- *Oracle Database Utilities* for specific table syntax used with SecureFiles LOBs
- *Oracle Database Utilities* for details on using Oracle Data Pump

Managing Temporary LOBs

The database keeps track of temporary LOBs in each session, and provides a v$ view called v$temporary_lobs. Using the session ID, the application can determine which user owns the temporary LOB. As a database administrator, you can use this view to

monitor and guide any emergency cleanup of temporary space used by temporary LOBs.

Managing Temporary Tablespace for Temporary LOBs

Temporary tablespaces are used to store temporary LOB data. As a database administrator, you control data storage resources for temporary LOB data by controlling user access to temporary tablespaces and by the creation of different temporary tablespaces.

> **See Also:** *Oracle Database Administrator's Guide* for details on managing temporary tablespaces

Managing BFILEs

This section describes administrative tasks for managing databases that contain `BFILE`s.

Rules for Using Directory Objects and BFILEs

When you create a directory object or `BFILE` objects, ensure that the following conditions are met:

- The operating system file must not be a symbolic or hard link.
- The operating system directory path named in the Oracle DIRECTORY object must be an existing operating system directory path.
- The operating system directory path named in the Oracle DIRECTORY object should not contain any symbolic links in its components.

Setting Maximum Number of Open BFILEs

A limited number of `BFILE`s can be open simultaneously in each session. The initialization parameter, `SESSION_MAX_OPEN_FILES`, defines an upper limit on the number of simultaneously open files in a session.

The default value for this parameter is 10. Using this default, you can open a maximum of 10 files at the same time in each session. To alter this limit, the database administrator must change the parameter value in the `init.ora` file**.** For example:

```
SESSION_MAX_OPEN_FILES=20
```

If the number of unclosed files reaches the `SESSION_MAX_OPEN_FILES` value, then you cannot open additional files in the session. To close all open files, use the `DBMS_LOB.FILECLOSEALL` call.

Changing Tablespace Storage for a LOB

As the database administrator, you can use the following techniques to change the default storage for a LOB after the table has been created:

- **Using ALTER TABLE... MODIFY**: You can change LOB tablespace storage as follows:

Note:

- The `ALTER TABLE` syntax for modifying an existing LOB column uses the `MODIFY LOB` clause, not the `LOB...STORE AS` clause. The `LOB...STORE AS` clause is only for newly added LOB columns.
- There are two kinds of LOB storage clauses: `LOB_storage_clause` and `modify_LOB_storage_clause`. In the `ALTER TABLE MODIFY LOB` statement, you can only specify the `modify_LOB_storage_clause`.

```
ALTER TABLE test MODIFY
  LOB (lob1)
    STORAGE  (
    NEXT         4M
    MAXEXTENTS   100
    PCTINCREASE  50
           )
```

- **Using ALTER TABLE... MOVE**: You can also use the `MOVE` clause of the `ALTER TABLE` statement to change LOB tablespace storage. For example:

```
ALTER TABLE test MOVE
    TABLESPACE tbs1
    LOB (lob1, lob2)
    STORE AS (
          TABLESPACE tbs2
          DISABLE STORAGE IN ROW);
```

4

Using Oracle LOB Storage

This chapter describes how to use Oracle LOB storage and the two types of LOB storage, SecureFiles LOB storage and BasicFiles LOB storage. The chapter describes how to design, create, and modify tables with LOB column types, and where needed, indicates which storage type to use.

This chapter contains these topics:

About LOB Storage

Earlier Oracle database releases supported only one type of LOB storage. In Oracle Database 11*g*, SecureFiles LOB storage was introduced; the original storage type was given the name BasicFiles LOB storage and became the default.

LOBs created using BasicFiles LOB storage became known as BasicFiles LOBs and LOBs created using SecureFiles LOB storage were named SecureFiles LOBs. The `CREATE TABLE` statement added new keywords to indicate the differences: `BASICFILE` specifies BasicFiles LOB storage and `SECUREFILE` specifies SecureFiles LOB storage.

Beginning with Oracle Database 12*c*, SecureFiles LOB storage became the default in the `CREATE TABLE` statement. If no storage type is explicitly specified, new LOB columns use SecureFiles LOB storage.

Throughout this guide, the term LOB can represent LOBs of either storage type unless the storage type is explicitly indicated, by name or by reference to archiving or linking, which apply only to the SecureFiles LOB storage type.

Initialization and compatibility are discussed in "Initialization, Compatibility, and Upgrading" on page 4-22

The following sections discuss the two storage types in detail:

BasicFiles LOB Storage

You must use BasicFiles LOB storage for LOB storage in tablespaces that are not managed with Automatic Segment Space Management (ASSM).

SecureFiles LOB Storage

SecureFiles LOBs can only be created in tablespaces managed with Automatic Segment Space Management (ASSM), unlike BasicFiles LOB storage.

SecureFiles LOB storage is designed to provide much better performance and scalability compared to BasicFiles LOBs and to meet or exceed the performance capabilities of traditional network file systems.

SecureFiles LOB storage supports three features that are not available with the BasicFiles LOB storage option: compression, deduplication, and encryption.

Oracle recommends that you enable compression, deduplication, and encryption through the `CREATE TABLE` statement. If you enable these features through the `ALTER TABLE` statement, *all* SecureFiles LOB data in the table is read, modified, and written; this can cause the database to lock the table during a potentially lengthy operation, though there are online capabilities in the `ALTER TABLE` statement which can help you avoid this issue.

Note: These features have specific licensing requirements as described in *Oracle Database Licensing Information*.

This section contains the following topics:

- About Advanced LOB Compression
- About Advanced LOB Deduplication
- About SecureFiles Encryption

About Advanced LOB Compression

Advanced LOB Compression transparently analyzes and compresses SecureFiles LOB data to save disk space and improve performance.

License Requirement: You must have a license for the Oracle Advanced Compression Option to implement Advanced LOB Compression.

See Also:

- "CREATE TABLE with Advanced LOB Compression" on page 4-12
- "ALTER TABLE with Advanced LOB Compression" on page 4-19

About Advanced LOB Deduplication

Advanced LOB Deduplication enables Oracle Database to automatically detect duplicate LOB data within a LOB column or partition, and conserve space by storing only one copy of the data.

License Requirement: You must have a license for the Oracle Advanced Compression Option to implement Advanced LOB Deduplication.

Note also that Oracle Streams does not support SecureFiles LOBs that are deduplicated.

See Also:

- "CREATE TABLE with Advanced LOB Deduplication" on page 4-13
- "ALTER TABLE with Advanced LOB Deduplication" on page 4-20

About SecureFiles Encryption

SecureFiles Encryption introduces a new encryption facility for LOBs. The data is encrypted using Transparent Data Encryption (TDE), which allows the data to be stored securely, and still allows for random read and write access.

License Requirement: You must have a license for the Oracle Advanced Security Option to implement SecureFiles Encryption.

See Also:

- "CREATE TABLE with SecureFiles Encryption" on page 4-15
- "ALTER TABLE with SecureFiles Encryption" on page 4-21

Using CREATE TABLE with LOB Storage

This section discusses how to use the `CREATE TABLE` statement with LOB storage and describes the parameters that work with SecureFiles or BasicFiles LOB storage, or both.

The `SHRINK` option is not supported for SecureFiles LOBs.

Example 4–1 provides the syntax for `CREATE TABLE` in Backus Naur (BNF) notation, with LOB-specific parameters in bold. See "CREATE TABLE LOB Storage Parameters" on page 4-9 for these parameter descriptions and the `CREATE TABLE` statement, *Oracle Database SQL Language Reference*, for complete reference.

Example 4–1 BNF for CREATE TABLE

Keywords are in bold.

CREATE [**GLOBAL TEMPORARY**] **TABLE**

[schema.]table **OF**

[schema.]object_type

[(relational_properties)]

[**ON COMMIT** { **DELETE** | **PRESERVE** } **ROWS**]

[OID_clause]

[OID_index_clause]

[physical_properties]

[table_properties] ;

<relational_properties> ::=

{ column_definition

| { out_of_line_constraint

| out_of_line_ref_constraint

```
  | supplemental_logging_props
  }
}
  [, { column_definition
     | { out_of_line_constraint
       | out_of_line_ref_constraint
       | supplemental_logging_props
       }
  ]...

<column_definition> ::=
column data_type [ SORT ]
     [ DEFAULT expr ]
     [ ENCRYPT encryption_spec ]
     [ ( inline_constraint [ inline_constraint ] ... )
     | inline_ref_constraint
     ]

<data_type> ::=
{ Oracle_built_in_datatypes
| ANSI_supported_datatypes
| user_defined_types
| Oracle_supplied_types
}

<Oracle_built_in_datatypes> ::=
{ character_datatypes
| number_datatypes
| long_and_raw_datatypes
| datetime_datatypes
| large_object_datatypes
| rowid_datatypes
}

<large_object_datatypes> ::=
{ BLOB | CLOB | NCLOB| BFILE }
```

```
<table_properties> ::=
 [ column_properties ]
 [ table_partitioning_clauses ]
 [ CACHE | NOCACHE ]
 [ parallel_clause ]
 [ ROWDEPENDENCIES | NOROWDEPENDENCIES ]
 [ enable_disable_clause ]
 [ enable_disable_clause ]...
 [ row_movement_clause ]
 [ AS subquery ]

<column_properties> ::=
 { object_type_col_properties
 | nested_table_col_properties
 | { varray_col_properties | LOB_storage_clause }
   [ (LOB_partition_storage
       [, LOB_partition_storage ]...
     )
   ]
 | XMLType_column_properties
 }
 [ { object_type_col_properties
   | nested_table_col_properties
   | { varray_col_properties | LOB_storage_clause }
     [ ( LOB_partition_storage
         [, LOB_partition_storage ]...
       )
     ]
   | XMLType_column_properties
   }
 ]...

<LOB_partition_storage> ::=
 PARTITION partition
 { LOB_storage_clause | varray_col_properties }
   [ LOB_storage_clause | varray_col_properties ]...
 [ ( SUBPARTITION subpartition
```

```
    { LOB_storage_clause | varray_col_properties }
     [ LOB_storage_clause
     | varray_col_properties
     ]...
   )
 ]

<LOB_storage_clause> ::=
 LOB
 { (LOB_item [, LOB_item ]...)
    STORE AS [ SECUREFILE | BASICFILE ] (LOB_storage_parameters)
 | (LOB_item)
    STORE AS [ SECUREFILE | BASICFILE ]
     { LOB_segname (LOB_storage_parameters)
     | LOB_segname
     | (LOB_storage_parameters)
     }
 }

<LOB_storage_parameters> ::=
 { TABLESPACE tablespace
 | { LOB_parameters [ storage_clause ]
   }
 | storage_clause
 }
   [ TABLESPACE tablespace
   | { LOB_parameters [ storage_clause ]
     }
   ]...

<LOB_parameters> ::=
 [ { ENABLE | DISABLE } STORAGE IN ROW
 | CHUNK integer
 | PCTVERSION integer
 | RETENTION [ { MAX | MIN integer | AUTO | NONE } ]
 | FREEPOOLS integer
 | LOB_deduplicate_clause
```

```
 | LOB_compression_clause
 | LOB_encryption_clause
 | { CACHE |  NOCACHE | CACHE READS } [ logging_clause ] } }
 ]

<logging_clause> ::=
 { LOGGING | NOLOGGING | FILESYSTEM_LIKE_LOGGING }

<storage_clause> ::=
 STORAGE
 ({ INITIAL integer [ K | M ]
  | NEXT integer [ K | M ]
  | MINEXTENTS integer
  | MAXEXTENTS { integer | UNLIMITED }
  | PCTINCREASE integer
  | FREELISTS integer
  | FREELIST GROUPS integer
  | OPTIMAL [ integer [ K | M ]
          | NULL
          ]
  | BUFFER_POOL { KEEP | RECYCLE | DEFAULT }
  }
   [ INITIAL integer [ K | M ]
   | NEXT integer [ K | M ]
   | MINEXTENTS integer
   | MAXEXTENTS { integer | UNLIMITED }
   | MAXSIZE { { integer { K | M | G | T | P } } | UNLIMITED }
   | PCTINCREASE integer
   | FREELISTS integer
   | FREELIST GROUPS integer
   | OPTIMAL [ integer [ K | M ]
           | NULL
           ]
   | BUFFER_POOL { KEEP | RECYCLE | DEFAULT }
   ]...
 )
```

```
<LOB_deduplicate_clause> ::=
 { DEDUPLICATE
 | KEEP_DUPLICATES
 }

<LOB_compression_clause> ::=
 { COMPRESS [ HIGH | MEDIUM | LOW ]
 | NOCOMPRESS }

<LOB_encryption_clause> ::=
 { ENCRYPT [ USING 'encrypt_algorithm' ]
   [ IDENTIFIED BY password ]
 | DECRYPT
 }

<XMLType_column_properties> ::=
XMLTYPE [ COLUMN ] column
  [ XMLType_storage ]
  [ XMLSchema_spec ]

<XMLType_storage> ::=
STORE AS
  { OBJECT RELATIONAL
  | [ SECUREFILE | BASICFILE ] { CLOB | BINARY XML }
      [ { LOB_segname [ (LOB_parameters) ]
        | LOB_parameters
        }
        ]

<varray_col_properties> ::=
VARRAY varray_item
  { [ substitutable_column_clause ]
    STORE AS [ SECUREFILE | BASICFILE ] LOB
      { [ LOB_segname ] (LOB_parameters)
      | LOB_segname
      }
  | substitutable_column_clause
```

```
}
```

CREATE TABLE LOB Storage Parameters

Table 4–1 summarizes the parameters of the `CREATE TABLE` statement that relate to LOB storage, where necessary noting whether a parameter is specific to BasicFiles LOB or SecureFiles LOB storage.

Table 4–1 Parameters of CREATE TABLE Statement Related to LOBs

Parameter	Description
`BASICFILE`	Specifies BasicFiles LOB storage, the original architecture for LOBs. If you set the Oracle Database compatibility mode to 10*g*, the LOB storage behavior is identical to that of Oracle Database 10*g* (parameter `BASICFILE` did not yet exist). If you set the compatibility mode to Oracle Database 11*g*, the same LOB functionality is enabled by default, with the `BASICFILE` parameter specified for completeness. Starting with Oracle Database 12*c*, you must explicitly specify the parameter `BASICFILE` to use the BasicFiles LOB storage type. Otherwise, the `CREATE TABLE` statement uses SecureFiles LOB, the current default. See "Initialization, Compatibility, and Upgrading" on page 4-22. For BasicFiles LOBs, specifying any of the SecureFiles LOB options results in an error.
`SECUREFILE`	Specifies SecureFiles LOBs storage. Starting with Oracle Database 12*c*, the SecureFiles LOB storage type, specified by the parameter `SECUREFILE`, is the default. A SecureFiles LOB can only be created in a tablespace managed with Automatic Segment Space Management (ASSM).
`CHUNK`	For BasicFiles LOBs, specifies the chunk size when creating a table that stores LOBs. `CHUNK` is one or more Oracle blocks and corresponds to the data size used by Oracle Database when accessing or modifying the LOB. For SecureFiles LOBs, it is an advisory size provided for backward compatibility.

Table 4–1 (Cont.) Parameters of CREATE TABLE Statement Related to LOBs

Parameter	Description
RETENTION	Configures the LOB column to store old versions of LOB data in a specified manner. In Oracle Database Release 12*c*, this parameter specifies the retention policy. RETENTION has these possible values: ■ MAX specifies that the system keep old versions of LOB data blocks until the space used by the segment has reached the size specified in the MAXSIZE parameter. If MAXSIZE is not specified, MAX behaves like AUTO. ■ MIN specifies that the system keep old versions of LOB data blocks for the specified number of seconds. ■ NONE specifies that there is no retention period and space can be reused in any way deemed necessary. ■ AUTO specifies that the system manage the space as efficiently as possible weighing both time and space needs. For details of the RETENTION parameter used with BasicFiles LOBs, see "RETENTION Parameter for BasicFiles LOBs" on page 11-8.
MAXSIZE	Specifies the upper limit of storage space that a LOB may use. If this amount of space is consumed, new LOB data blocks are taken from the pool of old versions of LOB data blocks as needed, regardless of time requirements.
FREEPOOLS	Specifies the number of FREELIST groups for BasicFiles LOBs, if the database is in automatic undo mode. Under Release 12*c* compatibility, this parameter is ignored when SecureFiles LOBs are created.
LOGGING, NOLOGGING, or FILESYSTEM_LIKE_LOGGING	Specifies logging options: ■ LOGGING specifies logging the creation of the LOB and subsequent inserts into the LOB, in the redo log file. LOGGING is the default. ■ NOLOGGING specifies no logging. ■ FILESYSTEM_LIKE_LOGGING specifies that the system only logs the metadata. This is similar to metadata journaling of file systems, which reduces mean time to recovery from failures. FILESYSTEM_LIKE_LOGGING ensures that data is completely recoverable (an instance recovery) after a server failure. *This option is invalid for BasicFiles LOBs.*

Table 4–1 (Cont.) Parameters of CREATE TABLE Statement Related to LOBs

Parameter	Description
	For SecureFiles LOBs, the following applies: ■ The NOLOGGING setting is converted internally to FILESYSTEM_LIKE_LOGGING. ■ The LOGGING setting is similar to the data journaling of file systems. ■ Both the LOGGING and FILESYSTEM_LIKE_LOGGING settings provide a complete transactional file system. See "LOGGING / NOLOGGING Parameter for BasicFiles LOBs" on page 11-9 and "Ensuring Read Consistency" on page 22-39. For a non-partitioned object, the value specified for this clause is the actual physical attribute of the segment associated with the object. For partitioned objects, the value specified for this clause is the default physical attribute of the segments associated with all partitions specified in the CREATE statement (and in subsequent ALTER ... ADD PARTITION statements), unless you specify the logging attribute in the PARTITION description. **CAUTION:** For LOB segments with NOLOGGING or FILESYSTEM_LIKE_LOGGING settings, it is possible that data can change on the disk during a backup operation. This results in read inconsistency. To avoid this situation, ensure that changes to LOB segments are saved in the redo log file by setting LOGGING for LOB storage. NOLOGGING and FILESYSTEM_LIKE_LOGGING SecureFiles are recoverable after an instance failure, but not after a media failure. LOGGING SecureFiles are recoverable after both instance and media failures. See the *Oracle Database Backup and Recovery User's Guide* for a discussion of data protection, media failure, and instance failure.
FREELISTS or FREELIST GROUPS	Specifies the number of process freelists or freelist groups, respectively, allocated to the segment; NULL for partitioned tables. Under Release 12*c* compatibility, these parameters are ignored when SecureFiles LOBs are created.
PCTVERSION	Specifies the percentage of used BasicFiles LOB data space that may be occupied by old versions of the LOB data pages. Under Release 12*c* compatibility, this parameter is ignored when SecureFiles LOBs are created.
COMPRESS or NOCOMPRESS	The COMPRESS option turns on Advanced LOB Compression, and NOCOMPRESS turns it off. Note that setting table or index compression does not affect Advanced LOB Compression.
DEDUPLICATE or KEEP_DUPLICATES	The DEDUPLICATE option enables Advanced LOB Deduplication; it specifies that SecureFiles LOB data that is identical in two or more rows in a LOB column, partition or subpartition must share the same data blocks. The database combines SecureFiles LOBs with identical content into a single copy, reducing storage and simplifying storage management. The opposite of this option is KEEP_DUPLICATES.

Table 4–1 (Cont.) Parameters of CREATE TABLE Statement Related to LOBs

Parameter	Description
ENCRYPT or DECRYPT	The ENCRYPT option turns on SecureFiles Encryption, and encrypts all SecureFiles LOB data using Oracle Transparent Data Encryption (TDE). The DECRYPT options turns off SecureFiles Encryption.

CREATE TABLE and SecureFiles LOB Features

This section provides usage notes and examples for features specific to SecureFiles LOBs used with CREATE TABLE.

> **Note:** Clauses in example discussions refer to the Backus Naur (BNF) notation "BNF for CREATE TABLE" on page 4-3. Parameters are described in "CREATE TABLE LOB Storage Parameters" on page 4-9.

This section covers:

- CREATE TABLE with Advanced LOB Compression
- CREATE TABLE with Advanced LOB Deduplication
- CREATE TABLE with SecureFiles Encryption

CREATE TABLE with Advanced LOB Compression

This section discusses Advanced LOB Compression when used in the CREATE TABLE statement.

This section contains the following topics:

- Usage Notes for Advanced LOB Compression
- Examples of CREATE TABLE and Advanced LOB Compression

Usage Notes for Advanced LOB Compression Consider the following when using the CREATE TABLE statement and Advanced LOB Compression:

- Advanced LOB Compression is performed on the server and enables random reads and writes to LOB data. Compression utilities on the client, like utl_compress, cannot provide random access.
- Advanced LOB Compression does not enable table or index compression. Conversely, table and index compression do not enable Advanced LOB Compression.
- The LOW, MEDIUM, and HIGH options provide varying degrees of compression. The higher the compression, the higher the latency incurred. The HIGH setting incurs more work, but compresses the data better. The default is MEDIUM.

  The LOW compression option uses an extremely lightweight compression algorithm that removes the majority of the CPU cost that is typical with file compression. Compressed SecureFiles LOBs at the LOW level provide a very efficient choice for SecureFiles LOB storage. SecureFiles LOBs compressed at LOW generally consume less CPU time and less storage than BasicFiles LOBs, and typically help the application run faster because of a reduction in disk I/O.
- Compression can be specified at the partition level. The CREATE TABLE lob_storage_clause enables specification of compression for partitioned tables on a per-partition basis.

- The DBMS_LOB.SETOPTIONS procedure can enable and disable compression on individual SecureFiles LOBs. See *Oracle Database PL/SQL Packages and Types Reference* for further information.

Examples of CREATE TABLE and Advanced LOB Compression The following examples demonstrate how to issue CREATE TABLE statements for specific compression scenarios.

Example 4–2 Creating a SecureFiles LOB Column with LOW Compression

```
CREATE TABLE t1 (a CLOB)
    LOB(a) STORE AS SECUREFILE(
    COMPRESS LOW
    CACHE
    NOLOGGING
  );
```

Example 4–3 Creating a SecureFiles LOB Column with MEDIUM (default) Compression

```
CREATE TABLE t1 ( a CLOB)
    LOB(a) STORE AS SECUREFILE (
         COMPRESS
         CACHE
         NOLOGGING
    );
```

Example 4–4 Creating a SecureFiles LOB Column with HIGH Compression

```
CREATE TABLE t1 ( a CLOB)
    LOB(a) STORE AS SECUREFILE (
         COMPRESS HIGH
         CACHE
    );
```

Example 4–5 Creating a SecureFiles LOB Column with Disabled Compression

```
CREATE TABLE t1 ( a CLOB)
    LOB(a) STORE AS SECUREFILE (
         NOCOMPRESS
         CACHE
    );
```

Example 4–6 Creating a SecureFiles LOB Column with Compression on One Partition

```
CREATE TABLE t1 ( REGION VARCHAR2(20), a BLOB)
     LOB(a) STORE AS SECUREFILE (
        CACHE
     )
     PARTITION BY LIST (REGION) (
          PARTITION p1 VALUES ('x', 'y')
               LOB(a) STORE AS SECUREFILE (
                   COMPRESS
                ),
          PARTITION p2 VALUES (DEFAULT)
     );
```

CREATE TABLE with Advanced LOB Deduplication

This section discusses Advanced LOB Deduplication when used in the CREATE TABLE statement.

This section contains the following topics:

- Usage Notes for Advanced LOB Deduplication
- Examples of CREATE TABLE and Advanced LOB Deduplication

Usage Notes for Advanced LOB Deduplication Consider the following when using `CREATE TABLE` and Advanced LOB Deduplication:

- Identical LOBs are good candidates for deduplication. Copy operations can avoid data duplication by enabling deduplication.
- Duplicate detection happens within a LOB segment. Duplicate detection does not span partitions or subpartitions for partitioned and subpartitioned LOB columns.
- Deduplication can be specified at a partition level. The `CREATE TABLE lob_storage_clause` enables specification for partitioned tables on a per-partition basis.
- The `DBMS_LOB.SETOPTIONS` procedure can enable or disable deduplication on individual LOBs.

Examples of CREATE TABLE and Advanced LOB Deduplication The following examples demonstrate how to issue `CREATE TABLE` statements for specific deduplication scenarios.

Example 4–7 Creating a SecureFiles LOB Column with Deduplication

```
CREATE TABLE t1 ( a CLOB)
    LOB(a) STORE AS SECUREFILE (
        DEDUPLICATE
        CACHE
    );
```

Example 4–8 Creating a SecureFiles LOB Column with Disabled Deduplication

```
CREATE TABLE t1 ( a CLOB)
    LOB(a) STORE AS SECUREFILE (
         KEEP_DUPLICATES
         CACHE
    );
```

Example 4–9 Creating a SecureFiles LOB Column with Deduplication on One Partition

```
CREATE TABLE t1 ( REGION VARCHAR2(20), a BLOB)
     LOB(a) STORE AS SECUREFILE (
           CACHE
)
PARTITION BY LIST (REGION) (
     PARTITION p1 VALUES ('x', 'y')
          LOB(a) STORE AS SECUREFILE (
               DEDUPLICATE
          ),
     PARTITION p2 VALUES (DEFAULT)
);
```

Example 4–10 Creating a SecureFiles LOB column with Deduplication Disabled on One Partition

```
CREATE TABLE t1 ( REGION VARCHAR2(20), ID NUMBER, a BLOB)
     LOB(a) STORE AS SECUREFILE (
           DEDUPLICATE
           CACHE
)
```

```
PARTITION BY RANGE (REGION)
      SUBPARTITION BY HASH(ID) SUBPARTITIONS 2 (
        PARTITION p1 VALUES LESS THAN (51)
           lob(a) STORE AS a_t2_p1
           (SUBPARTITION t2_p1_s1 lob(a) STORE AS a_t2_p1_s1,
            SUBPARTITION t2_p1_s2 lob(a) STORE AS a_t2_p1_s2),
        PARTITION p2 VALUES LESS THAN (MAXVALUE)
           lob(a) STORE AS a_t2_p2 ( KEEP_DUPLICATES )
           (SUBPARTITION t2_p2_s1 lob(a) STORE AS a_t2_p2_s1,
            SUBPARTITION t2_p2_s2 lob(a) STORE AS a_t2_p2_s2)
      );
```

CREATE TABLE with SecureFiles Encryption

This section discusses SecureFiles Encryption when used in the `CREATE TABLE` statement.

This section contains the following topics:

- Usage Notes for SecureFiles Encryption
- Examples of CREATE TABLE and SecureFiles Encryption

Usage Notes for SecureFiles Encryption Consider the following when using `CREATE TABLE` and SecureFiles Encryption:

- Transparent Data Encryption (TDE) supports encryption of LOB data types.
- Encryption is performed at the block level.
- The *`encrypt_algorithm`* indicates the name of the encryption algorithm. Valid algorithms are: `AES192` (default), `3DES168`, `AES128`, and `AES256`.
- The column encryption key is derived from `PASSWORD`, if specified.
- The default for LOB encryption is `SALT`. `NO SALT` is not supported.
- All LOBs in the LOB column are encrypted.
- `DECRYPT` keeps the LOBs in clear text.
- LOBs can be encrypted only on a per-column basis, similar to TDE. All partitions within a LOB column are encrypted.
- Key management controls the ability to encrypt or decrypt.
- TDE is not supported by the traditional `import` and `export` utilities or by transportable-tablespace-based `export`. Use the Data Pump `expdb` and `impdb` utilities with encrypted columns instead.

> **See Also:** "*Oracle Database Advanced Security Guide* for information about using the `ADMINISTER KEY MANAGEMENT` statement to create TDE keystores

Examples of CREATE TABLE and SecureFiles Encryption The following examples demonstrate how to issue `CREATE TABLE` statements for specific encryption scenarios.

Example 4–11 Creating a SecureFiles LOB Column with a Specific Encryption Algorithm

```
CREATE TABLE t1 ( a CLOB ENCRYPT USING 'AES128')
    LOB(a) STORE AS SECUREFILE (
         CACHE
    );
```

Example 4–12 Creating a SecureFiles LOB column with encryption for all partitions

```
CREATE TABLE t1 ( REGION VARCHAR2(20), a BLOB)
  LOB(a) STORE AS SECUREFILE (
  ENCRYPT USING 'AES128'
  NOCACHE
  FILESYSTEM_LIKE_LOGGING
)
PARTITION BY LIST (REGION) (
PARTITION p1 VALUES ('x', 'y'),
PARTITION p2 VALUES (DEFAULT)
);
```

Example 4–13 Creating a SecureFiles LOB Column with Encryption Based on a Password Key

```
CREATE TABLE t1 ( a CLOB ENCRYPT IDENTIFIED BY foo)
    LOB(a) STORE AS SECUREFILE (
        CACHE
    );
```

The following example has the same result because the encryption option can be set in the `LOB_deduplicate_clause` section of the statement:

```
CREATE TABLE t1 (a CLOB)
    LOB(a) STORE AS SECUREFILE (
        CACHE
        ENCRYPT
        IDENTIFIED BY foo
    );
```

Example 4–14 Creating a SecureFiles LOB Column with Disabled Encryption

```
CREATE TABLE t1 ( a CLOB )
    LOB(a) STORE AS SECUREFILE (
        CACHE DECRYPT
    );
```

Using ALTER TABLE with LOB Storage

You can modify LOB storage with an `ALTER TABLE` statement, or with online redefinition using the `DBMS_REDEFINITION` package. See *Oracle Database PL/SQL Packages and Types Reference*.

You can use `ALTER TABLE` or online redefinition to enable compression, deduplication, or encryption features for a LOB column. The `ALTER TABLE` statement supports online operations (see Oracle Database SQL Language Reference) and Oracle Database supports parallel operations on SecureFiles LOBs columns, making this a resource-efficient approach.

As an alternative to `ALTER TABLE`, you can use online redefinition to enable one or more of these features. As with `ALTER TABLE`, online redefinition of SecureFiles LOB columns can be executed in parallel.

See "Migrating Columns from BasicFiles LOBs to SecureFiles LOBs" on page 4-23 for details of online redefinition.

Note that the `SHRINK` option is not supported for SecureFiles LOBs.

Example 4–15 provides the syntax for `ALTER TABLE` in Backus Naur (BNF) notation, with LOB-specific parameters in bold. See "CREATE TABLE LOB Storage Parameters" on page 4-9 for these parameter descriptions and the `ALTER TABLE` statement, *Oracle*

Database SQL Language Reference, for complete reference.

Example 4–15 BNF for the ALTER TABLE Statement

LOB-specific keywords are in bold.

```
ALTER TABLE [ schema.]table
 [ alter_table_properties
 | column_clauses
 | constraint_clauses
 | alter_table_partitioning
 | alter_external_table_clauses
 | move_table_clause
 ]
  [ enable_disable_clause
  | { ENABLE | DISABLE }
    { TABLE LOCK | ALL TRIGGERS }
  [ enable_disable_clause
  | { ENABLE | DISABLE }
    { TABLE LOCK | ALL TRIGGERS }
  ]...
 ] ;


<column_clauses> ::=
 { { add_column_clause
   | modify_column_clause
   | drop_column_clause
   }
   [ add_column_clause
   | modify_column_clause
   | drop_column_clause
   ]...
 | rename_column_clause
 | modify_collection_retrieval
   [ modify_collection_retrieval ]...
 | modify_LOB_storage_clause
   [ modify_LOB_storage_clause ] ...
 | alter_varray_col_properties
   [ alter_varray_col_properties ]
```

```
 }

<modify_LOB_storage_clause> ::=
MODIFY LOB (LOB_item) ( modify_LOB_parameters )

<modify_LOB_parameters> ::=
{ storage_clause
| PCTVERSION integer
| FREEPOOLS integer
| REBUILD FREEPOOLS
| LOB_retention_clause
| LOB_deduplicate_clause
| LOB_compression_clause
| { ENCRYPT encryption_spec | DECRYPT }
| { CACHE
  | { NOCACHE | CACHE READS } [ logging_clause ]
  }
| allocate_extent_clause
| shrink_clause
| deallocate_unused_clause
} ...
```

ALTER TABLE LOB Storage Parameters

Table 4–2 summarizes the parameters of the `ALTER TABLE` statement that relate to LOB storage, where necessary noting whether a parameter is specific to BasicFiles LOB or SecureFiles LOB storage.

Table 4–2 Parameters of ALTER TABLE Statement Related to LOBs

Parameter	Description
`RETENTION`	Configures the LOB column to store old versions of LOB data in a specified manner. Altering `RETENTION` only affects space created after the `ALTER TABLE` statement runs.
`COMPRESS` or `NOCOMPRESS`	Enables or disables Advanced LOB Compression. All LOBs in the LOB segment are altered with the new setting.
`DEDUPLICATE` or `KEEP_DUPLICATES`	Enables or disables Advanced LOB Deduplication. The option `DEDUPLICATE` enables you to specify that LOB data that is identical in two or more rows in a LOB column share the same data blocks. The database combines LOBs with identical content into a single copy, reducing storage and simplifying storage management. The opposite of this option is `KEEP_DUPLICATES`.

Table 4–2 (Cont.) Parameters of ALTER TABLE Statement Related to LOBs

Parameter	Description
ENCRYPT or DECRYPT	Enables or disables SecureFiles LOB encryption. Alters all LOBs in the LOB segment with the new setting. A LOB segment can be only altered to enable or disable LOB encryption. That is, ALTER cannot be used to update the encryption algorithm or the encryption key. Update the encryption algorithm or encryption key using the ALTER TABLE REKEY syntax.

ALTER TABLE SecureFiles LOB Features

This section provides usage notes and examples for using features specific to SecureFiles LOBs with ALTER TABLE.

> **Note:** Clauses in example discussions refer to the Backus Naur (BNF) notation "BNF for the ALTER TABLE Statement" on page 4-17. Parameters are described in "ALTER TABLE LOB Storage Parameters" on page 4-18.

This section covers these topics:

- ALTER TABLE with Advanced LOB Compression
- ALTER TABLE with Advanced LOB Deduplication
- ALTER TABLE with SecureFiles Encryption

ALTER TABLE with Advanced LOB Compression

This section discusses Advanced LOB Compression when used in the ALTER TABLE statement and contains the following topics:

- Usage Notes for Advanced LOB Compression
- Examples of ALTER TABLE and Advanced LOB Compression

Usage Notes for Advanced LOB Compression Consider the following when using ALTER TABLE and Advanced LOB Compression:

- This syntax alters the compression mode of the LOB column.
- The DBMS_LOB.SETOPTIONS procedure can enable or disable compression on individual LOBs.
- Compression may be specified either at the table level or the partition level.
- The LOW, MEDIUM, and HIGH options provide varying degrees of compression. The higher the compression, the higher the latency incurred. The HIGH setting incurs more work, but compresses the data better. The default is MEDIUM. See "CREATE TABLE with Advanced LOB Compression" on page 4-12.

Examples of ALTER TABLE and Advanced LOB Compression The following examples demonstrate how to issue ALTER TABLE statements for specific compression scenarios.

Example 4–16 Altering a SecureFiles LOB Column to Enable LOW Compression

```
ALTER TABLE t1 MODIFY
   LOB(a) (
           COMPRESS LOW
```

```
   );
```

Example 4–17 Altering a SecureFiles LOB Column to Disable Compression

```
ALTER TABLE t1 MODIFY
    LOB(a) (
         NOCOMPRESS
    );
```

Example 4–18 Altering a SecureFiles LOB Column to Enable HIGH Compression

```
ALTER TABLE t1 MODIFY
    LOB(a) (
         COMPRESS HIGH
    );
```

Example 4–19 Altering a SecureFiles LOB Column to Enable Compression on One partition

```
ALTER TABLE t1 MODIFY PARTITION p1
    LOB(a) (
         COMPRESS HIGH
    );
```

ALTER TABLE with Advanced LOB Deduplication

This section discusses Advanced LOB Deduplication in reference to the `ALTER TABLE` statement and contains the following topics:

- Usage Notes for Advanced LOB Deduplication
- Examples of ALTER TABLE and Advanced LOB Deduplication

Usage Notes for Advanced LOB Deduplication Consider the following when using `ALTER TABLE` and Advanced LOB Deduplication:

- The `ALTER TABLE` syntax can enable or disable LOB-level deduplication.
- This syntax alters the deduplication mode of the LOB column.
- The `DBMS_LOB.SETOPTIONS` procedure can enable or disable deduplication on individual LOBs.
- Deduplication can be specified at a table level or partition level. Deduplication does not span across partitioned LOBs.

Examples of ALTER TABLE and Advanced LOB Deduplication The following examples demonstrate how to issue `ALTER TABLE` statements for specific deduplication scenarios.

Example 4–20 Altering a SecureFiles LOB Column to Disable Deduplication

```
ALTER TABLE t1 MODIFY
     LOB(a) (
         KEEP_DUPLICATES
    );
```

Example 4–21 Altering a SecureFiles LOB Column to Enable Deduplication

```
ALTER TABLE t1 MODIFY
    LOB(a) (
         DEDUPLICATE
    );
```

Example 4–22 Altering a SecureFiles LOB Column to Enable Deduplication on One Partition

```
ALTER TABLE t1 MODIFY PARTITION p1
    LOB(a) (
         DEDUPLICATE
    );
```

ALTER TABLE with SecureFiles Encryption

This section discusses SecureFiles Encryption when used in the `ALTER TABLE` statement.

This section contains the following topics:

- Usage Notes for SecureFiles Encryption
- Examples of ALTER TABLE and SecureFiles Encryption

Usage Notes for SecureFiles Encryption Consider the following when using `ALTER TABLE` and SecureFiles Encryption:

- `ALTER TABLE` enables and disables SecureFiles Encryption. This syntax also allows the user to re-key LOB columns with a new key or algorithm.
- `ENCRYPT` and `DECRYPT` options enable or disable encryption on all LOBs in the specified SecureFiles LOB column.
- The default for LOB encryption is `SALT`. `NO SALT` is not supported.
- The `DECRYPT` option converts encrypted columns to clear text form.
- Key management controls the ability to encrypt or decrypt.
- LOBs can be encrypted only on a per-column basis. A partitioned LOB has either all partitions encrypted or not encrypted.

Examples of ALTER TABLE and SecureFiles Encryption The following examples demonstrate how to issue `ALTER TABLE` statements for specific encryption scenarios.

Example 4–23 Altering a SecureFiles LOB Column by Encrypting Based on a Specific Algorithm

Enable LOB encryption using `3DES168`.

```
ALTER TABLE t1 MODIFY
      ( a CLOB ENCRYPT USING '3DES168');
```

This is another example of enabling LOB encryption using `3DES168`.

```
ALTER TABLE t1 MODIFY LOB(a)
      (ENCRYPT USING '3DES168');
```

Example 4–24 Altering a SecureFiles LOB Column by Encrypting Based on a Password Key

Enable encryption on a SecureFiles LOB column and build the encryption key using a password.

```
ALTER TABLE t1 MODIFY
    ( a CLOB ENCRYPT IDENTIFIED BY foo);
```

Example 4–25 Altering a SecureFiles LOB Column by Re-keying the Encryption

To re-encrypt the LOB column with a new key, re-key the table.

```
ALTER TABLE t1 REKEY USING '3DES168';
```

Initialization, Compatibility, and Upgrading

This section discusses LOB initialization and compatibility parameters and how they interact with each other.

This section covers these topics:

- Compatibility and Upgrading
- Initialization Parameter for SecureFiles LOBs

Compatibility and Upgrading

All features described in this document are enabled with compatibility set to `11.2.0.0.0` or higher. There is no downgrade capability after `11.2.0.0.0` is set.

If you want to upgrade BasicFiles LOBs to SecureFiles LOBs, you must use typical methods for upgrading data (CTAS/ITAS, online redefinition, export/import, column to column copy, or using a view and a new column). Most of these solutions require twice the disk space used by the data in the input LOB column. However, partitioning and taking these actions on a partition-by-partition basis lowers the disk space requirements.

Initialization Parameter for SecureFiles LOBs

Using the `DB_SECUREFILE` initialization parameter, the database administrator can modify the initial settings that the `COMPATIBILITY` parameter sets as default, changing the circumstances under which SecureFiles LOBs or BasicFiles LOBs are created or allowed. The `DB_SECUREFILE` parameter is typically set in the file `init.ora`. See "Compatibility and Upgrading" on page 4-22.

See Also: *Oracle Database Reference*

The `DB_SECUREFILE` initialization parameter is dynamic and can be modified with the `ALTER SYSTEM` statement. Example 4–26 shows the format for changing the parameter value:

Example 4–26 Setting DB_SECUREFILE parameter through ALTER SYSTEM

```
ALTER SYSTEM SET DB_SECUREFILE = 'ALWAYS';
```

The valid values for `DB_SECUREFILE` are:

- `NEVER` prevents SecureFiles LOBs from being created. If `NEVER` is specified, any LOBs that are specified as SecureFiles LOBs are created as BasicFiles LOBs. If storage options are not specified, the BasicFiles LOB defaults are used. All SecureFiles LOB-specific storage options and features such as compress, encrypt, or deduplicate throw an exception.
- `IGNORE` disallows SecureFiles LOBs and ignores any errors that forcing BasicFiles LOBs with SecureFiles LOBs options might cause. If `IGNORE` is specified, the `SECUREFILE` keyword and all SecureFiles LOB options are ignored.
- `PERMITTED` allows SecureFiles LOBs to be created, if specified by users. Otherwise, BasicFiles LOBs are created.

- PERFERRED attempts to create a SecureFiles LOB unless BasicFiles LOB is explicitly specified for the LOB or the parent LOB (if the LOB is in a partition or sub-partition). PREFERRED is the default value starting with Oracle Database 12*c*.
- ALWAYS attempts to create SecureFiles LOBs but creates any LOBs not in ASSM tablespaces as BasicFiles LOBs, unless the SECUREFILE parameter is explicitly specified. Any BasicFiles LOB storage options specified are ignored, and the SecureFiles LOB defaults are used for all storage options not specified.
- FORCE attempts to create all LOBs as SecureFiles LOBs even if users specify BASICFILE. This option is not recommended. Instead, PREFERRED or ALWAYS should be used.

Migrating Columns from BasicFiles LOBs to SecureFiles LOBs

This section presents methods of migrating LOBs columns.

Preventing Generation of REDO Data When Migrating to SecureFiles LOBs

Migrating BasicFiles LOB columns generates redo data, which can cause performance problems.

Redo changes for the table are logged during the migration process if the CREATE TABLE statement had the LOGGING clause set.

Redo changes for a column being converted from BasicFiles LOB to SecureFiles LOB are logged if LOGGING is the storage setting for the SecureFiles LOB column. The logging setting (LOGGING or NOLOGGING) for the LOB column is inherited from the tablespace in which the LOB is created.

To prevent redo space generation during migration, specify the NOLOGGING storage parameter for any new SecureFiles LOB columns. You can turn LOGGING on when the migration is complete.

Online Redefinition for BasicFiles LOBs

Online redefinition is the recommended method for migration of BasicFiles LOBs to SecureFiles LOBs. It can be done at the table or partition level.

Online Redefinition Advantages

- No requirement to take the table or partition offline
- Can be done in parallel

Online Redefinition Disadvantages

- Additional storage equal to the entire table or partition required and all LOB segments must be available
- Global indexes must be rebuilt

Using Online Redefinition for Migrating Tables with BasicFiles LOBs

You can also migrate a table using Online Redefinition. Online Redefinition has the advantage of not requiring the table to be off line, but it requires additional free space equal to or even slightly greater than the space used by the table. Example 4–27 demonstrates how to migrate a table using Online Redefinition.

Example 4–27 Example of Online Redefinition

```
REM Grant privileges required for online redefinition.
GRANT EXECUTE ON DBMS_REDEFINITION TO pm;
GRANT ALTER ANY TABLE TO pm;
GRANT DROP ANY TABLE TO pm;
GRANT LOCK ANY TABLE TO pm;
GRANT CREATE ANY TABLE TO pm;
GRANT SELECT ANY TABLE TO pm;
REM Privileges required to perform cloning of dependent objects.
GRANT CREATE ANY TRIGGER TO pm;
GRANT CREATE ANY INDEX TO pm;
CONNECT pm
//  ALTER SESSION FORCE parallel dml;
DROP TABLE cust;
CREATE TABLE cust(c_id NUMBER PRIMARY KEY,
    c_zip NUMBER,
    c_name VARCHAR(30) DEFAULT NULL,
    c_lob CLOB
);
INSERT INTO cust VALUES(1, 94065, 'hhh', 'ttt');
-- Creating Interim Table
-- There is no requirement to specify constraints because they are
-- copied over from the original table.
CREATE TABLE cust_int(c_id NUMBER NOT NULL,
    c_zip NUMBER,
    c_name VARCHAR(30) DEFAULT NULL,
    c_lob CLOB
) LOB(c_lob) STORE AS SECUREFILE (NOCACHE FILESYSTEM_LIKE_LOGGING);
DECLARE
    col_mapping VARCHAR2(1000);
BEGIN
-- map all the columns in the interim table to the original table
    col_mapping :=
    'c_id c_id , '||
    'c_zip c_zip , '||
    'c_name c_name, '||
    'c_lob c_lob';
DBMS_REDEFINITION.START_REDEF_TABLE('pm', 'cust', 'cust_int', col_mapping);
END;
/
DECLARE
    error_count pls_integer := 0;
BEGIN
    DBMS_REDEFINITION.COPY_TABLE_DEPENDENTS('pm', 'cust', 'cust_int',
      1, TRUE,TRUE,TRUE,FALSE, error_count);
    DBMS_OUTPUT.PUT_LINE('errors := ' || TO_CHAR(error_count));
END;
/
EXEC DBMS_REDEFINITION.FINISH_REDEF_TABLE('pm', 'cust', 'cust_int');
-- Drop the interim table
DROP TABLE cust_int;
DESC cust;
-- The following insert statement fails. This illustrates
-- that the primary key constraint on the c_id column is
-- preserved after migration.
INSERT INTO cust VALUES(1, 94065, 'hhh', 'ttt');
SELECT * FROM cust;
```

Parallel Online Redefinition

You can redefine a SecureFiles LOB column in parallel, if the system has sufficient resources for parallel execution.

For parallel execution of online redefinition, add the following statement after the connect statement in Example 4–27, "Example of Online Redefinition" in the last section:

```
ALTER SESSION FORCE PARALLEL DML;
```

PL/SQL Packages for LOBs and DBFS

This section discusses PL/SQL packages that are used with BasicFiles LOBs and SecureFiles LOBs, with an emphasis on changes made to accommodate SecureFiles LOBs and DBFS.

This section includes the following topics:

- Using DBMS_LOB with SecureFiles LOBs and DBFS
- DBMS_SPACE Package

Using DBMS_LOB with SecureFiles LOBs and DBFS

The DBMS_LOB package provides subprograms to operate on, or access and manipulate specific parts of a LOB or complete LOBs. The package applies to both SecureFiles LOB and BasicFiles LOB. See *Oracle Database PL/SQL Packages and Types Reference*, DBMS_LOB package for the complete details of this package.

The following sections describe modifications made to the DBMS_LOB constants and subprograms with the addition of SecureFiles LOB and Database File System (DBFS), which is introduced in Chapter 5.

DBMS_LOB Constants Used with SecureFiles LOBs and DBFS

Table 4–3 lists constants that support **DBFS Link** interfaces. For complete information about constants used in the PL/SQL DBMS_LOB package, see *Oracle Database PL/SQL Packages and Types Reference*.

Table 4–3 DBMS_LOB Constants That Support DBFS Link Interfaces

Constant	Description
DBFS_LINK_NEVER	DBFS link state value
DBFS_LINK_YES	DBFS link state value
DBFS_LINK_NO	DBFS link state value
DBFS_LINK_CACHE	Flag used by COPY_DBFS_LINK() and MOVE_DBFS_LINK().
DBFS_LINK_NOCACHE	Flag used by COPY_DBFS_LINK() and MOVE_DBFS_LINK().
DBFS_LINK_PATH_MAX_SIZE	The maximum length of DBFS pathnames; 1024.
CONTENTTYPE_MAX_SIZE	The maximum 1-byte ASCII characters for content type; 128.

DBMS_LOB Subprograms Used with SecureFiles LOBs and DBFS

Table 4–4 summarizes changes made to PL/SQL package DBMS_LOB subprograms.

Be aware that some of the DBMS_LOB operations that existed before Oracle Database 11*g* Release 2 throw an exception error if the LOB is a DBFS link. To remedy this problem,

modify your applications to explicitly replace the DBFS link with a LOB by calling the DBMS_LOB.COPY_FROM_LINK procedure before they make these calls. When the call completes, then the application can move the updated LOB back to DBFS using the DBMS_LOB.MOVE_TO_DBFS_LINK procedure, if necessary.

Other DBMS_LOB operations that existed before Oracle Database 11*g* Release 2 work transparently if the DBFS Link is in a file system that supports streaming. Note that these operations fail if streaming is either not supported or disabled.

Table 4–4 DBMS_LOB Subprograms

Subprogram	Description
APPEND	Appends the contents of the source LOB to the destination LOB See *Oracle Database PL/SQL Packages and Types Reference*
COMPARE	Compares two LOBs in full or in parts See *Oracle Database PL/SQL Packages and Types Reference*
CONVERTTOBLOB	Converts the character data of a CLOB or NCLOB into the specified character set and writes it in binary format to a destination BLOB See *Oracle Database PL/SQL Packages and Types Reference*
CONVERTTOCLOB	Converts the binary data of a BLOB into the specified character set and writes it in character format to a destination CLOB or NCLOB See *Oracle Database PL/SQL Packages and Types Reference*
COPY	Copies all or part of the source LOB to the destination LOB See *Oracle Database PL/SQL Packages and Types Reference*
COPY_DBFS_LINK	Copies an existing DBFS link into a new LOB See *Oracle Database PL/SQL Packages and Types Reference*
COPY_FROM_DBFS_LINK	Copies the specified LOB data from DBFS HSM Store into the database See *Oracle Database PL/SQL Packages and Types Reference*
DBFS_LINK_GENERATE_PATHNAME	Returns a unique file path name for creating a DBFS Link See *Oracle Database PL/SQL Packages and Types Reference*
ERASE	Erases all or part of a LOB See *Oracle Database PL/SQL Packages and Types Reference*
FRAGMENT_DELETE	Deletes a specified fragment of the LOB See *Oracle Database PL/SQL Packages and Types Reference*
FRAGMENT_INSERT	Inserts a fragment of data into the LOB See *Oracle Database PL/SQL Packages and Types Reference*
FRAGMENT_MOVE	Moves a fragment of a LOB from one location in the LOB to another location See *Oracle Database PL/SQL Packages and Types Reference*
FRAGMENT_REPLACE	Replaces a fragment of a LOB with new data See *Oracle Database PL/SQL Packages and Types Reference*
GET_DBFS_LINK	Returns the DBFS path name for a LOB See *Oracle Database PL/SQL Packages and Types Reference*
GET_DBFS_LINK_STATE	Returns the linking state of a LOB See *Oracle Database PL/SQL Packages and Types Reference*

Table 4–4 (Cont.) DBMS_LOB Subprograms

Subprogram	Description
GETCONTENTTYPE	Retrieves the content type string of the LOB data See *Oracle Database PL/SQL Packages and Types Reference*
GETOPTIONS	Retrieves the previously set options of a specific LOB See *Oracle Database PL/SQL Packages and Types Reference* See Also the *Oracle Call Interface Programmer's Guide* for more information on the corresponding OCI LOB function OCILobGetContentType().
ISSECUREFILE	Determines if a LOB is a SecureFiles LOB See *Oracle Database PL/SQL Packages and Types Reference*
LOADBLOBFROMFILE	Loads BFILE data into a BLOB See *Oracle Database PL/SQL Packages and Types Reference*
LOADCLOBFROMFILE	Loads BFILE data into a CLOB If the CLOB is linked, an exception is thrown. See *Oracle Database PL/SQL Packages and Types Reference*
LOADFROMFILE	Loads BFILE data into a LOB See *Oracle Database PL/SQL Packages and Types Reference*
MOVE_TO_DBFS_LINK	Moves the specified LOB data from the database into DBFS HSM Store See *Oracle Database PL/SQL Packages and Types Reference*
READ	Reads data from a LOB See *Oracle Database PL/SQL Packages and Types Reference*
SET_DBFS_LINK	Links a LOB with a DBFS path name See *Oracle Database PL/SQL Packages and Types Reference*
SETCONTENTTYPE	Sets the content type string of the LOB data See *Oracle Database PL/SQL Packages and Types Reference*
SETOPTIONS	Sets new options for a specific LOB See *Oracle Database PL/SQL Packages and Types Reference* See also *Oracle Call Interface Programmer's Guide* for more information on the corresponding OCI LOB function OCILobSetContentType().
SUBSTR	Returns a fragment of a LOB See *Oracle Database PL/SQL Packages and Types Reference*
TRIM	Trims the LOB to a specified length See *Oracle Database PL/SQL Packages and Types Reference*
WRITE	Writes data to a LOB See *Oracle Database PL/SQL Packages and Types Reference*
WRITEAPPEND	Appends data to the end of a LOB See *Oracle Database PL/SQL Packages and Types Reference*

DBMS_SPACE Package

The DBMS_SPACE PL/SQL package enables you to analyze segment growth and space requirements.

DBMS_SPACE.SPACE_USAGE()

The existing `DBMS_SPACE.SPACE_USAGE` procedure is overloaded to return information about LOB space usage. It returns the amount of disk space in blocks used by all the SecureFiles LOBs in the LOB segment. See *Oracle Database PL/SQL Packages and Types Reference* for more information.

Part II

Database File System (DBFS)

This part covers issues that you must consider when designing applications that use Database File System (DBFS) and DBFS content stores. Note: In most situations, the DBFS requires SecureFiles LOBs, which are discussed in Chapter 4, "Using Oracle LOB Storage". SecureFiles is the default storage mechanism for LOBs starting with Oracle Database 12*c*.

This part contains these chapters:

- Chapter 5, "Introducing the Database File System"
- Chapter 6, "DBFS SecureFiles Store"
- Chapter 7, "DBFS Hierarchical Store"
- Chapter 8, "DBFS Content API"
- Chapter 9, "Creating Your Own DBFS Store"
- Chapter 10, "Using DBFS"

5

Introducing the Database File System

This chapter contains these topics:

Why a Database File System?

Conceptually, a database file system is a file system interface placed on top of files and directories that are stored in database tables.

Applications commonly use the standard SQL data types, `BLOB`s and `CLOB`s, to store and retrieve files in the Oracle Database, files such as medical images, invoice images, documents, videos, and other files. Oracle Database provides much better security, availability, robustness, transactional capability, and scalability than traditional file systems. Files stored in the database along with relational data are automatically backed up, synchronized to the disaster recovery site using Data Guard, and recovered together.

Database File System (DBFS) is a feature of Oracle Database that makes it easier for users to access and manage files stored in the database. With this interface, access to files in the database is no longer limited to programs specifically written to use `BLOB` and `CLOB` programmatic interfaces. Files in the database can now be transparently accessed using any operating system (OS) program that acts on files. For example, ETL (extraction, transformation, and loading) tools can transparently store staging files in the database and file-based applications can benefit from database features such as Maximum Availability Architecture (MAA) without any changes to the applications.

What is Database File System (DBFS)?

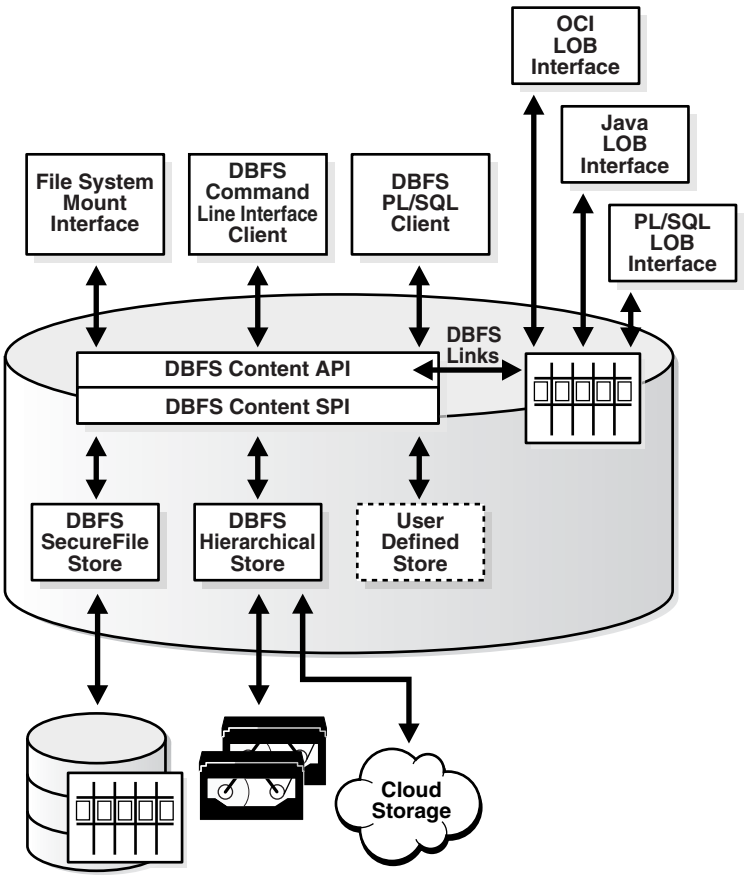
Database File System (DBFS) creates a standard file system interface on top of files and directories that are stored in database tables. DBFS is similar to NFS in that it provides a shared network file system that looks like a local file system and has both a server component and a client component.

At the core of DBFS is the DBFS Content API, a PL/SQL interface in the Oracle Database. It connects to the DBFS Content SPI, a programmatic interface which allows for the support of different types of storage.

At the programming level, the client calls the DBFS Content API to perform a specific function, such as delete a file. The DBFS Content API *delete file* function then calls the DBFS Content SPI to perform that function.

In a user-defined DBFS, the user must implement a delete function based on the specifications in the DBFS Content SPI, along with other functions in the specification.

Figure 5–1 Database File System (DBFS)



DBFS Server

In DBFS, the file server is the Oracle database. Files are stored as SecureFiles LOBs in database tables. An implementation of a file system in the database is called a DBFS content store, for example, the DBFS SecureFiles Store. A DBFS content store allows each database user to create one or more file systems that can be mounted by clients. Each file system has its own dedicated tables that hold the file system content.

> **See Also:** Chapter 8, "DBFS Content API" for information about content stores

The DBFS Content SPI supports different types of stores, as follows:

- DBFS SecureFiles Store: A DBFS content store that uses a table with a SecureFiles LOB column to store the file system data. It implements POSIX-like filesystem capabilities. See Chapter 9, "Creating Your Own DBFS Store".
- DBFS Hierarchical Store: A DBFS content store that allows files to be written to any tape storage units supported by Oracle Recovery Manager (RMAN) or to a cloud storage system. See Chapter 7, "DBFS Hierarchical Store".

- User-defined Store: A content store defined by the user. This allows users to program their own filesystems inside Oracle Database without writing any OS code.

See Chapter 9, "Creating Your Own DBFS Store" for a description of the DBFS Content SPI.

DBFS Client

For client systems, the Database File System offers several access methods.

- PL/SQL Client Interface

  Database applications can access files in the DBFS store directly, through the PL/SQL interface. The PL/SQL interface allows database transactions and read consistency to span relational and file data.

- DBFS Client Command-Line Interface

  A client command-line interface named `dbfs_client` runs on each file system client computer. `dbfs_client` allows users to copy files in and out of the database from any host on the network. It implements simple file system commands such as list and copy in a manner that is similar to shell utilities `ls` and `rcp`. The command interface creates a direct connection to the database without requiring an OS mount of DBFS. Chapter 10, "Using DBFS".

- File System Mount Interface

  On Linux and Solaris, the `dbfs_client` also includes a mount interface that uses the Filesystem in User Space (`FUSE`) kernel module to implement a file-system **mount point** with transparent access to the files stored in the database. This does not require any changes to the Linux or Solaris kernels. It receives standard file system calls from the `FUSE` kernel module and translates them into OCI calls to the PL/SQL procedures in the DBFS content store. See "DBFS Mounting Interface (Linux and Solaris Only)" on page 10-5.

- DBFS Links

  DBFS Links, Database File System Links, are references from SecureFiles LOB locators to files stored outside the database.

  DBFS Links can be used to migrate SecureFiles from existing tables to other storage. See "Database File System Links" on page 7-13 for information on using DBFS Links and "PL/SQL Packages for LOBs and DBFS" on page 4-25 for lists of useful `DBMS_LOB` constants and methods.

What Is a Content Store?

A content store is a collection of documents, each identified by a unique absolute path name, represented as a slash (`/`) followed by one or more component names that are each separated by a slash. Some stores may implement only a flat namespace, others might implement directories or folders implicitly, while still others may implement a comprehensive file system-like collection of entities. These may include hierarchical directories, files, symbolic links, hard links, references, and so on. They often include a rich set of metadata associated with documents, and a rich set of behaviors with respect to security, access control, locking, versioning, content addressing, retention control, and so on.

Because stores are typically designed and evolve independently of each other, applications that use a specific store are either written and packaged by the developers

of the store or else require the user to employ a store-specific API. Designers who create a store-specific API must have a detailed knowledge of the schema of the database tables that are used to implement the store.

6

DBFS SecureFiles Store

This chapter describes the DBFS SecureFiles Store and how to set up and use it.

This chapter contains the following topics:

- Setting Up a SecureFiles Store
- Using a DBFS SecureFiles Store File System
- About DBFS SecureFiles Store Package, DBMS_DBFS_SFS

Setting Up a SecureFiles Store

This section shows how to set up a SecureFiles Store.

This section contains these topics:

- Managing Permissions
- Creating a SecureFiles File System Store
- Initializing SecureFiles Store File Systems
- Comparing SecureFiles LOBs to BasicFiles LOBs

Managing Permissions

You must use a regular database user for all operational access to the Content API and stores. Do not use `SYS` or `SYSTEM` users or `SYSDBA` or `SYSOPER` system privileges. For better security and separation of duty, only allow specific trusted users the ability to manage DBFS Content API operations.

You must grant each user the `DBFS_ROLE` role. Otherwise, the user is not authorized to use the DBFS Content API. A user with suitable administrative privileges (or `SYSDBA`) can grant the role to additional users as needed.

Because of the way roles, access control, and definer and invoker rights interact in the database, it may be necessary to explicitly grant various permissions (typically execute permissions) on DBFS Content API types (SQL types with the `DBMS_DBFS_CONTENT_`*xxx* prefix) and packages (typically only `DBMS_DBFS_CONTENT` and `DBMS_DBFS_SFS`) to users who might otherwise have the `DBFS_ROLE` role.

These explicit, direct grants are normal and to be expected, and can be provided as needed and on demand.

To manage permissions:

1. Create or determine DBFS Content API target users.

This example uses this user and password: `sfs_demo/password`

At minimum, this database user must have the `CREATE SESSION`, `CREATE RESOURCE`, and `CREATE VIEW` privileges.

2. Grant the `DBFS_ROLE` role to the user.

```
CONNECT / as sysdba
GRANT dbfs_role TO sfs_demo;
```

This sets up the DBFS Content API for any database user who has the `DBFS_ROLE` role.

Creating a SecureFiles File System Store

This section describes how to create a SecureFiles file system store.

The `CREATEFILESYSTEM` procedure auto-commits before and after its execution (like a DDL). The method `CREATESTORE` is a wrapper around `CREATEFILESYSTEM`.

See *Oracle Database PL/SQL Packages and Types Reference* for `DBMS_DBFS_SFS` syntax details.

To create a SecureFiles File System Store:

1. Create the necessary stores to be accessed using the DBFS Content API:

```
DECLARE
  BEGIN
    DBMS_DBFS_SFS.CREATEFILESYSTEM(
      store_name => 'FS1',
      tbl_name => 'T1',
      tbl_tbs => null,
      use_bf => false
    );
    COMMIT;
  END;
/
```

where:

- `store_name` is any arbitrary, user-unique name.
- `tbl_name` is a valid table name, created in the current schema.
- `tbl_tbs` is a valid tablespace name used for the store table and its dependent segments, such as indexes, LOBs, or nested tables. The default is `NULL` and specifies a tablespace of the current schema.
- use_bf specifies that BasicFiles LOBs should be used, if `true`, or not used, if `false`.

2. Register this store with the DBFS Content API as a new store managed by the SecureFiles Store provider.

```
CONNECT sfs_demo
Enter password:password
DECLARE
  BEGIN
    DBMS_DBFS_CONTENT.REGISTERSTORE(
      store_name    => 'FS1',
      provider_name => 'anything',
      provider_package => 'dbms_dbfs_sfs'
    );
    COMMIT;
```

```
  END;
/
```

where:

- `store_name` is SecureFiles Store `FS1`, which uses table `SFS_DEMO.T1`.
- `provider_name` is ignored.
- `provider_package` is `DBMS_DBFS_SFS`, for SecureFiles Store reference provider.

This operation associates the SecureFiles Store `FS1` with the `DBMS_DBFS_SFS` provider.

3. Mount the store at suitable a mount-point.

```
CONNECT sfs_demo
Enter password: password
DECLARE
  BEGIN
    DBMS_DBFS_CONTENT.MOUNTSTORE(
      store_name    => 'FS1',
      store_mount   => 'mnt1'
    );
    COMMIT;
  END;
/
```

where:

- `store_name` is SecureFiles Store `FS1`, which uses table `SFS_DEMO.T1`.
- `store_mount` is the **mount point**.

4. [Optional] To see the results of the preceding steps, you can use the following statements.

   - To verify SecureFiles Store tables and file systems:

     ```
     SELECT * FROM TABLE(DBMS_DBFS_SFS.LISTTABLES);
     SELECT * FROM TABLE(DBMS_DBFS_SFS.LISTFILESYSTEMS);
     ```

   - To verify ContentAPI Stores and mounts:

     ```
     SELECT * FROM TABLE(DBMS_DBFS_CONTENT.LISTSTORES);
     SELECT * FROM TABLE(DBMS_DBFS_CONTENT.LISTMOUNTS);
     ```

   - To verify SecureFiles Store features:

     ```
     var fs1f number;
     exec :fs1f := dbms_dbfs_content.getFeaturesByName('FS1');
     select * from table(dbms_dbfs_content.decodeFeatures(:fs1f));
     ```

   - To verify resource and property views:

     ```
     SELECT * FROM DBFS_CONTENT;
     SELECT * FROM DBFS_CONTENT_PROPERTIES;
     ```

You should never directly access tables that hold data for a SecureFiles Store file systems, even through the `DBMS_DBFS_SFS` package methods. The correct way to access the file systems is as follows:

- For procedural operations: Use the DBFS Content API (`DBMS_DBFS_CONTENT` methods).

- For SQL operations: Use the resource and property views (`DBFS_CONTENT` and `DBFS_CONTENT_PROPERTIES`).

Initializing SecureFiles Store File Systems

The procedure `INITFS()` truncates and re-initializes the table associated with the SecureFiles Store `store_name`. The procedure executes like a DDL, auto-committing before and after its execution.

The following example uses file system `FS1` and table `SFS_DEMO.T1`.

```
CONNECT sfs_demo;
Enter password: password
EXEC DBMS_DBFS_SFS.INITFS(store_name => 'FS1');
```

Comparing SecureFiles LOBs to BasicFiles LOBs

SecureFiles LOBs are only available in Oracle Database 11*g* Release 1 and higher. They are not available in earlier releases.

You must use BasicFiles LOB storage for LOB storage in tablespaces that are not managed with Automatic Segment Space Management (ASSM).

Compatibility must be at least 11.1.0.0 to use SecureFiles LOBs.

Additionally, you need to specify the following in `DBMS_DBFS_SFS.CREATEFILESYSTEM`:

- To use SecureFiles LOBs (the default), specify `use_bf => false`.
- To use BasicFiles LOBs, specify `use_bf => true`.

Using a DBFS SecureFiles Store File System

This section describes how to use a SecureFiles Store file system.

This section covers these topics:

- Working with DBFS Content API
- Dropping SecureFiles Store File Systems

Working with DBFS Content API

Assuming the steps in "Setting Up a SecureFiles Store" on page 6-1 have been executed the DBFS Content API permissions set, and at least one SecureFiles Store reference file system is created and mounted under the mount point `/mnt1`, you can create a new file and directory elements as demonstrated in Example 6–1.

Example 6–1 Working with DBFS Content API

```
CONNECT tjones
Enter password: password

DECLARE
   ret integer;
   b   blob;
   str varchar2(1000)  := '' || chr(10) ||

'#include <stdio.h>' || chr(10) ||
'' || chr(10) ||
'int main(int argc, char** argv)' || chr(10) ||
'{' || chr(10) ||
```

```
'    (void) printf("hello world\n");' || chr(10) ||
'    RETURN 0;' || chr(10) ||
'}' || chr(10) ||
'';

    BEGIN
        ret := dbms_fuse.fs_mkdir('/mnt1/src');
        ret := dbms_fuse.fs_creat('/mnt1/src/hello.c', content => b);
        dbms_lob.writeappend(b, length(str), utl_raw.cast_to_raw(str));
        COMMIT;
    END;
    /
    SHOW ERRORS;

    -- verify newly created directory and file
    SELECT pathname, pathtype, length(filedata),
        utl_raw.cast_to_varchar2(filedata)
        FROM dbfs_content
            WHERE pathname LIKE '/mnt1/src%'
            ORDER BY pathname;
```

The file system can be populated and accessed from PL/SQL with `DBMS_DBFS_CONTENT`. The file system can be accessed read-only from SQL using the `dbfs_content` and `dbfs_content_properties` views.

The file system can also be populated and accessed using regular file system APIs and UNIX utilities when mounted using FUSE, or by the standalone `dbfs_client` tool (in environments where `FUSE` is either unavailable or not set up).

Dropping SecureFiles Store File Systems

Use the `unmountStore` method to drop SecureFiles Store file systems.

This method removes all stores referring to the file system from the metadata tables, and drops the underlying file system table. The procedure executes like a DDL, auto-committing before and after its execution.

To drop a SecureFiles Store file system

1. Unmount the store.

```
CONNECT sfs_demo
Enter password: password
DECLARE
  BEGIN
    DBMS_DBFS_CONTENT.UNMOUNTSTORE(
      store_name    => 'FS1',
      store_mount   => 'mntl';
    );
    COMMIT;
END;
/
```

   where:

   - `store_name` is SecureFiles Store `FS1`, which uses table `SFS_DEMO.T1`.
   - `store_mount` is the mount point.

2. Unregister the stores.

```
CONNECT sfs_demo
```

```
Enter password: password
EXEC DBMS_DBFS_CONTENT.UNREGISTERSTORE(store_name => 'FS1');
COMMIT;
```

where `store_name` is SecureFiles Store `FS1`, which uses table `SFS_DEMO.T1`.

3. Drop the file system.

```
CONNECT sfs_demo/******;
EXEC DBMS_DBFS_SFS.DROPFILESYSTEM(store_name => 'FS1');
COMMIT;
```

where `store_name` is SecureFiles Store `FS1`, which uses table `SFS_DEMO.T1`.

About DBFS SecureFiles Store Package, DBMS_DBFS_SFS

The DBFS SecureFiles Store package (`DBMS_DBFS_SFS`) is a store provider for `DBMS_DBFS_CONTENT` that supports SecureFiles LOB storage for DBFS content. To use the `DBMS_DBFS_SFS` package, you must be granted the `DBFS_ROLE` role.

The SecureFiles Store provider is a default implementation of the DBFS Content API (and is a standard example of a store provider that conforms to the Provider SPI) that enables applications that already use LOBs as columns in their schema, to access the `BLOB` columns. This enables existing applications to easily add PL/SQL provider implementations and provide access through the DBFS Content API without changing their schemas or their business logic. Additionally, applications can read and write content that is stored in other (third party) stores through the standard DBFS Content API interface. See Chapter 9, "Creating Your Own DBFS Store" and *Oracle Database PL/SQL Packages and Types Reference* for more information about the Provider SPI defined in `DBMS_DBFS_CONTENT_SPI`.

In a SecureFiles Store, the underlying user data is stored in SecureFiles LOBs and metadata such as pathnames, IDs, and properties are stored as columns in relational tables. See "SecureFiles LOB Storage" on page 4-2 for advanced features of SecureFiles LOBs.

See *Oracle Database PL/SQL Packages and Types Reference* for more information about the `DBMS_DBFS_SFS` package.

7

DBFS Hierarchical Store

This chapter describes the DBFS Hierarchical Store and related store wallet management.

This chapter contains the following topics:

About the Hierarchical Store Package, DBMS_DBFS_HS

The Oracle DBFS Hierarchical Store package (`DBMS_DBFS_HS`) is a store provider for `DBMS_DBFS_CONTENT` that supports hierarchical storage for DBFS content. The package stores content in two external storage devices: tape and the Amazon S3 web service, and associated metadata (or properties) in the database. The DBFS HS may cache frequently accessed content in database tables to improve performance.

The `DBMS_DBFS_HS` package must be used in conjunction with the `DBMS_DBFS_CONTENT` package to manage Hierarchical Storage Management for SecureFiles LOBs using DBFS Links. Using this package, data that is less frequently used can be migrated to a cheaper external device such as tape, achieving significant reduction in storage costs.

The `DBMS_DBFS_HS` package can also be plugged in to the `DBMS_DBFS_CONTENT` package, as a store provider, to implement a tape file system, if the associated external storage device is tape, or a cloud file system, if the associated external storage device is the Amazon S3 storage service.

The `DBMS_DBFS_HS` package provides you the ability to use tape as a storage tier when implementing Information Lifecycle Management (ILM) for database tables or content. The package also supports other forms of storage targets including Web Services like Amazon S3. This service enables users to store data in the database on tape and other forms of storage. The data on tape or Amazon S3 is part of the Oracle Database and all standard APIs can access it, but only through the database.

`DBMS_DBFS_HS` has additional interfaces needed to manage the external storage device and the cache associated with each store.

To use the package `DBMS_DBFS_HS`, you must be granted the `DBFS_ROLE` role.

See Also: *Oracle Database PL/SQL Packages and Types Reference*, for details of the `DBMS_DBFS_HS` Package

Setting up the Store

This section demonstrates how to manage a Hierarchical Store wallet and how to set up, register, and mount a hierarchical Store.

This section covers these topics:

- Managing a HS Store Wallet
- Creating, Registering, and Mounting the Store

Managing a HS Store Wallet

Use the command-line utility `mkstore` to create and manage wallets, using the following commands:

- Create a wallet

  ```
  mkstore -wrl wallet_location -create
  ```

- Add a `KEY` alias

  Specify the `access_key` and `secret_key` aliases by enclosing them within single quotes.

  ```
  mkstore -wrl wallet_location -createCredential alias 'access_key' 'secret_key'
  ```

  For example:

  ```
  mkstore -wrl /home/user1/mywallet -createCredential mykey 'abc' 'xyz'
  ```

- Delete a KEY alias

  ```
  mkstore -wrl wallet_location -deleteCredential alias
  ```

  For example:

  ```
  mkstore -wrl /home/user1/mywallet -deleteCredential mykey
  ```

  See Also:

  - *Oracle Database Advanced Security Guide* for more about creation and management of wallets

Creating, Registering, and Mounting the Store

This section describes how to set up a hierarchical file system store.

To set up a hierarchical file system store:

1. Call `createStore`.

   ```
   DBMS_DBFS_HS.createStore( store_name, store_type, tbl_name, tbs_name, cache_
   size, lob_cache_quota, optimal_tarball_size, schema_name);
   ```

2. Set mandatory and optional properties using the following interface:

   ```
   DBMS_DBFS_HS.setStoreProperty(StoreName, PropertyName, PropertyValue);
   ```

   For `store_type` = `STORETYPE_TAPE`, mandatory properties are:

PROPNAME_DEVICELIBRARY, PROPNAME_MEDIAPOOL, PROPNAME_CACHESIZE.

PROPNAME_CACHESIZE is already set by createStore.

You can change the value of PROPNAME_CACHESIZE using reconfigCache.

Optional properties are:

PROPNAME_OPTTARBALLSIZE, PROPNAME_READCHUNKSIZE, PROPNAME_WRITECHUNKSIZE, PROPNAME_STREAMABLE.

For store_type = STORETYPE_AMAZONS3 mandatory properties are:

PROPNAME_DEVICELIBRARY, PROPNAME_CACHESIZE, PROPNAME_S3HOST,PROPNAME_BUCKET, PROPNAME_LICENSEID, PROPNAME_WALLET.

PROPNAME_CACHESIZE is set by createStore. You can change the value of PROPNAME_CACHESIZE using reconfigCache.

Optional properties are:

PROPNAME_OPTTARBALLSIZE, PROPNAME_READCHUNKSIZE, PROPNAME_WRITECHUNKSIZE, PROPNAME_STREAMABLE, PROPNAME_HTTPPROXY.

3. Register the store with DBFS Content API using:

```
DBMS_DBFS_CONTENT.registerStore(store_name, provider_name, provider_package);
```

Note: provider_package is the dbms_dbfs_hs package.

4. Mount the stores for access using:

```
DBMS_DBFS_CONTENT.mountStore(store_name, store_mount, singleton,principal,
  owner, acl, asof, read_only);
```

Using the Hierarchical Store

The Hierarchical Store can be used as an independent file system or as an archive solution for SecureFiles LOBs.

This section covers the following topics:

- Using Hierarchical Store as a File System
- Using Hierarchical Store as an Archive Solution For SecureFiles LOBs
- Dropping a Hierarchical Store
- Using Compression with the Hierarchical Store
- Using Tape
- Using Amazon S3

Using Hierarchical Store as a File System

Use the DBMS_DBFS_CONTENT package to create, update, read, and delete file system entries in the store.

Refer to Chapter 8, "DBFS Content API" for details.

Using Hierarchical Store as an Archive Solution For SecureFiles LOBs

Use the DBMS_LOB package to archive SecureFiles LOBs in a tape or S3 store, as described in "PL/SQL Packages for LOBs and DBFS" on page 4-25.

To free space in the cache or to force cache resident contents to be written to external storage device, call:

```
DBMS_DBFS_HS.storePush(store_name);
```

Dropping a Hierarchical Store

To drop a hierarchical store, call:

```
DBMS_DBFS_HS.dropStore(store_name, opt_flags);
```

Using Compression with the Hierarchical Store

The DBFS hierarchical store has the ability to store its files in compressed form using the SETPROPERTY method and the property PROPNAME_COMPRESSLVL to specify the compression level.

Valid values are:

- PROPVAL_COMPLVL_NONE: No compression
- PROPVAL_COMPLVL_LOW: LOW compression
- PROPVAL_COMPLVL_MEDIUM: MEDIUM compression
- PROPVAL_COMPLVL_HIGH: HIGH compression

Generally, the compression level LOW has the best performance while still providing a good compression ratio. Compression levels MEDIUM and HIGH provide significantly better compression ratios, but compression times can be correspondingly longer. Oracle recommends using NONE or LOW when write performance is critical, such as when files in the DBFS HS store are updated frequently. If space is critical and the best possible compression ratio is desired, use MEDIUM or HIGH.

Files are compressed as they are paged out of the cache into the staging area (before they are subsequently pushed into the back end tape or S3 storage). Therefore, compression also benefits by storing smaller files in the staging area and effectively increasing the total available capacity of the staging area.

Using Tape

The following example program configures and uses a tape store.

Valid values must be substituted in some places, indicated by <...>, for the program to run successfully.

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* DBMS_DBFS_HS documentation for complete details about the methods and their parameters

```
Rem Example to configure and use a Tape store.
Rem
Rem hsuser should be a valid database user who has been granted
Rem the role dbfs_role.

connect hsuser/hsuser
```

```
Rem The following block sets up a STORETYPE_TAPE store with
Rem DBMS_DBFS_HS acting as the store provider.

declare
storename varchar2(32) ;
tblname varchar2(30) ;
tbsname varchar2(30) ;
lob_cache_quota number := 0.8 ;
cachesz number ;
ots number ;
begin
cachesz := 50 * 1048576 ;
ots := 1048576 ;
storename := 'tapestore10' ;
tblname := 'tapetbl10' ;
tbsname := '<TBS_3>' ; -- Substitute a valid tablespace name

-- Create the store.
-- Here tbsname is the tablespace used for the store,
-- tblname is the table holding all the store entities,
-- cachesz is the space used by the store to cache content
--   in the tablespace,
-- lob_cache_quota is the fraction of cachesz allocated
--   to level-1 cache and
-- ots is minimum amount of content that is accumulated
--   in level-2 cache before being stored on tape
dbms_dbfs_hs.createStore(
  storename,
  dbms_dbfs_hs.STORETYPE_TAPE,
  tblname, tbsname, cachesz,
  lob_cache_quota, ots) ;

dbms_dbfs_hs.setstoreproperty(
  storename,
  dbms_dbfs_hs.PROPNAME_SBTLIBRARY,
  '<ORACLE_HOME/work/libobkuniq.so>') ;
  -- Substitute your ORACLE_HOME path

dbms_dbfs_hs.setstoreproperty(
  storename,
  dbms_dbfs_hs.PROPNAME_MEDIAPOOL,
  '<0>') ;  -- Substitute valid value

dbms_dbfs_hs.setstoreproperty(
  storename,
  dbms_dbfs_hs.PROPNAME_COMPRESSLEVEL,
  'NONE') ;

-- Please refer to DBMS_DBFS_CONTENT documentation
-- for details about this method
dbms_dbfs_content.registerstore(
  storename,
  'tapeprvder10',
  'dbms_dbfs_hs') ;

-- Please refer to DBMS_DBFS_CONTENT documentation
-- for details about this method
dbms_dbfs_content.mountstore(storename, 'tapemnt10') ;
end ;
/
```

```
Rem The following code block does file operations
Rem using DBMS_DBFS_CONTENT on the store configured
Rem in the previous code block

connect hsuser/hsuser

declare
  path varchar2(256) ;
  path_pre varchar2(256) ;
  mount_point varchar2(32) ;
  store_name varchar2(32) ;
  prop1 dbms_dbfs_content_properties_t ;
  prop2 dbms_dbfs_content_properties_t ;
  mycontent blob := empty_blob() ;
  buffer varchar2(1050) ;
  rawbuf raw(1050) ;
  outcontent blob := empty_blob() ;
  itemtype integer ;
  pflag integer ;
  filecnt integer ;
  iter integer ;
  offset integer ;
  rawlen integer ;
begin

  mount_point := '/tapemnt10' ;
  store_name := 'tapestore10' ;
  path_pre := mount_point ||'/file' ;


-- We create 10 empty files in the following loop
  filecnt := 0 ;
  loop
    exit when filecnt = 10 ;
    path := path_pre || to_char(filecnt) ;
    mycontent := empty_blob() ;
    prop1 := null ;

    -- Please refer to DBMS_DBFS_CONTENT documentation
    -- for details about this method
    dbms_dbfs_content.createFile(
      path, prop1, mycontent) ; -- Create the file

    commit ;
    filecnt := filecnt + 1 ;
  end loop ;

  -- We populate the newly created files with content
  -- in the following loop
  pflag := dbms_dbfs_content.prop_data +
           dbms_dbfs_content.prop_std  +
           dbms_dbfs_content.prop_opt  ;

  buffer := 'Oracle provides an integrated management '  ||
            'solution for managing Oracle database with '||
            'a unique top-down application management '  ||
            'approach. With new self-managing '          ||
            'capabilities, Oracle eliminates time-'      ||
            'consuming, error-prone administrative '     ||
```

```
          'tasks, so database administrators can '     ||
          'focus on strategic business objectives '    ||
          'instead of performance and availability '   ||
          'fire drills. Oracle Management Packs for '  ||
          'Database provide signifiCant cost and time-'||
          'saving capabilities for managing Oracle '   ||
          'Databases. Independent studies demonstrate '||
          'that Oracle Database is 40 percent easier ' ||
          'to manage over DB2 and 38 percent over '    ||
          'SQL Server.';

rawbuf := utl_raw.cast_to_raw(buffer) ;
rawlen := utl_raw.length(rawbuf) ;
offset := 1 ;
filecnt := 0 ;
loop
  exit when filecnt = 10 ;
  path := path_pre || to_char(filecnt) ;
  prop1 := null;

  -- Append buffer to file
  -- Please refer to DBMS_DBFS_CONTENT documentation
  -- for details about this method
  dbms_dbfs_content.putpath(
    path, prop1, rawlen,
    offset, rawbuf) ;

  commit ;
  filecnt := filecnt + 1 ;
end loop ;

-- Clear out level 1 cache
dbms_dbfs_hs.flushCache(store_name) ;
commit ;

-- Do write operation on even-numbered files.
-- Do read operation on odd-numbered files.
filecnt := 0 ;
loop
  exit when filecnt = 10;
  path := path_pre || to_char(filecnt) ;
  if mod(filecnt, 2) = 0 then
    -- Get writable file
    -- Please refer to DBMS_DBFS_CONTENT documentation
    -- for details about this method
    dbms_dbfs_content.getPath(
      path, prop2, outcontent, itemtype,
      pflag, null, true) ;

    buffer := 'Agile businesses want to be able to '    ||
              'quickly adopt new technologies, whether '||
              'operating systems, servers, or '         ||
              'software, to help them stay ahead of '   ||
              'the competition. However, change often ' ||
              'introduces a period of instability into '||
              'mission-critical IT systems. Oracle '    ||
              'Real Application Testing-with Oracle '   ||
              'Database 11g Enterprise Edition-allows ' ||
              'businesses to quickly adopt new '        ||
              'technologies while eliminating the '     ||
```

```
                'risks associated with change. Oracle '   ||
                'Real Application Testing combines a '    ||
                'workload capture and replay feature '    ||
                'with an SQL performance analyzer to '    ||
                'help you test changes against real-life '||
                'workloads, and then helps you fine-tune '||
                'the changes before putting them into'    ||
                'production. Oracle Real Application '    ||
                'Testing supports older versions of '     ||
                'Oracle Database, so customers running '  ||
                'Oracle Database 9i and Oracle Database ' ||
                '10g can use it to accelerate their '     ||
                'database upgrades. ';

      rawbuf := utl_raw.cast_to_raw(buffer) ;
      rawlen := utl_raw.length(rawbuf) ;

      -- Modify file content
      -- Please refer to DBMS_DBFS_CONTENT documentation
      -- for details about this method
      dbms_lob.write(outcontent, rawlen, 10, rawbuf);
      commit ;
    else
      -- Read the file
      -- Please refer to DBMS_DBFS_CONTENT documentation
      -- for details about this method
      dbms_dbfs_content.getPath(
        path, prop2, outcontent, itemtype, pflag) ;
    end if ;
    filecnt := filecnt + 1 ;
  end loop ;

  -- Delete the first 2 files
  filecnt := 0;

  loop
    exit when filecnt = 2 ;
    path := path_pre || to_char(filecnt) ;
    -- Delete file
    -- Please refer to DBMS_DBFS_CONTENT documentation
    -- for details about this method
    dbms_dbfs_content.deleteFile(path) ;
    commit ;
    filecnt := filecnt + 1 ;
  end loop ;


  -- Move content staged in database to Amazon S3 store
  dbms_dbfs_hs.storePush(store_name) ;
  commit ;

end ;
/
```

Using Amazon S3

The following example program configures and uses an Amazon S3 store.

Valid values must be substituted in some places, indicated by <...>, for the program to run successfully.

See Also: *Oracle Database PL/SQL Packages and Types Reference* DBMS_DBFS_HS documentation for complete details about the methods and their parameters

```
Rem Example to configure and use an Amazon S3 store.
Rem
Rem hsuser should be a valid database user who has been granted
Rem the role dbfs_role.

connect hsuser/hsuser

Rem The following block sets up a STORETYPE_AMAZONS3 store with
Rem DBMS_DBFS_HS acting as the store provider.

declare
storename varchar2(32) ;
tblname varchar2(30) ;
tbsname varchar2(30) ;
lob_cache_quota number := 0.8 ;
cachesz number ;
ots number ;
begin
cachesz := 50 * 1048576 ;
ots := 1048576 ;
storename := 's3store10' ;
tblname := 's3tbl10' ;
tbsname := '<TBS_3>' ; -- Substitute a valid tablespace name

-- Create the store.
-- Here tbsname is the tablespace used for the store,
-- tblname is the table holding all the store entities,
-- cachesz is the space used by the store to cache content
--   in the tablespace,
-- lob_cache_quota is the fraction of cachesz allocated
--   to level-1 cache and
-- ots is minimum amount of content that is accumulated
--   in level-2 cache before being stored in AmazonS3
dbms_dbfs_hs.createStore(
  storename,
  dbms_dbfs_hs.STORETYPE_AMAZONS3,
  tblname, tbsname, cachesz,
  lob_cache_quota, ots) ;

dbms_dbfs_hs.setstoreproperty(storename,
  dbms_dbfs_hs.PROPNAME_SBTLIBRARY,
  '<ORACLE_HOME/work/libosbws11.so>');
  -- Substitute your ORACLE_HOME path

dbms_dbfs_hs.setstoreproperty(
  storename,
  dbms_dbfs_hs.PROPNAME_S3HOST,
  's3.amazonaws.com') ;

dbms_dbfs_hs.setstoreproperty(
  storename,
  dbms_dbfs_hs.PROPNAME_BUCKET,
  'oras3bucket10') ;

dbms_dbfs_hs.setstoreproperty(
  storename,
```

```
  dbms_dbfs_hs.PROPNAME_WALLET,
  'LOCATION=file:<ORACLE_HOME>/work/wlt CREDENTIAL_ALIAS=a_key') ;
  -- Substitute your ORACLE_HOME path

dbms_dbfs_hs.setstoreproperty(
  storename,
  dbms_dbfs_hs.PROPNAME_LICENSEID,
  '<xxxxxxxxxxxxxxxx>') ; -- Substitute a valid SBT license id

dbms_dbfs_hs.setstoreproperty(
  storename,
  dbms_dbfs_hs.PROPNAME_HTTPPROXY,
  '<http://www-proxy.mycompany.com:80/>') ;
  -- Substitute valid value. If a proxy is not used,
  -- then this property need not be set.

dbms_dbfs_hs.setstoreproperty(
  storename,
  dbms_dbfs_hs.PROPNAME_COMPRESSLEVEL,
  'NONE') ;

dbms_dbfs_hs.createbucket(storename) ;

-- Please refer to DBMS_DBFS_CONTENT documentation
-- for details about this method
dbms_dbfs_content.registerstore(
  storename,
  's3prvder10',
  'dbms_dbfs_hs') ;

-- Please refer to DBMS_DBFS_CONTENT documentation
-- for details about this method
dbms_dbfs_content.mountstore(
  storename,
  's3mnt10') ;
end ;
/

Rem The following code block does file operations
Rem using DBMS_DBFS_CONTENT on the store configured
Rem in the previous code block

connect hsuser/hsuser

declare
path varchar2(256) ;
path_pre varchar2(256) ;
mount_point varchar2(32) ;
store_name varchar2(32) ;
prop1 dbms_dbfs_content_properties_t ;
prop2 dbms_dbfs_content_properties_t ;
mycontent blob := empty_blob() ;
buffer varchar2(1050) ;
rawbuf raw(1050) ;
outcontent blob := empty_blob() ;
itemtype integer ;
pflag integer ;
filecnt integer ;
iter integer ;
offset integer ;
```

```
rawlen integer ;
begin

  mount_point := '/s3mnt10' ;
  store_name := 's3store10' ;
  path_pre := mount_point ||'/file' ;

  -- We create 10 empty files in the following loop
  filecnt := 0 ;
  loop
    exit when filecnt = 10 ;
    path := path_pre || to_char(filecnt) ;
    mycontent := empty_blob() ;
    prop1 := null ;

    -- Please refer to DBMS_DBFS_CONTENT documentation
    -- for details about this method
    dbms_dbfs_content.createFile(
      path, prop1, mycontent) ; -- Create the file

    commit ;
    filecnt := filecnt + 1 ;
  end loop ;

  -- We populate the newly created files with content
  -- in the following loop
  pflag := dbms_dbfs_content.prop_data +
           dbms_dbfs_content.prop_std  +
           dbms_dbfs_content.prop_opt  ;

  buffer := 'Oracle provides an integrated management '  ||
            'solution for managing Oracle database with '||
            'a unique top-down application management '  ||
            'approach. With new self-managing '          ||
            'capabilities, Oracle eliminates time-'      ||
            'consuming, error-prone administrative '     ||
            'tasks, so database administrators can '     ||
            'focus on strategic business objectives '    ||
            'instead of performance and availability '   ||
            'fire drills. Oracle Management Packs for '  ||
            'Database provide signifiCant cost and time-'||
            'saving capabilities for managing Oracle '   ||
            'Databases. Independent studies demonstrate '||
            'that Oracle Database is 40 percent easier ' ||
            'to manage over DB2 and 38 percent over '    ||
            'SQL Server.';

  rawbuf := utl_raw.cast_to_raw(buffer) ;
  rawlen := utl_raw.length(rawbuf) ;
  offset := 1 ;
  filecnt := 0 ;
  loop
    exit when filecnt = 10 ;
    path := path_pre || to_char(filecnt) ;
    prop1 := null;

    -- Append buffer to file
    -- Please refer to DBMS_DBFS_CONTENT documentation
    -- for details about this method
    dbms_dbfs_content.putpath(
```

```
    path, prop1, rawlen,
    offset, rawbuf) ;

  commit ;
  filecnt := filecnt + 1 ;
end loop ;

-- Clear out level 1 cache
dbms_dbfs_hs.flushCache(store_name) ;
commit ;

-- Do write operation on even-numbered files.
-- Do read operation on odd-numbered files.
filecnt := 0 ;
loop
  exit when filecnt = 10;
  path := path_pre || to_char(filecnt) ;
  if mod(filecnt, 2) = 0 then
    -- Get writable file
    -- Please refer to DBMS_DBFS_CONTENT documentation
    -- for details about this method
    dbms_dbfs_content.getPath(
      path, prop2, outcontent, itemtype,
      pflag, null, true) ;

    buffer := 'Agile businesses want to be able to '    ||
              'quickly adopt new technologies, whether '||
              'operating systems, servers, or '         ||
              'software, to help them stay ahead of '   ||
              'the competition. However, change often ' ||
              'introduces a period of instability into '||
              'mission-critical IT systems. Oracle '    ||
              'Real Application Testing-with Oracle '   ||
              'Database 11g Enterprise Edition-allows ' ||
              'businesses to quickly adopt new '        ||
              'technologies while eliminating the '     ||
              'risks associated with change. Oracle '   ||
              'Real Application Testing combines a '    ||
              'workload capture and replay feature '    ||
              'with an SQL performance analyzer to '    ||
              'help you test changes against real-life '||
              'workloads, and then helps you fine-tune '||
              'the changes before putting them into'    ||
              'production. Oracle Real Application '    ||
              'Testing supports older versions of '     ||
              'Oracle Database, so customers running '  ||
              'Oracle Database 9i and Oracle Database ' ||
              '10g can use it to accelerate their '     ||
              'database upgrades. ';

    rawbuf := utl_raw.cast_to_raw(buffer) ;
    rawlen := utl_raw.length(rawbuf) ;

    -- Modify file content
    -- Please refer to DBMS_DBFS_CONTENT documentation
    -- for details about this method
    dbms_lob.write(outcontent, rawlen, 10, rawbuf);
    commit ;
  else
    -- Read the file
```

```
      -- Please refer to DBMS_DBFS_CONTENT documentation
      -- for details about this method
      dbms_dbfs_content.getPath(
        path, prop2, outcontent, itemtype, pflag) ;
    end if ;
    filecnt := filecnt + 1 ;
  end loop ;

  -- Delete the first 2 files
  filecnt := 0;

  loop
    exit when filecnt = 2 ;
    path := path_pre || to_char(filecnt) ;
    -- Delete file
    -- Please refer to DBMS_DBFS_CONTENT documentation
    -- for details about this method
    dbms_dbfs_content.deleteFile(path) ;
    commit ;
    filecnt := filecnt + 1 ;
  end loop ;


  -- Move content staged in database to Amazon S3 store
  dbms_dbfs_hs.storePush(store_name) ;
  commit ;

end ;
/
```

Database File System Links

This section introduces Database File System Links. It contains the following topics:

- Overview of Database File System Links
- Creating Database File System Links
- Copying Database File System Links
- Copying a Linked LOB Between Tables
- Online Redefinition and DBFS Links
- Transparent Read

Overview of Database File System Links

DBFS Links provide the ability to transparently store SecureFiles LOBs in a location separate from the segment where the LOB would normally be stored, and instead store a link to the LOB in the segment. The link must reference a path that uses DBFS to locate the LOB when accessed. This means that the LOB could be stored on another file system, or on a tape system, or in the cloud, or any other location that can be accessed using DBFS.

When a user or application tries to access a SecureFiles LOB that has been stored outside the segment using a DBFS Link, the behavior can vary depending on the attempted operation and the characteristics of the DBFS store that is holding the LOB:

- Read:

If the LOB is not already cached in a local area in the database, then it can be read directly from the DBFS content store that holds it, if the content store allows streaming access based on the setting of the `PROPNAME_STREAMABLE` parameter. If the content store does not allow streaming access, then the entire LOB will first be read into a local area in the database, where it will be stored for a period of time for future access.

- Write:

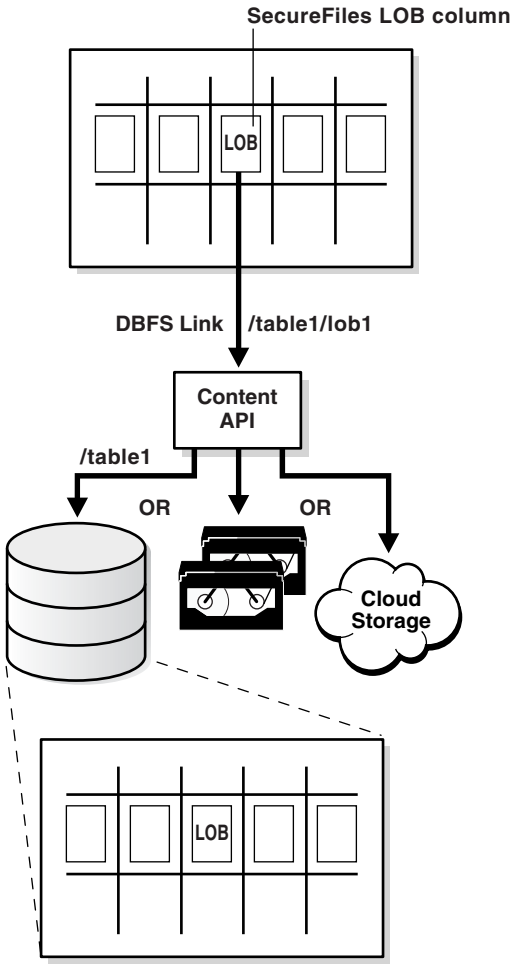
  If the LOB is not already cached in a local area in the database, then it will first be read into the database, modified as needed, and then written back to the DBFS content store defined in the DBFS Link for the LOB in question.

- Delete:

  When a SecureFiles LOB that is stored through a DBFS Link is deleted, the DBFS Link is deleted from the table, but the LOB itself is NOT deleted from the DBFS content store. Or it is more complex, based on the characteristics/settings, of the DBFS content store in question.

DBFS Links enable the use of SecureFiles LOBs to implement Hierarchical Storage Management (HSM) in conjunction with the DBFS Hierarchical Store (DBFS HS). HSM is a process by which the database moves rarely used or unused data from faster, more expensive, and smaller storage to slower, cheaper, and higher capacity storage.

Figure 7–1 Database File System Link



Creating Database File System Links

Database File System Links require the creation of a Database File System through the use of the DBFS Content package, `DBMS_DBFS_CONTENT`.

Oracle provides several methods for creating a DBFS Link:

- Move SecureFiles LOB data into a specified DBFS pathname and store the reference to the new location in the LOB.

  Call `DBMS_LOB.MOVE_TO_DBFS_LINK()` with LOB and DBFS path name arguments, and the system creates the specified DBFS HSM Store if it does not exist, copies data from the SecureFiles LOB into the specified DBFS HSM Store, removes data from the SecureFiles LOB, and stores the file path name for subsequent access through this LOB.

- Copy or create a reference to an existing file.

  Call `DBMS_LOB.COPY_DBFS_LINK()` to copy a link from an existing DBFS Link. If there is any data in the destination SecureFiles LOB, the system removes this data and stores a copy of the reference to the link in the destination SecureFiles LOB.

- Call `DBMS_LOB.SET_DBFS_LINK()`, which assumes that the data for the link is stored in the specified DBFS path name.

  The system removes data in the specified SecureFiles LOB and stores the link to the DBFS path name.

Creating a DBFS Link impacts which operations may be performed and how. Any `DBMS_LOB` operations that modify the contents of a LOB will throw an exception if the underlying lob has been moved into a DBFS Link. The application must explicitly replace the DBFS Link with a LOB by calling `DBMS_LOB.COPY_FROM_LINK()` before making these calls. When completed, the application can move the updated LOB back to DBFS using `DBMS_LOB.MOVE_TO_DBFS_LINK()`, if needed. Other `DBMS_LOB` operations that existed before Oracle Database 11*g* Release 2 work transparently if the DBFS Link is in a file system that supports streaming. Note that these operations fail if streaming is either not supported or disabled.

If the DBFS Link file is modified through DBFS interfaces directly, the change is reflected in subsequent reads of the SecureFiles LOB. If the file is deleted through DBFS interfaces, then an exception occurs on subsequent reads.

For the database, it is also possible that a DBA may not want to store all of the data stored in a SecureFiles LOB HSM during export and import. Oracle has the ability to export and import only the Database File System Links. The links are fully qualified identifiers that provide access to the stored data, when entered into a SecureFiles LOB or registered on a SecureFiles LOB in a different database. This ability to export and import a link is similar to the common file system functionality of symbolic links.

The newly imported link is only available as long as the source, the stored data, is available, or until the first retrieval occurs on the imported system. The application is responsible for stored data retention. If the application system removes data from the store that still has a reference to it, the database throws an exception when the referencing SecureFiles LOB(s) attempt to access the data. Oracle also supports continuing to keep the data in the database after migration out to a DBFS store as a cached copy. It is up to the application to purge these copies in compliance with its retention policies.

Copying Database File System Links

The API `DBMS_LOB.COPY_DBFS_LINK(DSTLOB, SRCLOB, FLAGS)` provides the ability to copy a linked SecureFiles LOB. By default, the LOB is not obtained from the DBFS HSM Store during this operation; this is a copy-by-reference operation that exports the DBFS path name (at source side) and imports it (at destination side). The `flags` argument can dictate that the destination has a local copy in the database and references the LOB data in the DBFS HSM Store.

Copying a Linked LOB Between Tables

`CREATE TABLE ... AS SELECT (CTAS)` and `INSERT TABLE ... AS SELECT (ITAS)` copies any DBFS Links that are stored in any SecureFiles LOBs in the source table to the destination table.

Online Redefinition and DBFS Links

Online redefinition copies any DBFS Links that are stored in any SecureFiles LOBs in the table being redefined.

Transparent Read

DBFS Links have the ability to read from a linked SecureFiles LOB even if the data is not cached in the database. This is done by reading the data from the content store where the data is currently stored, and streaming that data back to the user application as if it were being read from the SecureFiles LOB segment. This allows seamless access to the DBFS Linked data without the prerequisite first call to `DBMS_LOB.COPY_FROM_DBFS_LINK()`.

Whether or not transparent read is available for a particular SecureFiles LOB is determined by the `DBFS_CONTENT` store where the data resides. This feature is always enabled for `DBFS_SFS` stores, and by default for `DBFS_HS` stores. To disable transparent read for `DBFS_HS` store, set the `PROPNAME_STREAMABLE` parameter to `FALSE`.

See Also: *Oracle Database PL/SQL Packages and Types Reference*

The DBMS_DBFS_HS Package

This section covers these topics:

- Constants for DBMS_DBFS_HS Package
- Methods for DBMS_DBFS_HS Package

Constants for DBMS_DBFS_HS Package

See *Oracle Database PL/SQL Packages and Types Reference* for details of constants used by `DBMS_DBFS_HS` PL/SQL package

Methods for DBMS_DBFS_HS Package

Table 7–1 summarizes the `DBMS_DBFS_HS` PL/SQL package methods. See *Oracle Database PL/SQL Packages and Types Reference* for all details of this package.

Table 7–1 Methods of the DBMS_DBFS_HS PL/SQL Packages

Method	Description
CLEANUPUNUSEDBACKUPFILES	Removes files that are created on the external storage device if they have no current content. *Oracle Database PL/SQL Packages and Types Reference*
CREATEBUCKET	Creates an AWS bucket, for use with the STORETYPE_AMAZON3 store. *Oracle Database PL/SQL Packages and Types Reference*
CREATESTORE	Creates a DBFS HS store. *Oracle Database PL/SQL Packages and Types Reference*
DEREGSTORECOMMAND	Removes a command (message) that was associated with a store. *Oracle Database PL/SQL Packages and Types Reference*
DROPSTORE	Deletes a previously created DBFS HS store. *Oracle Database PL/SQL Packages and Types Reference*
FLUSHCACHE	Flushes out level 1 cache to level 2 cache, increasing space in level 1. *Oracle Database PL/SQL Packages and Types Reference*
GETSTOREPROPERTY	Retrieves the values of a property of a store in the database. *Oracle Database PL/SQL Packages and Types Reference*
RECONFIGCACHE	Reconfigures the parameters of the database cache used by the store. *Oracle Database PL/SQL Packages and Types Reference*
REGISTERSTORECOMMAND	Registers commands (messages) for a store so they are sent to the Media Manager of an external storage device. *Oracle Database PL/SQL Packages and Types Reference* .
SENDCOMMAND	Sends a command (message) to the Media Manager of an external storage device. *Oracle Database PL/SQL Packages and Types Reference*
SETSTOREPROPERTY	Associates name/value properties with a registered Hierarchical Store. *Oracle Database PL/SQL Packages and Types Reference*
STOREPUSH	Pushes locally cached data to an archive store. *Oracle Database PL/SQL Packages and Types Reference*

Views for DBFS Hierarchical Store

This section lists views for DBFS Hierarchical Stores.

> **See Also:** *Oracle Database Reference* for the columns and data types of these views

DBA Views

These views for DBFS Hierarchical Store are available:

- DBA_DBFS_HS

  This view shows all Database File System (DBFS) hierarchical stores

- DBA_DBFS_HS_PROPERTIES

  This view shows modifiable properties of all Database File System (DBFS) hierarchical stores.

- DBA_DBFS_HS_FIXED_PROPERTIES

  This view shows non-modifiable properties of all Database File System (DBFS) hierarchical stores.

- DBA_DBFS_HS_COMMANDS

  This view shows all the registered store commands for all Database File System (DBFS) hierarchical stores.

User Views

These views for DBFS Hierarchical Store are available:

- USER_DBFS_HS

  This view shows all Database File System (DBFS) hierarchical stores owned by the current user.

- USER_DBFS_HS_PROPERTIES

  This view shows modifiable properties of all Database File System (DBFS) hierarchical stores owned by current user.

- USER_DBFS_HS_FIXED_PROPERTIES

  This view shows non-modifiable properties of all Database File System (DBFS) hierarchical stores owned by current user.

- USER_DBFS_HS_COMMANDS

  This view shows all the registered store commands for all Database File system (DBFS) hierarchical stores owned by current user.

- USER_DBFS_HS_FILES

  This view shows files in the Database File System (DBFS) hierarchical store owned by the current user and their location on the backend device.

8

DBFS Content API

This chapter explains how developers can enable applications to use DBFS.

This chapter contains these topics:

- Overview of DBFS Content API
- Stores and DBFS Content API

Overview of DBFS Content API

The DBFS Content API (`DBMS_DBFS_CONTENT`) is a client-side programmatic API package which allows applications to use DBFS. These applications can be written in SQL, PL/SQL, JDBC, OCI, and other programming environments.

The DBFS Content API is a collection of methods that provide a file system-like abstraction. It is backed by one or more DBFS Store Providers. The *Content* in the DBFS Content interface refers to a file, including metadata, and it can either map to a SecureFiles `LOB` (and other columns) in a table or be dynamically created by user-written plug-ins in Java or PL/SQL that run inside the database. The plug-in form is referred to as a *provider*.

> **Note:** The DBFS Content API includes the SecureFiles Store Provider, `DBMS_DBFS_SFS`, a default implementation that enables applications that already use LOBs as columns in their schema, to access the `LOB` columns as files. See Chapter 6, "DBFS SecureFiles Store".

Examples of possible providers include:

- Packaged applications that want to surface data through files.
- Custom applications developers would like to use to leverage the file system interface. For example, an application that stores medical images.

Stores and DBFS Content API

The DBFS Content API takes the common features of various stores and forms them into a simple interface that can be used to build portable client applications, while allowing different stores to implement the set of features they choose.

The DBFS Content API aggregates the path namespace of one or more stores into a single unified namespace, using the first component of the path name to disambiguate the namespace and then presents it to client applications. This allows clients to access

the underlying documents using either a full absolute path name represented by a single string, in this form:

```
/store-name/store-specific-path-name
```

or a store-qualified path name as a string 2-tuple, in this form:

```
["store-name","/store-specific-path-name"]
```

The DBFS Content API then takes care of correctly dispatching various operations on path names to the appropriate stores and integrating the results back into the client-desired namespace.

Store providers must conform to the store provider interface (SPI) as declared by the package `DBMS_DBFS_CONTENT_SPI`. See Chapter 9, "Creating Your Own DBFS Store" for further information.

See *Oracle Database PL/SQL Packages and Types Reference* for `DBMS_DBFS_CONTENT` package syntax reference.

Getting Started with DBMS_DBFS_CONTENT Package

`DBMS_DBFS_CONTENT` is part of the Oracle Database, starting with Oracle Database 11g Release 2, and does not need to be installed.

See *Oracle Database PL/SQL Packages and Types Reference* for more information.

DBFS Content API Role

Access to the content operational and administrative API (packages, types, tables, and so on) is available through `DBFS_ROLE`. This role can be granted to all users as needed.

Path Name Constants and Types

Path name constants are modeled after their SecureFiles LOBs store counterparts. See *Oracle Database PL/SQL Packages and Types Reference* for path name constants and their types.

Path Properties

Every path name in a store is associated with a set of properties. For simplicity and generality, each property is identified by a string name, has a string value (possibly `null` if not set or undefined or unsupported by a specific store implementation), and a value typecode, a numeric discriminant for the actual type of value held in the value string.

Coercing property values to strings has the advantage of making the various interfaces uniform and compact (and can even simplify implementation of the underlying stores), but has the potential for information loss during conversions to and from strings.

It is expected that clients and stores use well-defined database conventions for these conversions and use the `typecode` field as appropriate.

PL/SQL types `path_t` and `name_t` are portable aliases for strings that can represent pathnames and component names,

A typecode is a numeric value representing the true type of a string-coerced property value. Simple scalar types (numbers, dates, timestamps, etc.) can be depended on by clients and must be implemented by stores.

Since standard RDBMS typecodes are positive integers, the DBMS_DBFS_CONTENT interface allows negative integers to represent client-defined types by negative typecodes. These typecodes do not conflict with standard typecodes, are maintained persistently and returned to the client as needed, but need not be interpreted by the DBFS content API or any particular store. Portable client applications should not use user-defined typecodes as a back door way of passing information to specific stores.

See *Oracle Database PL/SQL Packages and Types Reference* for details of the DBMS_DBFS_CONTENT constants and properties and the DBMS_DBFS_CONTENT_PROPERTY_T package.

Content IDs

Content IDs are unique identifiers that represent a path in the store. See *Oracle Database PL/SQL Packages and Types Reference* for details of the DBMS_DBFS_CONTENT Content ID constants and properties

Path Name Types

Stores can store and provide access to four types of entities: type_file, type_directory, type_directory, and type_reference.

Not all stores must implement all directories, links, or references. See "Store Features" on page 8-3, and *Oracle Database PL/SQL Packages and Types Reference* for details of the DBMS_DBFS_CONTENT constants and path name types.

Store Features

In order to provide a common programmatic interface to as many different types of stores as possible, the DBFS Content API leaves some of the behavior of various operations to individual store providers to define and implement.

The DBFS Content API remains rich and conducive to portable applications by allowing different store providers (and different stores) to describe themselves as a feature set. A feature set is a bit mask indicating which features they support and which ones they do not. With this, it is possible, although tricky, for client applications to compensate for the feature deficiencies of specific stores by implementing additional logic on the client side, and deferring complex operations to stores capable of supporting them.

See *Oracle Database PL/SQL Packages and Types Reference* for details of the store features and constants.

Lock Types

Stores that support locking should implement three types of locks: lock_read_only, lock_write_only, lock_read_write.

User locks (any of these types) can be associated with user-supplied lock_data. The store does not interpret the data, but client applications can use it for their own purposes (for example, the user data could indicate the time at which the lock was placed, and the client application might use this later to control its actions.

In the simplest locking model, a lock_read_only prevents all explicit modifications to a path name (but allows implicit modifications and changes to parent/child path names). A lock_write_only prevents all explicit reads to the path name, but allows implicit reads and reads to parent/child path names. A lock_read_write allows both.

All locks are associated with a principal user who performs the locking operation; stores that support locking are expected to preserve this information and use it to perform read/write lock checking (see opt_locker).

See *Oracle Database PL/SQL Packages and Types Reference* for details of the lock types and constants.

Standard Properties

Standard properties are well-defined, mandatory properties associated with all content path names, which all stores must support, in the manner described by the DBFS Content API. Stores created against tables with a fixed schema may choose reasonable defaults for as many of these properties as needed, and so on.

All standard properties informally use the `std` namespace. Clients and stores should avoid using this namespace to define their own properties to prevent conflicts in the future.

See *Oracle Database PL/SQL Packages and Types Reference* for details of the standard properties and constants.

Optional Properties

Optional properties are well-defined but non-mandatory properties associated with all content path names that all stores are free to support (but only in the manner described by the DBFS Content API). Clients should be prepared to deal with stores that support none of the optional properties.

All optional properties informally use the `opt` namespace. Clients and stores must avoid using this namespace to define their own properties to prevent conflicts in the future.

Oracle Database PL/SQL Packages and Types Reference for details of the optional properties and constants.

User-Defined Properties

You can define your own properties for use in your application. Ensure that the namespace prefixes do not conflict with each other or with the DBFS standard or optional properties.

Property Access Flags

DBFS Content API methods to get and set properties can use combinations of property access flags to fetch properties from different namespaces in a single API call.

See *Oracle Database PL/SQL Packages and Types Reference* for details of the property access flags and constants.

Exceptions

DBFS Content API operations can raise any one of the top-level exceptions.

Clients can program against these specific exceptions in their error handlers without worrying about the specific store implementations of the underlying error signalling code.

Store service providers, should try to trap and wrap any internal exceptions into one of the exception types, as appropriate.

Oracle Database PL/SQL Packages and Types Reference for details of the Exceptions.

Property Bundles

- The `property_t` record type describes a single (value, typecode) property value tuple; the property name is implied.

- properties_t is a name-indexed hash table of property tuples. The implicit hash-table association between the index and the value allows the client to build up the full dbms_dbfs_content_property_t tuples for a properties_t.

There is an approximate correspondence between dbms_dbfs_content_property_t and property_t. The former is a SQL object type that describes the full property tuple, while the latter is a PL/SQL record type that describes only the property value component.

There is an approximate correspondence between dbms_dbfs_content_properties_t and properties_t. The former is a SQL nested table type, while the latter is a PL/SQL hash table type.

Dynamic SQL calling conventions force the use of SQL types, but PL/SQL code may be implemented more conveniently in terms of the hash-table types.

DBFS Content API provides convenient utility functions to convert between dbms_dbfs_content_properties_t and properties_t.

The function DBMS_DBFS_CONTENT.PROPERTIEST2H converts a DBMS_DBFS_CONTENT_PROPERTIES_T value to an equivalent properties_t value, and the function DBMS_DBFS_CONTENT.PROPERTIESH2T converts a properties_t value to an equivalent DBMS_DBFS_CONTENT_PROPERTIES_T value.

See *Oracle Database PL/SQL Packages and Types Reference* for details of the PROPERTY_T record type.

Store Descriptors

- A store_t is a record that describes a store registered with, and managed by the DBFS Content API (see Administrative APIs in "Administrative and Query APIs" on page 8-5).
- A mount_t is a record that describes a store mount point and its properties.

Clients can query the DBFS Content API for the list of available stores, determine which store handles accesses to a given path name, and determine the feature set for the store.

See *Oracle Database PL/SQL Packages and Types Reference* for details of the STORE_T record type.

Administrative and Query APIs

Administrative clients and content providers are expected to register content stores with the DBFS Content API. Additionally, administrative clients are expected to mount stores into the top-level namespace of their choice.

The registration and unregistration of a store is separated from the mount and unmount of a store because it is possible for the same store to be mounted multiple times at different **mount point**s (and this is under client control).

See *Oracle Database PL/SQL Packages and Types Reference* for the summary of DBMS_DBFS_CONTENT package methods.

This section covers the following topics:

- Registering a Content Store
- Unregistering a Content Store
- Mounting a Registered Store
- Unmounting a Previously Mounted Store

- Listing all Available Stores and Their Features
- Listing all Available Mount Points
- Looking Up Specific Stores and Their Features

Registering a Content Store

Procedure `REGISTERSTORE()` registers a new store backed by a provider that uses `provider_package` as the store service provider (conforming to the `DBMS_DBFS_CONTENT_SPI` package signature).

This method is designed for use by service providers after they have created a new store. Store names must be unique.

See *Oracle Database PL/SQL Packages and Types Reference* for details of the `REGISTERSTORE()` method.

Unregistering a Content Store

Procedure `UNREGISTERSTORE()` unregisters a previously registered store, invalidating all mount points associated with it. Once unregistered, all access to the store and its mount points are not guaranteed to work, although a consistent read may provide a temporary illusion of continued access.

If the `ignore_unknown` argument is `true`, attempts to unregister unknown stores do not raise an exception.

See *Oracle Database PL/SQL Packages and Types Reference* for details of the `UNREGISTERSTORE()` method.

Mounting a Registered Store

Procedure `MOUNTSTORE()` mounts a registered store and binds it to the mount point.

Once mounted, access to path names of the form `/`*`store_mount`*`/xyz` is redirected to *`store_name`* and its content provider.

Store mount points must be unique, and a syntactically valid path name component (that is, a `name_t` with no embedded `/`).

If a mount point is not specified (that is, is `null`), the DBFS Content API attempts to use the store name itself as the mount point name (subject to the uniqueness and syntactic constraints).

A special empty mount point is available for single stores, that is, a scenario where the DBFS Content API manages a single back-end store. In such cases, the client can directly deal with full path names of the form `/xyz` because there is no ambiguity in how to redirect these accesses.

The same store can be mounted multiple times, obviously at different mount points.

Mount properties can be used to specify the DBFS Content API execution environment, that is, default values of the principal, owner, `ACL`, and `asof`, for a particular mount point. Mount properties can also be used to specify a read-only store.

See *Oracle Database PL/SQL Packages and Types Reference* for details of the `MOUNTSTORE()` method.

Unmounting a Previously Mounted Store

Procedure `UNMOUNTSTORE()` unmounts a previously mounted store, either by name or by mount point. Single stores can be unmounted only by store name because they

have no mount points. Attempting to unmount a store by name unmounts all mount points associated with the store.

Once unmounted, all access to the store or mount-point is not guaranteed to work although a consistent read may provide a temporary illusion of continued access. If the `ignore_unknown` argument is `true`, attempts to unregister unknown stores or mounts does not raise an exception.

See *Oracle Database PL/SQL Packages and Types Reference* for details of the `UNMOUNTSTORE` method.

Listing all Available Stores and Their Features

Function `LISTSTORES()` lists all the available stores. The `store_mount` field of the returned records is set to `null` because mount points are separate from stores themselves

See *Oracle Database PL/SQL Packages and Types Reference* for details of the `LISTSTORES` Function.

Listing all Available Mount Points

Function `LISTMOUNTS()` lists all available mount points, their backing stores, and the store features. A single mount results in a single returned row, with its `store_mount` field set to `null`.

See *Oracle Database PL/SQL Packages and Types Reference* for details of the `LISTMOUNTS()` method.

Looking Up Specific Stores and Their Features

The `GETSTOREBY`*`XXX`*`()` and `GETFEATUREBY`*`XXX`*`()` functions look up the path name, store name, or mount point of the store.

See *Oracle Database PL/SQL Packages and Types Reference* for details of the `DBMS_DBFS_CONTENT` methods.

DBFS Content API Space Usage

Clients can query file system space usage statistics using the `SPACEUSAGE()` method. Providers are expected to support this method for their stores and to make a best effort determination of space usage, especially if the store consists of multiple tables, indexes, LOBs, and so on

- `blksize` is the natural tablespace block size that holds the store; if multiple tablespaces with different block sizes are used, any valid block size is acceptable.
- `tbytes` is the total size of the store in bytes, and `fbytes` is the free or unused size of the store in bytes. These values are computed over all segments that comprise the store.
- `nfile`, `ndir`, `nlink`, and `nref` count the number of currently available files, directories, links, and references in the store.

Database objects can grow dynamically. Therefore, it is not easy to estimate the division between free space and used space.

A space usage query on the top level root directory returns a combined summary of the space usage of all available distinct stores under it. If the same store is mounted multiple times, it is counted only once.

See *Oracle Database PL/SQL Packages and Types Reference* for details of the `SPACEUSAGE()` method.

DBFS Content API Session Defaults

Normal client access to the DBFS Content API executes with an implicit context that consists of:

- The `principal` invoking the current operation
- The `owner` for all new elements created (implicitly or explicitly) by the current operation
- The `ACL` for all new elements created (implicitly or explicitly) by the current operation
- The `ASOF` timestamp at which the underlying read-only operation (or its read-only sub-components) execute

All of this information can be passed in explicitly through arguments to the various DBFS Content API method calls, allowing the client fine-grained control over individual operations.

The DBFS Content API also allows clients to set session duration defaults for the context that are automatically inherited by all operations for which the defaults are not explicitly overridden.

All of the context defaults start out as `null` and can be cleared by setting them to `null`.

See *Oracle Database PL/SQL Packages and Types Reference* for details of the `DBMS_DBFS_CONTENT` methods.

DBFS Content API Interface Versioning

To allow for the DBFS Content API itself to evolve, an internal numeric API version increases with each change to the public API.

See *Oracle Database PL/SQL Packages and Types Reference* for details of the `GETVERSION()` method.

DBFS Content API Notes on Path Names

Clients of the DBFS Content API refer to store items through absolute path names. These path names may be full (a single string of the form `/mount_point/pathname`), or store-qualified (a tuple of the form `(store_name, pathname)`, where the path name is rooted within the store namespace).

Clients may use either naming scheme as it suits them, and can use both naming methods within their programs.

If path names are returned by DBFS Content API calls, the exact values being returned depend on the naming scheme used by the client in the call. For example, a listing or search on a fully qualified directory name returns items with their fully qualified path names, while a listing or search on a store-qualified directory name returns items whose path names are store-specific, and the store-qualification is implied.

The implementation of the DBFS Content API internally manages the normalization and inter-conversion between these two naming schemes.

DBFS Content API Creation Operations

The provider SPI must be implemented in such a way that when clients invoke the DBFS Content API, it causes the SPI to create directory, file, link, and reference elements (subject to store feature support).

All of the creation methods require a valid path name and can optionally specify properties to be associated with the path name as it is created. It is also possible for clients to fetch back item properties after the creation completes, so that automatically generated properties, such as `std_creation_time`, are immediately available to clients. The exact set of properties fetched back is controlled by the various `prop_xxx` bit masks in `prop_flags`.

Links and references require an additional path name to associate with the primary path name. File path names can optionally specify a `BLOB` value to initially populate the underlying file content, and the provided `BLOB` may be any valid LOB, either temporary or permanent. On creation, the underlying `LOB` is returned to the client if `prop_data` is specified in `prop_flags`.

Non-directory path names require that their parent directory be created first. Directory path names themselves can be recursively created. This means that the path name hierarchy leading up to a directory can be created in one call.

Attempts to create paths that already exist produce an error, except for path names that are soft-deleted. In these cases, the soft-deleted item is implicitly purged, and the new item creation is attempted.

Stores and their providers that support contentID-based access accept an explicit store name and a `NULL` path to create a new content element. The contentID generated for this element is available by means of the `OPT_CONTENT_ID` property (see *Oracle Database PL/SQL Packages and Types Reference*, `DBMS_DBFS_CONTENT` Constants - Optional Properties). The `PROP_OPT` property in the `prop_flags` parameter automatically implies contentID-based creation.

The newly created element may also have an internally generated path name if the `FEATURE_LAZY_PATH` property is not supported and this path is available by way of the `STD_CANONICAL_PATH` property (see *Oracle Database PL/SQL Packages and Types Reference* `DBMS_DBFS_CONTENT` Constants - Standard Properties).

Only file elements are candidates for contentID-based access.

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* for details of the `DBMS_DBFS_CONTENT()` methods

DBFS Content API Deletion Operations

The provider SPI must be implemented in such a way that when clients invoke the DBFS Content API, it causes the SPI to delete directory, file, link, and reference elements (subject to store feature support).

By default, the deletions are permanent, and remove successfully deleted items on transaction commit. However, repositories may also support soft-delete features. If requested by the client, soft-deleted items are retained by the store. They are not, however, typically visible in normal listings or searches. Soft-deleted items may be restored or explicitly purged.

Directory path names may be recursively deleted; the path name hierarchy below a directory may be deleted in one call. Non-recursive deletions can be performed only on empty directories. Recursive soft-deletions apply the soft-delete to all of the items being deleted.

Individual path names or all soft-deleted path names under a directory may be restored or purged using the RESTORE*XXX*() and PURGE*XXX*() methods.

Providers that support filtering can use the provider filter to identify subsets of items to delete; this makes most sense for bulk operations such as deleteDirectory(), RESTOREALL(), and PURGEALL(), but all of the deletion-related operations accept a filter argument.

Stores and their providers that support contentID-based access can also allow deleting file items by specifying their contentID.

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* for details of the DBMS_DBFS_CONTENT() methods

DBFS Content API Path Get and Put Operations

Existing path items may be queried or updated by simple GET*XXX*() and PUT*XXX*() methods.

All path names allow their metadata to be read and modified. On completion of the call, the client can request that specific properties be fetched through prop_flags.

File path names allow their data to be read and modified. On completion of the call, the client can request a new BLOB locator through the prop_data bit masks in prop_flags; these may be used to continue data access.

Files can also be read and written without using BLOB locators, by explicitly specifying logical offsets, buffer amounts, and a suitably sized buffer.

Update accesses must specify the forUpdate flag. Access to link path names may be implicitly and internally dereferenced by stores, subject to feature support, if the deref flag is specified. Oracle does not recommend this practice because symbolic links are not guaranteed to resolve.

The read method GETPATH() where forUpdate is false accepts a valid asof timestamp parameter that can be used by stores to implement flashback-style queries.

Mutating versions of the GETPATH() and the PUTPATH() methods do not support asof modes of operation.

The DBFS Content API does not have an explicit COPY() operation because a copy is easily implemented as a combination of a GETPATH() followed by a CREATE*XXX*() with appropriate data or metadata transfer across the calls. This allows copies across stores, while an internalized copy operation cannot provide this facility.

See *Oracle Database PL/SQL Packages and Types Reference* for details of the DBMS_DBFS_CONTENT methods.

DBFS Content API Rename and Move Operations

Path names can be renamed or moved, possibly across directory hierarchies and mount points, but only within the same store.

Non-directory path names previously accessible by oldPath can be renamed as a single item subsequently accessible by newPath, assuming that newPath does not exist.

If newPath exists and is not a directory, the rename implicitly deletes the existing item before renaming oldPath. If newPath exists and is a directory, oldPath is moved into the target directory.

Directory path names previously accessible by `oldPath` can be renamed by moving the directory and all of its children to `newPath` (if it does not exist) or as children of `newPath` (if it exists and is a directory).

Because the semantics of rename and move is complex with respect to non-existent or existent and non-directory or directory targets, clients may choose to implement complex rename and move operations as sequences of simpler moves or copies.

Stores and their providers that support contentID-based access and lazy path name binding also support the *Oracle Database PL/SQL Packages and Types Reference* `SETPATH` procedure that associates an existing contentID with a new "path".

See *Oracle Database PL/SQL Packages and Types Reference* for details of the `DBMS_DBFS_CONTENT.RENAMEPATH()` methods.

Directory Listings

- A `list_item_t` is a tuple of path name, component name, and type representing a single element in a directory listing.
- A `path_item_t` is a tuple describing a store, mount qualified path in a content store, with all standard and optional properties associated with it.
- A `prop_item_t` is a tuple describing a store, mount qualified path in a content store, with all user-defined properties associated with it, expanded out into individual tuples of name, value, and type.

See *Oracle Database PL/SQL Packages and Types Reference* for details of data structures.

DBFS Content API Directory Navigation and Search

Clients of the DBFS Content API can list or search the contents of directory path names, with these optional modes:

- searching recursively in sub-directories
- seeing soft-deleted items
- using flashback `asof` a provided timestamp
- filtering items in and out within the store based on list or search predicates.

The DBFS Content API currently only returns list items; clients explicitly use one of the `getPath()` methods to access the properties or content associated with an item, as appropriate.

See *Oracle Database PL/SQL Packages and Types Reference* for details of the `DBMS_DBFS_CONTENT` methods.

DBFS Content API Locking Operations

Clients of the DBFS Content API can apply user-level locks to any valid path name, subject to store feature support, associate the lock with user data, and subsequently unlock these path names. The status of locked items is available through various optional properties.

If a store supports user-defined lock checking, it is responsible for ensuring that lock and unlock operations are performed in a consistent manner.

Oracle Database PL/SQL Packages and Types Reference for details of the `DBMS_DBFS_CONTENT` methods.

DBFS Content API Access Checks

Function `CHECKACCESS()` checks if a given path name (`path`, `pathtype`, `store_name`) can be manipulated by an operation, such as the various `op_xxx` opcodes) by `principal`, as described in "DBFS Content API Locking Operations" on page 8-11

This is a convenience function for the client; a store that supports access control still internally performs these checks to guarantee security.

See *Oracle Database PL/SQL Packages and Types Reference* for details of the `DBMS_DBFS_CONTENT` methods.

DBFS Content API Abstract Operations

All of the operations in the DBFS Content API are represented as abstract `opcodes`. Clients can use these `opcodes` to directly and explicitly invoke the `CHECKACCESS()` method which verifies if a particular operation can be invoked by a given principal on a particular path name.

An `op_acl()` is an implicit operation invoked during an `op_create()` or `op_put()` call, which specifies a `std_acl` property. The operation tests to see if the principal is allowed to set or change the `ACL` of a store item.

`op_delete()` represents the soft-deletion, purge, and restore operations.

The source and destination operations of a rename or move operation are separated, although stores are free to unify these `opcodes` and to also treat a rename as a combination of delete and create.

`op_store` is a catch-all category for miscellaneous store operations that do not fall under any of the other operational APIs.

See "DBFS Content API Access Checks" on page 8-12 and *Oracle Database PL/SQL Packages and Types Reference* for details of the `DBMS_DBFS_CONTENT` Constants - Operation Codes.

DBFS Content API Path Normalization

Function `NORMALIZEPATH()` performs the following steps:

1. Verifies that the path name is absolute (starts with a `/`).
2. Collapses multiple consecutive `/`s into a single `/`.
3. Strips trailing `/`s.
4. Breaks store-specific normalized path names into two components: the parent path name and the trailing component name.
5. Breaks fully qualified normalized path names into three components: store name, parent path name, and trailing component name.

Note that the root path `/` is special: its parent path name is also `/`, and its component name is `null`. In fully qualified mode, it has a `null` store name unless a singleton mount has been created, in which case the appropriate store name is returned.

The return value is always the completely normalized store-specific or fully qualified path name.

See *Oracle Database PL/SQL Packages and Types Reference* for details of the `DBMS_DBFS_CONTENT.RENAMEPATH()` methods.

DBFS Content API Statistics Support

DBFS Content API statistics are expensive to collect and maintain persistently. DBFS has support for buffering statistics in memory for a maximum of `flush_time` centiseconds or a maximum of `flush_count` operations, whichever limit is reached first), at which time the buffers are implicitly flushed to disk.

Clients can also explicitly invoke a flush using `flushStats`. An implicit flush also occurs when statistics collection is disabled.

`setStats` is used to enable and disable statistics collection; the client can optionally control the flush settings by specifying non-`null` values for the time and count parameters.

Oracle Database PL/SQL Packages and Types Reference for details of the `DBMS_DBFS_CONTENT` methods.

DBFS Content API Tracing Support

DBFS Content API tracing is a generic tracing facility that may be used by any DBFS Content API user (both clients and providers). The DBFS Content API dispatcher itself uses the tracing facility.

Trace information is written to the foreground trace file, with varying levels of detail as specified by the trace level arguments. The global trace level consists of two components: `severity` and `detail`. These can be thought of as additive bit masks.

The `severity` component allows the separation of top-level as compared to low-level tracing of different components, and allows the amount of tracing to be increased as needed. There are no semantics associated with different levels, and users are free to set the trace level at any severity they choose, although a good rule of thumb would be to use severity 1 for top-level API entry and exit traces, severity 2 for internal operations, and severity 3 or greater for very low-level traces.

The `detail` component controls how much additional information the trace reports with each trace record: timestamps, short-stack, and so on. Example 8–1 demonstrates how to enable tracing using the DBFS Content APIs.

Example 8–1 DBFS Content Tracing

```
function    getTrace
        return  integer;
    procedure   setTrace(
        trclvl      in              integer);
    function    traceEnabled(
        sev         in              integer)
        return  integer;
    procedure   trace(
        sev         in              integer,
        msg0        in              varchar2,
        msg1        in              varchar     default '',
        msg2        in              varchar     default '',
        msg3        in              varchar     default '',
        msg4        in              varchar     default '',
        msg5        in              varchar     default '',
        msg6        in              varchar     default '',
        msg7        in              varchar     default '',
        msg8        in              varchar     default '',
        msg9        in              varchar     default '',
        msg10       in              varchar     default '');
```

See *Oracle Database PL/SQL Packages and Types Reference* for details of the `DBMS_DBFS_CONTENT` methods.

Resource and Property Views

The following views describe the structure and properties of Content API.

- The `DBFS_CONTENT` views; see *Oracle Database Reference*
- The `DBFS_CONTENT_PROPERTIES` views; see *Oracle Database Reference*

9

Creating Your Own DBFS Store

This chapter describes how to create your own DBFS Store.

This chapter contains these topics:

- Overview of DBFS Store Creation and Use
- DBFS Content Store Provider Interface (DBFS Content SPI)
- Creating a Custom Provider

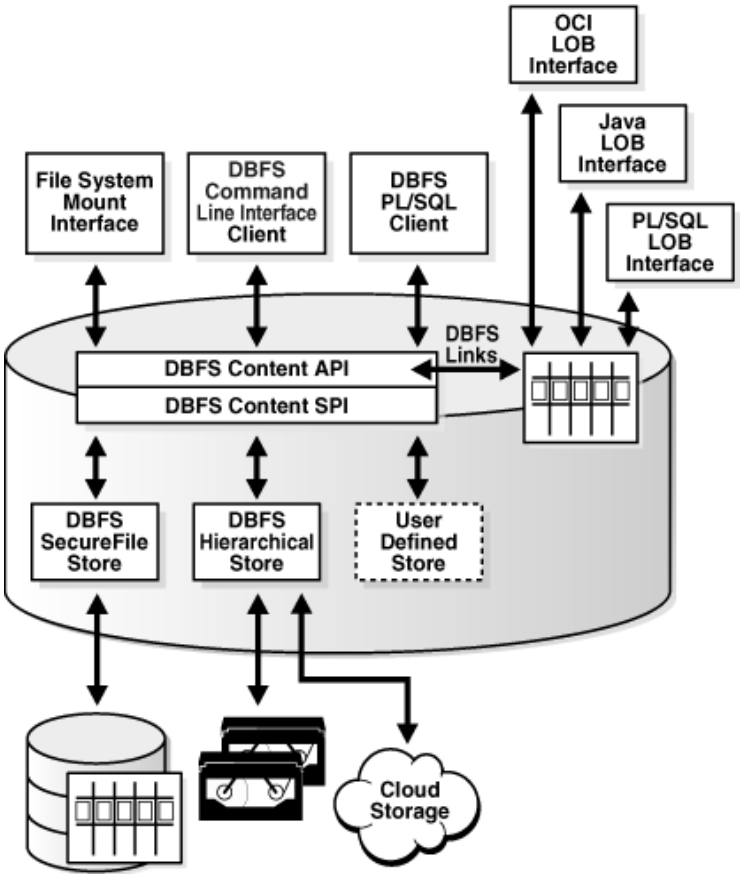
Overview of DBFS Store Creation and Use

In order to customize a DBFS store, you must implement the DBFS Content SPI (`DBMS_DBFS_CONTENT_SPI`). This is the basis for existing stores such as the DBFS SecureFiles Store and the DFFS Hierarchical Store, as well as any user-defined DBFS stores that you create.

Client-side applications, such the PL/SQL interface, invoke functions and procedures in the DBFS Content API. The DBFS Content API then invokes corresponding subprograms in the DBFS Content SPI to create stores and perform other related functions. See Chapter 8, "DBFS Content API".

Once you create your DBFS store, you run it much the same way that you would a SecureFiles Store, as described in Chapter 6, "DBFS SecureFiles Store".

Figure 9–1 Database File System (DBFS)



DBFS Content Store Provider Interface (DBFS Content SPI)

The DBFS Content SPI (Store Provider Interface) is a specification only and has no package body. The package body must be implemented in order to respond to calls from the DBFS Content API. In other words, DBFS Content SPI is a collection of required program specifications which you must implement using the method signatures and semantics indicated. You may add additional functions and procedures to suit your needs. Your implementation may implement other methods and expose other interfaces, but the DBFS Content API will not use these interfaces.

The DBFS Content SPI references various elements such as constants, types, and exceptions defined by the DBFS Content API (package `DBMS_DBFS_CONTENT`).

The main distinction in the method-naming conventions is that all path name references are always store-qualified, that is, the notion of mount points and full absolute path names have been normalized and converted to store-qualified path names by the DBFS Content API before it invokes any of the Provider SPI methods.

Because the DBFS Content API and Provider SPI is a one-to-many pluggable architecture, the DBFS Content API uses dynamic SQL to invoke methods in the Provider SPI; this may lead to run time errors if your Provider SPI implementation does not follow the Provider SPI specification in this document.

There are no explicit initial or final methods to indicate when the DBFS Content API plugs and unplugs a particular Provider SPI. Provider SPIs must be able to auto-initialize themselves at any SPI entry point.

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* for syntax of the `DBMS_DBFS_CONTENT_SPI` package
- See the file `$ORACLE_HOME/rdbms/admin/dbmscapi.sql` for more information

Creating a Custom Provider

This section describes an example store provider for DBFS that exposes a relational table containing a `BLOB` column as a flat, non-hierarchical filesystem, that is, a collection of named files.

The rest of this section assumes that you have installed the Oracle Database 12*c* and are familiar with DBFS concepts, and have installed and used `dbfs_client` and `FUSE` to mount and access filesystems backed by the standard SFS store provider.

The TaBleFileSystem Store Provider ("tbfs") does not aim to be feature-rich or even complete---it does however provide a sufficient demonstration of what it takes for users of DBFS to write their own custom providers that expose their table(s) through `dbfs_client` to traditional filesystem programs.

The TBFS can be used as a skeleton for such custom providers, or can be used as a learning tool for users to become familiar with the DBFS and its SPI.

This section contains:

Mechanics

This section contains these topics:

Installation and Setup

The TBFS consists of the following SQL files:

`tbfs.sql` top-level driver script

`tbl.sql` script to create a test user, tablespace, the table backing the filesystem, and so on.

`spec.sql` the SPI specification of the tbfs

`body.sql` the SPI implementation of the tbfs

`capi.sql` DBFS register/mount script

To install the TBFS, just run `tbfs.sql` as `SYSDBA`, in the directory that contains all of the above files. `tbfs.sql` will load the other SQL files in the proper sequence.

Ignoring any name conflicts, all of the SQL files should load without any compilation errors. All SQL files should also load without any run time errors, depending on the value of the "plsql_warnings" init.ora parameter, you may see various innocuous warnings.

If there are any name conflicts (tablespace name TBFS, datafile name"tbfs.f", user name TBFS, package name TBFS), the appropriate references in the various SQL files must be changed consistently.

TBFS Use

Once the TBFS has been installed, `dbfs_client` connected as user TBFS will see a simple, non-hierarchical, filesystem backed by an RDBMS table (TBFS.TBFST).

Files can be added or removed from this filesystem through SQL (that is, through DML on the underlying table), through Unix utilities (mediated by `dbfs_client`), or through PL/SQL (using the DBFS APIs).

Changes to the filesystem made through any of the access methods will be visible, in a transactionally consistent manner (that is, at commit/rollback boundaries) to all of the other access methods.

TBFS Internals

The TBFS is necessarily simple since its primary purpose is to serve as a teaching and learning example. However, the implementation shows the path towards a robust, production-quality custom SPI that can plug into the DBFS, and expose existing relational data as Unix filesystems.

The TBFS makes various simplifications in order to remain concise (however, these should not be taken as inviolable limitations of DBFS or the SPI):

- The TBFS SPI package handles only a single table with a hard-coded name (TBFS.TBFST). It is possible to use dynamic SQL and additional configuration information to have a single SPI package support multiple tables, each as a separate filesystem (or even to unify data in multiple tables into a single filesystem).
- The TBFS does not support filesystem hierarchies; it imposes a flat namespace: a collection of files, identified by a simple item name, under a virtual "/" root directory. Implementing directory hierarchies is significantly more complex because it requires the store provider to manage parent/child relationships in a consistent manner.

  Moreover, existing relational data (the kind of data that TBFS is attempting to expose as a filesystem) does not typically have inter-row relationships that form a natural directory/file hierarchy.
- Because the TBFS supports only a flat namespace, most methods in the SPI are unimplemented, and the method bodies raise a `dbms_dbfs_content.unsupported_operation` exception. This exception is also a good starting point for you to write your own custom SPI. You can start with a simple SPI skeleton cloned from the `DBMS_DBFS_CONTENT_SPI` package, default all method bodies to ones that raise this exception, and subsequently fill in more realistic implementations incrementally.

- The table underlying the TBFS is close to being the simplest possible structure (a key/name column and a LOB column). This means that various properties used or expected by DBFS and `dbfs_client` must be generated dynamically (the TBFS implementation shows how this is done for the `std:guid` property).

  Other properties (such as Unix-style timestamps) are not implemented at all. This still allows a surprisingly functional filesystem to be implemented, but when you write your own custom SPIs, you can easily incorporate support for additional DBFS properties by expanding the structure of their underlying table(s) to include additional columns as needed, or by using existing columns in their existing tables to provide the values for these DBFS properties.
- The TBFS does not implement a rename/move method; adding support for this (a suitable `UPDATE` statement in the `renamePath` method) is left as an exercise for the user.
- The TBFS example uses the string "tbfs" in multiple places (tablespace, datafile, user, package, and even filesystem name). All these uses of "tbfs" belong in different namespaces—identifying which namespace corresponds to a specific occurrence of the string. "tbfs" in these examples is also a good learning exercise to make sure that the DBFS concepts are clear in your mind.

TBFS.SQL

The TBFS.SQL script is the top level driver script.

```
set echo on;

@tbl
@spec
@body
@capi

quit;
```

TBL.SQL

The TBL.SQL script creates a test user, a tablespace, the table that backs the filesystem and so on.

```
connect / as sysdba

create tablespace tbfs datafile 'tbfs.f' size 100m
    reuse autoextend on
    extent management local
    segment space management auto;

create user tbfs identified by tbfs;
alter user tbfs default tablespace tbfs;
grant connect, resource, dbfs_role to tbfs;


connect tbfs/tbfs;

drop table tbfst;
purge recyclebin;

create table tbfst(
    key     varchar2(256)
            primary key
```

```
            check           (instr(key, '/') = 0),
    data    blob)
        tablespace tbfs
    lob(data)
        store as securefile
            (tablespace tbfs);

grant select on tbfst to dbfs_role;
grant insert on tbfst to dbfs_role;
grant delete on tbfst to dbfs_role;
grant update on tbfst to dbfs_role;
```

spec.sql

The `spec.sql` script provide the SPI specification of the tbfs.

```
connect / as sysdba;

create or replace package tbfs
    authid current_user
as



    /*
     * Lookup store features (see dbms_dbfs_content.feature_XXX). Lookup
     * store id.
     *
     * A store ID identifies a provider-specific store, across
     * registrations and mounts, but independent of changes to the store
     * contents.
     *
     * I.e. changes to the store table(s) should be reflected in the
     * store ID, but re-initialization of the same store table(s) should
     * preserve the store ID.
     *
     * Providers should also return a "version" (either specific to a
     * provider package, or to an individual store) based on a standard
     * <a.b.c> naming convention (for <major>, <minor>, and <patch>
     * components).
     *
     */

    function    getFeatures(
        store_name          in      varchar2)
            return  integer;

    function    getStoreId(
        store_name          in      varchar2)
            return  number;

    function    getVersion(
        store_name          in      varchar2)
            return  varchar2;



    /*
     * Lookup pathnames by (store_name, std_guid) or (store_mount,
     * std_guid) tuples.
```

```
 *
 * If the underlying "std_guid" is found in the underlying store,
 * this function returns the store-qualified pathname.
 *
 * If the "std_guid" is unknown, a "null" value is returned. Clients
 * are expected to handle this as appropriate.
 *
 */

function    getPathByStoreId(
    store_name          in      varchar2,
    guid                in      integer)
        return  varchar2;



/*
 * DBFS SPI: space usage.
 *
 * Clients can query filesystem space usage statistics via the
 * "spaceUsage()" method. Providers are expected to support this
 * method for their stores (and to make a best effort determination
 * of space usage---esp. if the store consists of multiple
 * tables/indexes/lobs, etc.).
 *
 * "blksize" is the natural tablespace blocksize that holds the
 * store---if multiple tablespaces with different blocksizes are
 * used, any valid blocksize is acceptable.
 *
 * "tbytes" is the total size of the store in bytes, and "fbytes" is
 * the free/unused size of the store in bytes. These values are
 * computed over all segments that comprise the store.
 *
 * "nfile", "ndir", "nlink", and "nref" count the number of
 * currently available files, directories, links, and references in
 * the store.
 *
 * Since database objects are dynamically growable, it is not easy
 * to estimate the division between "free" space and "used" space.
 *
 */

procedure   spaceUsage(
    store_name  in              varchar2,
    blksize     out             integer,
    tbytes      out             integer,
    fbytes      out             integer,
    nfile       out             integer,
    ndir        out             integer,
    nlink       out             integer,
    nref        out             integer);



/*
 * DBFS SPI: notes on pathnames.
 *
 * All pathnames used in the SPI are store-qualified, i.e. a 2-tuple
 * of the form (store_name, pathname) (where the pathname is rooted
 * within the store namespace).
```

```
 *
 *
 * Stores/providers that support contentID-based access (see
 * "feature_content_id") also support a form of addressing that is
 * not based on pathnames. Items are identified by an explicit store
 * name, a "null" pathname, and possibly a contentID specified as a
 * parameter or via the "opt_content_id" property.
 *
 * Not all operations are supported with contentID-based access, and
 * applications should depend only on the simplest create/delete
 * functionality being available.
 *
 */



/*
 * DBFS SPI: creation operations
 *
 * The SPI must allow the DBFS API to create directory, file, link,
 * and reference elements (subject to store feature support).
 *
 *
 * All of the creation methods require a valid pathname (see the
 * special exemption for contentID-based access below), and can
 * optionally specify properties to be associated with the pathname
 * as it is created. It is also possible for clients to fetch-back
 * item properties after the creation completes (so that
 * automatically generated properties (e.g. "std_creation_time") are
 * immediately available to clients (the exact set of properties
 * fetched back is controlled by the various "prop_xxx" bitmasks in
 * "prop_flags").
 *
 *
 * Links and references require an additional pathname to associate
 * with the primary pathname.
 *
 * File pathnames can optionally specify a BLOB value to use to
 * initially populate the underlying file content (the provided BLOB
 * may be any valid lob: temporary or permanent). On creation, the
 * underlying lob is returned to the client (if "prop_data" is
 * specified in "prop_flags").
 *
 * Non-directory pathnames require that their parent directory be
 * created first. Directory pathnames themselves can be recursively
 * created (i.e. the pathname hierarchy leading up to a directory
 * can be created in one call).
 *
 *
 * Attempts to create paths that already exist is an error; the one
 * exception is pathnames that are "soft-deleted" (see below for
 * delete operations)---in these cases, the soft-deleted item is
 * implicitly purged, and the new item creation is attempted.
 *
 *
 * Stores/providers that support contentID-based access accept an
 * explicit store name and a "null" path to create a new element.
 * The contentID generated for this element is available via the
 * "opt_content_id" property (contentID-based creation automatically
 * implies "prop_opt" in "prop_flags").
```

```
 *
 * The newly created element may also have an internally generated
 * pathname (if "feature_lazy_path" is not supported) and this path
 * is available via the "std_canonical_path" property.
 *
 * Only file elements are candidates for contentID-based access.
 *
 */

procedure   createFile(
    store_name  in              varchar2,
    path        in              varchar2,
    properties  in out nocopy   dbms_dbfs_content_properties_t,
    content     in out nocopy   blob,
    prop_flags  in              integer,
    ctx         in              dbms_dbfs_content_context_t);

procedure   createLink(
    store_name  in              varchar2,
    srcPath     in              varchar2,
    dstPath     in              varchar2,
    properties  in out nocopy   dbms_dbfs_content_properties_t,
    prop_flags  in              integer,
    ctx         in              dbms_dbfs_content_context_t);

procedure   createReference(
    store_name  in              varchar2,
    srcPath     in              varchar2,
    dstPath     in              varchar2,
    properties  in out nocopy   dbms_dbfs_content_properties_t,
    prop_flags  in              integer,
    ctx         in              dbms_dbfs_content_context_t);

procedure   createDirectory(
    store_name  in              varchar2,
    path        in              varchar2,
    properties  in out nocopy   dbms_dbfs_content_properties_t,
    prop_flags  in              integer,
    recurse     in              integer,
    ctx         in              dbms_dbfs_content_context_t);



/*
 * DBFS SPI: deletion operations
 *
 * The SPI must allow the DBFS API to delete directory, file, link,
 * and reference elements (subject to store feature support).
 *
 *
 * By default, the deletions are "permanent" (get rid of the
 * successfully deleted items on transaction commit), but stores may
 * also support "soft-delete" features. If requested by the client,
 * soft-deleted items are retained by the store (but not typically
 * visible in normal listings or searches).
 *
 * Soft-deleted items can be "restore"d, or explicitly purged.
 *
 *
 * Directory pathnames can be recursively deleted (i.e. the pathname
```

```
 * hierarchy below a directory can be deleted in one call).
 * Non-recursive deletions can be performed only on empty
 * directories. Recursive soft-deletions apply the soft-delete to
 * all of the items being deleted.
 *
 *
 * Individual pathnames (or all soft-deleted pathnames under a
 * directory) can be restored or purged via the restore and purge
 * methods.
 *
 *
 * Providers that support filtering can use the provider "filter" to
 * identify subsets of items to delete---this makes most sense for
 * bulk operations (deleteDirectory, restoreAll, purgeAll), but all
 * of the deletion-related operations accept a "filter" argument.
 *
 *
 * Stores/providers that support contentID-based access can also
 * allow file items to be deleted by specifying their contentID.
 *
 */

procedure   deleteFile(
    store_name  in              varchar2,
    path        in              varchar2,
    filter      in              varchar2,
    soft_delete in              integer,
    ctx         in              dbms_dbfs_content_context_t);

procedure   deleteContent(
    store_name  in              varchar2,
    contentID   in              raw,
    filter      in              varchar2,
    soft_delete in              integer,
    ctx         in              dbms_dbfs_content_context_t);

procedure   deleteDirectory(
    store_name  in              varchar2,
    path        in              varchar2,
    filter      in              varchar2,
    soft_delete in              integer,
    recurse     in              integer,
    ctx         in              dbms_dbfs_content_context_t);

procedure   restorePath(
    store_name  in              varchar2,
    path        in              varchar2,
    filter      in              varchar2,
    ctx         in              dbms_dbfs_content_context_t);

procedure   purgePath(
    store_name  in              varchar2,
    path        in              varchar2,
    filter      in              varchar2,
    ctx         in              dbms_dbfs_content_context_t);

procedure   restoreAll(
    store_name  in              varchar2,
    path        in              varchar2,
    filter      in              varchar2,
```

```
    ctx         in              dbms_dbfs_content_context_t);

procedure   purgeAll(
    store_name  in              varchar2,
    path        in              varchar2,
    filter      in              varchar2,
    ctx         in              dbms_dbfs_content_context_t);



/*
 * DBFS SPI: path get/put operations.
 *
 * Existing path items can be accessed (for query or for update) and
 * modified via simple get/put methods.
 *
 * All pathnames allow their metadata (i.e. properties) to be
 * read/modified. On completion of the call, the client can request
 * (via "prop_flags") specific properties to be fetched as well.
 *
 * File pathnames allow their data (i.e. content) to be
 * read/modified. On completion of the call, the client can request
 * (via the "prop_data" bitmaks in "prop_flags") a new BLOB locator
 * that can be used to continue data access.
 *
 * Files can also be read/written without using BLOB locators, by
 * explicitly specifying logical offsets/buffer-amounts and a
 * suitably sized buffer.
 *
 *
 * Update accesses must specify the "forUpdate" flag. Access to link
 * pathnames can be implicitly and internally deferenced by stores
 * (subject to feature support) if the "deref" flag is
 * specified---however, this is dangerous since symbolic links are
 * not always resolvable.
 *
 *
 * The read methods (i.e. "getPath" where "forUpdate" is "false"
 * also accepts a valid "asof" timestamp parameter that can be used
 * by stores to implement "as of" style flashback queries. Mutating
 * versions of the "getPath" and the "putPath" methods do not
 * support as-of modes of operation.
 *
 *
 * "getPathNowait" implies a "forUpdate", and, if implemented (see
 * "feature_nowait"), allows providers to return an exception
 * (ORA-54) rather than wait for row locks.
 *
 */

procedure   getPath(
    store_name  in              varchar2,
    path        in              varchar2,
    properties  in out nocopy   dbms_dbfs_content_properties_t,
    content     out    nocopy   blob,
    item_type   out             integer,
    prop_flags  in              integer,
    forUpdate   in              integer,
    deref       in              integer,
    ctx         in              dbms_dbfs_content_context_t);
```

```
procedure   getPathNowait(
    store_name  in              varchar2,
    path        in              varchar2,
    properties  in out nocopy   dbms_dbfs_content_properties_t,
    content     out    nocopy   blob,
    item_type   out             integer,
    prop_flags  in              integer,
    deref       in              integer,
    ctx         in              dbms_dbfs_content_context_t);

procedure   getPath(
    store_name  in              varchar2,
    path        in              varchar2,
    properties  in out nocopy   dbms_dbfs_content_properties_t,
    amount      in out          number,
    offset      in              number,
    buffer      out    nocopy   raw,
    prop_flags  in              integer,
    ctx         in              dbms_dbfs_content_context_t);

procedure   getPath(
    store_name  in              varchar2,
    path        in              varchar2,
    properties  in out nocopy   dbms_dbfs_content_properties_t,
    amount      in out          number,
    offset      in              number,
    buffers     out    nocopy   dbms_dbfs_content_raw_t,
    prop_flags  in              integer,
    ctx         in              dbms_dbfs_content_context_t);

procedure   putPath(
    store_name  in              varchar2,
    path        in              varchar2,
    properties  in out nocopy   dbms_dbfs_content_properties_t,
    content     in out nocopy   blob,
    item_type   out             integer,
    prop_flags  in              integer,
    ctx         in              dbms_dbfs_content_context_t);

procedure   putPath(
    store_name  in              varchar2,
    path        in              varchar2,
    properties  in out nocopy   dbms_dbfs_content_properties_t,
    amount      in              number,
    offset      in              number,
    buffer      in              raw,
    prop_flags  in              integer,
    ctx         in              dbms_dbfs_content_context_t);

procedure   putPath(
    store_name  in              varchar2,
    path        in              varchar2,
    properties  in out nocopy   dbms_dbfs_content_properties_t,
    written     out             number,
    offset      in              number,
    buffers     in              dbms_dbfs_content_raw_t,
    prop_flags  in              integer,
    ctx         in              dbms_dbfs_content_context_t);
```

```
/*
 * DBFS SPI: rename/move operations.
 *
 * Pathnames can be renamed or moved, possibly across directory
 * hierarchies and mount-points, but within the same store.
 *
 *
 * Non-directory pathnames previously accessible via "oldPath" are
 * renamed as a single item subsequently accessible via "newPath";
 * assuming that "newPath" does not already exist.
 *
 * If "newPath" exists and is not a directory, the rename implicitly
 * deletes the existing item before renaming "oldPath". If "newPath"
 * exists and is a directory, "oldPath" is moved into the target
 * directory.
 *
 *
 * Directory pathnames previously accessible via "oldPath" are
 * renamed by moving the directory and all of its children to
 * "newPath" (if it does not already exist) or as children of
 * "newPath" (if it exists and is a directory).
 *
 *
 * Stores/providers that support contentID-based access and lazy
 * pathname binding also support the "setPath" method that
 * associates an existing "contentID" with a new "path".
 *
 */

procedure   renamePath(
    store_name  in              varchar2,
    oldPath     in              varchar2,
    newPath     in              varchar2,
    properties  in out nocopy   dbms_dbfs_content_properties_t,
    ctx         in              dbms_dbfs_content_context_t);

procedure   setPath(
    store_name  in              varchar2,
    contentID   in              raw,
    path        in              varchar2,
    properties  in out nocopy   dbms_dbfs_content_properties_t,
    ctx         in              dbms_dbfs_content_context_t);



/*
 * DBFS SPI: directory navigation and search.
 *
 * The DBFS API can list or search the contents of directory
 * pathnames, optionally recursing into sub-directories, optionally
 * seeing soft-deleted items, optionally using flashback "as of" a
 * provided timestamp, and optionally filtering items in/out within
 * the store based on list/search predicates.
 *
 */

function    list(
    store_name  in              varchar2,
```

```
    path        in              varchar2,
    filter      in              varchar2,
    recurse     in              integer,
    ctx         in              dbms_dbfs_content_context_t)
        return  dbms_dbfs_content_list_items_t
            pipelined;

function    search(
    store_name  in              varchar2,
    path        in              varchar2,
    filter      in              varchar2,
    recurse     in              integer,
    ctx         in              dbms_dbfs_content_context_t)
        return  dbms_dbfs_content_list_items_t
            pipelined;



/*
 * DBFS SPI: locking operations.
 *
 * Clients of the DBFS API can apply user-level locks to any valid
 * pathname (subject to store feature support), associate the lock
 * with user-data, and subsequently unlock these pathnames.
 *
 * The status of locked items is available via various optional
 * properties (see "opt_lock*" above).
 *
 *
 * It is the responsibility of the store (assuming it supports
 * user-defined lock checking) to ensure that lock/unlock operations
 * are performed in a consistent manner.
 *
 */

procedure   lockPath(
    store_name  in              varchar2,
    path        in              varchar2,
    lock_type   in              integer,
    lock_data   in              varchar2,
    ctx         in              dbms_dbfs_content_context_t);

procedure   unlockPath(
    store_name  in              varchar2,
    path        in              varchar2,
    ctx         in              dbms_dbfs_content_context_t);



/*
 * DBFS SPI: access checks.
 *
 * Check if a given pathname (store_name, path, pathtype) can be
 * manipulated by "operation (see the various
 * "dbms_dbfs_content.op_xxx" opcodes) by "principal".
 *
 * This is a convenience function for the DBFS API; a store that
 * supports access control still internally performs these checks to
 * guarantee security.
 *
```

```
     */

    function    checkAccess(
        store_name  in              varchar2,
        path        in              varchar2,
        pathtype    in              integer,
        operation   in              varchar2,
        principal   in              varchar2)
            return  integer;
end;
/
show errors;

create or replace public synonym tbfs
    for sys.tbfs;

grant execute on tbfs
    to dbfs_role;
```

body.sql

The `body.sql` script provides the SPI implementation of the tbfs.

```
connect / as sysdba;

create or replace package body tbfs
as



    /*
     * Lookup store features (see dbms_dbfs_content.feature_XXX). Lookup
     * store id.
     *
     * A store ID identifies a provider-specific store, across
     * registrations and mounts, but independent of changes to the store
     * contents.
     *
     * I.e. changes to the store table(s) should be reflected in the
     * store ID, but re-initialization of the same store table(s) should
     * preserve the store ID.
     *
     * Providers should also return a "version" (either specific to a
     * provider package, or to an individual store) based on a standard
     * <a.b.c> naming convention (for <major>, <minor>, and <patch>
     * components).
     *
     */

    function    getFeatures(
        store_name          in      varchar2)
            return  integer
    is
    begin
        return dbms_dbfs_content.feature_locator;
    end;

    function    getStoreId(
        store_name          in      varchar2)
            return  number
```

```
is
begin
    return 1;
end;

function    getVersion(
    store_name          in      varchar2)
        return  varchar2
is
begin
    return '1.0.0';
end;



/*
 * Lookup pathnames by (store_name, std_guid) or (store_mount,
 * std_guid) tuples.
 *
 * If the underlying "std_guid" is found in the underlying store,
 * this function returns the store-qualified pathname.
 *
 * If the "std_guid" is unknown, a "null" value is returned. Clients
 * are expected to handle this as appropriate.
 *
 */

function    getPathByStoreId(
    store_name          in      varchar2,
    guid                in      integer)
        return  varchar2
is
begin
    raise dbms_dbfs_content.unsupported_operation;
end;



/*
 * DBFS SPI: space usage.
 *
 * Clients can query filesystem space usage statistics via the
 * "spaceUsage()" method. Providers are expected to support this
 * method for their stores (and to make a best effort determination
 * of space usage---esp. if the store consists of multiple
 * tables/indexes/lobs, etc.).
 *
 * "blksize" is the natural tablespace blocksize that holds the
 * store---if multiple tablespaces with different blocksizes are
 * used, any valid blocksize is acceptable.
 *
 * "tbytes" is the total size of the store in bytes, and "fbytes" is
 * the free/unused size of the store in bytes. These values are
 * computed over all segments that comprise the store.
 *
 * "nfile", "ndir", "nlink", and "nref" count the number of
 * currently available files, directories, links, and references in
 * the store.
 *
 * Since database objects are dynamically growable, it is not easy
```

```
 * to estimate the division between "free" space and "used" space.
 *
 */

procedure   spaceUsage(
    store_name  in              varchar2,
    blksize     out             integer,
    tbytes      out             integer,
    fbytes      out             integer,
    nfile       out             integer,
    ndir        out             integer,
    nlink       out             integer,
    nref        out             integer)
is
    nblks       number;
begin
    select count(*) into nfile
        from tbfs.tbfst;
    ndir  := 0;
    nlink := 0;
    nref  := 0;

    select sum(bytes) into tbytes
        from user_segments;
    select sum(blocks) into nblks
        from user_segments;
    blksize := tbytes/nblks;
    fbytes  := 0;                                   /* change as needed */
end;



/*
 * DBFS SPI: notes on pathnames.
 *
 * All pathnames used in the SPI are store-qualified, i.e. a 2-tuple
 * of the form (store_name, pathname) (where the pathname is rooted
 * within the store namespace).
 *
 *
 * Stores/providers that support contentID-based access (see
 * "feature_content_id") also support a form of addressing that is
 * not based on pathnames. Items are identified by an explicit store
 * name, a "null" pathname, and possibly a contentID specified as a
 * parameter or via the "opt_content_id" property.
 *
 * Not all operations are supported with contentID-based access, and
 * applications should depend only on the simplest create/delete
 * functionality being available.
 *
 */



/*
 * DBFS SPI: creation operations
 *
 * The SPI must allow the DBFS API to create directory, file, link,
 * and reference elements (subject to store feature support).
 *
```

```
 *
 * All of the creation methods require a valid pathname (see the
 * special exemption for contentID-based access below), and can
 * optionally specify properties to be associated with the pathname
 * as it is created. It is also possible for clients to fetch-back
 * item properties after the creation completes (so that
 * automatically generated properties (e.g. "std_creation_time") are
 * immediately available to clients (the exact set of properties
 * fetched back is controlled by the various "prop_xxx" bitmasks in
 * "prop_flags").
 *
 *
 * Links and references require an additional pathname to associate
 * with the primary pathname.
 *
 * File pathnames can optionally specify a BLOB value to use to
 * initially populate the underlying file content (the provided BLOB
 * may be any valid lob: temporary or permanent). On creation, the
 * underlying lob is returned to the client (if "prop_data" is
 * specified in "prop_flags").
 *
 * Non-directory pathnames require that their parent directory be
 * created first. Directory pathnames themselves can be recursively
 * created (i.e. the pathname hierarchy leading up to a directory
 * can be created in one call).
 *
 *
 * Attempts to create paths that already exist is an error; the one
 * exception is pathnames that are "soft-deleted" (see below for
 * delete operations)---in these cases, the soft-deleted item is
 * implicitly purged, and the new item creation is attempted.
 *
 *
 * Stores/providers that support contentID-based access accept an
 * explicit store name and a "null" path to create a new element.
 * The contentID generated for this element is available via the
 * "opt_content_id" property (contentID-based creation automatically
 * implies "prop_opt" in "prop_flags").
 *
 * The newly created element may also have an internally generated
 * pathname (if "feature_lazy_path" is not supported) and this path
 * is available via the "std_canonical_path" property.
 *
 * Only file elements are candidates for contentID-based access.
 *
 */

procedure   createFile(
    store_name  in              varchar2,
    path        in              varchar2,
    properties  in out nocopy   dbms_dbfs_content_properties_t,
    content     in out nocopy   blob,
    prop_flags  in              integer,
    ctx         in              dbms_dbfs_content_context_t)
is
    guid        number;
begin
    if (path = '/') then
        raise dbms_dbfs_content.invalid_path;
    end if;
```

```
    if content is null then
        content := empty_blob();
    end if;

    begin
        insert into tbfs.tbfst values (substr(path,2), content)
            returning data into content;
    exception
        when dup_val_on_index then
            raise dbms_dbfs_content.path_exists;
    end;

    select ora_hash(path) into guid from dual;

    properties := dbms_dbfs_content_properties_t(
        dbms_dbfs_content_property_t(
            'std:length',
            to_char(dbms_lob.getlength(content)),
            dbms_types.TYPECODE_NUMBER),
        dbms_dbfs_content_property_t(
            'std:guid',
            to_char(guid),
            dbms_types.TYPECODE_NUMBER));
end;

procedure   createLink(
    store_name  in              varchar2,
    srcPath     in              varchar2,
    dstPath     in              varchar2,
    properties  in out nocopy   dbms_dbfs_content_properties_t,
    prop_flags  in              integer,
    ctx         in              dbms_dbfs_content_context_t)
is
begin
    raise dbms_dbfs_content.unsupported_operation;
end;

procedure   createReference(
    store_name  in              varchar2,
    srcPath     in              varchar2,
    dstPath     in              varchar2,
    properties  in out nocopy   dbms_dbfs_content_properties_t,
    prop_flags  in              integer,
    ctx         in              dbms_dbfs_content_context_t)
is
begin
    raise dbms_dbfs_content.unsupported_operation;
end;

procedure   createDirectory(
    store_name  in              varchar2,
    path        in              varchar2,
    properties  in out nocopy   dbms_dbfs_content_properties_t,
    prop_flags  in              integer,
    recurse     in              integer,
    ctx         in              dbms_dbfs_content_context_t)
is
begin
    raise dbms_dbfs_content.unsupported_operation;
```

```
end;



/*
 * DBFS SPI: deletion operations
 *
 * The SPI must allow the DBFS API to delete directory, file, link,
 * and reference elements (subject to store feature support).
 *
 *
 * By default, the deletions are "permanent" (get rid of the
 * successfully deleted items on transaction commit), but stores may
 * also support "soft-delete" features. If requested by the client,
 * soft-deleted items are retained by the store (but not typically
 * visible in normal listings or searches).
 *
 * Soft-deleted items can be "restore"d, or explicitly purged.
 *
 *
 * Directory pathnames can be recursively deleted (i.e. the pathname
 * hierarchy below a directory can be deleted in one call).
 * Non-recursive deletions can be performed only on empty
 * directories. Recursive soft-deletions apply the soft-delete to
 * all of the items being deleted.
 *
 *
 * Individual pathnames (or all soft-deleted pathnames under a
 * directory) can be restored or purged via the restore and purge
 * methods.
 *
 *
 * Providers that support filtering can use the provider "filter" to
 * identify subsets of items to delete---this makes most sense for
 * bulk operations (deleteDirectory, restoreAll, purgeAll), but all
 * of the deletion-related operations accept a "filter" argument.
 *
 *
 * Stores/providers that support contentID-based access can also
 * allow file items to be deleted by specifying their contentID.
 *
 */

procedure   deleteFile(
    store_name  in              varchar2,
    path        in              varchar2,
    filter      in              varchar2,
    soft_delete in              integer,
    ctx         in              dbms_dbfs_content_context_t)
is
begin
    if (path = '/') then
        raise dbms_dbfs_content.invalid_path;
    end if;

    if ((soft_delete <> 0)      or
        (filter is not null))   then
        raise dbms_dbfs_content.unsupported_operation;
    end if;
```

```
    delete from tbfs.tbfst t
        where ('/' || t.key) = path;

    if sql%rowcount <> 1 then
        raise dbms_dbfs_content.invalid_path;
    end if;
end;

procedure   deleteContent(
    store_name  in              varchar2,
    contentID   in              raw,
    filter      in              varchar2,
    soft_delete in              integer,
    ctx         in              dbms_dbfs_content_context_t)
is
begin
    raise dbms_dbfs_content.unsupported_operation;
end;

procedure   deleteDirectory(
    store_name  in              varchar2,
    path        in              varchar2,
    filter      in              varchar2,
    soft_delete in              integer,
    recurse     in              integer,
    ctx         in              dbms_dbfs_content_context_t)
is
begin
    raise dbms_dbfs_content.unsupported_operation;
end;

procedure   restorePath(
    store_name  in              varchar2,
    path        in              varchar2,
    filter      in              varchar2,
    ctx         in              dbms_dbfs_content_context_t)
is
begin
    raise dbms_dbfs_content.unsupported_operation;
end;

procedure   purgePath(
    store_name  in              varchar2,
    path        in              varchar2,
    filter      in              varchar2,
    ctx         in              dbms_dbfs_content_context_t)
is
begin
    raise dbms_dbfs_content.unsupported_operation;
end;

procedure   restoreAll(
    store_name  in              varchar2,
    path        in              varchar2,
    filter      in              varchar2,
    ctx         in              dbms_dbfs_content_context_t)
is
begin
    raise dbms_dbfs_content.unsupported_operation;
end;
```

```
procedure   purgeAll(
    store_name  in              varchar2,
    path        in              varchar2,
    filter      in              varchar2,
    ctx         in              dbms_dbfs_content_context_t)
is
begin
    raise dbms_dbfs_content.unsupported_operation;
end;



/*
 * DBFS SPI: path get/put operations.
 *
 * Existing path items can be accessed (for query or for update) and
 * modified via simple get/put methods.
 *
 * All pathnames allow their metadata (i.e. properties) to be
 * read/modified. On completion of the call, the client can request
 * (via "prop_flags") specific properties to be fetched as well.
 *
 * File pathnames allow their data (i.e. content) to be
 * read/modified. On completion of the call, the client can request
 * (via the "prop_data" bitmaks in "prop_flags") a new BLOB locator
 * that can be used to continue data access.
 *
 * Files can also be read/written without using BLOB locators, by
 * explicitly specifying logical offsets/buffer-amounts and a
 * suitably sized buffer.
 *
 *
 * Update accesses must specify the "forUpdate" flag. Access to link
 * pathnames can be implicitly and internally deferenced by stores
 * (subject to feature support) if the "deref" flag is
 * specified---however, this is dangerous since symbolic links are
 * not always resolvable.
 *
 *
 * The read methods (i.e. "getPath" where "forUpdate" is "false"
 * also accepts a valid "asof" timestamp parameter that can be used
 * by stores to implement "as of" style flashback queries. Mutating
 * versions of the "getPath" and the "putPath" methods do not
 * support as-of modes of operation.
 *
 *
 * "getPathNowait" implies a "forUpdate", and, if implemented (see
 * "feature_nowait"), allows providers to return an exception
 * (ORA-54) rather than wait for row locks.
 *
 */

procedure   getPath(
    store_name  in              varchar2,
    path        in              varchar2,
    properties  in out nocopy   dbms_dbfs_content_properties_t,
    content     out    nocopy   blob,
    item_type   out             integer,
    prop_flags  in              integer,
```

```
    forUpdate   in              integer,
    deref       in              integer,
    ctx         in              dbms_dbfs_content_context_t)
is
    guid        number;
begin
    if (deref <> 0) then
        raise dbms_dbfs_content.unsupported_operation;
    end if;

    select ora_hash(path) into guid from dual;

    if (path = '/') then
        if (forUpdate <> 0) then
            raise dbms_dbfs_content.unsupported_operation;
        end if;

        content    := null;
        item_type  := dbms_dbfs_content.type_directory;
        properties := dbms_dbfs_content_properties_t(
        dbms_dbfs_content_property_t(
            'std:guid',
            to_char(guid),
            dbms_types.TYPECODE_NUMBER));

        return;
    end if;

    begin
        if (forUpdate <> 0) then
            select t.data into content from tbfs.tbfst t
                where ('/' || t.key) = path
                for update;
        else
            select t.data into content from tbfs.tbfst t
                where ('/' || t.key) = path;
        end if;
    exception
        when no_data_found then
            raise dbms_dbfs_content.invalid_path;
    end;

    item_type  := dbms_dbfs_content.type_file;
    properties := dbms_dbfs_content_properties_t(
        dbms_dbfs_content_property_t(
            'std:length',
            to_char(dbms_lob.getlength(content)),
            dbms_types.TYPECODE_NUMBER),
        dbms_dbfs_content_property_t(
            'std:guid',
            to_char(guid),
            dbms_types.TYPECODE_NUMBER));
end;

procedure   getPathNowait(
    store_name  in              varchar2,
    path        in              varchar2,
    properties  in out nocopy   dbms_dbfs_content_properties_t,
    content     out    nocopy   blob,
    item_type   out             integer,
```

```
    prop_flags  in              integer,
    deref       in              integer,
    ctx         in              dbms_dbfs_content_context_t)
is
begin
    raise dbms_dbfs_content.unsupported_operation;
end;

procedure   getPath(
    store_name  in              varchar2,
    path        in              varchar2,
    properties  in out nocopy   dbms_dbfs_content_properties_t,
    amount      in out          number,
    offset      in              number,
    buffer      out    nocopy   raw,
    prop_flags  in              integer,
    ctx         in              dbms_dbfs_content_context_t)
is
    content     blob;
    guid        number;
begin
    if (path = '/') then
        raise dbms_dbfs_content.unsupported_operation;
    end if;

    begin
        select t.data into content from tbfs.tbfst t
            where ('/' || t.key) = path;
    exception
        when no_data_found then
            raise dbms_dbfs_content.invalid_path;
    end;

    select ora_hash(path) into guid from dual;
    dbms_lob.read(content, amount, offset, buffer);

    properties := dbms_dbfs_content_properties_t(
        dbms_dbfs_content_property_t(
            'std:length',
            to_char(dbms_lob.getlength(content)),
            dbms_types.TYPECODE_NUMBER),
        dbms_dbfs_content_property_t(
            'std:guid',
            to_char(guid),
            dbms_types.TYPECODE_NUMBER));
end;

procedure   getPath(
    store_name  in              varchar2,
    path        in              varchar2,
    properties  in out nocopy   dbms_dbfs_content_properties_t,
    amount      in out          number,
    offset      in              number,
    buffers     out    nocopy   dbms_dbfs_content_raw_t,
    prop_flags  in              integer,
    ctx         in              dbms_dbfs_content_context_t)
is
begin
    raise dbms_dbfs_content.unsupported_operation;
end;
```

```
procedure   putPath(
    store_name  in              varchar2,
    path        in              varchar2,
    properties  in out nocopy   dbms_dbfs_content_properties_t,
    content     in out nocopy   blob,
    item_type   out             integer,
    prop_flags  in              integer,
    ctx         in              dbms_dbfs_content_context_t)
is
    guid        number;
begin
    if (path = '/') then
        raise dbms_dbfs_content.unsupported_operation;
    end if;

    if content is null then
        content := empty_blob();
    end if;

    update tbfs.tbfst t
        set t.data = content
        where ('/' || t.key) = path
        returning t.data into content;

    if sql%rowcount <> 1 then
        raise dbms_dbfs_content.invalid_path;
    end if;

    select ora_hash(path) into guid from dual;

    item_type  := dbms_dbfs_content.type_file;
    properties := dbms_dbfs_content_properties_t(
        dbms_dbfs_content_property_t(
            'std:length',
            to_char(dbms_lob.getlength(content)),
            dbms_types.TYPECODE_NUMBER),
        dbms_dbfs_content_property_t(
            'std:guid',
            to_char(guid),
            dbms_types.TYPECODE_NUMBER));
end;

procedure   putPath(
    store_name  in              varchar2,
    path        in              varchar2,
    properties  in out nocopy   dbms_dbfs_content_properties_t,
    amount      in              number,
    offset      in              number,
    buffer      in              raw,
    prop_flags  in              integer,
    ctx         in              dbms_dbfs_content_context_t)
is
    content     blob;
    guid        number;
begin
    if (path = '/') then
        raise dbms_dbfs_content.unsupported_operation;
    end if;
```

```
    begin
        select t.data into content from tbfs.tbfst t
            where ('/' || t.key) = path
            for update;
    exception
        when no_data_found then
            raise dbms_dbfs_content.invalid_path;
    end;

    select ora_hash(path) into guid from dual;
    dbms_lob.write(content, amount, offset, buffer);

    properties := dbms_dbfs_content_properties_t(
        dbms_dbfs_content_property_t(
            'std:length',
            to_char(dbms_lob.getlength(content)),
            dbms_types.TYPECODE_NUMBER),
        dbms_dbfs_content_property_t(
            'std:guid',
            to_char(guid),
            dbms_types.TYPECODE_NUMBER));
end;

procedure   putPath(
    store_name  in              varchar2,
    path        in              varchar2,
    properties  in out nocopy   dbms_dbfs_content_properties_t,
    written     out             number,
    offset      in              number,
    buffers     in              dbms_dbfs_content_raw_t,
    prop_flags  in              integer,
    ctx         in              dbms_dbfs_content_context_t)
is
begin
    raise dbms_dbfs_content.unsupported_operation;
end;



/*
 * DBFS SPI: rename/move operations.
 *
 * Pathnames can be renamed or moved, possibly across directory
 * hierarchies and mount-points, but within the same store.
 *
 *
 * Non-directory pathnames previously accessible via "oldPath" are
 * renamed as a single item subsequently accessible via "newPath";
 * assuming that "newPath" does not already exist.
 *
 * If "newPath" exists and is not a directory, the rename implicitly
 * deletes the existing item before renaming "oldPath". If "newPath"
 * exists and is a directory, "oldPath" is moved into the target
 * directory.
 *
 *
 * Directory pathnames previously accessible via "oldPath" are
 * renamed by moving the directory and all of its children to
 * "newPath" (if it does not already exist) or as children of
 * "newPath" (if it exists and is a directory).
```

```
 *
 *
 * Stores/providers that support contentID-based access and lazy
 * pathname binding also support the "setPath" method that
 * associates an existing "contentID" with a new "path".
 *
 */

procedure   renamePath(
    store_name  in              varchar2,
    oldPath     in              varchar2,
    newPath     in              varchar2,
    properties  in out nocopy   dbms_dbfs_content_properties_t,
    ctx         in              dbms_dbfs_content_context_t)
is
begin
    raise dbms_dbfs_content.unsupported_operation;
end;

procedure   setPath(
    store_name  in              varchar2,
    contentID   in              raw,
    path        in              varchar2,
    properties  in out nocopy   dbms_dbfs_content_properties_t,
    ctx         in              dbms_dbfs_content_context_t)
is
begin
    raise dbms_dbfs_content.unsupported_operation;
end;



/*
 * DBFS SPI: directory navigation and search.
 *
 * The DBFS API can list or search the contents of directory
 * pathnames, optionally recursing into sub-directories, optionally
 * seeing soft-deleted items, optionally using flashback "as of" a
 * provided timestamp, and optionally filtering items in/out within
 * the store based on list/search predicates.
 *
 */

function    list(
    store_name  in              varchar2,
    path        in              varchar2,
    filter      in              varchar2,
    recurse     in              integer,
    ctx         in              dbms_dbfs_content_context_t)
        return  dbms_dbfs_content_list_items_t
            pipelined
is
begin
    for rws in (select * from tbfs.tbfst)
    loop
        pipe row(dbms_dbfs_content_list_item_t(
            '/' || rws.key, rws.key, dbms_dbfs_content.type_file));
    end loop;
end;
```

```
function    search(
    store_name  in              varchar2,
    path        in              varchar2,
    filter      in              varchar2,
    recurse     in              integer,
    ctx         in              dbms_dbfs_content_context_t)
        return  dbms_dbfs_content_list_items_t
            pipelined
is
begin
    raise dbms_dbfs_content.unsupported_operation;
end;



/*
 * DBFS SPI: locking operations.
 *
 * Clients of the DBFS API can apply user-level locks to any valid
 * pathname (subject to store feature support), associate the lock
 * with user-data, and subsequently unlock these pathnames.
 *
 * The status of locked items is available via various optional
 * properties (see "opt_lock*" above).
 *
 *
 * It is the responsibility of the store (assuming it supports
 * user-defined lock checking) to ensure that lock/unlock operations
 * are performed in a consistent manner.
 *
 */

procedure   lockPath(
    store_name  in              varchar2,
    path        in              varchar2,
    lock_type   in              integer,
    lock_data   in              varchar2,
    ctx         in              dbms_dbfs_content_context_t)
is
begin
    raise dbms_dbfs_content.unsupported_operation;
end;

procedure   unlockPath(
    store_name  in              varchar2,
    path        in              varchar2,
    ctx         in              dbms_dbfs_content_context_t)
is
begin
    raise dbms_dbfs_content.unsupported_operation;
end;



/*
 * DBFS SPI: access checks.
 *
 * Check if a given pathname (store_name, path, pathtype) can be
 * manipulated by "operation (see the various
 * "dbms_dbfs_content.op_xxx" opcodes) by "principal".
```

```
     *
     * This is a convenience function for the DBFS API; a store that
     * supports access control still internally performs these checks to
     * guarantee security.
     *
     */

    function    checkAccess(
        store_name  in              varchar2,
        path        in              varchar2,
        pathtype    in              integer,
        operation   in              varchar2,
        principal   in              varchar2)
            return  integer
    is
    begin
        return 1;
    end;
end;
/
show errors;
```

capi.sql

The `capi.sql` script registers and mounts the DBFS.

```
connect tbfs/tbfs;

exec dbms_dbfs_content.registerStore('MY_TBFS', 'table', 'TBFS');
exec dbms_dbfs_content.mountStore('MY_TBFS', singleton => true);
commit;
```

10
Using DBFS

This chapter describes how to implement the DBFS File System.

This chapter contains these topics:

- DBFS Installation
- Creating a DBFS File System
- Accessing a DBFS File System
- DBFS Administration
- Shrinking and Reorganizing DBFS Filesystems

DBFS Installation

DBFS is a part of Oracle database installation and is installed under `ORACLE_HOME`.

`$ORACLE_HOME/rdbms/admin` contains these DBFS utility scripts:

- Content API (CAPI)
- SecureFiles Store (SFS)

`$ORACLE_HOME/bin` contains:

- `dbfs_client` executable

`$ORACLE_HOME/rdbms/admin` contains:

- SQL (`.plb` extension) scripts for the content store

Creating a DBFS File System

This section covers these topics:

- Privileges Required to Create a DBFS File System
- Advantages of Non-Partitioned Versus Partitioned DBFS File Systems
- Creating a Non-Partitioned File System
- Creating a Partitioned File System
- Dropping a File System

Privileges Required to Create a DBFS File System

Database users must have at least these privileges to create a file system:

- GRANT CONNECT
- CREATE SESSION
- RESOURCE, CREATE TABLE
- CREATE PROCEDURE
- DBFS_ROLE

Advantages of Non-Partitioned Versus Partitioned DBFS File Systems

You can create either non-partitioned or partitioned file systems. Partitioning is the best performing and most scalable way to create a file system in DBFS and is the default.

Space cannot be shared between partitions, so it is possible for one partition to run out of space even when other partitions have space. This is usually not an issue if the file system size is big compared to the size of the individual files. However, if file sizes are a big percentage of the file system size, it may result in the ENOSPC error even if the file system is not full.

Another implication of partitioning is that a *rename* operation can require rewriting the file, which can be expensive if the file is big.

Creating a Non-Partitioned File System

You can create a file system by running DBFS_CREATE_FILESYSTEM.SQL while logged in as a user with DBFS administrator privileges. By default, the file system is non-partitioned. For example:

```
$ sqlplus dbfs_user/@db_server
     @$ORACLE_HOME/rdbms/admin/dbfs_create_filesystem.sql tablespace_name
     file_system_name
```

The following example creates a file system called staging_area in an existing tablespace dbfs_tbspc.

```
$ sqlplus dbfs_user/db_server
   @$ORACLE_HOME/rdbms/admin/dbfs_create_filesystem.sql
   dbfs_tbspc staging_area
```

Creating a Partitioned File System

Partitioning creates multiple physical segments in the database, and files are distributed randomly in these partitions.

You can create a partitioned file system by running DBFS_CREATE_FILESYSTEM_ ADVANCED.SQL while logged in as a user with DBFS administrator privileges.

The following example creates a partitioned file system called staging_area in the existing tablespace dbfs_tbspc.

```
$ sqlplus dbfs_user/@db_server
       @$ORACLE_HOME/rdbms/admin/dbfs_create_filesystem_advanced.sql dbfs_tbspc
       staging_area nocompress nodeduplicate noencrypt partition
```

Dropping a File System

You can drop a file system by running DBFS_DROP_FILESYSTEM.SQL, for example:

```
$ sqlplus dbfs_user/@db_server @$ORACLE_HOME/rdbms/admin/dbfs_drop_filesystem.sql
file_system_name
```

Accessing a DBFS File System

This section discusses how you access a DBFS file system, including prerequisites, access interfaces, and security.

DBFS Client Prerequisites

This section lists prerequisites for using the DBFS File System Client, named `dbfs_client`, which runs on each system that will access DBFS filesystems.

- The `dbfs_client` host must have the Oracle client libraries installed.
- The `dbfs_client` can be used as a direct RDBMS client using the DBFS Command Interface on Linux, Linux.X64, Solaris, Solaris64, AIX, HPUX and Windows platforms.
- The `dbfs_client` can only be used as a mount client on Linux, Linux.X64, and Solaris 11 platforms. This requires the following:
  - `dbfs_client` host must have the `FUSE` Linux package or the Solaris `libfuse` package installed.
  - A group named `fuse` must be created, with the user name that runs the `dbfs_client` as a member.

  See "DBFS Mounting Interface (Linux and Solaris Only)" on page 10-5 for further details.

Using the DBFS Client Command-Line Interface

The DBFS client command-line interface allows users direct access to files stored in DBFS. Users can copy files in and out of the DBFS filesystem from any host on the network as well as perform other pre-defined commands.

The command-line interface has slightly better performance than the DBFS client mount interface because it does not mount the file system, thus bypassing the user space file system. However, it is not transparent to applications.

The DBFS client mount interface allows DBFS to be mounted through a file system mount point thus providing transparent access to files stored in DBFS with generic file system operations.

This section contains the following topics:

- Removing Files and Directories

Creating Content Store Paths

All DBFS content store paths must be preceded by `dbfs:` for example: `dbfs:/staging_area/file1`. All database path names specified must be absolute paths. To run DBFS commands, specify `--command` to the DBFS client.

```
dbfs_client db_user@db_server --command command [switches] [arguments]
```

where:

- *command* is the executable command, such as `ls`, `cp`, `mkdir`, or `rm`.
- *switches* are specific for each command.
- *arguments* are file names or directory names, and are specific for each command.

Note that `dbfs_client` returns a nonzero value in case of failure.

Creating a Directory

You can use the `mkdir` command to create a new directory.

```
dbfs_client db_user@db_server --command mkdir directory_name
```

where:

- *directory_name* is the name of the directory created. For example:

```
$ dbfs_client ETLUser@DBConnectString --command mkdir dbfs:/staging_area/dir1
```

Listing a Directory

You can use the `ls` command to list the contents of a directory.

```
dbfs_client db_user@db_server --command ls [switches] target
```

where

- *target* is the listed directory.
- *switches* is any combination of the following:
  - *-a* shows all files, including '.' and '..'.
  - *-l* shows the long listing format: name of each file, the file type, permissions, and size.
  - *-R* lists subdirectories recursively.

For example:

```
$ dbfs_client ETLUser@DBConnectString --command  ls dbfs:/staging_area/dir1

$ dbfs_client ETLUser@DBConnectString --command  ls -l -a -R dbfs:/staging_area/dir1
```

Copying Files and Directories

You can use the `cp` command to copy files or directories from the source location to the destination location. It also supports recursive copy of directories.

```
dbfs_client db_user@db_server --command cp [switches] source destination
```

where:

- *source* is the source location.

- *destination* is the destination location.
- *switches* is either -R or -r, the options to recursively copy all source contents into the destination directory.

The following example copies the contents of the local directory, 01-01-10-dump recursively into a directory in DBFS:

```
$ dbfs_client ETLUser@DBConnectString --command cp -R  01-01-10-dump dbfs:/staging_area/
```

The following example copies the file hello.txt from DBFS to a local file Hi.txt:

```
$ dbfs_client ETLUser@DBConnectString --command cp dbfs:/staging_area/hello.txt Hi.txt
```

Removing Files and Directories

You can use the command rm to delete a file or directory. It also supports recursive delete of directories.

```
dbfs_client db_user@db_server --command rm [switches] target
```

where:

- *target* is the listed directory.
- *switches* is either -R or -r, the options to recursively delete all contents.

For example:

```
$ dbfs_client ETLUser@DBConnectString --command rm  dbfs:/staging_area/srcdir/hello.txt

$ dbfs_client ETLUser@DBConnectString --command rm -R  dbfs:/staging_area/dir1
```

DBFS Mounting Interface (Linux and Solaris Only)

This section discusses how to mount DBFS using dbfs_client. It applies to Linux and Solaris. The instructions indicate different requirements for the two platforms.

This section contains the following topics:

Installing FUSE on Solaris 11 SRU7 and Later

To use dbfs_client as a mount client in Solaris 11 SRU7 and later, you must install FUSE.

Run the following package as root.

```
pkg install libfuse
```

Mounting the DBFS Store

To mount a DBFS store, run the dbfs_client program. Ensure that LD_LIBRARY_PATH has the correct path to the Oracle client libraries before calling this program.

The dbfs_client program does not return until the file system is unmounted.

For the most secure method of specifying the password, see "Using Oracle Wallet with DBFS Client" on page 10-16.

Solaris-Specific Privileges On Solaris, the user must have the Solaris privilege `PRIV_SYS_MOUNT` to perform mount and unmount operations on DBFS filesystems.

Edit `/etc/user_attr` and add or modify the user entry (assuming the user is Oracle) as follows:

```
oracle::::type=normal;project=group.dba;defaultpriv=basic,priv_sys_mount;;auth
 s=solaris.smf.*
```

About the Mount Command for Solaris and Linux The `dbfs_client` command has the following syntax:

```
dbfs_client db_user@db_server [-o option_1 -o option_2 ...] mount_point
```

where the mandatory parameters are:

- *db_user* is the name of the database user who owns the DBFS content store file system.
- *db_server* is a valid connect string to the Oracle Database server, such as `hrdb_host:1521/hrservice`.
- *mount_point* is the path where the Database File System is mounted. Note that all file systems owned by the database user are visible at the mount point.

The options are:

- `direct_io`: To bypass the OS page cache and provide improved performance for large files. Programs in the file system cannot be executed with this option. Oracle recommends this option when DBFS is used as an ETL staging area.
- `wallet`: To run the DBFS client in the background. The Wallet must be configured to get its credentials.
- `failover`: To fail over the DBFS client to surviving database instances without data loss. Expect some performance cost on writes, especially for small files.
- `allow_root`: To allow the root user to access the filesystem. You must set the `user_allow_other` parameter in the `/etc/fuse.conf` configuration file.
- `allow_other`: To allow other users to access the filesystem. You must set the `user_allow_other` parameter in the `/etc/fuse.conf` configuration file.
- `rw`: To mount the filesystem as read-write. This is the default setting.
- `ro`: To mount the filesystem as read-only. Files cannot be modified.
- `trace_level=`*n* sets the trace level. Trace levels are:
  - `1 DEBUG`
  - `2 INFO`
  - `3 WARNING`
  - `4 ERROR`: The default tracing level. It outputs diagnostic information only when an error happens. It is recommended that this tracing level is always enabled.
  - `5 CRITICAL`

- `trace_file=STR`: Specifies the tracing log file, where `STR` can be either a *`file_name`* or `syslog`.
- `trace_size=`*`trcfile_size`*: Specifies size of the trace file in MB. By default, `dbfs_client` rotates tracing output between two 10MB files. Specifying `0` for `trace_size` sets the maximum size of the trace file to unlimited.

Mounting Examples This section contains these examples:

- Example 10–1, "Mounting a File System"
- Example 10–2, "Mounting a File System with Password at Command Prompt"
- Example 10–3, "Mounting a File System with Password Read from a File"

Example 10–1 Mounting a File System

To use this example with a wallet (see "Using Oracle Wallet with DBFS Client" on page 10-16) configure the `LD_LIBRARY_PATH` and *`ORACLE_HOME`* environment variables correctly.

1. Login as admin user.
2. Mount the DBFS store. (Oracle recommends that you do not perform this step as root user.)

   ```
   % dbfs_client @/dbfsdb -o wallet,rw,user,direct_io /mnt/dbfs
   ```

3. [Optional] To test if the previous step was successful, as admin user, list the `dbfs` directory.

   ```
   $ ls /mnt/tdbfs
   ```

Example 10–2 Mounting a File System with Password at Command Prompt

To mount a file system using `dbfs_client` by entering the password on the command prompt:

```
$ dbfs_client ETLUser@DBConnectString /mnt/dbfs
  password: xxxxxxx
```

Example 10–3 Mounting a File System with Password Read from a File

The following example mounts a file system and frees the terminal. It reads the password from a file:

```
$ nohup dbfs_client ETLUser@DBConnectString /mnt/dbfs < passwordfile.f &
$ ls -l /mnt/dbfs
drwxrwxrwx 10 root root 0 Feb  9 17:28 staging_area
```

Unmounting a File System

In Linux, you can run `fusermount` to unmount file systems.

```
$ fusermount -u <mount point>
```

In Solaris, you can run `umount` to unmount file systems.

```
$ umount -u <mount point>
```

Mounting DBFS Through fstab Utility for Linux

In Linux, you can configure `fstab` utility to use `dbfs_client` to mount a DBFS filesystem. To mount DBFS through `/etc/fstab`, you must use Oracle Wallet for authentication.

To mount DBFS through fstab:

1. Login as `root` user.
2. Change the user and group of `dbfs_client` to user `root` and group `fuse`.

   ```
   # chown root.fuse $ORACLE_HOME/bin/dbfs_client
   ```

3. Set the `setuid` bit on `dbfs_client` and restrict `execute` privileges to the user and group only.

   ```
   # chmod u+rwxs,g+rx-w,o-rwx dbfs_client
   ```

4. Create a symbolic link to `dbfs_client` in `/sbin` as "`mount.dbfs`".

   ```
   $ ln -s $ORACLE_HOME/bin/dbfs_client /sbin/mount.dbfs
   ```

5. Create a new Linux group called "`fuse`".
6. Add the Linux user that is running the DBFS Client to the `fuse` group.
7. Add the following line to `/etc/fstab`:

   ```
   /sbin/mount.dbfs#db_user@db_server mount_point fuse rw,user,noauto 0 0
   ```

   For example:

   ```
   /sbin/mount.dbfs#/@DBConnectString /mnt/dbfs fuse rw,user,noauto 0 0
   ```

8. The Linux user can mount the DBFS file system using the standard Linux `mount` command. For example:

   ```
   $ mount /mnt/dbfs
   ```

   Note that `FUSE` does not currently support `automount`.

Mounting DBFS Through the vfstab Utility for Solaris

On Solaris, file systems are commonly configured using the `vfstab` utility.

To mount DBFS through `vfstab`:

1. Create a mount shell script `mount_dbfs.sh` to use to start `dbfs_client`. All the environment variables that are required for Oracle RDBMS must be exported. These environment variables include `TNS_ADMIN`, `ORACLE_HOME`, and `LD_LIBRARY_PATH`. For example:

   ```
   #!/bin/ksh
   export TNS_ADMIN=/export/home/oracle/dbfs/tnsadmin
   export ORACLE_HOME=/export/home/oracle/11.2.0/dbhome_1
   export DBFS_USER=dbfs_user
   export DBFS_PASSWD=/tmp/passwd.f
   export DBFS_DB_CONN=dbfs_db
   export O=$ORACLE_HOME
   export LD_LIBRARY_PATH=$O/lib:$O/rdbms/lib:/usr/lib:/lib:$LD_LIBRARY_PATH
   export NOHUP_LOG=/tmp/dbfs.nohup

   (nohup $ORACLE_HOME/bin/dbfs_client $DBFS_USER@$DBFS_DB_CONN < $DBFS_PASSWD
            2>&1 & ) &
   ```

2. Add an entry for DBFS to `/etc/vfstab`. Specify the `mount_dbfs.sh` script for the `device_to_mount`. Specify `uvfs` for the `FS_type`. Specify no `formount_at_boot`. Specify mount options as needed. For example:

   ```
   /usr/local/bin/mount_dbfs.sh - /mnt/dbfs uvfs - no rw,allow_other
   ```

3. User can mount the DBFS file system using the standard Solaris mount command. For example:

   ```
   $ mount /mnt/dbfs
   ```

4. User can unmount the DBFS file system using the standard Solaris umount command. For example:

   ```
   $ umount /mnt/dbfs
   ```

Restrictions on Mounted File Systems

DBFS supports most file system operations with these exceptions:

- `ioctl`
- locking
- asynchronous I/O through `libaio`
- `O_DIRECT` file opens
- hard links, pipes
- other special file modes

Memory-mapped files are supported except in shared-writable mode. For performance reasons, DBFS does not update the file access time every time file data or the file data attributes are read.

You cannot run programs from a DBFS-mounted file system if the `direct_io` option is specified.

Oracle does not support exporting DBFS file systems using NFS or Samba.

File System Security Model

The database manages security in DBFS and does use not the operating system security model.

DBFS operates under a security model where all file systems created by a user are private to that user, by default. Oracle recommends maintaining this model. Because operating system users and Oracle Database users are different, it is possible to allow multiple operating system users to mount a single DBFS filesystem. These mounts may potentially have different mount options and permissions. For example, `user1` may mount a DBFS filesystem as `READ ONLY`, and `user2` may mount it as `READ WRITE`. However, Oracle Database views both users as having the same privileges because they would be accessing the filesystem as the same database user.

Access to a database file system requires a database login as a database user with privileges on the tables that underlie the file system.The database administrator grants access to a file system to database users, and different database users may have different `READ` or `UPDATE` privileges to the file system. The database administrator has access to all files stored in the DBFS file system.

On each client computer, access to a DBFS mount point is limited to the operating system user that mounts the file system. This, however, does not limit the number of users who can access the DBFS file system, because many users may separately mount the same DBFS file system.

DBFS only performs database privilege checking. Linux performs operating system file-level permission checking when a DBFS file system is mounted. DBFS does not perform this check either when using the command interface or when using the PL/SQL interface directly.

This following sections explore DBFS security in more detail:

- Enabling Shared Root Access
- Enabling DBFS Access Among Multiple Database Users

Enabling Shared Root Access

The operating system user who mounts the file system may allow root access to the file system by specifying the `allow_root` option. This option requires that `/etc/fuse.conf` file contain the `user_allow_other` field, as demonstrated in Example 10–4.

Example 10–4 Enabling Root Access for Other Users

```
# Allow users to specify the 'allow_root' mount option.
user_allow_other
```

Enabling DBFS Access Among Multiple Database Users

Some circumstances may require that multiple database users access the same filesystem. For example, the database user that owns the filesystem may be a privileged user and sharing its user credentials may pose a security risk. To mitigate this, DBFS allows multiple database users to share a subset of the filesystem state.

While DBFS registrations and mounts made through the DBFS content API are private to each user, the underlying filesystem and the tables on which they rely may be shared across users. After this is done, the individual filesystems may be independently mounted and used by different database users, either through SQL/PLSQL, or through `dbfs_client` APIs.

In the following example, user `user1` is able to modify the filesystem, and user `user2` can see these changes. Here, `user1` is the database user that creates a filesystem, and `user2` is the database user that eventually uses `dbfs_client` to mount and access the filesystem. Both `user1` and `user2` must have the `DBFS_ROLE` privilege.

To establish DBFS access sharing across multiple database users:

1. Connect as the user who creates the filesystem.

```
sys@tank as sysdba> connect user1
Connected.
```

2. Create the filesystem `user1_FS`, register the store, and mount it as `user1_mt`.

```
user1@tank> exec dbms_dbfs_sfs.createFilesystem('user1_FS');
user1@tank> exec dbms_dbfs_content.registerStore('user1_FS', 'posix', 'DBMS_
DBFS_SFS');
user1@tank> exec dbms_dbfs_content.mountStore('user1_FS', 'user1_mnt');
user1@tank> commit;
```

3. [Optional] You may check that the previous step has completed successfully by viewing all mounts.

```
user1@tank> select * from table(dbms_dbfs_content.listMounts);

STORE_NAME           |   STORE_ID|PROVIDER_NAME
---------------------|- ---------|----------------------------------------
PROVIDER_PKG         |PROVIDER_ID|PROVIDER_VERSION     |STORE_FEATURES
---------------------|-----------|---------------------|--------------------
STORE_GUID
----------
STORE_MOUNT
--------------------------------------------------------------------------------
CREATED
--------------------------------------------------------------------------------
MOUNT_PROPERTIES(PROPNAME, PROPVALUE, TYPECODE)
--------------------------------------------------------------------------------
user1_FS             | 1362968596|posix
"DBMS_DBFS_SFS"      | 3350646887|0.5.0               | 12714135  141867344
user1_mnt
01-FEB-10 09.44.25.357858 PM
DBMS_DBFS_CONTENT_PROPERTIES_T(
  DBMS_DBFS_CONTENT_PROPERTY_T('principal', (null), 9),
  DBMS_DBFS_CONTENT_PROPERTY_T('owner', (null), 9),
  DBMS_DBFS_CONTENT_PROPERTY_T('acl', (null), 9),
  DBMS_DBFS_CONTENT_PROPERTY_T('asof', (null), 187),
  DBMS_DBFS_CONTENT_PROPERTY_T('read_only', '0', 2))
```

4. [Optional] Connect as the user who will use the `dbfs_client`.

```
user1@tank> connect user2
Connected.
```

5. [Optional] Note that `user2` cannot see `user1`'s DBFS state, as he has no mounts.

```
user2@tank> select * from table(dbms_dbfs_content.listMounts);
```

6. While connected as `user1`, export filesystem `user1_FS` for access to any user with `DBFS_ROLE` privilege.

```
user1@tank> exec dbms_dbfs_sfs.exportFilesystem('user1_FS');
user1@tank> commit;
```

7. Connect as the user who will use the `dbfs_client`.

```
user1@tank> connect user2
Connected.
```

8. As `user2`, view all available tables.

```
user2@tank> select * from table(dbms_dbfs_sfs.listTables);

SCHEMA_NAME               |TABLE_NAME                |PTABLE_NAME
--------------------------|--------------------------|------------------
VERSION#
--------------------------------
CREATED
--------------------------------------------------------------------------------
FORMATTED
--------------------------------------------------------------------------------
PROPERTIES(PROPNAME, PROPVALUE, TYPECODE)
--------------------------------------------------------------------------------
```

```
user1                          |SFS$_FST_11                |SFS$_FSTP_11
0.5.0
01-FEB-10 09.43.53.497856 PM
01-FEB-10 09.43.53.497856 PM
(null)
```

9. As `user2`, register and mount the store, but do not re-create the `user1_FS` filesystem.

```
user2@tank> exec dbms_dbfs_sfs.registerFilesystem(
   'user2_FS', 'user1', 'SFS$_FST_11');
user2@tank> exec dbms_dbfs_content.registerStore(
   'user2_FS', 'posix', 'DBMS_DBFS_SFS');
user2@tank> exec dbms_dbfs_content.mountStore(
   'user2_FS', 'user2_mnt');
user2@tank> commit;
```

10. [Optional] As `user2`, you may check that the previous step has completed successfully by viewing all mounts.

```
user2@tank> select * from table(dbms_dbfs_content.listMounts);

STORE_NAME           |   STORE_ID|PROVIDER_NAME
---------------------|- ---------|----------------------------------------
PROVIDER_PKG         |PROVIDER_ID|PROVIDER_VERSION     |STORE_FEATURES
---------------------|-----------|---------------------|--------------------
STORE_GUID
----------
STORE_MOUNT
--------------------------------------------------------------------------------
CREATED
--------------------------------------------------------------------------------
MOUNT_PROPERTIES(PROPNAME, PROPVALUE, TYPECODE)
--------------------------------------------------------------------------------
user2_FS             | 1362968596|posix
"DBMS_DBFS_SFS"      | 3350646887|0.5.0                | 12714135  141867344
user1_mnt
01-FEB-10 09.46.16.013046 PM
DBMS_DBFS_CONTENT_PROPERTIES_T(
  DBMS_DBFS_CONTENT_PROPERTY_T('principal', (null), 9),
  DBMS_DBFS_CONTENT_PROPERTY_T('owner', (null), 9),
  DBMS_DBFS_CONTENT_PROPERTY_T('acl', (null), 9),
  DBMS_DBFS_CONTENT_PROPERTY_T('asof', (null), 187),
  DBMS_DBFS_CONTENT_PROPERTY_T('read_only', '0', 2))
```

11. [Optional] List path names for `user2` and `user1`. Note that another mount, `user2_mnt`, for store `user2_FS`, is available for `user2`. However, the underlying filesystem data is the same for `user2` as for `user1`.

```
user2@tank> select pathname from dbfs_content;

PATHNAME
--------------------------------------------------------------------------------
/user2_mnt
/user2_mnt/.sfs/tools
/user2_mnt/.sfs/snapshots
/user2_mnt/.sfs/content
/user2_mnt/.sfs/attributes
/user2_mnt/.sfs/RECYCLE
/user2_mnt/.sfs
```

```
user2@tank> connect user1
Connected.

user1@tank> select pathname from dbfs_content;

PATHNAME
--------------------
/user1_mnt
/user1_mnt/.sfs/tools
/user1_mnt/.sfs/snapshots
/user1_mnt/.sfs/content
/user1_mnt/.sfs/attributes
/user1_mnt/.sfs/RECYCLE
/user1_mnt/.sfs
```

12. In filesystem `user1_FS`, `user1` creates file *xxx*.

```
user1@tank> var ret number;
user1@tank> var data blob;
user1@tank> exec :ret := dbms_fuse.fs_create('/user1_mnt/xxx', content =>
:data);
user1@tank> select :ret from dual;
      :RET
----------
         0
```

13. [Optional] Write to file *xxx*, created in the previous step.

```
user1@tank> var buf varchar2(100);
user1@tank> exec :buf := 'hello world';
user1@tank> exec dbms_lob.writeappend(:data, length(:buf), utl_raw.cast_to_
raw(:buf));
user1@tank> commit;
```

14. [Optional] Show that file *xxx* exists, and contains the appended data.

```
user1@tank> select pathname, utl_raw.cast_to_varchar2(filedata)
  from dbfs_content where filedata is not null;

PATHNAME
--------------------------------------------------------------------------------
UTL_RAW.CAST_TO_VARCHAR2(FILEDATA)
--------------------------------------------------------------------------------
/user1_mnt/xxx
hello world
```

15. User `user2` sees the same file in their own DBFS-specific path name and mount prefix.

```
user1@tank> connect user2
Connected.

user2@tank> select pathname, utl_raw.cast_to_varchar2(filedata) from
  dbfs_content where filedata is not null;

PATHNAME
--------------------------------------------------------------------------------
UTL_RAW.CAST_TO_VARCHAR2(FILEDATA)
--------------------------------------------------------------------------------
/user2_mnt/xxx
hello world
```

After the export and register pairing completes, both users behave as equals with regard to their usage of the underlying tables. The `exportFilesystem()` procedure manages the necessary grants for access to the same data, which is shared between schemas. After `user1` calls `exportFilesystem()`, filesystem access may be granted to any user with `DBFS_ROLE`. Note that a different role can be specified.

Subsequently, `user2` may create a new DBFS filesystem that shares the same underlying storage as the `user1_FS` filesystem, by invoking `dbms_dbfs_sfs.registerFilesystem()`, `dbms_dbfs_sfs.registerStore()`, and `dmbs_dbfs_sfs.mountStore()` procedure calls.

When multiple database users share a filesystem, they must ensure that all database users unregister their interest in the filesystem before the owner (here, `user1`) drops the filesystem.

Oracle does not recommend that you run the DBFS as root.

HTTP, WebDAV, and FTP Access to DBFS

This section discusses support for components which enable HTTP, WebDAV, and FTP access to DBFS over the Internet, using various XML DB server protocols.

This sections covers these topics:

- Internet Access to DBFS Through XDB
- Web Distributed Authoring and Versioning (WebDAV) Access
- FTP Access to DBFS
- HTTP Access to DBFS

Internet Access to DBFS Through XDB

To provide database users who have DBFS authentication with a hierarchical file system-like view of registered and mounted DBFS stores, stores are displayed under the path `/dbfs`.

For guidelines on DBFS store creation, registration, deregistration, mount, unmount and deletion, see Chapter 8, "DBFS Content API".

The `/dbfs` folder is a virtual folder because the resources in its subtree are stored in DBFS stores, not the XDB repository. XDB issues a `dbms_dbfs_content.list()` command for the root path name "/" (with invoker rights) and receives a list of store access points as subfolders in the `/dbfs` folder. The list is comparable to `store_mount` parameters passed to `dbms_dbfs_content.mountStore()`. FTP and WebDAV users can navigate to these stores, while HTTP and HTTPS users access URLs from browsers.

Note that features implemented by the XDB repository, such as repository events, resource configurations, and ACLs, are not available for the `/dbfs` folder.

Web Distributed Authoring and Versioning (WebDAV) Access

WebDAV is an IETF standard protocol that provides users with a file-system-like interface to a repository over the Internet.

WebDAV server folders are typically accessed through Web Folders on Microsoft Windows (2000/NT/XP/Vista/7, and so on). You can access a resource using its fully qualified name, for example, `/dbfs/sfs1/dir1/file1.txt`, where `sfs1` is the name of a DBFS store.

You need to set up WebDAV on Windows to access the DBFS filesystem.

See Also: *Oracle XML DB Developer's Guide*

The user authentication required to access the DBFS virtual folder is the same as for the XDB repository.

When a WebDAV client connects to a WebDAV server for the first time, the user is typically prompted for a username and password, which the client uses for all subsequent requests. From a protocol point-of-view, every request contains authentication information, which XDB uses to authenticate the user as a valid database user. If the user does not exist, the client does not get access to the DBFS store or the XDB repository. Upon successful authentication, the database user becomes the current user in the session.

XDB supports both basic authentication and digest authentication. For security reasons, it is highly recommended that HTTPS transport be used if basic authentication is enabled.

FTP Access to DBFS

FTP access to DBFS uses the standard FTP clients found on most Unix-based distributions. FTP is a file transfer mechanism built on client-server architecture with separate control and data connections.

FTP users are authenticated as database users. The protocol, as outlined in RFC 959, uses clear text user name and password for authentication. Therefore, FTP is not a secure protocol.

The following commands are supported for DBFS:

- `USER`: Authentication username
- `PASS`: Authentication password
- `CWD`: Change working directory
- `CDUP`: Change to Parent directory
- `QUIT`: Disconnect
- `PORT`: Specifies an address and port to which the server should connect
- `PASV`: Enter passive mode
- `TYPE`: Sets the transfer mode, such as, ASCII or Binary
- `RETR`: Transfer a copy of the file
- `STOR`: Accept the data and store the data as a file at the server site
- `RNFR`: Rename From
- `RNTO`: Rename To
- `DELE`: Delete file
- `RMD`: Remove directory
- `MKD`: Make a directory
- `PWD`: Print working directory
- `LIST`: Listing of a file or directory. Default is current directory.
- `NLST`: Returns file names in a directory
- `HELP`: Usage document

- SYST: Return system type
- FEAT: Gets the feature list implemented by the server
- NOOP: No operation (used for keep-alives)
- EPRT: Extended address (IPv6) and port to which the server should connect
- EPSV: Enter extended passive mode (IPv6)

HTTP Access to DBFS

Users have read-only access through HTTP/HTTPS protocols. Users point their browsers to a DBFS store using the XDB HTTP server with a URL such as `https://hostname:port/dbfs/sfs1` where `sfs1` is a DBFS store name.

DBFS Administration

This sections describes the DBFS administration tools.

This section contains the following topics:

- Using Oracle Wallet with DBFS Client
- Performing DBFS Diagnostics
- Managing DBFS Client Failover
- Sharing and Caching DBFS
- Backing up DBFS
- Small File Performance of DBFS
- Enabling Advanced SecureFiles LOB Features for DBFS

Using Oracle Wallet with DBFS Client

An Oracle Wallet allows the DBFS client to mount a DBFS store without requiring the user to enter a password. Refer to *Oracle Database Enterprise User Security Administrator's Guide* for more information about creation and management of wallets. The "`/@`" syntax means to use the wallet, as shown in Step 7.

To create an Oracle Wallet and use it with dbfs_client:

1. Create a directory for the wallet. For example:

```
mkdir $ORACLE_HOME/oracle/wallet
```

2. Create an auto-login wallet.

```
mkstore -wrl $ORACLE_HOME/oracle/wallet -create
```

3. Add the wallet location in the client's `sqlnet.ora` file:

```
WALLET_LOCATION = (SOURCE = (METHOD = FILE) (METHOD_DATA = (DIRECTORY =
 $ORACLE_HOME/oracle/wallet) ) )
```

4. Add the following parameter in the client's `sqlnet.ora` file:

```
SQLNET.WALLET_OVERRIDE = TRUE
```

5. Create credentials:

```
mkstore -wrl wallet_location -createCredential db_connect_string username
```

password

For example:

```
mkstore -wrl $ORACLE_HOME/oracle/wallet -createCredential DBConnectString scott
password
```

6. Add the connection alias to your `tnsnames.ora` file.
7. Use `dbfs_client` with Oracle Wallet.

   For example:

```
$ dbfs_client -o wallet /@DBConnectString /mnt/dbfs
```

Performing DBFS Diagnostics

The `dbfs_client` program supports multiple levels of tracing to help diagnose problems. It can either output traces to a file or to `/var/log/messages` using the `syslog` daemon on Linux. When tracing to a file, it keeps two trace files on disk. `dbfs_client` rotates the trace files automatically and limits disk usage to 20 MB.

By default, tracing is turned off except for critical messages which are always logged to `/var/log/messages`.

If `dbfs_client` is not able to connect to the Oracle Database, enable tracing using `trace_level` and `trace_file` options. Tracing prints additional messages to log file for easier debugging.

DBFS uses Oracle Database for storing files. Sometimes Oracle server issues are propagated to `dbfs_client` as errors. If there is a `dbfs_client` error, please view the Oracle server logs to see if that is the root cause.

Managing DBFS Client Failover

In cases of failure of one database instance in an Oracle RAC cluster, `dbfs_client` can failover to one of the other existing database instances. For `dbfs_client` failover to work correctly, you must modify the Oracle database service and specify failover parameters, as demonstrated in Example 10–5.

Example 10–5 Enabling DBFS Client Failover Events

```
exec DBMS_SERVICE.MODIFY_SERVICE(service_name => 'service_name',
             aq_ha_notifications => true,
             failover_method => 'BASIC',
             failover_type => 'SELECT',
             failover_retries => 180,
             failover_delay => 1);
```

To ensure no data loss during failover of the DBFS connection after a failure of the back-end Oracle database instance, specify the `-o failover` mount option, as demonstrated in Example 10–6. In this case, cached *writes* may be lost if the client loses the connection. However, back-end failover to other Oracle RAC instances or standby databases does not cause lost writes.

Example 10–6 Preventing Data Loss During Failover Events

```
$ dbfs_client database_user@database_server -o failover /mnt/dbfs
```

Sharing and Caching DBFS

Multiple copies of `dbfs_client` can access the same shared file system. The sharing and caching semantics are similar to NFS in using a behavior referred to as *close-to-open cache consistency*. The default mode caches writes on the client and flushes them after a timeout or after the user closes the file. Also, writes to a file only appear to clients that open the file after the writer closed the file.

To bypass client-side write caching, specify `O_SYNC` when the file is opened. To force writes in the cache to disk call `fsync`.

Backing up DBFS

There are two alternatives for backing up DBFS. You can back up the tables that underlie the file system at the database level or use a file system backup utility, such as Oracle Secure Backup, through a mount point.

This section contains the following topics:

- Backing up DBFS at the Database Level
- Backing up DBFS through a File System Utility

Backing up DBFS at the Database Level

An advantage of backing up the tables at the database level is that the files in the file system are always consistent with the relational data in the database. A full restore and recover of the database also fully restores and recovers the file system with no data loss. During a point-in-time recovery of the database, the files are recovered to the specified time. As usual with database backup, modifications that occur during the backup do not affect the consistency of a restore. The entire restored file system is always consistent with respect to a specified time stamp.

Backing up DBFS through a File System Utility

The advantage of backing up the file system using a file system backup utility is that individual files can be restored from backup more easily. Any changes made to the restored files after the last backup are lost.

You must specify the `allow_root` mount option if backups are scheduled using the Oracle Secure Backup Administrative Server.

Small File Performance of DBFS

Like any shared file system, the performance of DBFS for small files lags the performance of a local file system. Each file data or metadata operation in DBFS must go through the `FUSE` user mode file system and then be forwarded across the network to the database. Therefore, each operation that is not cached on the client takes a few milliseconds to run in DBFS.

For operations that involve an input/output (IO) to disk, the time delay overhead is masked by the wait for the disk IO. Naturally, larger IOs have a lower percentage overhead than smaller IOs. The network overhead is more noticeable for operations that do not issue a disk IO.

When you compare the operations on a few small files with a local file system, the overhead is not noticeable, but operations that affect thousands of small files incur a much more noticeable overhead. For example, listing a single directory or looking at a single file produce near instantaneous response, while searching across a directory tree with many thousands of files results in a larger relative overhead.

Enabling Advanced SecureFiles LOB Features for DBFS

DBFS offers the following advanced features available with SecureFiles LOBs: compression, deduplication, encryption, and partitioning. For example, DBFS can be configured as a compressed file system with partitioning. At the time of creating a DBFS file system, you must specify the set of enabled features for the file system. See Chapter 4, "Using Oracle LOB Storage" and "Creating a Partitioned File System" on page 10-2 for more information about the features of SecureFiles LOBs.

Example 10–7 Enabling Advanced Secure Files LOB Features for DBFS

```
$ sqlplus @dbfs_create_filesystem_advanced tablespace_name file_systemname
  [compress-high | compress-medium | compress-low | nocompress]
  [deduplicate | nodeduplicate]
  [encrypt | noencrypt]
  [partition | non-partition]
```

Shrinking and Reorganizing DBFS Filesystems

A DBFS Filesystem uses Online Filesystem Reorganization to shrink itself, enabling the release of allocated space back to the containing tablespace.

DBFS filesystems, like other database segments, grow dynamically with the addition or enlargement of files and directories. Growth occurs with the allocation of space from the tablespace that holds the DBFS filesystem to the various segments that make up the filesystem.

However, even if files and directories in the DBFS filesystem are deleted, the allocated space is not released back to the containing tablespace, but continues to exist and be available for other DBFS entities. A process called Online Filesystem Reorganization solves this problem by shrinking the DBFS Filesystem.

The DBFS Online Filesystem Reorganization utility internally uses the Oracle Database online redefinition facility, with the original filesystem and a temporary placeholder corresponding to the base and interim objects in the online redefinition model.

> **See Also:** *Oracle Database Administrator's Guide* for further information about online redefinition

This section covers these topics:

- Advantages of Online Filesystem Reorganization
- Determining Availability of Online Filesystem Reorganization
- Invoking Online Filesystem Reorganization

Advantages of Online Filesystem Reorganization

DBFS Online Filesystem Reorganization is a powerful data movement facility with these advantages:

- **It is online:** When reorganization is taking place, the filesystem remains fully available for read and write operations for all applications.
- **It can reorganize the structure:** The underlying physical structure and organization of the DBFS filesystem can be changed in many ways, such as:
  - A non-partitioned filesystem can be converted to a partitioned filesystem and vice-versa.

  - Special SecureFiles LOB properties can be selectively enabled or disabled in any combination, including the compression, encryption, and deduplication properties.
  - The data in the filesystem can be moved across tablespaces or within the same tablespace.
- **It can reorganize multiple filesystems concurrently:** Multiple different filesystems can be reorganized at the same time, if no temporary filesystems have the same name and the tablespaces have enough free space, typically, twice the space requirement for each filesystem being reorganized.

Determining Availability of Online Filesystem Reorganization

DBFS for Oracle Database 12*c* and later supports online filesystem reorganization. Some earlier versions also support the facility. To determine if your version does, query for a specific function in the DBFS PL/SQL packages, as shown below:

```
$ sqlplus / as sysdba
SELECT * FROM dba_procedures
WHERE   owner = 'SYS'
    and object_name = 'DBMS_DBFS_SFS'
    and procedure_name = 'REORGANIZEFS';

OWNER
--------------------------------------------------------------------------------
OBJECT_NAME
--------------------------------------------------------------------------------
PROCEDURE_NAME
--------------------------------------------------------------------------------
OBJECT_ID|SUBPROGRAM_ID|OVERLOAD                                |OBJECT_TYPE
|AGG|PIP
----------|-------------|--------------------------------------|-------------|---|---
IMPLTYPEOWNER
--------------------------------------------------------------------------------
IMPLTYPENAME
--------------------------------------------------------------------------------
PAR|INT|DET|AUTHID
---|---|---|------------
SYS
DBMS_DBFS_SFS
REORGANIZEFS
     11424|           52|(null)                                 |PACKAGE      |NO
|NO
(null)
(null)
NO |NO |NO |CURRENT_USER
```

If this query returns a single row similar to the one in this example, the DBFS installation supports Online Filesystem Reorganization. If the query does not return any rows, then the DBFS installation should either be upgraded or requires a patch for bug-10051996.

Invoking Online Filesystem Reorganization

To invoke an Online Filesystem Reorganization, do the following:

1. Create a temporary DBFS filesystem with the desired new organization and structure: including the desired target tablespace (which may be the same

tablespace as the filesystem being reorganized), desired target SecureFiles LOB storage properties (compression, encryption, or deduplication), and so on.

2. Invoke the PL/SQL procedure to reorganize the DBFS filesystem using the newly-created temporary filesystem for data movement.
3. Once the reorganization procedure completes, drop the temporary filesystem.

The example below reorganizes DBFS filesystem `FS1` in tablespace `TS1` into a new tablespace `TS2`, using a temporary filesystem named `TMP_FS`, where all filesystems belong to database user `dbfs_user`:

```
$ cd $ORACLE_HOME/rdbms/admin
$ sqlplus dbfs_user/***

@dbfs_create_filesystem TS2 TMP_FS
EXEC DBMS_DBFS_SFS.REORGANIZEFS('FS1', 'TMP_FS');
@dbfs_drop_filesystem TMP_FS
QUIT;
```

where:

- `TMP_FS` can have any valid name. It is intended as a temporary placeholder and can be dropped (as shown in the example above) or retained as a fully materialized point-in-time snapshot of the original filesystem.
- `FS1` is the original filesystem and is unaffected by the attempted reorganization. It remains usable for all DBFS operations, including SQL, PL/SQL, and `dbfs_client` mounts and commandline, during the reorganization. At the end of the reorganization, FS1 has the new structure and organization used to create `TMP_FS` and vice versa (`TMP_FS` will have the structure and organization originally used for `FS1`). If the reorganization fails for any reason, DBFS attempts to clean up the internal state of `FS1`.
- `TS2` needs enough space to accommodate all active (non-deleted) files and directories in `FS1`.
- `TS1` needs at least twice the amount of space being used by `FS1` if the filesystem is moved within the same tablespace as part of a shrink.

Part III

Application Design with LOBs

This part covers issues that you must consider when designing LOB applications.

This part contains these chapters:

11

LOB Storage with Applications

This chapter describes issues specific to tables that contain LOB columns, with both the SECUREFILE and BASICFILE parameters. If a feature applies to only one of the two kinds of LOB, it is so stated.

This chapter contains these topics:

Creating Tables That Contain LOBs

When creating tables that contain LOBs, use the guidelines described in the following sections:

Initializing Persistent LOBs to NULL or Empty

You can set a persistent LOB — that is, a LOB column in a table, or a LOB attribute in an object type that you defined— to be NULL or empty:

- **Setting a Persistent LOB to NULL:** A LOB set to NULL has no locator. A NULL value is stored in the row in the table, not a locator. This is the same process as for all other data types.
- **Setting a Persistent LOB to Empty:** By contrast, an empty LOB stored in a table is a LOB of zero length that has a locator. So, if you SELECT from an empty LOB column or attribute, then you get back a locator which you can use to populate the LOB with data using supported programmatic environments, such as OCI or PL/SQL(DBMS_LOB). See Chapter 13, "Overview of Supplied LOB APIs" for more information on supported environments.

Details for these options are given in the following discussions.

Setting a Persistent LOB to NULL

You may want to set a persistent LOB value to NULL upon inserting the row in cases where you do not have the LOB data at the time of the INSERT or if you want to use a SELECT statement, such as the following, to determine whether the LOB holds a NULL value:

```
SELECT COUNT (*) FROM print_media WHERE ad_graphic IS NOT NULL;

SELECT COUNT (*) FROM print_media WHERE ad_graphic IS NULL;
```

Note that you cannot call OCI or DBMS_LOB functions on a NULL LOB, so you must then use an SQL UPDATE statement to reset the LOB column to a non-NULL (or empty) value.

The point is that you cannot make a function call from the supported programmatic environments on a LOB that is NULL. These functions only work with a locator, and if the LOB column is NULL, then there is no locator in the row.

Setting a Persistent LOB to Empty

You can initialize a persistent LOB to EMPTY rather that NULL. Doing so, enables you to obtain a locator for the LOB instance without populating the LOB with data. To set a persistent LOB to EMPTY, use the SQL function EMPTY_BLOB() or EMPTY_CLOB() in the INSERT statement:

```
INSERT INTO a_table VALUES (EMPTY_BLOB());
```

As an alternative, you can use the RETURNING clause to obtain the LOB locator in one operation rather than calling a subsequent SELECT statement:

```
DECLARE
   Lob_loc  BLOB;
BEGIN
   INSERT INTO a_table VALUES (EMPTY_BLOB()) RETURNING blob_col INTO Lob_loc;
   /* Now use the locator Lob_loc to populate the BLOB with data */
END;
```

Initializing LOBs

You can initialize the LOBs in print_media by using the following INSERT statement:

```
INSERT INTO print_media VALUES (1001, EMPTY_CLOB(), EMPTY_CLOB(), NULL,
    EMPTY_BLOB(), EMPTY_BLOB(), NULL, NULL, NULL, NULL);
```

This sets the value of ad_sourcetext, ad_fltextn, ad_composite, and ad_photo to an empty value, and sets ad_graphic to NULL.

Initializing Persistent LOB Columns and Attributes to a Value

You can initialize the LOB column or LOB attributes to a value that contains more than 4G bytes of data, the limit before release 10.2.

See Also: Chapter 20, "Data Interface for Persistent LOBs"

Initializing BFILEs to NULL or a File Name

A BFILE can be initialized to NULL or to a filename. To do so, you can use the BFILENAME() function.

See Also: "BFILENAME and Initialization" on page 21-5.

Restriction on First Extent of a LOB Segment

The first extent of any segment requires *at least 2 blocks* (if `FREELIST GROUPS` was 0). That is, the initial extent size of the segment should be at least 2 blocks. LOBs segments are different because they need *at least 3 blocks* in the first extent if the LOB is a BasicFiles LOB and 16 blocks if the LOB is a SecureFiles LOB.

If you try to create a LOB segment in a permanent dictionary managed tablespace with initial = 2 blocks, then it still works because it is possible for segments in permanent dictionary-managed tablespaces to override the default storage setting of the tablespaces.

But if uniform locally managed tablespaces or dictionary managed tablespaces of the temporary type, or locally managed temporary tablespaces have an extent size of 2 blocks, then LOB segments cannot be created in these tablespaces. This is because in these tablespace types, extent sizes are fixed and the default storage setting of the tablespaces is not ignored.

Choosing a LOB Column Data Type

When selecting a data type, consider the following three topics:

LOBs Compared to LONG and LONG RAW Types

Table 11–1 lists the similarities and differences between LOBs, LONGs, and LONG RAW types.

Table 11–1 LOBs Vs. LONG RAW

LOB Data Type	LONG and LONG RAW Data Type
You can store multiple LOBs in a single row	You can store only one `LONG` or `LONG RAW` in each row.
`LOB`s can be attributes of a user-defined data type	This is not possible with either a `LONG` or `LONG RAW`
Only the LOB locator is stored in the table column; `BLOB` and `CLOB` data can be stored in separate tablespaces and `BFILE` data is stored as an external file. For inline LOBs, the database stores LOBs that are less than approximately 4000 bytes of data in the table column.	In the case of a `LONG` or `LONG RAW` the entire value is stored in the table column.
When you access a LOB column, you can choose to fetch the locator or the data.	When you access a `LONG` or `LONG RAW`, the entire value is returned.
A LOB can be up to 128 terabytes or more in size depending on your block size.	A `LONG` or `LONG RAW` instance is limited to 2 gigabytes in size.
There is greater flexibility in manipulating data in a random, piece-wise manner with LOBs. LOBs can be accessed at random offsets.	Less flexibility in manipulating data in a random, piece-wise manner with `LONG or LONG RAW data`.`LONG`s must be accessed from the beginning to the desired location.
You can replicate LOBs in both local and distributed environments.	Replication in both local and distributed environments is not possible with a `LONG` or `LONG RAW` (see *Oracle Database Advanced Replication)*

Storing Varying-Width Character Data in LOBs

Varying-width character data in CLOB and NCLOB data types is stored in an internal format that is compatible with UCS2 Unicode character set format. This ensures that there is no storage loss of character data in a varying-width format. Also note the following if you are using LOBs to store varying-width character data:

- You can create tables containing CLOB and NCLOB columns even if you use a varying-width CHAR or NCHAR database character set.
- You can create a table containing a data type that has a CLOB attribute regardless of whether you use a varying-width CHAR database character set.

Implicit Character Set Conversions with LOBs

For CLOB and NCLOB instances used in OCI (Oracle Call Interface), or any of the programmatic environments that access OCI functionality, character set conversions are implicitly performed when translating from one character set to another.

The DBMS_LOB.LOADCLOBFROMFILE API, performs an implicit conversion from binary data to character data when loading to a CLOB or NCLOB. With the exception of DBMS_LOB.LOADCLOBFROMFILE, LOB APIs do not perform implicit conversions from binary data to character data.

For example, when you use the DBMS_LOB.LOADFROMFILE API to populate a CLOB or NCLOB, you are populating the LOB with binary data from a BFILE. In this case, you must perform character set conversions on the BFILE data before calling DBMS_LOB.LOADFROMFILE.

See Also: *Oracle Database Globalization Support Guide* for more detail on character set conversions.

Note: The database character set cannot be changed from a single-byte to a multibyte character set if there are populated user-defined CLOB columns in the database tables. The national character set cannot be changed between AL16UTF16 and UTF8 if there are populated user-defined NCLOB columns in the database tables.

LOB Storage Parameters

This section summarizes LOB storage characteristics to consider when designing tables with LOB storage. For a discussion of SECUREFILE parameters:

See Also:

- "Using CREATE TABLE with LOB Storage" on page 4-3
- "Using ALTER TABLE with LOB Storage" on page 4-16

Inline and Out-of-Line LOB Storage

LOB columns store locators that reference the location of the actual LOB value. Depending on the column properties you specify when you create the table, and depending the size of the LOB, actual LOB values are stored either in the table row (inline) or outside of the table row (out-of-line).

LOB values are stored out-of-line when any of the following situations apply:

- If you explicitly specify DISABLE STORAGE IN ROW for the LOB storage clause when you create the table.
- If the size of the LOB is greater than approximately 4000 bytes (4000 minus system control information), regardless of the LOB storage properties for the column.
- If you update a LOB that is stored out-of-line and the resulting LOB is less than approximately 4000 bytes, it is still stored out-of-line.

LOB values are stored inline when any of the following conditions apply:

- When the size of the LOB stored in the given row is small, approximately 4000 bytes or less, and you either explicitly specify ENABLE STORAGE IN ROW or the LOB storage clause when you create the table, or when you do not specify this parameter (which is the default).
- When the LOB value is NULL (regardless of the LOB storage properties for the column).

Using the default LOB storage properties (inline storage) can allow for better database performance; it avoids the overhead of creating and managing out-of-line storage for smaller LOB values. If LOB values stored in your database are frequently small in size, then using inline storage is recommended.

Note:

- LOB locators are always stored in the row.
- A LOB locator always exists for any LOB instance regardless of the LOB storage properties or LOB value - NULL, empty, or otherwise.
- If the LOB is created with DISABLE STORAGE IN ROW properties and the BasicFiles LOB holds any data, then a minimum of one CHUNK of out-of-line storage space is used; even when the size of the LOB is less than the CHUNK size.
- If a LOB column is initialized with EMPTY_CLOB() or EMPTY_BLOB(), then no LOB value exists, not even NULL. The row holds a LOB locator only. No additional LOB storage is used.
- LOB storage properties do not affect BFILE columns. BFILE data is always stored in operating system files outside the database.

Defining Tablespace and Storage Characteristics for Persistent LOBs

When defining LOBs in a table, you can explicitly indicate the tablespace and storage characteristics for each *persistent LOB* column.

To create a BasicFiles LOB, the BASICFILE keyword is optional but is recommended for clarity, as shown in the following example:

```
CREATE TABLE ContainsLOB_tab (n NUMBER, c CLOB)
      lob (c) STORE AS BASICFILE segname (TABLESPACE lobtbs1 CHUNK 4096
                        PCTVERSION 5
                        NOCACHE LOGGING
                        STORAGE (MAXEXTENTS 5)
                       );
```

For SecureFiles, the SECUREFILE keyword is necessary, as shown in the following example (assuming TABLESPACE lobtbs1 is ASSM):

```
CREATE TABLE ContainsLOB_tab1 (n NUMBER, c CLOB)
      lob (c) STORE AS SECUREFILE sfsegname (TABLESPACE lobtbs1
                      RETENTION AUTO
                      CACHE LOGGING
                      STORAGE (MAXEXTENTS 5)
                     );
```

Note: There are no tablespace or storage characteristics that you can specify for *external* LOBs (BFILEs) as they are not stored in the database.

If you must modify the LOB storage parameters on an existing LOB column, then use the ALTER TABLE ... MOVE statement. You can change the RETENTION, PCTVERSION, CACHE, NOCACHE LOGGING, NOLOGGING, or STORAGE settings. You can also change the TABLESPACE using the ALTER TABLE ... MOVE statement.

Assigning a LOB Data Segment Name

As shown in the in the previous example, specifying a name for the LOB data segment makes for a much more intuitive working environment. When querying the LOB data dictionary views USER_LOBS, ALL_LOBS, DBA_LOBS (see *Oracle Database Reference*), you see the LOB data segment that you chose instead of system-generated names.

LOB Storage Characteristics for LOB Column or Attribute

LOB storage characteristics that can be specified for a LOB column or a LOB attribute include the following:

- TABLESPACE
- PCTVERSION or RETENTION

  Note that you can specify either PCTVERSION or RETENTION for BasicFiles LOBs, but not both. For SecureFiles, only the RETENTION parameter can be specified.
- CACHE/NOCACHE/CACHE READS
- LOGGING/NOLOGGING
- CHUNK
- ENABLE/DISABLE STORAGE IN ROW
- STORAGE

For most users, defaults for these storage characteristics are sufficient. If you want to fine-tune LOB storage, then consider the following guidelines.

See Also:

- STORAGE clause in *Oracle Database SQL Language Reference*
- RETENTION parameter in *Oracle Database SQL Language Reference*

TABLESPACE and LOB Index

The LOB index is an internal structure that is strongly associated with LOB storage. This implies that a user may not drop the LOB index and rebuild it.

Note: The LOB index cannot be altered.

The system determines which tablespace to use for LOB data and LOB index depending on your specification in the LOB storage clause:

- If you do *not* specify a tablespace for the LOB data, then the tablespace of the table is used for the LOB data and index.
- If you specify a tablespace for the LOB data, then both the LOB data and index use the tablespace that was specified.

Tablespace for LOB Index in Non-Partitioned Table

When creating a table, if you specify a tablespace for the LOB index for a non-partitioned table, then your specification of the tablespace is ignored and the LOB index is co-located with the LOB data. Partitioned LOBs do not include the LOB index syntax.

Specifying a separate tablespace for the LOB storage segments enables a decrease in contention on the tablespace of the table.

PCTVERSION

When a BasicFiles LOB is modified, a new version of the BasicFiles LOB page is produced in order to support consistent read of prior versions of the BasicFiles LOB value.

`PCTVERSION` is the percentage of all used BasicFiles LOB data space that can be occupied by old versions of BasicFiles LOB data pages. As soon as old versions of BasicFiles LOB data pages start to occupy more than the `PCTVERSION` amount of used BasicFiles LOB space, Oracle Database tries to reclaim the old versions and reuse them. In other words, `PCTVERSION` is the percent of used BasicFiles LOB data blocks that is available for versioning old BasicFiles LOB data.

PCTVERSION has a default of 10 (%), a minimum of 0, and a maximum of 100.

To decide what value `PCTVERSION` should be set to, consider the following:

- How often BasicFiles LOBs are updated?
- How often the updated BasicFiles LOBs are read?

Table 11–2, " Recommended PCTVERSION Settings" provides some guidelines for determining a suitable `PCTVERSION` value given an update percentage of 'X'.

Table 11–2 Recommended PCTVERSION Settings

BasicFiles LOB Update Pattern	BasicFiles LOB Read Pattern	PCTVERSION
Updates X% of LOB data	Reads updated LOBs	X%
Updates X% of LOB data	Reads LOBs but not the updated LOBs	0%
Updates X% of LOB data	Reads both updated and non-updated LOBs	2X%
Never updates LOB	Reads LOBs	0%

If your application requires several BasicFiles LOB updates concurrent with heavy reads of BasicFiles LOB columns, then consider using a higher value for `PCTVERSION`, such as 20%.

Setting `PCTVERSION` to twice the default value allows more free pages to be used for old versions of data pages. Because large queries may require consistent reads of BasicFiles LOB columns, it may be useful to retain old versions of BasicFiles LOB pages. In this

case, BasicFiles LOB storage may grow because the database does not reuse free pages aggressively.

If persistent BasicFiles LOB instances in your application are created and written just once and are primarily read-only afterward, then updates are infrequent. In this case, consider using a lower value for PCTVERSION, such as 5% or lower.

The more infrequent and smaller the BasicFiles LOB updates are, the less space must be reserved for old copies of BasicFiles LOB data. If existing BasicFiles LOBs are known to be read-only, then you could safely set PCTVERSION to 0% because there would never be any pages needed for old versions of data.

RETENTION Parameter for BasicFiles LOBs

As an alternative to the PCTVERSION parameter, you can specify the RETENTION parameter in the LOB storage clause of the CREATE TABLE or ALTER TABLE statement. Doing so, configures the LOB column to store old versions of LOB data for a *period of time*, rather than using a percentage of the table space. For example:

```
CREATE TABLE ContainsLOB_tab (n NUMBER, c CLOB)
      lob (c) STORE AS BASICFILE segname (TABLESPACE lobtbs1 CHUNK 4096
                        RETENTION
                        NOCACHE LOGGING
                        STORAGE (MAXEXTENTS 5)
                       );
```

The RETENTION parameter is designed for use with UNDO features of the database, such as Flashback Versions Query. When a LOB column has the RETENTION property set, old versions of the LOB data are retained for the amount of time specified by the UNDO_RETENTION parameter.

Note the following with respect to the RETENTION parameter:

- UNDO SQL is not enabled for LOB columns as it is with other data types. You must set the RETENTION property on a LOB column to use Undo SQL on LOB data.
- You cannot set the value of the RETENTION parameter explicitly. The amount of time for retention of LOB versions in determined by the UNDO_RETENTION parameter.
- Usage of the RETENTION parameter is only supported in Automatic Undo Management mode. You must configure your table for use with Automatic Undo Management before you can set RETENTION on a LOB column. ASSM is required for LOB RETENTION to be in effect for BasicFiles LOBs. The RETENTION parameter of the SQL (in the STORE AS clause) is silently ignored if the BasicFiles LOB resides in an MSSM tablespace.
- The LOB storage clause can specify RETENTION or PCTVERSION, but not both.

See Also:

- *Oracle Database Development Guide* for more information on using flashback features of the database.
- *Oracle Database SQL Language Reference* for details on LOB storage clause syntax.

RETENTION Parameter for SecureFiles LOBs

Specifying the RETENTION parameter for SecureFiles indicates that the database manages consistent read data for the SecureFiles storage dynamically, taking into account factors such as the UNDO mode of the database.

- Specify MAX if the database is in FLASHBACK mode to limit the size of the LOB UNDO retention in bytes. If you specify MAX, then you must also specify the MAXSIZE clause in the storage_clause.
- Specify AUTO if you want to retain UNDO sufficient for consistent read purposes only. This is the default.
- Specify NONE if no UNDO is required for either consistent read or flashback purposes.

The default RETENTION for SecureFiles is AUTO.

CACHE / NOCACHE / CACHE READS

When creating tables that contain LOBs, use the cache options according to the guidelines in Table 11–3, " When to Use CACHE, NOCACHE, and CACHE READS":

Table 11–3 When to Use CACHE, NOCACHE, and CACHE READS

Cache Mode	Read	Write
CACHE READS	Frequently	Once or occasionally
CACHE	Frequently	Frequently
NOCACHE (default)	Once or occasionally	Never

CACHE / NOCACHE / CACHE READS: LOB Values and Buffer Cache

- CACHE: Oracle places LOB pages in the buffer cache for faster access.
- NOCACHE: As a parameter in the STORE AS clause, NOCACHE specifies that LOB values are not brought into the buffer cache.
- CACHE READS: LOB values are brought into the buffer cache only during read and not during write operations.

NOCACHE is the default for both SecureFiles and BasicFiles LOBs.

> **Note:** Using the CACHE option results in improved performance when reading and writing data from the LOB column. However, it can potentially age other non-LOB pages out of the buffer cache prematurely.

LOGGING / NOLOGGING Parameter for BasicFiles LOBs

[NO]LOGGING has a similar application with regard to using LOBs as it does for other table operations. In the usual case, if the [NO]LOGGING clause is omitted, then this means that neither NOLOGGING nor LOGGING is specified and the logging attribute of the table or table partition defaults to the logging attribute of the tablespace in which it resides.

For LOBs, there is a further alternative depending on how CACHE is stipulated.

- **CACHE is specified** and [NO]LOGGING clause is omitted. LOGGING is automatically implemented (because you cannot have CACHE NOLOGGING).
- **CACHE is not specified** and [NO]LOGGING clause is omitted. The process defaults in the same way as it does for tables and partitioned tables. That is, the

[NO]LOGGING value is obtained from the tablespace in which the LOB segment resides.

The following issues should also be kept in mind.

LOBs Always Generate Undo for LOB Index Pages

Regardless of whether LOGGING or NOLOGGING is set, LOBs never generate rollback information (undo) for LOB data pages because old LOB data is stored in versions. Rollback information that is created for LOBs tends to be small because it is only for the LOB index page changes.

When LOGGING is Set Oracle Generates Full Redo for LOB Data Pages

NOLOGGING is intended to be used when a customer does not care about media recovery. Thus, if the disk/tape/storage media fails, then you cannot recover your changes from the log because the changes were never logged.

NOLOGGING is Useful for Bulk Loads or Inserts. For instance, when loading data into the LOB, if you do not care about redo and can just start the load over if it fails, set the LOB data segment storage characteristics to NOCACHE NOLOGGING. This provides good performance for the initial load of data.

Once you have completed loading data, if necessary, use ALTER TABLE to modify the LOB storage characteristics for the LOB data segment for normal LOB operations, for example, to CACHE or NOCACHE LOGGING.

Note: CACHE implies that you also get LOGGING.

LOGGING/FILESYSTEM_LIKE_LOGGING for SecureFiles LOBs

NOLOGGING or LOGGING has a similar application with regard to using SecureFiles as LOGGING/NOLOGGING does for other table operations. In the usual case, if the logging_clause is omitted, then the SecureFiles inherits its logging attribute from the tablespace in which it resides. In this case, if NOLOGGING is the default value, the SecureFiles defaults to FILESYSTEM_LIKE_LOGGING.

Note: Using the CACHE option results in improved performance when reading and writing data from the LOB column. However, it can potentially age other non-LOB pages out of the buffer cache prematurely.

CACHE Implies LOGGING

For SecureFiles, there is a further alternative depending on how CACHE is specified:

- CACHE is specified and the LOGGING clause is omitted, then LOGGING is used.
- CACHE is not specified and the logging_clause is omitted. Then the process defaults in the same way as it does for tables and partitioned tables. That is, the LOGGING value is obtained from the tablespace in which the LOB value resides. If the tablespace is NOLOGGING, then the SecureFiles defaults to FILESYSTEM_LIKE_LOGGING.

The following issues should also be kept in mind.

SecureFiles and an Efficient Method of Generating REDO and UNDO

This means that Oracle Database determines if it is more efficient to generate REDO and UNDO for the change to a block, similar to heap blocks, or if it generates a version and full REDO of the new block similar to BasicFiles LOBs.

FILESYSTEM_LIKE_LOGGING is Useful for Bulk Loads or Inserts

For instance, when loading data into the LOB, if you do not care about REDO and can just start the load over if it fails, set the LOB data segment storage characteristics to FILESYSTEM_LIKE_LOGGING. This provides good performance for the initial load of data.

Once you have completed loading data, if necessary, use ALTER TABLE to modify the LOB storage characteristics for the LOB data segment for normal LOB operations. For example, to CACHE or NOCACHE LOGGING.

CHUNK

A chunk is one or more Oracle blocks. You can specify the chunk size for the BasicFiles LOB when creating the table that contains the LOB. This corresponds to the data size used by Oracle Database when accessing or modifying the LOB value. Part of the chunk is used to store system-related information and the rest stores the LOB value. The API you are using has a function that returns the amount of space used in the LOB chunk to store the LOB value. In PL/SQL use DBMS_LOB.GETCHUNKSIZE. In OCI, use OCILobGetChunkSize().

Note: If the tablespace block size is the same as the database block size, then CHUNK is also a multiple of the database block size. The default CHUNK size is equal to the size of one tablespace block, and the maximum value is 32K.

See Also: "Terabyte-Size LOB Support" on page 12-21 for information about maximum LOB sizes

Choosing the Value of CHUNK

Once the value of CHUNK is chosen (when the LOB column is created), it cannot be changed. Hence, it is important that you choose a value which optimizes your storage and performance requirements. For SecureFiles, CHUNK is an advisory size and is provided for backward compatibility purposes.

Space Considerations The value of CHUNK does not matter for LOBs that are stored inline. This happens when ENABLE STORAGE IN ROW is set, and the size of the LOB locator and the LOB data is less than approximately 4000 bytes. However, when the LOB data is stored out-of-line, it always takes up space in multiples of the CHUNK parameter. This can lead to a large waste of space if your data is small, but the CHUNK is set to a large number. Table 11–4, " Data Size and CHUNK Size" illustrates this point:

Table 11–4 Data Size and CHUNK Size

Data Size	CHUNK Size	Disk Space Used to Store the LOB	Space Utilization (Percent)
3500 enable storage in row	irrelevant	3500 in row	100
3500 disable storage in row	32 KB	32 KB	10
3500 disable storage in row	4 KB	4 KB	90
33 KB	32 KB	64 KB	51
2 GB +10	32 KB	2 GB + 32 KB	99+

Performance Considerations Accessing lobs in big chunks is more efficient. You can set `CHUNK` to the data size most frequently accessed or written. For example, if only one block of LOB data is accessed at a time, then set `CHUNK` to the size of one block. If you have big LOBs, and read or write big amounts of data, then choose a large value for `CHUNK`.

Set INITIAL and NEXT to Larger than CHUNK

If you explicitly specify storage characteristics for the LOB, then make sure that `INITIAL` and `NEXT` for the LOB data segment storage are set to a size that is larger than the `CHUNK` size. For example, if the database block size is 2KB and you specify a `CHUNK` of 8KB, then make sure that `INITIAL` and `NEXT` are bigger than 8KB and preferably considerably bigger (for example, at least 16KB).

Put another way: If you specify a value for `INITIAL`, `NEXT`, or the LOB `CHUNK` size, then make sure they are set in the following manner:

- `CHUNK <= NEXT`
- `CHUNK <= INITIAL`

ENABLE or DISABLE STORAGE IN ROW Clause

You use the `ENABLE | DISABLE STORAGE IN ROW` clause to indicate whether the LOB should be stored inline (in the row) or out-of-line.

> **Note:** You may not alter this specification once you have made it: if you `ENABLE STORAGE IN ROW`, then you cannot alter it to `DISABLE STORAGE IN ROW` and vice versa.

The default is `ENABLE STORAGE IN ROW`.

Guidelines for ENABLE or DISABLE STORAGE IN ROW

The maximum amount of LOB data stored in the row is the maximum `VARCHAR2` size (4000). This includes the control information and the LOB value. If you indicate that the LOB should be stored in the row, once the LOB value and control information is larger than approximately 4000, then the LOB value is automatically moved out of the row.

This suggests the following guidelines:

The default, ENABLE STORAGE IN ROW, is usually the best choice for the following reasons:

- **Small LOBs:** If the LOB is small (less than approximately 4000 bytes), then the whole LOB can be read while reading the row without extra disk I/O.
- **Large LOBs:** If the LOB is big (greater than approximately 4000 bytes), then the control information is still stored in the row if ENABLE STORAGE IN ROW is set, even after moving the LOB data out of the row. This control information could enable us to read the out-of-line LOB data faster.

However, in some cases DISABLE STORAGE IN ROW is a better choice. This is because storing the LOB in the row increases the size of the row. This impacts performance if you are doing a lot of base table processing, such as full table scans, multi-row accesses (range scans), or many UPDATE/SELECT to columns other than the LOB columns.

Indexing LOB Columns

This section discusses different techniques you can use to index LOB columns.

Note: After you move a LOB column any existing table indexes must be rebuilt.

Using Domain Indexing on LOB Columns

You might be able to improve the performance of queries by building indexes specifically attuned to your domain. Extensibility interfaces provided with the database allow for domain indexing, a framework for implementing such domain specific indexes.

Note: You cannot build a B-tree or bitmap index on a LOB column.

See Also: *Oracle Database Data Cartridge Developer's Guide* for information on building domain specific indexes.

Indexing LOB Columns Using a Text Index

Depending on the nature of the contents of the LOB column, one of the Oracle Text options could also be used for building indexes. For example, if a text document is stored in a `CLOB` column, then you can build a text index to speed up the performance of text-based queries over the `CLOB` column.

See Also: *Oracle Text Application Developer's Guide* for an example of using a `CLOB` column to store text data

Function-Based Indexes on LOBs

A function-based index is an index built on an expression. It extends your indexing capabilities beyond indexing on a column. A function-based index increases the variety of ways in which you can access data.

Function-based indexes cannot be built on nested tables or LOB columns. However, you can build function-based indexes on VARRAYs.

Like extensible indexes and domain indexes on LOB columns, function-based indexes are also automatically updated when a DML operation is performed on the LOB

column. Function-based indexes are also updated when any extensible index is updated.

> **See Also:** *Oracle Database Development Guide* for more information on using function-based indexes.

Extensible Indexing on LOB Columns

The database provides *extensible indexing*, a feature which enables you to define new index types as required. This is based on the concept of cooperative indexing where a data cartridge and the database build and maintain indexes for data types such as text and spatial for example, for On-line-Analytical Processing (OLAP).

The cartridge is responsible for defining the index structure, maintaining the index content during load and update operations, and searching the index during query processing. The index structure can be stored in Oracle as heap-organized, or an index-organized table, or externally as an operating system file.

To support this structure, the database provides an *indextype*. The purpose of an indextype is to enable efficient search and retrieval functions for complex domains such as text, spatial, image, and OLAP by means of a data cartridge. An indextype is analogous to the sorted or bit-mapped index types that are built-in within the Oracle Server. The difference is that an indextype is implemented by the data cartridge developer, whereas the Oracle kernel implements built-in indexes. Once a new indextype has been implemented by a data cartridge developer, end users of the data cartridge can use it just as they would built-in indextypes.

When the database system handles the physical storage of domain indexes, data cartridges

- Define the format and content of an index. This enables cartridges to define an index structure that can accommodate a complex data object.
- Build, delete, and update a domain index. The cartridge handles building and maintaining the index structures. Note that this is a significant departure from the medicine indexing features provided for simple SQL data types. Also, because an index is modeled as a collection of tuples, in-place updating is directly supported.
- Access and interpret the content of an index. This capability enables the data cartridge to become an integral component of query processing. That is, the content-related clauses for database queries are handled by the data cartridge.

By supporting extensible indexes, the database significantly reduces the effort needed to develop high-performance solutions that access complex data types such as LOBs.

Extensible Optimizer

The extensible optimizer functionality allows authors of user-defined functions and indexes to create statistics collections, selectivity, and cost functions. This information is used by the optimizer in choosing a query plan. The cost-based optimizer is thus extended to use the user-supplied information.

Extensible indexing functionality enables you to define new operators, index types, and domain indexes. For such user-defined operators and domain indexes, the extensible optimizer functionality allows users to control the three main components used by the optimizer to select an execution plan: *statistics*, *selectivity*, and *cost*.

> **See Also:** *Oracle Database Data Cartridge Developer's Guide*

Oracle Text Indexing Support for XML

You can create Oracle Text indexes on `CLOB` columns and perform queries on XML data.

See Also:

- *Oracle XML Developer's Kit Programmer's Guide*
- *Oracle Text Reference*
- *Oracle Text Application Developer's Guide*

Manipulating LOBs in Partitioned Tables

You can partition tables that contain LOB columns. As a result, LOBs can take advantage of all of the benefits of partitioning including the following:

- LOB segments can be spread between several tablespaces to balance I/O load and to make backup and recovery more manageable.
- LOBs in a partitioned table become easier to maintain.
- LOBs can be partitioned into logical groups to speed up operations on LOBs that are accessed as a group.

This section describes some of the ways you can manipulate LOBs in partitioned tables.

Partitioning a Table Containing LOB Columns

LOBs are supported in RANGE partitioned, LIST partitioned, and HASH partitioned tables. Composite heap-organized tables can also have LOBs.

You can partition a table containing LOB columns using the following techniques:

- When the table is created using the `PARTITION BY ...` clause of the `CREATE TABLE` statement.
- Adding a partition to an existing table using the `ALTER TABLE ... ADD PARTITION` clause.
- Exchanging partitions with a table that has partitioned LOB columns using the `ALTER TABLE ... EXCHANGE PARTITION` clause. Note that `EXCHANGE PARTITION` can only be used when both tables have the same storage attributes, for example, both tables store LOBs out-of-line.

Creating LOB partitions at the same time you create the table (in the `CREATE TABLE` statement) is recommended. If you create partitions on a LOB column when the table is created, then the column can hold LOBs stored either inline or out-of-line LOBs.

After a table is created, new LOB partitions can only be created on LOB columns that are stored out-of-line. Also, partition maintenance operations, `SPLIT PARTITION` and `MERGE PARTITIONS`, only work on LOB columns that store LOBs out-of-line.

See Also: "Restrictions for LOBs in Partitioned Index-Organized Tables" on page 11-17 for additional information on LOB restrictions.

Note that once a table is created, storage attributes cannot be changed. See "LOB Storage Parameters" on page 11-4 for more information about LOB storage attributes.

Creating an Index on a Table Containing Partitioned LOB Columns

To improve the performance of queries, you can create indexes on partitioned LOB columns. For example:

```
CREATE INDEX index_name
   ON table_name (LOB_column_1, LOB_column_2, ...) LOCAL;
```

Note that only domain and function-based indexes are supported on LOB columns. Other types of indexes, such as unique indexes are not supported with LOBs.

Moving Partitions Containing LOBs

You can move a LOB partition into a different tablespace. This is useful if the tablespace is no longer large enough to hold the partition. To do so, use the `ALTER TABLE ... MOVE PARTITION` clause. For example:

```
ALTER TABLE current_table MOVE PARTITION partition_name
   TABLESPACE destination_table_space
   LOB (column_name) STORE AS (TABLESPACE current_tablespace);
```

Splitting Partitions Containing LOBs

You can split a partition containing LOBs into two equally sized partitions using the `ALTER TABLE ... SPLIT PARTITION` clause. Doing so permits you to place one or both new partitions in a new tablespace. For example:

```
ALTER TABLE table_name SPLIT PARTITION partition_name
   AT (partition_range_upper_bound)
   INTO (PARTITION partition_name,
      PARTITION new_partition_name TABLESPACE new_tablespace_name
         LOB (column_name) STORE AS (TABLESPACE tablespace_name)
         ... ;
```

Merging Partitions Containing LOBs

You can merge partitions that contain LOB columns using the `ALTER TABLE ... MERGE PARTITIONS` clause. This technique is useful for reclaiming unused partition space. For example:

```
ALTER TABLE table_name
   MERGE PARTITIONS partition_1, partition_2
   INTO PARTITION new_partition TABLESPACE new_tablespace_name
      LOB (column_name) store as (TABLESPACE tablespace_name)
     ... ;
```

LOBs in Index Organized Tables

Index Organized Tables (IOTs) support internal and external LOB columns. For the most part, SQL DDL, DML, and piece wise operations on LOBs in IOTs produce the same results as those for normal tables. The only exception is the default semantics of LOBs during creation. The main differences are:

- ***Tablespace Mapping***: By default, or unless specified otherwise, the LOB data and index segments are created in the tablespace in which the primary key index segments of the index organized table are created.
- ***Inline as Compared to Out-of-Line Storage***: By default, all LOBs in an index organized table created without an overflow segment are stored out of line. In other words, if an index organized table is created without an overflow segment,

then the LOBs in this table have their default storage attributes as DISABLE STORAGE IN ROW. If you forcibly try to specify an ENABLE STORAGE IN ROW clause for such LOBs, then SQL raises an error.

On the other hand, if an overflow segment has been specified, then LOBs in index organized tables exactly mimic their semantics in conventional tables (see "Defining Tablespace and Storage Characteristics for Persistent LOBs" on page 11-5).

Example of Index Organized Table (IOT) with LOB Columns

Consider the following example:

```
CREATE TABLE iotlob_tab (c1 INTEGER PRIMARY KEY, c2 BLOB, c3 CLOB, c4
VARCHAR2(20))
  ORGANIZATION INDEX
    TABLESPACE iot_ts
    PCTFREE 10 PCTUSED 10 INITRANS 1 MAXTRANS 1 STORAGE (INITIAL 4K)
    PCTTHRESHOLD 50 INCLUDING c2
  OVERFLOW
    TABLESPACE ioto_ts
    PCTFREE 10 PCTUSED 10 INITRANS 1 MAXTRANS 1 STORAGE (INITIAL 8K) LOB (c2)
    STORE AS lobseg (TABLESPACE lob_ts DISABLE STORAGE IN ROW
                     CHUNK 16384 PCTVERSION 10 CACHE STORAGE (INITIAL 2M)
                     INDEX lobidx_c1 (TABLESPACE lobidx_ts STORAGE (INITIAL 4K)));
```

Executing these statements results in the creation of an index organized table iotlob_tab with the following elements:

- A primary key index segment in the tablespace iot_ts,
- An overflow data segment in tablespace ioto_ts
- Columns starting from column C3 being explicitly stored in the overflow data segment
- BLOB (column C2) data segments in the tablespace lob_ts
- BLOB (column C2) index segments in the tablespace lobidx_ts
- CLOB (column C3) data segments in the tablespace iot_ts
- CLOB (column C3) index segments in the tablespace iot_ts
- CLOB (column C3) stored in line by virtue of the IOT having an overflow segment
- BLOB (column C2) explicitly forced to be stored out of line

Note: If no overflow had been specified, then both C2 and C3 would have been stored out of line by default.

Other LOB features, such as BFILEs and varying character width LOBs, are also supported in index organized tables, and their usage is the same as for conventional tables.

Restrictions for LOBs in Partitioned Index-Organized Tables

LOB columns are supported in range-, list-, and hash-partitioned index-organized tables with the following restrictions:

- Composite partitioned index-organized tables are not supported.

- Relational and object partitioned index-organized tables (partitioned by range, hash, or list) can hold LOBs stored as follows; however, partition maintenance operations, such as `MOVE`, `SPLIT`, and `MERGE` are not supported with:
  - VARRAY data types stored as LOB data types
  - Abstract data types with LOB attributes
  - Nested tables with LOB types

  See Also: Additional restrictions for LOB columns in general are given in "LOB Rules and Restrictions" on page 2-6.

Updating LOBs in Nested Tables

To update LOBs in a nested table, you must lock the row containing the LOB explicitly. To do so, you must specify the FOR UPDATE clause in the subquery prior to updating the LOB value.

Note that locking the row of a parent table does not lock the row of a nested table containing LOB columns.

Note: Nested tables containing LOB columns are the only data structures supported for creating collections of LOBs. You cannot create a `VARRAY` of any LOB data type.

12

Advanced Design Considerations

This chapter describes design considerations for more advanced application development issues.

This chapter contains these topics:

LOB Buffering Subsystem

The database provides a LOB buffering subsystem (LBS) for advanced OCI-based applications such as Data Cartridges, Web servers, and other client-based applications that must buffer the contents of one or more LOBs in the client address space. The client-side memory requirement for the buffering subsystem during its maximum usage is `512` KBytes. It is also the maximum amount that you can specify for a single read or write operation on a LOB that has been enabled for buffered access.

Advantages of LOB Buffering

The advantages of buffering, especially for client applications that perform a series of small reads and writes (often repeatedly) to specific regions of the LOB, are:

- Buffering enables deferred writes to the server. You can buffer up several writes in the LOB buffer in the client address space and eventually *flush* the buffer to the server. This reduces the number of network round-trips from your client application to the server, and hence, makes for better overall performance for LOB updates.
- Buffering reduces the overall number of LOB updates on the server, thereby reducing the number of LOB versions and amount of logging. This results in better overall LOB performance and disk space usage.

Guidelines for Using LOB Buffering

The following caveats apply to buffered LOB operations:

- Explicitly flush LOB buffer contents:

  The LOB buffering subsystem is not a cache. The contents of a LOB buffer are not always the same as the LOB value in the server. Unless you explicitly flush the contents of a LOB buffer, you do not see the results of your buffered writes reflected in the actual LOB on the server.

- Error recovery for buffered LOB operations is your responsibility:

  Owing to the deferred nature of the actual LOB update, error reporting for a particular buffered read or write operation is deferred until the next access to the server based LOB.

- LOB Buffering is Single User, Single Threaded:

  Transactions involving buffered LOB operations cannot migrate across user sessions — the LBS is a single user, single threaded system.

- Maintain logical savepoints to rollback to:

  Oracle does not guarantee transactional support for buffered LOB operations. To ensure transactional *semantics* for buffered LOB updates, you must maintain logical savepoints in your application to rollback all the changes made to the buffered LOB in the event of an error. You should always wrap your buffered LOB updates within a logical savepoint (see "OCI Example of LOB Buffering" on page 12-7).

- Ensure LOB is not updated by another bypassing transaction:

  In any given transaction, once you have begun updating a LOB using buffered writes, it is your responsibility to ensure that the same LOB is not updated through any other operation within the scope of the same transaction *that bypasses the buffering subsystem*.

  You could potentially do this by using an SQL statement to update the server-based LOB. Oracle cannot distinguish, and hence prevent, such an operation. This seriously affects the correctness and integrity of your application.

- Updating buffer-enabled LOB locators:

  Buffered operations on a LOB are done through its locator, just as in the conventional case. A locator that is enabled for buffering provides a consistent read version of the LOB, until you perform a write operation on the LOB through that locator. See also, "Read-Consistent Locators" on page 12-10.

  Once the locator becomes an updated locator by virtue of its being used for a buffered write, it always provides access to the most up-to-date version of the LOB *as seen through the buffering subsystem.* Buffering also imposes an additional significance to this updated locator; all further buffered writes to the LOB can be done *only through this updated locator*. Oracle returns an error if you attempt to write to the LOB through other locators enabled for buffering. See also, "Example of Updating LOBs Through Updated Locators" on page 12-12.

- Passing a buffer-enabled LOB locator an IN OUT or OUT parameter:

  You can pass an updated locator that was enabled for buffering as an `IN` parameter to a PL/SQL procedure. However, passing an `IN OUT` or an `OUT` parameter, or an attempt to return an updated locator, produces an error.

- You cannot assign an updated locator that was enabled for buffering to another locator:

  There are different ways that assignment of locators may occur: through `OCILobAssign()`, through assignment of PL/SQL variables, through

`OCIObjectCopy()` where the object contains the LOB attribute, and so on. Assigning a consistent read locator that was enabled for buffering to a locator that did not have buffering enabled, turns buffering on for the target locator. By the same token, assigning a locator that was not enabled for buffering to a locator that did have buffering enabled, turns buffering off for the target locator.

Similarly, if you `SELECT` into a locator for which buffering was originally enabled, then the locator becomes overwritten with the new locator value, thereby turning buffering off.

- When two or more locators point to the same LOB do not enable both for buffering:

  If two or more different locators point to the same LOB, then it is your responsibility to make sure that you do not enable both the locators for buffering. Otherwise Oracle does not guarantee the contents of the LOB.

- Buffer-enable LOBs do not support appends that create zero-byte fillers or spaces:

  Appending to the LOB value using buffered write(s) is allowed, but only if the starting offset of these write(s) is exactly one byte (or character) past the end of the `BLOB` (or `CLOB/NCLOB`). In other words, the buffering subsystem does not support appends that involve creation of zero-byte fillers or spaces in the server based LOB.

- For CLOBs, Oracle requires the client side character set form for the locator bind variable be the same as that of the LOB in the server:

  This is usually the case in most OCI LOB programs. The exception is when the locator is selected from a *remote* database, which may have a different character set form from the database which is currently being accessed by the `OCI` program. In such a case, an error is returned. If there is no character set form input by the user, then Oracle assumes it is `SQLCS_IMPLICIT`.

LOB Buffering Subsystem Usage

Here are some details of the LOB buffering subsystem:

LOB Buffer Physical Structure

Each user session has the following structure:

- Fixed page pool of 16 pages, shared by all LOBs accessed in buffering mode from that session.
- Each page has a fixed size of up to 32K bytes (not characters) where page size = n x `CHUNK` ~= 32K.

A LOB buffer consists of one or more of these pages, up to a maximum of 16 in each session. The maximum amount that you ought to specify for any given buffered read or write operation is 512K bytes, remembering that under different circumstances the maximum amount you may read/write could be smaller.

LOB Buffering Subsystem Usage Scenario

Consider that a LOB is divided into fixed-size, logical regions. Each page is mapped to one of these fixed size regions, and is in essence, their in-memory copy. Depending on the input `offset` and `amount` specified for a read or write operation, the database allocates one or more of the free pages in the page pool to the LOB buffer. A *free page* is one that has not been read or written by a buffered read or write operation.

For example, assuming a page size of 32KBytes:

- For an input offset of 1000 and a specified read/write amount of 30000, Oracle reads the first 32K byte region of the LOB into a page in the LOB buffer.
- For an input offset of 33000 and a read/write amount of 30000, the second 32K region of the LOB is read into a page.
- For an input offset of 1000, and a read/write amount of 35000, the LOB buffer contains two pages — the first mapped to the region 1 — 32K, and the second to the region 32K+1 — 64K of the LOB.

This mapping between a page and the LOB region is temporary until Oracle maps another region to the page. When you attempt to access a region of the LOB that is not available in full in the LOB buffer, Oracle allocates any available free page(s) from the page pool to the LOB buffer. If there are no free pages available in the page pool, then Oracle reallocates the pages as follows. It ages out the *least recently used* page among the *unmodified* pages in the LOB buffer and reallocates it for the current operation.

If no such page is available in the LOB buffer, then it ages out the least recently used page among the *unmodified* pages of *other* buffered LOBs in the same session. Again, if no such page is available, then it implies that all the pages in the page pool are *modified*, and either the currently accessed LOB, or one of the other LOBs, must be flushed. Oracle notifies this condition to the user as an error. Oracle *never* flushes and reallocates a modified page implicitly. You can either flush them explicitly, or discard them by disabling buffering on the LOB.

To illustrate the preceding discussion, consider two LOBs being accessed in buffered mode — L1 and L2, each with buffers of size 8 pages. Assume that 6 of the 8 pages in the L1 buffer are dirty, with the remaining 2 containing unmodified data read in from the server. Assume similar conditions in the `L2` buffer. Now, for the next buffered operation on `L1`, Oracle reallocates the least recently used page from the two unmodified pages in the `L1` buffer. Once all the 8 pages in the L1 buffer are used up for LOB writes, Oracle can service two more operations on L1 by allocating the two unmodified pages from the `L2` buffer using the least recently used policy. But for any further buffered operations on `L1` or `L2`, Oracle returns an error.

If all the buffers are dirty and you attempt another read from or write to a buffered LOB, then you receive the following error:

```
Error 22280: no more buffers available for operation
```

There are two possible causes:

1. All buffers in the buffer pool have been used up by previous operations.

   In this case, flush the LOBs through the locator that is being used to update the LOB.

2. You are trying to flush a LOB without any previous buffered update operations.

   In this case, write to the LOB through a locator enabled for buffering before attempting to flush buffers.

Flushing the LOB Buffer

The term flush refers to a set of processes. Writing data to the LOB in the buffer through the locator transforms the locator into an updated locator. After you have updated the LOB data in the buffer through the updated locator, a flush call performs the following actions:

- Writes the dirty pages in the LOB buffer to the server-based LOB, thereby updating the LOB value.

- Resets the updated locator to be a read-consistent locator.
- Frees the flushed buffers or turns the status of the buffer pages back from dirty to unmodified.

After the flush, the locator becomes a read-consistent locator and can be assigned to another locator (`L2 := L1`).

For instance, suppose you have two locators, `L1` and `L2`. Let us say that they are both read-consistent locators and consistent with the state of the LOB data in the server. If you then update the LOB by writing to the buffer, `L1` becomes an updated locator. `L1` and `L2` now refer to different versions of the LOB value. If you want to update the LOB in the server, then you must use `L1` to retain the read-consistent state captured in `L2`. The flush operation writes a new snapshot environment into the locator used for the flush. The important point to remember is that you must use the updated locator (`L1`), when you flush the LOB buffer. Trying to flush a read-consistent locator generates an error.

The technique you use to flush the LOB buffer determines whether data in the buffer is cleared and has performance implications as follows:

- In the default mode, data is retained in the pages that were modified when the flush operation occurs. In this case, when you read or write to the same range of bytes, no round-trip to the server is necessary. Note that flushing the buffer, in this context, does not clear the data in the buffer. It also does not return the memory occupied by the flushed buffer to the client address space.

  Note: Unmodified pages may now be aged out if necessary.

- In the second case, you set the flag parameter in `OCILobFlushBuffer()` to `OCI_LOB_BUFFER_FREE` to free the buffer pages, and so return the memory to the client address space. Flushing the buffer using this technique updates the LOB value on the server, returns a read-consistent locator, and frees the buffer pages.

Flushing the Updated LOB

It is very important to note that you must flush a LOB that has been updated through the LOB buffering subsystem in the following situations:

- Before committing the transaction
- Before migrating from the current transaction to another
- Before disabling buffering operations on a LOB
- Before returning from an external callout execution into the calling function, procedure, or method in PL/SQL

  Note that when the external callout is called from a PL/SQL block and the locator is passed as a parameter, all buffering operations, including the enable call, should be made within the callout itself. In other words, adhere to the following sequence:

  - Call the external callout
  - Enable the locator for buffering
  - Read or write using the locator
  - Flush the LOB
  - Disable the locator for buffering

- Return to the calling function, procedure, or method in PL/SQL

Remember that the database never implicitly flushes the LOB buffer.

Using Buffer-Enabled Locators

Note that there are several cases in which you can use buffer-enabled locators and others in which you cannot.

- When it is OK to Use Buffer-Enabled Locators:
  - ***OCI*** — A locator that is enabled for buffering can only be used with the following OCI APIs:

    ```
    OCILobRead2(), OCILobWrite2(), OCILobAssign(), OCILobIsEqual(),
    OCILobLocatorIsInit(), OCILobCharSetId(), OCILobCharSetForm()
    ```

- When it is Not OK to Use Buffer-Enabled Locators:

  The following OCI APIs return errors if used with a locator enabled for buffering:

  - OCI:

    ```
    OCILobCopy2(), OCILobAppend(), OCILobErase2(), OCILobGetLength2(),
    OCILobTrim2(), OCILobWriteAppend2()
    ```

    These APIs also return errors when used with a locator which has not been enabled for buffering, but the LOB that the locator represents is being accessed in buffered mode through some other locator.

  - PL/SQL (DBMS_LOB):

    An error is returned from `DBMS_LOB` APIs if the input lob locator has buffering enabled.

  As in the case of all other locators, buffer-enabled locators cannot span transactions.

Saving Locator State to Avoid a Reselect

Suppose you want to save the current state of the LOB before further writing to the LOB buffer. In performing updates while using LOB buffering, writing to an existing buffer does not make a round-trip to the server, and so does not refresh the snapshot environment in the locator. This would not be the case if you were updating the LOB directly without using LOB buffering. In that case, every update would involve a round-trip to the server, and so would refresh the snapshot in the locator.

Therefore to save the state of a LOB that has been written through the LOB buffer, follow these steps:

1. Flush the LOB, thereby updating the LOB and the snapshot environment in the locator (L1). At this point, the state of the locator (L1) and the LOB are the same.
2. Assign the locator (L1) used for flushing and updating to another locator (L2). At this point, the states of the two locators (L1 and L2), and the LOB are all identical.

L2 now becomes a read-consistent locator with which you are able to access the changes made through L1 up until the time of the flush, but not after. This assignment avoids incurring a round-trip to the server to reselect the locator into L2.

OCI Example of LOB Buffering

The following OCI pseudocode example is based on the PM schema included with the Oracle Database Sample Schemas.

```
OCI_BLOB_buffering_program()
{
   int            amount;
   int            offset;
   OCILobLocator  lbs_loc1, lbs_loc2, lbs_loc3;
   void          *buffer;
   int            bufl;

   -- Standard OCI initialization operations - logging on to
   -- server, creating and initializing bind variables...

   init_OCI();

   -- Establish a savepoint before start of LOB buffering subsystem
   -- operations
   exec_statement("savepoint lbs_savepoint");

   -- Initialize bind variable to BLOB columns from buffered
   -- access:
   exec_statement("select ad_composite into lbs_loc1 from Print_media
       where ad_id = 12001");
   exec_statement("select ad_composite into lbs_loc2 from Print_media
       where ad_id = 12001 for update");
   exec_statement("select ad_composite into lbs_loc2 from Print_media
       where ad_id = 12001 for update");

   -- Enable locators for buffered mode access to LOB:
   OCILobEnableBuffering(..., lbs_loc1);
   OCILobEnableBuffering(..., lbs_loc2);
   OCILobEnableBuffering(..., lbs_loc3);

   -- Read 4K bytes through lbs_loc1 starting from offset 1:
   amount = 4096; offset = 1; bufl = 4096;
   OCILobRead2(.., lbs_loc1, &amount, 0, offset, buffer, bufl, ...);
      if (exception)
          goto exception_handler;
          -- This reads the first 32K bytes of the LOB from
          -- the server into a page (call it page_A) in the LOB
          -- client-side buffer.
          -- lbs_loc1 is a read-consistent locator.

          -- Write 4K of the LOB throgh lbs_loc2 starting from
          -- offset 1:
          amount = 4096; offset = 1; bufl = 4096;
          buffer = populate_buffer(4096);
          OCILobWrite2(.., lbs_loc2, &amount, 0, offset, buffer, bufl, ..);

      if (exception)
          goto exception_handler;
          -- This reads the first 32K bytes of the LOB from
          -- the server into a new page (call it page_B) in the
          -- LOB buffer, and modify the contents of this page
          -- with input buffer contents.
          -- lbs_loc2 is an updated locator.
```

```
    -- Read 20K bytes through lbs_loc1 starting from
    -- offset 10K
    amount = 20480; offset = 10240;
    OCILobRead2(.., lbs_loc1, &amount, 0, offset, buffer, bufl, ..);

if (exception)
  goto exception_handler;
    -- Read directly from page_A into the user buffer.
    -- There is no round-trip to the server because the
    -- data is in the client-side buffer.

    -- Write 20K bytes through lbs_loc2 starting from offset
    -- 10K
    amount = 20480; offset = 10240; bufl = 20480;
    buffer = populate_buffer(20480);
    OCILobWrite2(.., lbs_loc2, &amount, 0, offset, buffer, bufl, ..);

if (exception)
    goto exception_handler;
    -- The contents of the user buffer are now written
    -- into page_B without involving a round-trip to the
    -- server.  This avoids making a new LOB version on the
    -- server and writing redo to the log.

    -- The following write through lbs_loc3 also
    -- results in an error:
    amount = 20000; offset = 1000; bufl = 20000;
    buffer = populate_buffer(20000);
    OCILobWrite2(.., lbs_loc3, amount, 0, offset,buffer, bufl, ..);

if (exception)
    goto exception_handler;
    -- No two locators can be used to update a buffered LOB
    -- through the buffering subsystem

-- The following update through lbs_loc3 also
-- results in an error
OCILobFileCopy(.., lbs_loc3, lbs_loc2, ..);

if (exception)
    goto exception_handler;
    -- Locators enabled for buffering cannot be used with
    -- operations like Append, Copy, Trim and so on
    -- When done, flush the LOB buffer to the server:
OCILobFlushBuffer(.., lbs_loc2, OCI_LOB_BUFFER_NOFREE);

if (exception)
   goto exception_handler;
   -- This flushes all the modified pages in the LOB buffer,
   -- and resets lbs_loc2 from updated to read-consistent
   -- locator. The modified pages remain in the buffer
   -- without freeing memory.  These pages can be aged
   -- out if necessary.

-- Disable locators for buffered mode access to LOB */
OCILobDisableBuffering(..., lbs_loc1);
OCILobDisableBuffering(..., lbs_loc2);
OCILobDisableBuffering(..., lbs_loc3);

if (exception)
```

```
        goto exception_handler;
        -- This disables the three locators for buffered access,
        -- and frees up the LOB buffer resources.
       exception_handler:
     handle_exception_reporting();
     exec_statement("rollback to savepoint lbs_savepoint");
}
```

Opening Persistent LOBs with the OPEN and CLOSE Interfaces

The OPEN and CLOSE interfaces enable you to explicitly open a persistent LOB instance. When you open a LOB instance with the OPEN interface, the instance remains open until you explicitly close the LOB using the CLOSE interface. The ISOPEN interface enables you to determine whether a persistent LOB is open.

Note that the open state of a LOB is associated with the LOB instance, not the LOB locator. The locator does not save any information indicating whether the LOB instance that it points to is open.

See Also: "Opening and Closing LOBs" on page 2-2.

Index Performance Benefits of Explicitly Opening a LOB

Explicitly opening a LOB instance can benefit performance of a persistent LOB in an indexed column.

If you do not explicitly open the LOB instance, then every modification to the LOB implicitly opens and closes the LOB instance. Any triggers on a domain index are fired each time the LOB is closed. Note that in this case, any domain indexes on the LOB are updated as soon as any modification to the LOB instance is made; the domain index is always valid and can be used at any time.

When you explicitly open a LOB instance, index triggers do not fire until you explicitly close the LOB. Using this technique can increase performance on index columns by eliminating unneeded indexing events until you explicitly close the LOB. Note that any index on the LOB column is not valid until you explicitly close the LOB.

Working with Explicitly Open LOB Instances

If you explicitly open a LOB instance, then you must close the LOB before you commit the transaction.

Committing a transaction on the open LOB instance causes an error. When this error occurs, the LOB instance is closed implicitly, any modifications to the LOB instance are saved, and the transaction is committed, but any indexes on the LOB column are not updated. In this situation, you must rebuild your indexes on the LOB column.

If you subsequently rollback the transaction, then the LOB instance is rolled back to its previous state, but the LOB instance is no longer explicitly open.

You must close any LOB instance that you explicitly open:

- Between DML statements that start a transaction, including `SELECT ... FOR UPDATE` and `COMMIT`
- Within an autonomous transaction block
- Before the end of a session (when there is no transaction involved)

If you do not explicitly close the LOB instance, then it is implicitly closed at the end of the session and no index triggers are fired.

Keep track of the open or closed state of LOBs that you explicitly open. The following actions cause an error:

- Explicitly opening a LOB instance that has been explicitly open earlier.
- Explicitly closing a LOB instance that is has been explicitly closed earlier.

This occurs whether you access the LOB instance using the same locator or different locators.

Read-Consistent Locators

Oracle Database provides the same read consistency mechanisms for LOBs as for all other database reads and updates of scalar quantities. Refer to *Oracle Database Concepts* for general information about read consistency. Read consistency has some special applications to LOB locators that you must understand. These applications are described in the following sections.

A Selected Locator Becomes a Read-Consistent Locator

A selected locator, regardless of the existence of the `FOR UPDATE` clause, becomes a *read-consistent locator,* and remains a read-consistent locator until the LOB value is updated through that locator. A read-consistent locator contains the snapshot environment as of the point in time of the `SELECT` operation.

This has some complex implications. Suppose you have created a read-consistent locator (`L1`) by way of a `SELECT` operation. In reading the value of the persistent LOB through `L1`, note the following:

- The LOB is read as of the point in time of the `SELECT` statement even if the `SELECT` statement includes a `FOR UPDATE`.
- If the LOB value is updated through a different locator (`L2`) in the same transaction, then `L1` does not see the `L2` updates.
- `L1` does not see committed updates made to the LOB through another transaction.
- If the read-consistent locator `L1` is copied to another locator `L2` (for example, by a PL/SQL assignment of two locator variables — `L2 := L1`), then `L2` becomes a read-consistent locator along with `L1` and any data read is read as of the point in time of the `SELECT` for `L1`.

You can use the existence of multiple locators to access different transformations of the LOB value. However, in doing so, you must keep track of the different values accessed by different locators.

Example of Updating LOBs and Read-Consistency

Read-consistent locators provide the same LOB value regardless of when the `SELECT` occurs.

The following example demonstrates the relationship between read-consistency and updating in a simple example. Using the `Print_media` table and PL/SQL, three `CLOB` instances are created as potential locators: `clob_selected`, `clob_update`, and `clob_copied`.

Observe these progressions in the code, from times t1 through t6:

- At the time of the first SELECT INTO (at t1), the value in ad_sourcetext is associated with the locator clob_selected.
- In the second operation (at t2), the value in ad_sourcetext is associated with the locator clob_updated. Because there has been no change in the value of ad_sourcetext between t1 and t2, both clob_selected and clob_updated are read-consistent locators that effectively have the same value even though they reflect snapshots taken at different moments in time.
- The third operation (at t3) copies the value in clob_selected to clob_copied. At this juncture, all three locators see the same value. The example demonstrates this with a series of DBMS_LOB.READ() calls.
- At time t4, the program uses DBMS_LOB.WRITE() to alter the value in clob_updated, and a DBMS_LOB.READ() reveals a new value.
- However, a DBMS_LOB.READ() of the value through clob_selected (at t5) reveals that it is a read-consistent locator, continuing to refer to the same value as of the time of its SELECT.
- Likewise, a DBMS_LOB.READ() of the value through clob_copied (at t6) reveals that it is a read-consistent locator, continuing to refer to the same value as clob_selected.

```
INSERT INTO PRINT_MEDIA VALUES (2056, 20020, EMPTY_BLOB(),
    'abcd', EMPTY_CLOB(), EMPTY_CLOB(), NULL, NULL, NULL, NULL);

COMMIT;

DECLARE
  num_var           INTEGER;
  clob_selected     CLOB;
  clob_updated      CLOB;
  clob_copied       CLOB;
  read_amount       INTEGER;
  read_offset       INTEGER;
  write_amount      INTEGER;
  write_offset      INTEGER;
  buffer            VARCHAR2(20);

BEGIN
  -- At time t1:
  SELECT ad_sourcetext INTO clob_selected
     FROM Print_media
     WHERE ad_id = 20020;

  -- At time t2:
  SELECT ad_sourcetext INTO clob_updated
     FROM Print_media
     WHERE ad_id = 20020
     FOR UPDATE;

  -- At time t3:
  clob_copied := clob_selected;
  -- After the assignment, both the clob_copied and the
  -- clob_selected have the same snapshot as of the point in time
  -- of the SELECT into clob_selected

  -- Reading from the clob_selected and the clob_copied does
  -- return the same LOB value. clob_updated also sees the same
  -- LOB value as of its select:
```

```
  read_amount := 10;
  read_offset := 1;
  dbms_lob.read(clob_selected, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_selected value: ' || buffer);
  -- Produces the output 'abcd'

  read_amount := 10;
  dbms_lob.read(clob_copied, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_copied value: ' || buffer);
  -- Produces the output 'abcd'

  read_amount := 10;
  dbms_lob.read(clob_updated, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_updated value: ' || buffer);
  -- Produces the output 'abcd'

  -- At time t4:
  write_amount := 3;
  write_offset := 5;
  buffer := 'efg';
  dbms_lob.write(clob_updated, write_amount, write_offset, buffer);

  read_amount := 10;
  dbms_lob.read(clob_updated, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_updated value: ' || buffer);
  -- Produces the output 'abcdefg'

  -- At time t5:
  read_amount := 10;
  dbms_lob.read(clob_selected, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_selected value: ' || buffer);
  -- Produces the output 'abcd'

  -- At time t6:
  read_amount := 10;
  dbms_lob.read(clob_copied, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_copied value: ' || buffer);
  -- Produces the output 'abcd'
END;
/
```

Example of Updating LOBs Through Updated Locators

When you update the value of the persistent LOB through the LOB locator (`L1`), `L1` (that is, the locator itself) is updated to contain the current snapshot environment as of the time after the operation was completed on the LOB value through locator `L1`. `L1` is then termed an updated locator. This operation enables you to see your own changes to the LOB value on the next read through the same locator, `L1`.

> **Note:** The snapshot environment in the locator is *not* updated if the locator is used to merely read the LOB value. It is only updated when you modify the LOB value through the locator using the PL/SQL `DBMS_LOB` package or the OCI LOB APIs.

Any committed updates made by a different transaction are seen by L1 only if your transaction is a read-committed transaction and if you use L1 to update the LOB value after the other transaction committed.

Note: When you update a persistent LOB value, the modification is always made to the most current LOB value.

Updating the value of the persistent LOB through any of the available methods, such as OCI LOB APIs or PL/SQL `DBMS_LOB` package, updates the LOB value *and then reselects* the locator that refers to the new LOB value.

Caution: Once you have selected out a LOB locator by whatever means, you can read from the locator but not write into it.

Note that updating the LOB value through SQL is merely an `UPDATE` statement. It is up to you to do the reselect of the LOB locator or use the `RETURNING` clause in the `UPDATE` statement so that the locator can see the changes made by the `UPDATE` statement. Unless you reselect the LOB locator or use the `RETURNING` clause, you may think you are reading the latest value when this is not the case. For this reason you should avoid mixing SQL DML with `OCI` and `DBMS_LOB` piecewise operations.

See Also: *Oracle Database PL/SQL Language Reference*

Example of Updating a LOB Using SQL DML and DBMS_LOB

Using table `Print_media` in the following example, a `CLOB` locator is created as `clob_selected`. Note the following progressions in the example, from times t1 through t3:

- At the time of the first `SELECT INTO` (at `t1`), the value in `ad_sourcetext` is associated with the locator `clob_selected`.
- In the second operation (at `t2`), the value in `ad_sourcetext` is modified through the `SQL UPDATE` statement, without affecting the `clob_selected` locator. The locator still sees the value of the LOB as of the point in time of the original `SELECT`. In other words, the locator does not see the update made using the SQL `UPDATE` statement. This is illustrated by the subsequent `DBMS_LOB.READ()` call.
- The third operation (at `t3`) re-selects the LOB value into the locator `clob_selected`. The locator is thus updated with the latest snapshot environment which allows the locator to see the change made by the previous SQL `UPDATE` statement. Therefore, in the next `DBMS_LOB.READ()`, an error is returned because the LOB value is empty, that is, it does not contain any data.

```
INSERT INTO Print_media VALUES (3247, 20010, EMPTY_BLOB(),
    'abcd', EMPTY_CLOB(), EMPTY_CLOB(), NULL, NULL, NULL, NULL);

COMMIT;

DECLARE
  num_var           INTEGER;
  clob_selected     CLOB;
  read_amount       INTEGER;
  read_offset       INTEGER;
  buffer            VARCHAR2(20);

BEGIN

  -- At time t1:
```

```
  SELECT ad_sourcetext INTO clob_selected
  FROM Print_media
  WHERE ad_id = 20010;

  read_amount := 10;
  read_offset := 1;
  dbms_lob.read(clob_selected, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_selected value: ' || buffer);
  -- Produces the output 'abcd'

  -- At time t2:
  UPDATE Print_media SET ad_sourcetext = empty_clob()
      WHERE ad_id = 20010;
  -- although the most current LOB value is now empty,
  -- clob_selected still sees the LOB value as of the point
  -- in time of the SELECT

  read_amount := 10;
  dbms_lob.read(clob_selected, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_selected value: ' || buffer);
  -- Produces the output 'abcd'

  -- At time t3:
  SELECT ad_sourcetext INTO clob_selected FROM Print_media WHERE
       ad_id = 20010;
  -- the SELECT allows clob_selected to see the most current
  -- LOB value

  read_amount := 10;
  dbms_lob.read(clob_selected, read_amount, read_offset, buffer);
  -- ERROR: ORA-01403: no data found
END;
/
```

Example of Using One Locator to Update the Same LOB Value

> **Note:** Avoid updating the same LOB with different locators. You may avoid many pitfalls if you use only one locator to update a given LOB value.

In the following example, using table `Print_media`, two `CLOB`s are created as potential locators: clob_updated and clob_copied.

Note these progressions in the example at times t1 through t5:

- At the time of the first `SELECT INTO` (at `t1`), the value in `ad_sourcetext` is associated with the locator `clob_updated`.
- The second operation (at time `t2`) copies the value in `clob_updated` to `clob_copied`. At this time, both locators see the same value. The example demonstrates this with a series of `DBMS_LOB.READ()` calls.
- At time `t3`, the program uses `DBMS_LOB.WRITE()` to alter the value in `clob_updated`, and a `DBMS_LOB.READ()` reveals a new value.
- However, a `DBMS_LOB.READ()` of the value through `clob_copied` (at time `t4`) reveals that it still sees the value of the LOB as of the point in time of the assignment from `clob_updated` (at `t2`).

- It is not until clob_updated is assigned to clob_copied (t5) that clob_copied sees the modification made by clob_updated.

```
INSERT INTO PRINT_MEDIA VALUES (2049, 20030, EMPTY_BLOB(),
    'abcd', EMPTY_CLOB(), EMPTY_CLOB(), NULL, NULL, NULL, NULL);

COMMIT;

DECLARE
  num_var          INTEGER;
  clob_updated     CLOB;
  clob_copied      CLOB;
  read_amount      INTEGER;
  read_offset      INTEGER;
  write_amount     INTEGER;
  write_offset     INTEGER;
  buffer           VARCHAR2(20);
BEGIN

-- At time t1:
  SELECT ad_sourcetext INTO clob_updated FROM PRINT_MEDIA
      WHERE ad_id = 20030
      FOR UPDATE;

  -- At time t2:
  clob_copied := clob_updated;
  -- after the assign, clob_copied and clob_updated see the same
  -- LOB value

  read_amount := 10;
  read_offset := 1;
  dbms_lob.read(clob_updated, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_updated value: ' || buffer);
  -- Produces the output 'abcd'

  read_amount := 10;
  dbms_lob.read(clob_copied, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_copied value: ' || buffer);
  -- Produces the output 'abcd'

  -- At time t3:
  write_amount := 3;
  write_offset := 5;
  buffer := 'efg';
  dbms_lob.write(clob_updated, write_amount, write_offset,
        buffer);

  read_amount := 10;
  dbms_lob.read(clob_updated, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_updated value: ' || buffer);
  -- Produces the output 'abcdefg'


  -- At time t4:
  read_amount := 10;
  dbms_lob.read(clob_copied, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_copied value: ' || buffer);
  -- Produces the output 'abcd'

  -- At time t5:
  clob_copied := clob_updated;
```

```
  read_amount := 10;
  dbms_lob.read(clob_copied, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_copied value: ' || buffer);
  -- Produces the output 'abcdefg'
END;
/
```

Example of Updating a LOB with a PL/SQL (DBMS_LOB) Bind Variable

When a LOB locator is used as the source to update another persistent LOB (as in a SQL `INSERT` or `UPDATE` statement, the `DBMS_LOB.COPY` routine, and so on), the snapshot environment in the source LOB locator determines the LOB value that is used as the source. If the source locator (for example `L1`) is a read-consistent locator, then the LOB value as of the time of the `SELECT` of `L1` is used. If the source locator (for example `L2`) is an updated locator, then the LOB value associated with the `L2` snapshot environment at the time of the operation is used.

In the following example, using the table `Print_media`, three `CLOB`s are created as potential locators: `clob_selected`, clob_updated, and clob_copied.

Note these progressions in the example at times `t1` through `t5`:

- At the time of the first `SELECT INTO` (at `t1`), the value in `ad_sourcetext` is associated with the locator `clob_updated`.
- The second operation (at `t2`) copies the value in `clob_updated` to `clob_copied`. At this juncture, both locators see the same value.
- Then (at `t3`), the program uses `DBMS_LOB.WRITE()` to alter the value in `clob_updated`, and a `DBMS_LOB.READ()` reveals a new value.
- However, a `DBMS_LOB.READ()` of the value through `clob_copied` (at `t4`) reveals that `clob_copied` does not see the change made by `clob_updated`.
- Therefore (at `t5`), when `clob_copied` is used as the source for the value of the `INSERT` statement, the value associated with `clob_copied` (for example, without the new changes made by `clob_updated`) is inserted. This is demonstrated by the subsequent `DBMS_LOB.READ()` of the value just inserted.

```
INSERT INTO PRINT_MEDIA VALUES (2056, 20020, EMPTY_BLOB(),
    'abcd', EMPTY_CLOB(), EMPTY_CLOB(), NULL, NULL, NULL, NULL);

COMMIT;

DECLARE
  num_var           INTEGER;
  clob_selected     CLOB;
  clob_updated      CLOB;
  clob_copied       CLOB;
  read_amount       INTEGER;
  read_offset       INTEGER;
  write_amount      INTEGER;
  write_offset      INTEGER;
  buffer            VARCHAR2(20);
BEGIN

  -- At time t1:
  SELECT ad_sourcetext INTO clob_updated FROM PRINT_MEDIA
      WHERE ad_id = 20020
      FOR UPDATE;
```

```
  read_amount := 10;
  read_offset := 1;
  dbms_lob.read(clob_updated, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_updated value: ' || buffer);
  -- Produces the output 'abcd'

  -- At time t2:
  clob_copied := clob_updated;


  -- At time t3:
  write_amount := 3;
  write_offset := 5;
  buffer := 'efg';
  dbms_lob.write(clob_updated, write_amount, write_offset, buffer);

  read_amount := 10;
  dbms_lob.read(clob_updated, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_updated value: ' || buffer);
  -- Produces the output 'abcdefg'
  -- note that clob_copied does not see the write made before
  -- clob_updated


  -- At time t4:
  read_amount := 10;
  dbms_lob.read(clob_copied, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_copied value: ' || buffer);
  -- Produces the output 'abcd'

  -- At time t5:
  -- the insert uses clob_copied view of the LOB value which does
  -- not include clob_updated changes
  INSERT INTO PRINT_MEDIA VALUES (2056, 20022, EMPTY_BLOB(),
    clob_copied, EMPTY_CLOB(), EMPTY_CLOB(), NULL, NULL, NULL, NULL)
    RETURNING ad_sourcetext INTO clob_selected;

  read_amount := 10;
  dbms_lob.read(clob_selected, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_selected value: ' || buffer);
  -- Produces the output 'abcd'
END;
/
```

LOB Locators and Transaction Boundaries

This section discusses the use of LOB locators in transactions and transaction IDs. A basic description of LOB locators and their operations is given in "LOB Locators and BFILE Locators" on page 2-3.

Note the following regarding LOB locators and transactions:

- Locators contain transaction IDs when:

  You Begin the Transaction, Then Select Locator: If you begin a transaction and subsequently select a locator, then the locator contains the transaction ID. Note that you can implicitly be in a transaction without explicitly beginning one. For example, `SELECT... FOR UPDATE` implicitly begins a transaction. In such a case, the locator contains a transaction ID.

- Locators Do Not Contain Transaction IDs When...
  - You are Outside the Transaction, Then Select Locator: By contrast, if you select a locator outside of a transaction, then the locator does not contain a transaction ID.
  - When Selected Prior to DML Statement Execution: A transaction ID is not assigned until the first DML statement executes. Therefore, locators that are selected prior to such a DML statement do not contain a transaction ID.

Reading and Writing to a LOB Using Locators

You can always read the LOB data using the locator irrespective of whether the locator contains a transaction ID.

- Cannot Write Using Locator:

  If the locator contains a transaction ID, then you cannot write to the LOB outside of that particular transaction.

- Can Write Using Locator:

  If the locator *does not* contain a transaction ID, then you can write to the LOB after beginning a transaction either explicitly or implicitly.

- Cannot Read or Write Using Locator With Serializable Transactions:

  If the locator contains a transaction ID of an older transaction, and the current transaction is serializable, then you cannot read or write using that locator.

- Can Read, Not Write Using Locator With Non-Serializable Transactions:

  If the transaction is non-serializable, then you can read, but not write outside of that transaction.

The following examples show the relationship between locators and *non-serializable* transactions

Selecting the Locator Outside of the Transaction Boundary

The following scenarios describe techniques for using locators in non-serializable transactions when the locator is selected outside of a transaction.

First Scenario:

1. Select the locator with no current transaction. At this point, the locator does not contain a transaction id.
2. Begin the transaction.
3. Use the locator to read data from the LOB.
4. Commit or rollback the transaction.
5. Use the locator to read data from the LOB.
6. Begin a transaction. The locator does not contain a transaction id.
7. Use the locator to write data to the LOB. This operation is valid because the locator did not contain a transaction id prior to the write. After this call, the locator contains a transaction id.

Second Scenario:

1. Select the locator with no current transaction. At this point, the locator does not contain a transaction id.
2. Begin the transaction. The locator does not contain a transaction id.
3. Use the locator to read data from the LOB. The locator does not contain a transaction id.
4. Use the locator to write data to the LOB. This operation is valid because the locator did not contain a transaction id prior to the write. After this call, the locator contains a transaction id. You can continue to read from or write to the LOB.
5. Commit or rollback the transaction. The locator continues to contain the transaction id.
6. Use the locator to read data from the LOB. This is a valid operation.
7. Begin a transaction. The locator contains the previous transaction id.
8. Use the locator to write data to the LOB. This write operation fails because the locator does not contain the transaction id that matches the current transaction.

Selecting the Locator Within a Transaction Boundary

The following scenarios describe techniques for using locators in non-serializable transactions when the locator is selected within a transaction.

First Scenario:

1. Select the locator within a transaction. At this point, the locator contains the transaction id.
2. Begin the transaction. The locator contains the previous transaction id.
3. Use the locator to read data from the LOB. This operation is valid even though the transaction id in the locator does not match the current transaction.

   See Also: "Read-Consistent Locators" on page 12-10 for more information about using the locator to read LOB data.

4. Use the locator to write data to the LOB. This operation fails because the transaction id in the locator does not match the current transaction.

Second Scenario:

1. Begin a transaction.
2. Select the locator. The locator contains the transaction id because it was selected within a transaction.
3. Use the locator to read from or write to the LOB. These operations are valid.
4. Commit or rollback the transaction. The locator continues to contain the transaction id.
5. Use the locator to read data from the LOB. This operation is valid even though there is a transaction id in the locator and the transaction was previously committed or rolled back.
6. Use the locator to write data to the LOB. This operation fails because the transaction id in the locator is for a transaction that was previously committed or rolled back.

LOB Locators Cannot Span Transactions

Modifying a persistent LOB value through the LOB locator using DBMS_LOB, OCI, or SQL INSERT or UPDATE statements changes the locator from a read-consistent locator to an updated locator. The INSERT or UPDATE statement automatically starts a transaction and locks the row. Once this has occurred, the locator cannot be used outside the current transaction to modify the LOB value. In other words, LOB locators that are used to write data cannot span transactions. However, the locator can be used to read the LOB value unless you are in a serializable transaction.

> **See Also:** "LOB Locators and Transaction Boundaries" on page 12-17, for more information about the relationship between LOBs and transaction boundaries.

In the following example, a CLOB locator is created: clob_updated

- At the time of the first SELECT INTO (at t1), the value in ad_sourcetext is associated with the locator clob_updated.
- The second operation (at t2), uses the DBMS_LOB.WRITE function to alter the value in clob_updated, and a DBMS_LOB.READ reveals a new value.
- The commit statement (at t3) ends the current transaction.
- Therefore (at t4), the subsequent DBMS_LOB.WRITE operation fails because the clob_updated locator refers to a different (already committed) transaction. This is noted by the error returned. You must re-select the LOB locator before using it in further DBMS_LOB (and OCI) modify operations.

Example of Locator Not Spanning a Transaction

```
INSERT INTO PRINT_MEDIA VALUES (2056, 20010, EMPTY_BLOB(),
    'abcd', EMPTY_CLOB(), EMPTY_CLOB(), NULL, NULL, NULL, NULL);

COMMIT;

DECLARE
  num_var          INTEGER;
  clob_updated     CLOB;
  read_amount      INTEGER;
  read_offset      INTEGER;
  write_amount     INTEGER;
  write_offset     INTEGER;
  buffer           VARCHAR2(20);

BEGIN
          -- At time t1:
     SELECT      ad_sourcetext
     INTO        clob_updated
     FROM        PRINT_MEDIA
     WHERE       ad_id = 20010
     FOR UPDATE;
     read_amount := 10;
     read_offset := 1;
     dbms_lob.read(clob_updated, read_amount, read_offset, buffer);
     dbms_output.put_line('clob_updated value: ' || buffer);
     -- This produces the output 'abcd'

     -- At time t2:
     write_amount := 3;
```

```
     write_offset := 5;
     buffer := 'efg';
     dbms_lob.write(clob_updated, write_amount, write_offset, buffer);
     read_amount := 10;
     dbms_lob.read(clob_updated, read_amount, read_offset, buffer);
     dbms_output.put_line('clob_updated value: ' || buffer);
     -- This produces the output 'abcdefg'

    -- At time t3:
    COMMIT;

     -- At time t4:
    dbms_lob.write(clob_updated , write_amount, write_offset, buffer);
    -- ERROR: ORA-22990: LOB locators cannot span transactions
END;
/
```

LOBs in the Object Cache

Consider these object cache issues for internal and external LOB attributes:

- Persistent LOB attributes: Creating an object in object cache, sets the LOB attribute to empty.

  When you create an object in the object cache that contains a persistent LOB attribute, the LOB attribute is implicitly set to empty. You may not use this empty LOB locator to write data to the LOB. You must first flush the object, thereby inserting a row into the table and creating an empty LOB — that is, a LOB with 0 length. Once the object is refreshed in the object cache (use `OCI_PIN_LATEST`), the real LOB locator is read into the attribute, and you can then call the OCI LOB API to write data to the LOB.

- External LOB (`BFILE`) attributes: Creating an object in object cache, sets the `BFILE` attribute to `NULL`.

  When creating an object with an external LOB (`BFILE`) attribute, the `BFILE` is set to `NULL`. It must be updated with a valid directory object name and file name before reading from the `BFILE`.

When you copy one object to another in the object cache with a LOB locator attribute, only the LOB locator is copied. This means that the LOB attribute in these two different objects contain exactly the same locator which refers to *one and the same* LOB *value*. Only when the target object is flushed is a separate, physical copy of the LOB value made, which is distinct from the source LOB value.

> **See Also:** "Example of Updating LOBs and Read-Consistency" on page 12-10 for a description of what version of the LOB value is seen by each object if a write is performed through one of the locators.

Therefore, in cases where you want to modify the LOB that was the target of the copy, ***you must flush the target object, refresh the target object, and then*** write to the LOB through the locator attribute.

Terabyte-Size LOB Support

Terabyte-size LOBs—LOBs up to a maximum size of 8 to 128 terabytes depending on your database block size—are supported by the following APIs:

- Java using JDBC (Java Database Connectivity)

- PL/SQL using the DBMS_LOB Package
- C using OCI (Oracle Call Interface)

You cannot create and use LOB instances of size greater than 4 gigabytes "terabyte-size LOBs"— in the following programmatic environments:

- COBOL using the Pro*COBOL Precompiler
- C or C++ using the Pro*C/C++ Precompiler

> **Note:** Oracle Database does not support `BFILE`s larger than 2^64-1 bytes (`UB8MAXVAL` in OCI) in any programmatic environment. Any additional file size limit imposed by your operating system also applies to `BFILE`s.

Maximum Storage Limit for Terabyte-Size LOBs

In supported environments, you can create and manipulate LOBs that are up to the maximum storage size limit for your database configuration.

Oracle Database lets you create tablespaces with block sizes different from the database block size, and the maximum size of a LOB depends on the size of the tablespace blocks. `CHUNK` is a parameter of LOB storage whose value is controlled by the block size of the tablespace in which the LOB is stored.

> **Note:** The `CHUNK` parameter does not apply to SecureFiles. It is only used for BasicFiles LOBs.

When you create a LOB column, you can specify a value for `CHUNK`, which is the number of bytes to be allocated for LOB manipulation. The value must be a multiple of the tablespace block size, or Oracle Database rounds up to the next multiple. (If the tablespace block size is the same as the database block size, then `CHUNK` is also a multiple of the database block size.)

The maximum allowable storage limit for your configuration depends on the tablespace block size setting, and is calculated as (4 gigabytes - 1) times the value obtained from `DBMS_LOB.GETCHUNKSIZE` or `OCILobGetChunkSize()`. This value, in number of bytes for `BLOB`s or number of characters for `CLOB`s, is actually less than the size of the `CHUNK` parameter due to internal storage overhead. With the current allowable range for the tablespace block size from 2K to 32K, the storage limit ranges from 8 terabytes to 128 terabytes.

For example, suppose your database block size is 32K bytes and you create a tablespace with a nonstandard block size of 8K. Further suppose that you create a table with a LOB column and specify a CHUNK size of 16K (which is a multiple of the 8K tablespace block size). Then the maximum size of a LOB in this column is (4 gigabytes - 1) * 16K.

See Also:

- *Oracle Database Administrator's Guide* for details on the initialization parameter setting for your database installation
- "CHUNK" on page 11-11

This storage limit applies to all LOB types in environments that support terabyte-size LOBs. However, note that CLOB and NCLOB types are sized in characters, while the BLOB type is sized in bytes.

Using Terabyte-Size LOBs with JDBC

You can use the LOB APIs included in the Oracle JDBC classes to access terabyte-size LOBs.

See Also: "Using Java (JDBC) to Work With LOBs" on page 13-23

Using Terabyte-Size LOBs with the DBMS_LOB Package

You can access terabyte-size LOBs with all APIs in the DBMS_LOB PL/SQL package. Use DBMS_LOB.GETCHUNKSIZE to obtain the value to be used in reading and writing LOBs. The number of bytes stored in a chunk is actually less than the size of the CHUNK parameter due to internal storage overhead. The DBMS_LOB.GET_STORAGE_LIMIT function returns the storage limit for your database configuration. This is the maximum allowable size for LOBs. BLOBs are sized in bytes, while CLOBs and NCLOBs are sized in characters.

See Also: *Oracle Database PL/SQL Packages and Types Reference* for details on the initialization parameter setting for your database installation.

Using Terabyte-Size LOBs with OCI

The Oracle Call Interface API provides a set of functions for operations on LOBs of all sizes. OCILobGetChunkSize() returns the value, in bytes for BLOBs, or in characters for CLOBs, to be used in reading and writing LOBs. For varying-width character sets, the value is the number of Unicode characters that fit. The number of bytes stored in a chunk is actually less than the size of the CHUNK parameter due to internal storage overhead. The function OCILobGetStorageLimit() returns the maximum allowable size, in bytes, of internal LOBs in the current database installation. If streaming mode is used, where the whole LOB is read, there is no requirement to get the chunk size.

See Also: *Oracle Call Interface Programmer's Guide*, the chapter "LOB and BFILE Operations", section "Using LOBs of Size Greater than 4GB" for details on OCI functions that support LOBs.

Guidelines for Creating Gigabyte LOBs

To create gigabyte LOBs in supported environments, use the following guidelines to make use of all available space in the tablespace for LOB storage:

- Single Data File Size Restrictions:

  There are restrictions on the size of a single data file for each operating system. For example, Solaris 2.5 only allows operating system files of up to 2 gigabytes. Hence, add more data files to the tablespace when the LOB grows larger than the maximum allowed file size of the operating system on which your Oracle Database runs.

- Set PCT INCREASE Parameter to Zero:

PCTINCREASE parameter in the LOB storage clause specifies the percent growth of the new extent size. When a LOB is being filled up piece by piece in a tablespace, numerous new extents get created in the process. If the extent sizes keep increasing by the default value of 50 percent every time, then extents become unmanageable and eventually waste space in the tablespace. Therefore, the PCTINCREASE parameter should be set to zero or a small value.

- Set MAXEXTENTS to a Suitable Value or UNLIMITED:

  The MAXEXTENTS parameter limits the number of extents allowed for the LOB column. A large number of extents are created incrementally as the LOB size grows. Therefore, the parameter should be set to a value that is large enough to hold all the LOBs for the column. Alternatively, you could set it to UNLIMITED.

- Use a Large Extent Size:

  For every new extent created, Oracle generates undo information for the header and other metadata for the extent. If the number of extents is large, then the rollback segment can be saturated. To get around this, choose a large extent size, say 100 megabytes, to reduce the frequency of extent creation, or commit the transaction more often to reuse the space in the rollback segment.

Creating a Tablespace and Table to Store Gigabyte LOBs

The following example illustrates how to create a tablespace and table to store gigabyte LOBs.

```
CREATE TABLESPACE lobtbs1 DATAFILE '/your/own/data/directory/lobtbs_1.dat'
SIZE 2000M REUSE ONLINE NOLOGGING DEFAULT STORAGE (MAXEXTENTS UNLIMITED);
ALTER TABLESPACE lobtbs1 ADD DATAFILE
'/your/own/data/directory/lobtbs_2.dat' SIZE 2000M REUSE;

CREATE TABLE print_media_backup
  (product_id NUMBER(6),
   ad_id NUMBER(6),
   ad_composite BLOB,
   ad_sourcetext CLOB,
   ad_finaltext CLOB,
   ad_fltextn NCLOB,
   ad_textdocs_ntab textdoc_tab,
   ad_photo BLOB,
   ad_graphic BLOB,
   ad_header adheader_typ)
   NESTED TABLE ad_textdocs_ntab STORE AS textdocs_nestedtab5
   LOB(ad_sourcetext) STORE AS (TABLESPACE lobtbs1 CHUNK 32768 PCTVERSION 0
                                NOCACHE NOLOGGING
                                STORAGE(INITIAL 100M NEXT 100M MAXEXTENTS
                                UNLIMITED PCTINCREASE 0));
```

Note the following with respect to this example:

- The storage clause in this example is specified in the CREATE TABLESPACE statement.
- You can specify the storage clause in the CREATE TABLE statement as an alternative.
- The storage clause is not allowed in the CREATE TEMPORARY TABLESPACE statement.
- Setting the PCTINCREASE parameter to 0 is recommended for gigabyte LOBs. For small, or medium size lobs, the default PCTINCREASE value of 50 is recommended as it reduces the number of extent allocations.

13

Overview of Supplied LOB APIs

This chapter contains these topics:

Programmatic Environments That Support LOBs

Table 13–1 lists the programmatic environments that support LOB functionality.

> **See Also:** APIs for supported LOB operations are described in detail in the following chapters:
>
> - Chapter 19, "Operations Specific to Persistent and Temporary LOBs"
> - Chapter 22, "Using LOB APIs"
> - Chapter 21, "LOB APIs for BFILE Operations"

Table 13–1 Programmatic Environments That Support LOBs

Language	Precompiler or Interface Program	Syntax Reference	In This Chapter See...
PL/SQL	DBMS_LOB Package	*Oracle Database PL/SQL Packages and Types Reference*	"Using PL/SQL (DBMS_LOB Package) to Work With LOBs" on page 13-5.
C	Oracle Call Interface for C (OCI)	*Oracle Call Interface Programmer's Guide*	"Using OCI to Work With LOBs" on page 13-8.
C++	Oracle Call Interface for C++ (OCCI)	*Oracle C++ Call Interface Programmer's Guide*	"Using C++ (OCCI) to Work With LOBs" on page 13-13.
C/C++	Pro*C/C++ Precompiler	*Pro*C/C++ Programmer's Guide*	"Using C/C++ (Pro*C) to Work With LOBs" on page 13-18.
COBOL	Pro*COBOL Precompiler	*Pro*COBOL Programmer's Guide*	"Using COBOL (Pro*COBOL) to Work With LOBs" on page 13-21.
Java	JDBC Application Programmatic Interface (API)	*Oracle Database JDBC Developer's Guide*.	"Using Java (JDBC) to Work With LOBs" on page 13-23.
ADO/OLE DB	Oracle Provider for OLE DB (OraOLEDB).	*Oracle Provider for OLE DB Developer's Guide for Microsoft Windows*	"Oracle Provider for OLE DB (OraOLEDB)" on page 13-42
.NET	Oracle Data Provider for .NET (ODP.NET)	*Oracle Data Provider for .NET Developer's Guide for Microsoft Windows*	"Overview of Oracle Data Provider for .NET (ODP.NET)" on page 13-42

Comparing the LOB Interfaces

Table 13–2 and Table 13–3 compare the eight LOB programmatic interfaces by listing their functions and methods used to operate on LOBs. The tables are split in two simply to accommodate all eight interfaces. The functionality of the interfaces, with regards to LOBs, is described in the following sections.

Table 13–2 Comparing the LOB Interfaces, 1 of 2

PL/SQL: DBMS_LOB (dbmslob.sql)	C (OCI) (ociap.h)	C++ (OCCI) (occiData.h). Also for Clob and Bfile classes.	Pro*C/C++ and Pro*COBOL
`DBMS_LOB.COMPARE`	N/A	N/A	N/A
`DBMS_LOB.INSTR`	N/A	N/A	N/A
`DBMS_LOB.SUBSTR`	N/A	N/A	N/A
`DBMS_LOB.APPEND`	`OCILobAppend()`	`Blob.append()`	`APPEND`
N/A (use PL/SQL assign operator)	`OCILobAssign()`		`ASSIGN`
N/A	`OCILobCharSetForm()`	`Clob.getCharsetForm` (`CLOB` only)	N/A
N/A	`OCILobCharSetId()`	`Clob.getCharsetId()` (`CLOB` only)	N/A
`DBMS_LOB.CLOSE`	`OCILobClose()`	`Blob.close()`	`CLOSE`

Table 13–2 (Cont.) Comparing the LOB Interfaces, 1 of 2

PL/SQL: DBMS_LOB (dbmslob.sql)	C (OCI) (ociap.h)	C++ (OCCI) (occiData.h). Also for Clob and Bfile classes.	Pro*C/C++ and Pro*COBOL
N/A	N/A	Clob.closeStream()	N/A
DBMS_LOB.COPY	OCILobCopy2()	Blob.copy()	COPY
N/A	OCILobDisableBuffering()	N/A	DISABLE BUFFERING
N/A	OCILobEnableBuffering()	N/A	ENABLE BUFFERING
DBMS_LOB.ERASE	OCILobErase2()	N/A	ERASE
DBMS_LOB.FILECLOSE	OCILobFileClose()	Clob.close()	CLOSE
DBMS_LOB.FILECLOSEALL	OCILobFileCloseAll()	N/A	FILE CLOSE ALL
DBMS_LOB.FILEEXISTS	OCILobFileExist()	Bfile.fileExists()	DESCRIBE [FILEEXISTS]
DBMS_LOB.GETCHUNKSIZE	OCILobGetChunkSize()	Blob.getChunkSize()	DESCRIBE [CHUNKSIZE]
DBMS_LOB.GET_STORAGE_LIMIT	OCILobGetStorageLimit()	N/A	N/A
DBMS_LOB.GETOPTIONS	OCILobGetOptions()	Blob/Clob::getOptions	N/A
DBMS_LOB.FILEGETNAME	OCILobFileGetName()	Bfile.getFileName() and Bfile.getDirAlias()	DESCRIBE *DIRECTORY, FILENAME*
DBMS_LOB.FILEISOPEN	OCILobFileIsOpen()	Bfile.isOpen()	DESCRIBE ISOPEN
DBMS_LOB.FILEOPEN	OCILobFileOpen()	Bfile.open()	OPEN
N/A (use BFILENAME operator)	OCILobFileSetName()	Bfile.setName()	FILE SET
N/A	OCILobFlushBuffer()	N/A	FLUSH BUFFER
DBMS_LOB.GETLENGTH	OCILobGetLength2()	Blob.length()	DESCRIBE LENGTH
N/A	OCILobIsEqual()	Use operator = ( )=/!=	N/A
DBMS_LOB.ISOPEN	OCILobIsOpen()	Blob.isOpen()	DESCRIBE ISOPEN
DBMS_LOB.LOADFROMFILE	OCILobLoadFromFile2()	Use overloadedcopy()	LOAD FROM FILE
N/A	OCILobLocatorIsInit()	Clob.isinitialized()	N/A
DBMS_LOB.OPEN	OCILobOpen()	Blob.open	OPEN
DBMS_LOB.READ	OCILobRead()	Blob.read	READ
DBMS_LOB.SETOPTIONS	OCILobSetOptions()	Blob/Clob::setOptions	N/A
DBMS_LOB.TRIM	OCILobTrim2()	Blob.trim	TRIM
DBMS_LOB.WRITE	OCILobWrite2	Blob.write	WRITEORALOB.
DBMS_LOB.WRITEAPPEND	OCILobWriteAppend2()	N/A	WRITE APPEND
DBMS_LOB.CREATETEMPORARY	OCILobCreateTemporary()	N/A	N/A
DBMS_LOB.FREETEMPORARY	OCILobFreeTemporary()	N/A	N/A
DBMS_LOB.ISTEMPORARY	OCILobIsTemporary()	N/A	N/A
N/A	OCILobLocatorAssign()	use operator = ( ) or copy constructor	N/A

Table 13–3 Comparing the LOB Interfaces, 2 of 2

PL/SQL: DBMS_LOB (dbmslob.sql)	Java (JDBC)	ODP.NET
DBMS_LOB.COMPARE	Use DBMS_LOB.	OracleClob.Compare
DBMS_LOB.INSTR	position	OracleClob.Search
DBMS_LOB.SUBSTR	getBytes for BLOBs or BFILEsgetSubString for CLOBs	N/A
DBMS_LOB.APPEND	Use length and then putBytes() or PutString()	OracleClob.Append
OCILobAssign()	N/A [use equal sign]	OracleClob.Clone
OCILobCharSetForm()	N/A	N/A
OCILobCharSetId()	N/A	N/A
DBMS_LOB.CLOSE	use DBMS_LOB.	OracleClob.Close
DBMS_LOB.COPY	Use read and write	OracleClob.CopyTo
OCILobDisableBuffering()	N/A	N/A
OCILobEnableBuffering()	N/A	N/A
DBMS_LOB.ERASE	Use DBMS_LOB.	OracleClob.Erase
DBMS_LOB.FILECLOSE	closeFile	OracleBFile.CloseFile
DBMS_LOB.FILECLOSEALL	Use DBMS_LOB.	N/A
DBMS_LOB.FILEEXISTS	fileExists	OracleBFile.FileExists
DBMS_LOB.GETCHUNKSIZE	getChunkSize	OracleClob.OptimumChunkSize
DBMS_LOB.FILEGETNAME	getDirAlias getName	OracleBFile.DirectoryName Oracle.BFile.FileName
DBMS_LOB.FILEISOPEN	Use DBMS_LOB.ISOPEN	OracleBFile.IsOpen
DBMS_LOB.FILEOPEN	openFile	OracleBFile.OpenFile
OCILobFileSetName()	Use BFILENAME	OracleBFile.DirectoryName Oracle.BFile.FileName
OCILobFlushBuffer()	N/A	N/A
DBMS_LOB.GETLENGTH	length	OracleClob.Length
N/A	equals()	N/A
DBMS_LOB.ISOPEN	use DBMS_LOB.ISOPEN()	OracleClob.IsInChunkWriteMode
DBMS_LOB.LOADFROMFILE	Use read and then write	N/A
DBMS_LOB.OPEN	Use DBMS_LOB.OPEN()	OracleClob.BeginChunkWrite
DBMS_LOB.READ	BLOB or BFILE: getBytes() and getBinaryStream() CLOB: getString() and getSubString() and getCharacterStream()	OracleClob.Read

Table 13–3 (Cont.) Comparing the LOB Interfaces, 2 of 2

PL/SQL: DBMS_LOB (dbmslob.sql)	Java (JDBC)	ODP.NET
DBMS_LOB.TRIM	Use DBMS_LOB.TRIM()	OracleClob.SetLength
DBMS_LOB.WRITE	BLOB: setBytes() and setBinaryStream()	OracleClob.Write
	CLOB: setString() and setCharacterStream()	
DBMS_LOB.WRITEAPPEND	Use length() and then putString() or putBytes()	OracleClob.Append
DBMS_LOB.CREATETEMPORARY	N/A	OracleClob constructors
DBMS_LOB.FREETEMPORARY	N/A	OracleClob.Dispose
DBMS_LOB.ISTEMPORARY	N/A	OracleClob.IsTemporary

Using PL/SQL (DBMS_LOB Package) to Work With LOBs

The PL/SQL DBMS_LOB package can be used for the following operations:

- **Internal persistent LOBs and Temporary LOBs**: Read and modify operations, either entirely or in a piece-wise manner.
- **BFILEs**: Read operations

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* for detailed documentation, including parameters, parameter types, return values, and example code.

Provide a LOB Locator Before Running the DBMS_LOB Routine

As described in more detail in the following, DBMS_LOB routines work based on *LOB locators*. For the successful completion of DBMS_LOB routines, you must provide an input locator representing a LOB that exists in the database tablespaces or external file system, *before* you call the routine.

- **Persistent LOBs:** First use SQL to define tables that contain LOB columns, and subsequently you can use SQL to initialize or populate the locators in these LOB columns.
- **External LOBs:** Define a DIRECTORY object that maps to a valid physical directory containing the external LOBs that you intend to access. These files must exist, and have READ permission for Oracle Server to process. If your operating system uses case-sensitive path names, then specify the directory in the correct case. See "Directory Objects" on page 21-3 for more information.

Once the LOBs are defined and created, you may then SELECT a LOB locator into a local PL/SQL LOB variable and use this variable as an input parameter to DBMS_LOB for access to the LOB value.

Examples provided with each DBMS_LOB routine illustrate this in the following sections.

Guidelines for Offset and Amount Parameters in DBMS_LOB Operations

The following guidelines apply to offset and amount parameters used in procedures in the DBMS_LOB PL/SQL package:

- For character data—in all formats, fixed-width and varying-width—the `amount` and `offset` parameters are in characters. This applies to operations on `CLOB` and `NCLOB` data types.
- For binary data, the `offset` and `amount` parameters are in bytes. This applies to operations on `BLOB` data types.
- When using the following procedures:
  - `DBMS_LOB.LOADFROMFILE`
  - `DBMS_LOB.LOADBLOBFROMFILE`
  - `DBMS_LOB.LOADCLOBFROMFILE`

  you cannot specify an amount parameter with a value larger than the size of the `BFILE` you are loading from. To load the entire `BFILE` with these procedures, you must specify either the exact size of the `BFILE`, or the maximum allowable storage limit.

  See Also:
  - "Loading a LOB with Data from a BFILE" on page 22-5
  - "Loading a BLOB with Data from a BFILE" on page 22-7
  - "Loading a CLOB or NCLOB with Data from a BFILE" on page 22-8

- When using `DBMS_LOB.READ`, the `amount` parameter can be larger than the size of the data. The amount should be less than or equal to the size of the buffer. The buffer size is limited to 32K.

  See Also: "Reading Data from a LOB" on page 22-12

Determining Character Set ID

To determine the character set ID, you must know the character set name (a user can select from the `V$NLS_VALID_VALUES` view, which lists the names of the character sets that are valid as database and national character sets). Then call the function `NLS_CHARSET_ID` with the desired character set name as the one string argument. The character set ID is returned as an integer. `UTF16` does not work because it has no character set name. Use character set ID = 1000 for `UTF16`. Although `UTF16` is not allowed as a database or national character set, the APIs in `DBMS_LOB` support it for database conversion purposes. `DBMS_LOB.LOADCLOBFROMFILE` and other procedures in `DBMS_LOB` take character set ID, not character set name, as an input.

See Also:
- *Oracle Database PL/SQL Packages and Types Reference* for details and supported Unicode encodings in the chapter on `DBMS_LOB.LOADCLOBFROMFILE`
- *Oracle Database Globalization Support Guide*, Appendix A, for supported languages

PL/SQL Functions and Procedures for LOBs

PL/SQL functions and procedures that operate on `BLOB`s, `CLOB`s, `NCLOB`s, and `BFILE`s are summarized in the following:

- To modify persistent LOB values, see Table 13–4
- To read or examine LOB values, see Table 13–5
- To create, free, or check on temporary LOBs, see Table 13–6
- For read-only functions on external LOBs (BFILEs), see Table 13–7
- To open or close a LOB, or check if LOB is open, see Table 13–8
- To perform archive management on SecureFiles, see "PL/SQL Packages for LOBs and DBFS" on page 4-25

PL/SQL Functions and Procedures to Modify LOB Values

Here is a table of DBMS_LOB procedures:

Table 13–4 PL/SQL: DBMS_LOB Procedures to Modify LOB Values

Function/Procedure	Description
APPEND	Appends the LOB value to another LOB
CONVERTTOBLOB	Converts a CLOB to a BLOB
CONVERTTOCLOB	Converts a BLOB to a CLOB
COPY	Copies all or part of a LOB to another LOB
ERASE	Erases part of a LOB, starting at a specified offset
FRAGMENT_DELETE	Delete the data from the LOB at the given offset for the given length
FRAGMENT_INSERT	Insert the given data (< 32KBytes) into the LOB at the given offset
FRAGMENT_MOVE	Move the given amount of bytes from the given offset to the new given offset
FRAGMENT_REPLACE	Replace the data at the given offset with the given data (< 32kBytes)
LOADFROMFILE	Load BFILE data into a persistent LOB
LOADCLOBFROMFILE	Load character data from a file into a LOB
LOADBLOBFROMFILE	Load binary data from a file into a LOB
SETOPTIONS	Sets LOB features (deduplication and compression)
TRIM	Trims the LOB value to the specified shorter length
WRITE	Writes data to the LOB at a specified offset
WRITEAPPEND	Writes data to the end of the LOB

PL/SQL Functions and Procedures for Introspection of LOBs

Table 13–5 PL/SQL: DBMS_LOB Procedures to Read or Examine Internal and External LOB values

Function/Procedure	Description
COMPARE	Compares the value of two LOBs
GETCHUNKSIZE	Gets the chunk size used when reading and writing. This only works on persistent LOBs and does not apply to external LOBs (BFILEs).
GETLENGTH	Gets the length of the LOB value.
GETOPTIONS	Returns options (deduplication, compression, encryption) for SecureFiles.
GET_STORAGE_LIMIT	Gets the LOB storage limit for the database configuration.
INSTR	Returns the matching position of the nth occurrence of the pattern in the LOB.

Table 13–5 (Cont.) PL/SQL: DBMS_LOB Procedures to Read or Examine Internal and External LOB values

Function/Procedure	Description
ISSECUREFILE	Returns TRUE if the BLOB or CLOB locator passed to it is for a SecureFiles or FALSE if it is not.
READ	Reads data from the LOB starting at the specified offset.
SETOPTIONS	Sets options (deduplication and compression) for a SecureFiles, overriding the default LOB column settings. Incurs a server round trip.
SUBSTR	Returns part of the LOB value starting at the specified offset.

PL/SQL Operations on Temporary LOBs

Table 13–6 PL/SQL: DBMS_LOB Procedures to Operate on Temporary LOBs

Function/Procedure	Description
CREATETEMPORARY	Creates a temporary LOB
ISTEMPORARY	Checks if a LOB locator refers to a temporary LOB
FREETEMPORARY	Frees a temporary LOB

PL/SQL Read-Only Functions and Procedures for BFILEs

Table 13–7 PL/SQL: DBMS_LOB Read-Only Procedures for BFILEs

Function/Procedure	Description
FILECLOSE	Closes the file. Use CLOSE() instead.
FILECLOSEALL	Closes all previously opened files
FILEEXISTS	Checks if the file exists on the server
FILEGETNAME	Gets the directory object name and file name
FILEISOPEN	Checks if the file was opened using the input BFILE locators. Use ISOPEN() instead.
FILEOPEN	Opens a file. Use OPEN() instead.

PL/SQL Functions and Procedures to Open and Close Internal and External LOBs

Table 13–8 PL/SQL: DBMS_LOB Procedures to Open and Close Internal and External LOBs

Function/Procedure	Description
OPEN	Opens a LOB
ISOPEN	Sees if a LOB is open
CLOSE	Closes a LOB

These procedures are described in detail for specific LOB operations, such as, INSERT a row containing a LOB, in "Opening Persistent LOBs with the OPEN and CLOSE Interfaces" on page 12-9.

Using OCI to Work With LOBs

Oracle Call Interface (OCI) LOB functions enable you to access and make changes to LOBs and to read data from BFILEs in C.

See Also: *Oracle Call Interface Programmer's Guide* chapter "LOB and BFILE Operations" for the details of all topics discussed in this section.

Prefetching of LOB Data, Length, and Chunk Size

To improve OCI access of smaller LOBs, LOB data can be prefetched and cached while also fetching the locator. This applies to internal LOBs, temporary LOBs, and BFILEs.

Setting the CSID Parameter for OCI LOB APIs

If you want to read or write data in 2-byte Unicode format, then set the csid (character set ID) parameter in OCILobRead2() and OCILobWrite2() to OCI_UTF16ID. The csid parameter indicates the character set id for the buffer parameter. You can set the csid parameter to any character set ID. If the csid parameter is set, then it overrides the NLS_LANG environment variable.

See Also:

- *Oracle Call Interface Programmer's Guide* for information on the OCIUnicodeToCharSet() function and details on OCI syntax in general.
- *Oracle Database Globalization Support Guide* for detailed information about implementing applications in different languages.

Fixed-Width and Varying-Width Character Set Rules for OCI

In OCI, for fixed-width client-side character sets, the following rules apply:

- CLOBs and NCLOBs: offset and amount parameters are always in characters
- BLOBs and BFILEs: offset and amount parameters are always in bytes

The following rules apply only to varying-width client-side character sets:

- **Offset parameter**:

  Regardless of whether the client-side character set is varying-width, the offset parameter is always as follows:

  - CLOBs and NCLOBs: in characters
  - BLOBs and BFILEs: in bytes

- **Amount parameter**:

  The amount parameter is always as follows:

  - When referring to a server-side LOB: in characters
  - When referring to a client-side buffer: in bytes

- **OCILobFileGetLength()**:

  Regardless of whether the client-side character set is varying-width, the output length is as follows:

  - CLOBs and NCLOBs: in characters
  - BLOBs and BFILEs: in bytes

- **OCILobRead2()**:

  With client-side character set of varying-width, CLOBs and NCLOBs:

- **Input amount** is in characters. Input amount refers to the number of characters to read from the server-side CLOB or NCLOB.
- **Output amount** is in bytes. Output amount indicates how many bytes were read into the buffer bufp.

- **OCILobWrite2()**: With client-side character set of varying-width, CLOBs and NCLOBs:
  - **Input amount** is in bytes. The input amount refers to the number of bytes of data in the input buffer bufp.
  - **Output amount** is in characters. The output amount refers to the number of characters written into the server-side CLOB or NCLOB.

Other Operations

For all other LOB operations, irrespective of the client-side character set, the amount parameter is in characters for CLOBs and NCLOBs. These include OCILobCopy2(), OCILobErase2(), OCILobLoadFromFile2(), and OCILobTrim2(). All these operations refer to the amount of LOB data on the server.

See Also: *Oracle Database Globalization Support Guide*

NCLOBs in OCI

NCLOBs are allowed as parameters in methods.

OCILobLoadFromFile2() Amount Parameter

When using OCILobLoadFromFile2() you cannot specify amount larger than the length of the BFILE. To load the entire BFILE, you can pass the value returned by OCILobGetStorageLimit().

OCILobRead2() Amount Parameter

To read to the end of a LOB using OCILobRead2(), you specify an amount equal to the value returned by OCILobGetStorageLimit(). See "Reading Data from a LOB" on page 22-12 for more information.

OCILobLocator Pointer Assignment

Special care must be taken when assigning OCILobLocator pointers in an OCI program—using the "=" assignment operator. Pointer assignments create a shallow copy of the LOB. After the pointer assignment, the source and target LOBs point to the same copy of data.

These semantics are different from using LOB APIs, such as OCILobAssign() or OCILobLocatorAssign() to perform assignments. When the these APIs are used, the locators logically point to independent copies of data after assignment.

For temporary LOBs, before performing pointer assignments, you must ensure that any temporary LOB in the target LOB locator is freed by calling OCIFreeTemporary(). In contrast, when OCILobLocatorAssign() is used, the original temporary LOB in the target LOB locator variable, if any, is freed automatically before the assignment happens.

LOB Locators in Defines and Out-Bind Variables in OCI

Before you reuse a LOB locator in a define or an out-bind variable in a SQL statement, you must free any temporary LOB in the existing LOB locator buffer using `OCIFreeTemporary()`.

OCI Functions That Operate on BLOBs, CLOBs, NCLOBs, and BFILEs

OCI functions that operate on `BLOB`s, `CLOB`s, `NCLOB`s, and `BFILE`s are as follows:

- To modify persistent LOBs, see Table 13–9
- To read or examine LOB values, see Table 13–10
- To create or free temporary LOB, or check if Temporary LOB exists, see Table 13–11
- For read only functions on external LOBs (`BFILE`s), see Table 13–12
- To operate on LOB locators, see Table 13–13
- For LOB buffering, see Table 13–14
- To open and close LOBs, see Table 13–15

OCI Functions to Modify Persistent LOB (BLOB, CLOB, and NCLOB) Values

Table 13–9 OCI Functions to Modify Persistent LOB (BLOB, CLOB, and NCLOB) Values

Function/Procedure	Description
`OCILobAppend()`	Appends LOB value to another LOB.
`OCILobArrayWrite()`	Writes data using multiple locators in one round trip.
`OCILobCopy2()`	Copies all or part of a LOB to another LOB.
`OCILobErase2()`	Erases part of a LOB, starting at a specified offset.
`OCILobLoadFromFile2()`	Loads `BFILE` data into a persistent LOB.
`OCILobSetContentType()`	Sets a content string in a SecureFiles.
`OCILObSetOptions()`	Enables option settings (deduplication and compression) for a SecureFiles.
`OCILobTrim2()`	Truncates a LOB.
`OCILobWrite2()`	Writes data from a buffer into a LOB, overwriting existing data.
`OCILobWriteAppend2()`	Writes data from a buffer to the end of the LOB.

OCI Functions to Read or Examine Persistent LOB and External LOB (BFILE) Values

Table 13–10 OCI Functions to Read or Examine persistent LOB and external LOB (BFILE) Values

Function/Procedure	Description
`OCILobArrayRead()`	Reads data using multiple locators in one round trip.
`OCILobGetChunkSize()`	Gets the chunk size used when reading and writing. This works on persistent LOBs and does not apply to external LOBs (`BFILE`s).
`OCILobGetContentType()`	Gets the content string for a SecureFiles.
`OCILobGetLength2()`	Returns the length of a LOB or a `BFILE`.

Table 13–10 (Cont.) OCI Functions to Read or Examine persistent LOB and external LOB (BFILE) Values

Function/Procedure	Description
`OCILObGetOptions()`	Obtains the enabled settings (deduplication, compression, encryption) for a given SecureFiles.
`OCILobGetStorageLimit()`	Gets the maximum length of an internal LOB.
`OCILobRead2()`	Reads a specified portion of a non-`NULL` LOB or a `BFILE` into a buffer.

OCI Functions for Temporary LOBs

Table 13–11 OCI Functions for Temporary LOBs

Function/Procedure	Description
`OCILobCreateTemporary()`	Creates a temporary LOB.
`OCILobIsTemporary()`	Sees if a temporary LOB exists.
`OCILobFreeTemporary()`	Frees a temporary LOB.

OCI Read-Only Functions for BFILEs

Table 13–12 OCI Read-Only Functions for BFILES

Function/Procedure	Description
`OCILobFileClose()`	Closes an open `BFILE`.
`OCILobFileCloseAll()`	Closes all open `BFILE`s.
`OCILobFileExists()`	Checks whether a `BFILE` exists.
`OCILobFileGetName()`	Returns the name of a `BFILE`.
`OCILobFileIsOpen()`	Checks whether a `BFILE` is open.
`OCILobFileOpen()`	Opens a `BFILE`.

OCI LOB Locator Functions

Table 13–13 OCI LOB-Locator Functions

Function/Procedure	Description
`OCILobAssign()`	Assigns one LOB locator to another.
`OCILobCharSetForm()`	Returns the character set form of a LOB.
`OCILobCharSetId()`	Returns the character set ID of a LOB.
`OCILobFileSetName()`	Sets the name of a `BFILE` in a locator.
`OCILobIsEqual()`	Checks whether two LOB locators refer to the same LOB.
`OCILobLocatorIsInit()`	Checks whether a LOB locator is initialized.

OCI LOB-Buffering Functions

Table 13–14 OCI LOB-Buffering Functions

Function/Procedure	Description
`OCILobDisableBuffering()`	Disables the buffering subsystem use.
`OCILobEnableBuffering()`	Uses the LOB buffering subsystem for subsequent reads and writes of LOB data.
`OCILobFlushBuffer()`	Flushes changes made to the LOB buffering subsystem to the database (server).

OCI Functions to Open and Close Internal and External LOBs

Table 13–15 OCI Functions to Open and Close Internal and External LOBs

Function/Procedure	Description
`OCILobOpen()`	Opens a LOB.
`OCILobIsOpen()`	Sees if a LOB is open.
`OCILobClose()`	Closes a LOB.

OCI LOB Examples

Further OCI examples are provided in:

- Chapter 22, "Using LOB APIs"
- Chapter 21, "LOB APIs for BFILE Operations"

See also Appendix B, "OCI Demonstration Programs" in *Oracle Call Interface Programmer's Guide*, for further OCI demonstration script listings.

Further Information About OCI

For further information and features of OCI, refer to the OTN Web site, `http://www.oracle.com/technology/` for OCI features and frequently asked questions.

Using C++ (OCCI) to Work With LOBs

Oracle C++ Call Interface (OCCI) is a C++ API for manipulating data in an Oracle database. OCCI is organized as an easy-to-use set of C++ classes that enable a C++ program to connect to a database, run SQL statements, insert/update values in database tables, retrieve results of a query, run stored procedures in the database, and access metadata of database schema objects. OCCI also provides a seamless interface to manipulate objects of user-defined types as C++ class instances.

Oracle C++ Call Interface (OCCI) is designed so that you can use OCI and OCCI together to build applications.

The OCCI API provides the following advantages over JDBC and ODBC:

- OCCI encompasses more Oracle functionality than JDBC. OCCI provides all the functionality of OCI that JDBC does not provide.
- OCCI provides *compiled* performance. With compiled programs, the source code is written as close to the computer as possible. Because JDBC is an *interpreted* API, it cannot provide the performance of a compiled API. With an interpreted program,

performance degrades as each line of code must be interpreted individually into code that is close to the computer.

- OCCI provides memory management with smart pointers. You do not have to be concerned about managing memory for OCCI objects. This results in robust higher performance application code.
- Navigational access of OCCI enables you to intuitively access objects and call methods. Changes to objects persist without writing corresponding SQL statements. If you use the client side cache, then the navigational interface performs better than the object interface.
- With respect to ODBC, the OCCI API is simpler to use. Because ODBC is built on the C language, OCCI has all the advantages C++ provides over C. Moreover, ODBC has a reputation as being difficult to learn. The OCCI, by contrast, is designed for ease of use.

You can use OCCI to make changes to an entire persistent LOB, or to pieces of the beginning, middle, or end of it, as follows:

- For reading from internal and external LOBs (`BFILE`s)
- For writing to persistent LOBs

OCCI Classes for LOBs

OCCI provides the following classes that allow you to use different types of LOB instances as objects in your C++ application:

- `Clob` class to access and modify data stored in internal `CLOB`s and `NCLOB`s
- `Blob` class to access and modify data stored in internal `BLOB`s
- `Bfile` class to access and read data stored in external LOBs (`BFILE`s)

> **See Also:** Syntax information on these classes and details on OCCI in general is available in the *Oracle C++ Call Interface Programmer's Guide*.

Clob Class

The Clob driver implements a `CLOB` object using an SQL LOB locator. This means that a CLOB object contains a logical pointer to the SQL `CLOB` data rather than the data itself.

The `CLOB` interface provides methods for getting the length of an SQL `CLOB` value, for materializing a `CLOB` value on the client, and getting a substring. Methods in the `ResultSet` and `Statement` interfaces such as `getClob()` and `setClob()` allow you to access SQL `CLOB` values.

> **See Also:** *Oracle C++ Call Interface Programmer's Guide* for detailed information on the Clob class.

Blob Class

Methods in the `ResultSet` and `Statement` interfaces, such as `getBlob()` and `setBlob()`, allow you to access SQL `BLOB` values. The `Blob` interface provides methods for getting the length of a SQL `BLOB` value, for materializing a `BLOB` value on the client, and for extracting a part of the `BLOB`.

See Also:

- *Oracle C++ Call Interface Programmer's Guide* for detailed information on the Blob class methods and details on instantiating and initializing a Blob object in your C++ application.
- *Oracle Database Globalization Support Guide* for detailed information about implementing applications in different languages.

Bfile Class

The `Bfile` class enables you to instantiate a `Bfile` object in your C++ application. You must then use methods of the `Bfile` class, such as the `setName()` method, to initialize the `Bfile` object which associates the object properties with an object of type `BFILE` in a `BFILE` column of the database.

See Also: *Oracle C++ Call Interface Programmer's Guide* for detailed information on the `Blob` class methods and details on instantiating and initializing an `Blob` object in your C++ application.

Fixed-Width Character Set Rules

In OCCI, for *fixed-width* client-side character sets, the following rules apply:

- `Clob`: offset and amount parameters are always in characters
- `Blob`: offset and amount parameters are always in bytes
- `Bfile`: offset and amount parameters are always in bytes

Varying-Width Character Set Rules

The following rules apply only to *varying-width* client-side character sets:

- **Offset parameter**: Regardless of whether the client-side character set is varying-width, the offset parameter is always as follows:
  - `Clob()`: in characters
  - `Blob()`: in bytes
  - `Bfile()`: in bytes
- **Amount parameter**: The amount parameter is always as follows:
  - `Clob`: in characters, when referring to a server-side LOB
  - `Blob`: in bytes, when referring to a client-side buffer
  - `Bfile`: in bytes, when referring to a client-side buffer
- **length()**: Regardless of whether the client-side character set is varying-width, the output length is as follows:
  - `Clob.length()`: in characters
  - `Blob.length()`: in bytes
  - `Bfile.length()`: in bytes
- **Clob.read() and Blob.read()**: With client-side character set of varying-width, `CLOB`s and `NCLOB`s:
  - ***Input amount*** is in characters. Input amount refers to the number of characters to read from the server-side `CLOB` or `NCLOB`.

    - ***Output amount*** is in bytes. Output amount indicates how many bytes were read into the OCCI buffer parameter, `buffer`.
- **Clob.write() and Blob.write()**: With client-side character set of varying-width, `CLOB`s and `NCLOB`s:
    - **Input amount** is in bytes. Input amount refers to the number of bytes of data in the OCCI input buffer, `buffer`.
    - **Output amount** is in characters. Output amount refers to the number of characters written into the server-side `CLOB` or `NCLOB`.

Offset and Amount Parameters for Other OCCI Operations

For all other OCCI LOB operations, irrespective of the client-side character set, the *amount parameter* is in characters for `CLOB`s and `NCLOB`s. These include the following:

- `Clob.copy()`
- `Clob.erase()`
- `Clob.trim()`
- For `LoadFromFile` functionality, overloaded `Clob.copy()`

All these operations refer to the amount of LOB data on the server.

> **See also:** *Oracle Database Globalization Support Guide*

NCLOBs in OCCI

- `NCLOB` instances are allowed as parameters in methods
- `NCLOB` instances are allowed as attributes in object types.

Amount Parameter for OCCI LOB copy() Methods

The `copy()` method on `Clob` and `Blob` enables you to load data from a `BFILE`. You can pass one of the following values for the `amount` parameter to this method:

- An amount smaller than the size of the `BFILE` to load a portion of the data
- An amount equal to the size of the `BFILE` to load all of the data
- The `UB8MAXVAL` constant to load all of the `BFILE` data

You cannot specify an amount larger than the length of the `BFILE`.

Amount Parameter for OCCI read() Operations

The `read()` method on an `Clob`, `Blob`, or `Bfile` object, reads data from a `BFILE`. You can pass one of the following values for the amount parameter to specify the amount of data to read:

- An amount smaller than the size of the `BFILE` to load a portion of the data
- An amount equal to the size of the `BFILE` to load all of the data
- `0` (zero) to read until the end of the `BFILE` in streaming mode

You cannot specify an amount larger than the length of the `BFILE`.

Further Information About OCCI

See Also:

- *Oracle C++ Call Interface Programmer's Guide*
- http://www.oracle.com/ search for articles and product information featuring OCCI.

OCCI Methods That Operate on BLOBs, BLOBs, NCLOBs, and BFILEs

OCCI methods that operate on BLOBs, CLOBs, NCLOBs, and BFILEs are as follows:

- To modify persistent LOBs, see Table 13–16
- To read or examine LOB values, see Table 13–17
- For read only methods on external LOBs (BFILEs), see Table 13–18
- Other LOB OCCI methods are described in Table 13–19
- To open and close LOBs, see Table 13–20

OCCI Methods to Modify Persistent LOB (BLOB, CLOB, and NCLOB) Values

Table 13–16 OCCI Clob and Blob Methods to Modify Persistent LOB (BLOB, CLOB, and NCLOB) Values

Function/Procedure	Description
Blob/Clob.append()	Appends CLOB or BLOB value to another LOB.
Blob/Clob.copy()	Copies all or part of a CLOB or BLOB to another LOB.
Blob/Clob.copy()	Loads BFILE data into a persistent LOB.
Blob/Clob.trim()	Truncates a CLOB or BLOB.
Blob/Clob.write()	Writes data from a buffer into a LOB, overwriting existing data.

OCCI Methods to Read or Examine Persistent LOB and BFILE Values

Table 13–17 OCCI Blob/Clob/Bfile Methods to Read or Examine persistent LOB and BFILE Values

Function/Procedure	Description
Blob/Clob.getChunkSize()	Gets the chunk size used when reading and writing. This works on persistent LOBs and does not apply to external LOBs (BFILEs).
Blob/Clob.getOptions()	Obtains settings for existing and newly created LOBs.
Blob/Clob.length()	Returns the length of a LOB or a BFILE.
Blob/Clob.read()	Reads a specified portion of a non-NULL LOB or a BFILE into a buffer.
Blob/Clob.setOptions()	Enables LOB settings for existing and newly created LOBs.

OCCI Read-Only Methods for BFILEs

Table 13–18 OCCI Read-Only Methods for BFILES

Function/Procedure	Description
`Bfile.close()`	Closes an open `BFILE`.
`Bfile.fileExists()`	Checks whether a `BFILE` exists.
`Bfile.getFileName()`	Returns the name of a `BFILE`.
`Bfile.getDirAlias()`	Gets the directory object name.
`Bfile.isOpen()`	Checks whether a `BFILE` is open.
`Bfile.open()`	Opens a `BFILE`.

Other OCCI LOB Methods

Table 13–19 Other OCCI LOB Methods

Methods	Description
`Clob/Blob/Bfile.operator=()`	Assigns one LOB locator to another. Use = or the copy constructor.
`Clob.getCharSetForm()`	Returns the character set form of a LOB.
`Clob.getCharSetId()`	Returns the character set ID of a LOB.
`Bfile.setName()`	Sets the name of a `BFILE`.
`Clob/Blob/Bfile.operator==()`	Checks whether two LOB refer to the same LOB.
`Clob/Blob/Bfile.isInitialized()`	Checks whether a LOB is initialized.

OCCI Methods to Open and Close Internal and External LOBs

Table 13–20 OCCI Methods to Open and Close Internal and External LOBs

Function/Procedure	Description
`Clob/Blob/Bfile.Open()`	Opens a LOB
`Clob/Blob/Bfile.isOpen()`	Sees if a LOB is open
`Clob/Blob/Bfile.Close()`	Closes a LOB

Using C/C++ (Pro*C) to Work With LOBs

You can make changes to an entire persistent LOB, or to pieces of the beginning, middle or end of a LOB by using embedded SQL. You can access both internal and external LOBs for read purposes, and you can *write* to persistent LOBs.

Embedded SQL statements allow you to access data stored in `BLOB`s, `CLOB`s, `NCLOB`s, and `BFILE`s. These statements are listed in the following tables, and are discussed in greater detail later in the chapter.

> **See Also:** *Pro*C/C++ Programmer's Guide* for detailed documentation, including syntax, host variables, host variable types and example code.

First Provide an Allocated Input Locator Pointer That Represents LOB

Unlike locators in PL/SQL, locators in Pro*C/C++ are mapped to locator pointers which are then used to refer to the LOB or BFILE value.

To successfully complete an embedded SQL LOB statement you must do the following:

1. Provide an allocated input locator pointer that represents a LOB that exists in the database tablespaces or external file system *before* you run the statement.
2. SELECT a LOB locator into a LOB locator pointer variable.
3. Use this variable in the embedded SQL LOB statement to access and manipulate the LOB value.

See Also: APIs for supported LOB operations are described in detail in:

- Chapter 19, "Operations Specific to Persistent and Temporary LOBs"
- Chapter 22, "Using LOB APIs"
- Chapter 21, "LOB APIs for BFILE Operations"

Pro*C/C++ Statements That Operate on BLOBs, CLOBs, NCLOBs, and BFILEs

Pro*C/C++ statements that operate on BLOBs, CLOBs, and NCLOBs are listed in the following tables:

- To modify persistent LOBs, see Table 13–21
- To read or examine LOB values, see Table 13–22
- To create or free temporary LOB, or check if Temporary LOB exists, see Table 13–23
- To operate close and 'see if file exists' functions on BFILEs, see Table 13–24
- To operate on LOB locators, see Table 13–25
- For LOB buffering, see Table 13–26
- To open or close LOBs or `BFILE`s, see Table 13–27

Pro*C/C++ Embedded SQL Statements to Modify Persistent LOB Values

Table 13–21 Pro*C/C++: Embedded SQL Statements to Modify Persistent LOB Values

Statement	Description
`APPEND`	Appends a LOB value to another LOB.
`COPY`	Copies all or a part of a LOB into another LOB.
`ERASE`	Erases part of a LOB, starting at a specified offset.
`LOAD FROM FILE`	Loads `BFILE` data into a persistent LOB at a specified offset.
`TRIM`	Truncates a LOB.
`WRITE`	Writes data from a buffer into a LOB at a specified offset.
`WRITE APPEND`	Writes data from a buffer into a LOB at the end of the LOB.

Pro*C/C++ Embedded SQL Statements for Introspection of LOBs

*Table 13–22 Pro*C/C++: Embedded SQL Statements for Introspection of LOBs*

Statement	Description
DESCRIBE [CHUNKSIZE]	Gets the chunk size used when writing. This works for persistent LOBs only. It does not apply to external LOBs (BFILEs).
DESCRIBE [LENGTH]	Returns the length of a LOB or a BFILE.
READ	reads a specified portion of a non-NULL LOB or a BFILE into a buffer.

Pro*C/C++ Embedded SQL Statements for Temporary LOBs

*Table 13–23 Pro*C/C++: Embedded SQL Statements for Temporary LOBs*

Statement	Description
CREATE TEMPORARY	Creates a temporary LOB.
DESCRIBE [ISTEMPORARY]	Sees if a LOB locator refers to a temporary LOB.
FREE TEMPORARY	Frees a temporary LOB.

Pro*C/C++ Embedded SQL Statements for BFILEs

*Table 13–24 Pro*C/C++: Embedded SQL Statements for BFILES*

Statement	Description
FILE CLOSE ALL	Closes all open BFILEs.
DESCRIBE [FILEEXISTS]	Checks whether a BFILE exists.
DESCRIBE [DIRECTORY,FILENAME]	Returns the directory object name and filename of a BFILE.

Pro*C/C++ Embedded SQL Statements for LOB Locators

*Table 13–25 Pro*C/C++ Embedded SQL Statements for LOB Locators*

Statement	Description
ASSIGN	Assigns one LOB locator to another.
FILE SET	Sets the directory object name and filename of a BFILE in a locator.

Pro*C/C++ Embedded SQL Statements for LOB Buffering

*Table 13–26 Pro*C/C++ Embedded SQL Statements for LOB Buffering*

Statement	Description
DISABLE BUFFERING	Disables the use of the buffering subsystem.
ENABLE BUFFERING	Uses the LOB buffering subsystem for subsequent reads and writes of LOB data.
FLUSH BUFFER	Flushes changes made to the LOB buffering subsystem to the database (server)

Pro*C/C++ Embedded SQL Statements to Open and Close LOBs

Table 13–27 Pro*C/C++ Embedded SQL Statements to Open and Close Persistent LOBs and External LOBs (BFILEs)

Statement	Description
`OPEN`	Opens a LOB or `BFILE`.
`DESCRIBE [ISOPEN]`	Sees if a LOB or `BFILE` is open.
`CLOSE`	Closes a LOB or `BFILE`.

Using COBOL (Pro*COBOL) to Work With LOBs

You can make changes to an entire persistent LOB, or to pieces of the beginning, middle or end of it by using embedded SQL. You can access both internal and external LOBs for read purposes, and you can also *write* to persistent LOBs.

Embedded SQL statements allow you to access data stored in `BLOB`s, `CLOB`s, `NCLOB`s, and `BFILE`s. These statements are listed in the following tables, and are discussed in greater detail later in the manual.

First Provide an Allocated Input Locator Pointer That Represents LOB

Unlike locators in PL/SQL, locators in Pro*COBOL are mapped to locator pointers which are then used to refer to the LOB or BFILE value. For the successful completion of an embedded SQL LOB statement you must perform the following:

1. Provide an *allocated* input locator pointer that represents a LOB that exists in the database tablespaces or external file system *before* you run the statement.
2. SELECT a LOB locator into a LOB locator pointer variable
3. Use this variable in an embedded SQL LOB statement to access and manipulate the LOB value.

> **See Also:** APIs for supported LOB operations are described in detail in:
>
> - Chapter 19, "Operations Specific to Persistent and Temporary LOBs"
> - Chapter 22, "Using LOB APIs"
> - Chapter 21, "LOB APIs for BFILE Operations"

Where the Pro*COBOL interface does not supply the required functionality, you can call OCI using C. Such an example is not provided here because such programs are operating system dependent.

> **See Also:** *Pro*COBOL Programmer's Guide* for detailed documentation, including syntax, host variables, host variable types, and example code.

Pro*COBOL Statements That Operate on BLOBs, CLOBs, NCLOBs, and BFILEs

The following Pro*COBOL statements operate on BLOBs, CLOBs, NCLOBs, and BFILEs:

- To modify persistent LOBs, see Table 13–28

- To read or examine internal and external LOB values, see Table 13–29
- To create or free temporary LOB, or check LOB locator, see Table 13–30
- To operate close and 'see if file exists' functions on BFILEs, see Table 13–31
- To operate on LOB locators, see Table 13–32
- For LOB buffering, see Table 13–33
- To open or close persistent LOBs or BFILEs, see Table 13–34

Pro*COBOL Embedded SQL Statements to Modify Persistent LOB Values

Table 13–28 Pro*COBOL Embedded SQL Statements to Modify LOB Values

Statement	Description
APPEND	Appends a LOB value to another LOB.
COPY	Copies all or part of a LOB into another LOB.
ERASE	Erases part of a LOB, starting at a specified offset.
LOAD FROM FILE	Loads BFILE data into a persistent LOB at a specified offset.
TRIM	Truncates a LOB.
WRITE	Writes data from a buffer into a LOB at a specified offset
WRITE APPEND	Writes data from a buffer into a LOB at the end of the LOB.

Pro*COBOL Embedded SQL Statements for Introspection of LOBs

Table 13–29 Pro*COBOL Embedded SQL Statements for Introspection of LOBs

Statement	Description
DESCRIBE [CHUNKSIZE]	Gets the Chunk size used when writing.
DESCRIBE [LENGTH]	Returns the length of a LOB or a BFILE.
READ	Reads a specified portion of a non-NULL LOB or a BFILE into a buffer.

Pro*COBOL Embedded SQL Statements for Temporary LOBs

Table 13–30 Pro*COBOL Embedded SQL Statements for Temporary LOBs

Statement	Description
CREATE TEMPORARY	Creates a temporary LOB.
DESCRIBE [ISTEMPORARY]	Sees if a LOB locator refers to a temporary LOB.
FREE TEMPORARY	Frees a temporary LOB.

Pro*COBOL Embedded SQL Statements for BFILEs

*Table 13–31 Pro*COBOL Embedded SQL Statements for BFILES*

Statement	Description
`FILE CLOSE ALL`	Closes all open `BFILE`s.
`DESCRIBE [FILEEXISTS]`	Checks whether a `BFILE` exists.
`DESCRIBE [DIRECTORY, FILENAME]`	Returns the directory object name and filename of a `BFILE`.

Pro*COBOL Embedded SQL Statements for LOB Locators

*Table 13–32 Pro*COBOL Embedded SQL Statements for LOB Locator Statements*

Statement	Description
`ASSIGN`	Assigns one LOB locator to another.
`FILE SET`	Sets the directory object name and filename of a `BFILE` in a locator.

Pro*COBOL Embedded SQL Statements for LOB Buffering

*Table 13–33 Pro*COBOL Embedded SQL Statements for LOB Buffering*

Statement	Description
`DISABLE BUFFERING`	Disables the use of the buffering subsystem.
`ENABLE BUFFERING`	Uses the LOB buffering subsystem for subsequent reads and writes of LOB data.
`FLUSH BUFFER`	Flushes changes made to the LOB buffering subsystem to the database (server)

Pro*COBOL Embedded SQL Statements for Opening and Closing LOBs and BFILEs

*Table 13–34 Pro*COBOL Embedded SQL Statements for Opening and Closing Persistent LOBs and BFILEs*

Statement	Description
`OPEN`	Opens a LOB or `BFILE`.
`DESCRIBE [ISOPEN]`	Sees if a LOB or `BFILE` is open.
`CLOSE`	Closes a LOB or `BFILE`.

Using Java (JDBC) to Work With LOBs

You can perform the following tasks on LOBs with Java (JDBC):

- Modifying Internal Persistent LOBs Using Java
- Reading Internal Persistent LOBs and External LOBs (BFILEs) With Java
- Calling DBMS_LOB Package from Java (JDBC)
- Referencing LOBs Using Java (JDBC)
- Create and Manipulate Temporary LOBs and Store Them in Tables as Permanent LOBs. See JDBC Temporary LOB APIs

Modifying Internal Persistent LOBs Using Java

You can make changes to an entire persistent LOB, or to pieces of the beginning, middle, or end of a persistent LOB in Java by means of the JDBC API using the classes:

- `oracle.sql.BLOB`
- `oracle.sql.CLOB`

These classes implement `java.sql.Blob` and `java.sql.Clob` interfaces according to the JDBC 3.0 specification, which has methods for LOB modification. They also include legacy Oracle proprietary methods for LOB modification. These legacy methods are marked as deprecated.

Starting in Oracle Database Release 11.1, the minimum supported version of the JDK is JDK5. To use JDK5, place `ojdbc5.jar` in your `CLASSPATH`. To use JDK6, place `ojdbc6.jar` in your `CLASSPATH`. `ojdbc5.jar` supports the JDBC 3.0 specification and `ojdbc6.jar` supports the JDBC4.0 specification which is new with JDK6.

Reading Internal Persistent LOBs and External LOBs (BFILEs) With Java

With JDBC you can use Java to read both internal persistent LOBs and external LOBs (`BFILE`s).

BLOB, CLOB, and BFILE Classes

- `BLOB` and `CLOB` Classes: In JDBC theses classes provide methods for performing operations on large objects in the database including `BLOB` and `CLOB` data types.
- `BFILE` Class: In JDBC this class provides methods for performing operations on `BFILE` data in the database.

The `BLOB`, `CLOB`, and `BFILE` classes encapsulate LOB locators, so you do not deal with locators but instead use methods and properties provided to perform operations and get state information.

Calling DBMS_LOB Package from Java (JDBC)

Any LOB functionality not provided by these classes can be accessed by a call to the PL/SQL `DBMS_LOB` package. This technique is used repeatedly in the examples throughout this manual.

LOB Prefetching to Improve Performance

The number of server round trips can be reduced by prefetching part of the data and metadata (length and chunk size) along with the LOB locator during the fetch. The `SELECT` parse, execution, and fetch occurs in one round trip. For large LOBs (larger than five times the prefetch size) less improvement is seen.

To configure the prefetch size, a connection property, `oracle.jdbc.defaultLobPrefetchSize`, defined as a constant in `oracle.jdbc.OracleConnection` can be used. Values can be -1 to disable prefetching, 0 to enable prefetching for metadata only, or any value greater than 0 which represents the number of bytes for `BLOB`s and characters for `CLOB`s, to be prefetched along with the locator during fetch operations.

You can change the prefetch size for a particular statement by using a method defined in `oracle.jdbc.OracleStatement`:

```
void setLobPrefetchSize(int size) throws SQLException;
```

The statement level setting overrides the setting at the connection level. This setting can also be overriden at the column level through the extended `defineColumnType` method, where the size represents the number of bytes (or characters for `CLOB`) to prefetch. The possible values are the same as for the connection property. The type must be set to `OracleTypes.CLOB` for a `CLOB` column and `OracleTypes.BLOB` for a `BLOB` column. This method throws `SQLException` if the value is less than -1. To complement the statement there is in `oracle.jdbc.OracleStatement`:

```
int getLobPrefetchSize();
```

Zero-Copy Input/Output for SecureFiles to Improve Performance

To improve the performance of SecureFiles, there is a Zero-copy Input/Output protocol on the server that is only available to network clients that support the new Net NS Data transfer protocol.

To determine if a LOB is a SecureFiles or not, use the method

```
public boolean isSecureFile() throws SQLException
```

If it is a SecureFiles, `TRUE` is returned.

Use this thin connection property to disable (by setting to `FALSE`) the Zero-copy Input/Output protocol:

```
oracle.net.useZeroCopyIO
```

Zero-Copy Input/Output on the Server

Oracle Net Services is now able to use data buffers provided by the users of Oracle Net Services without transferring the data into or out of its local buffers. The network buffers (at the NS layer) are bypassed and internal lob buffers are directly written on the network. The same applies to buffer reads.

This feature is only available to network clients that support the new NS Data packet (this is negotiated during the NS handshake). The thin driver supports the new NS protocol so that the server can use the zero-copy protocol and JavaNet exposes the zero-copy IO mechanism to the upper layer so that data copies are no longer required in the thin driver code.

Zero-Copy Input/Output in the JDBC Thin Driver

When you call the `BLOB.getBytes(long pos, int length, byte[] buffer)` API, the buffer provided is used at the JavaNet layer to read the bytes from the socket. The data is retrieved in one single round trip. Similarly, during a write operation, when you call `BLOB.setBytes(long pos, byte[] bytes)`, the buffer is directly written on the network at the JavaNet layer. So the data is written in one single round trip. The user buffer is sent as a whole.

JDBC-OCI Driver Considerations

The JDBC-OCI driver supports Zero-copy Input/Output in the server and in the network layer.

Referencing LOBs Using Java (JDBC)

You can get a reference to any of the preceding LOBs in the following two ways:

- As a column of an `OracleResultSet`
- As an `OUT` type PL/SQL parameter from an `OraclePreparedStatement`

Using OracleResultSet: BLOB and CLOB Objects Retrieved

When BLOB and CLOB objects are retrieved as a part of an OracleResultSet, these objects represent LOB locators of the currently selected row.

If the current row changes due to a move operation, for example, rset.next(), then the retrieved locator still refers to the original LOB row.

To retrieve the locator for the most current row, you must call getBLOB(), getCLOB(), or getBFILE() on the OracleResultSet each time a move operation is made depending on whether the instance is a BLOB, CLOB or BFILE.

JDBC Syntax References and Further Information

For further JDBC syntax and information about using JDBC with LOBs:

See Also:

- *Oracle Database JDBC Developer's Guide*,for detailed documentation, including parameters, parameter types, return values, and example code.
- http://www.oracle.com/technology/

JDBC Methods for Operating on LOBs

The following JDBC methods operate on BLOBs, CLOBs, and BFILEs:

- BLOBs:
  - To modify BLOB values, see Table 13–35
  - To read or examine BLOB values, see Table 13–36
  - For BLOB buffering, see Table 13–37
  - Temporary BLOBs: Creating, checking if BLOB is open, and freeing. See Table 13–45
  - Opening, closing, and checking if BLOB is open, see Table 13–45
  - Truncating BLOBs, see Table 13–48
  - BLOB streaming API, see Table 13–50
- CLOBs:
  - To read or examine CLOB values, see Table 13–39
  - For CLOB buffering, see Table 13–40
  - To modify CLOBs, see Table 13–50
- Temporary CLOBs:
  - Opening, closing, and checking if CLOB is open, see Table 13–46
  - Truncating CLOBs, see Table 13–49
  - CLOB streaming API, see Table 13–51
- BFILEs:
  - To read or examine BFILEs, see Table 13–41
  - For BFILE buffering, see Table 13–42
  - Opening, closing, and checking if BFILE is open, see Table 13–47

- BFILE streaming API, see Table 13–52

JDBC oracle.sql.BLOB Methods to Modify BLOB Values

Table 13–35 JDBC oracle.sql.BLOB Methods To Modify BLOB Values

Method	Description
`int setBytes(long, byte[])`	Inserts the byte array into the `BLOB`, starting at the given offset

JDBC oracle.sql.BLOB Methods to Read or Examine BLOB Values

Table 13–36 JDBC oracle.sql.BLOB Methods to Read or Examine BLOB Values

Method	Description
`byte[] getBytes(long, int)`	Gets the contents of the LOB as an array of bytes, given an offset
`long position(byte[],long)`	Finds the given byte array within the LOB, given an offset
`long position(Blob,long)`	Finds the given `BLOB` within the LOB
`public boolean equals(java.lang.Object)`	Compares this LOB with another. Compares the LOB locators.
`public long length()`	Returns the length of the LOB
`public int getChunkSize()`	Returns the `ChunkSize` of the LOB

JDBC oracle.sql.BLOB Methods and Properties for BLOB Buffering

Table 13–37 JDBC oracle.sql.BLOB Methods and Properties for BLOB Buffering

Method	Description
`public java.io.InputStream getBinaryStream())`	Streams the LOB as a binary stream
`public java.io.OutputStream setBinaryStream()`	Retrieves a stream that can be used to write to the `BLOB` value that this `Blob` object represents

JDBC oracle.sql.CLOB Methods to Modify CLOB Values

Table 13–38 JDBC oracle.sql.CLOB Methods to Modify CLOB Values

Method	Description
`int setString(long, java.lang.String)`	JDBC 3.0: Writes the given Java String to the `CLOB` value that this `Clob` object designates at the position `pos`.
`int putChars(long, char[])`	Inserts the character array into the LOB, starting at the given offset

JDBC oracle.sql.CLOB Methods to Read or Examine CLOB Value

Table 13–39 JDBC oracle.sql.CLOB Methods to Read or Examine CLOB Values

Method	Description
`java.lang.String getSubString(long, int)`	Returns a substring of the LOB as a string
`int getChars(long, int, char[])`	Reads a subset of the LOB into a character array
`long position(java.lang.String, long)`	Finds the given String within the LOB, given an offset
`long position(oracle.jdbc2.Clob, long)`	Finds the given `CLOB` within the LOB, given an offset
`long length()`	Returns the length of the LOB
`int getChunkSize()`	Returns the `ChunkSize` of the LOB

JDBC oracle.sql.CLOB Methods and Properties for CLOB Buffering

Table 13–40 JDBC oracle.sql.CLOB Methods and Properties for CLOB Buffering

Method	Description
`java.io.InputStream getAsciiStream()`	Implements the `Clob` interface method. Gets the `CLOB` value designated by this `Clob` object as a stream of ASCII bytes
`java.io.OutputStream setAsciiStream(long pos)`	JDBC 3.0: Retrieves a stream to be used to write ASCII characters to the `CLOB` value that this `Clob` object represents, starting at position `pos`
`java.io.Reader getCharacterStream()`	Reads the `CLOB` as a character stream
`java.io.Writer setCharacterStream(long pos)`	JDBC 3.0: Retrieves a stream to be used to write Unicode characters to the `CLOB` value that this `Clob` object represents, starting at position `pos`

JDBC oracle.sql.BFILE Methods to Read or Examine External LOB (BFILE) Values

Table 13–41 JDBC oracle.sql.BFILE Methods to Read or Examine External LOB (BFILE) Values

Method	Description
`byte[] getBytes(long, int)`	Gets the contents of the `BFILE` as an array of bytes, given an offset
`int getBytes(long, int, byte[])`	Reads a subset of the `BFILE` into a byte array
`long position(oracle.sql.BFILE, long)`	Finds the first appearance of the given `BFILE` contents within the LOB, from the given offset
`long position(byte[], long)`	Finds the first appearance of the given byte array within the `BFILE`, from the given offset
`long length()`	Returns the length of the `BFILE`
`boolean fileExists()`	Checks if the operating system file referenced by this `BFILE` exists
`public void openFile()`	Opens the operating system file referenced by this `BFILE`
`public void closeFile()`	Closes the operating system file referenced by this `BFILE`

Table 13–41 (Cont.) JDBC oracle.sql.BFILE Methods to Read or Examine External LOB (BFILE) Values

Method	Description
`public boolean isFileOpen()`	Checks if this `BFILE` is open
`public java.lang.String getDirAlias()`	Gets the directory object name for this `BFILE`
`public java.lang.String getName()`	Gets the file name referenced by this `BFILE`

JDBC oracle.sql.BFILE Methods and Properties for BFILE Buffering

Table 13–42 JDBC oracle.sql.BFILE Methods and Properties for BFILE Buffering

Method	Description
`public java.io.InputStream getBinaryStream()`	Reads `the BFILE` as a binary stream

JDBC Temporary LOB APIs

Oracle Database JDBC drivers contain APIs to create and close temporary LOBs. These APIs can replace workarounds that use the following procedures from the `DBMS_LOB` PL/SQL package in prior releases:

- `DBMS_LOB.createTemporary()`
- `DBMS_LOB.isTemporary()`
- `DBMS_LOB.freeTemporary()`

Table 13–43 JDBC: Temporary BLOB APIs

Methods	Description
`public static BLOB createTemporary(Connection conn, boolean cache, int duration) throws SQLException`	Creates a temporary `BLOB`
`public static boolean isTemporary(BLOB blob) throws SQLException`	Checks if the specified `BLOB` locator refers to a temporary `BLOB`
`public boolean isTemporary() throws SQLException`	Checks if the current `BLOB` locator refers to a temporary `BLOB`
`public static void freeTemporary(BLOB temp_blob) throws SQLException`	Frees the specified temporary `BLOB`
`public void freeTemporary() throws SQLException`	Frees the temporary `BLOB`

Table 13–44 JDBC: Temporary CLOB APIs

Methods	Description
`public static CLOB createTemporary(Connection conn, boolean cache, int duration) throws SQLException`	Creates a temporary `CLOB`
`public static boolean isTemporary(CLOB clob) throws SQLException`	Checks if the specified `CLOB` locator refers to a temporary `CLOB`

Table 13–44 (Cont.) JDBC: Temporary CLOB APIs

Methods	Description
`public boolean isTemporary() throws SQLException`	Checks if the current `CLOB` locator refers to a temporary `CLOB`
`public static void freeTemporary(CLOB temp_clob) throws SQLException`	Frees the specified temporary `CLOB`
`public void freeTemporary() throws SQLException`	Frees the temporary `CLOB`

JDBC: Opening and Closing LOBs

`oracle.sql.CLOB` class is the Oracle JDBC driver implementation of standard JDBC `java.sql.Clob` interface. Table 13–44 lists the Oracle extension APIs in `oracle.sql.CLOB` for accessing temporary CLOBs.

Oracle Database JDBC drivers contain APIs to explicitly open and close LOBs. These APIs replace previous techniques that use `DBMS_LOB.open()` and `DBMS_LOB.close()`.

JDBC: Opening and Closing BLOBs

`oracle.sql.BLOB` class is the Oracle JDBC driver implementation of standard JDBC java.sql.Blob interface. Table 13–45 lists the Oracle extension APIs in `oracle.sql.BLOB` that open and close BLOBs.

Table 13–45 JDBC: Opening and Closing BLOBs

Methods	Description
`public void open(int mode) throws SQLException`	Opens the `BLOB`
`public boolean isOpen() throws SQLException`	Sees if the `BLOB` is open
`public void close() throws SQLException`	Closes the `BLOB`

Opening the BLOB Using JDBC

To open a `BLOB`, your JDBC application can use the open method as defined in `oracle.sql.BLOB` class as follows:

```
/**
 * Open a BLOB in the indicated mode. Valid modes include MODE_READONLY,
 * and MODE_READWRITE. It is an error to open the same LOB twice.
 */
public void open (int mode) throws SQLException
```

Possible values of the mode parameter are:

```
public static final int MODE_READONLY
public static final int MODE_READWRITE
```

Each call to open opens the BLOB. For example:

```
BLOB blob = ...
blob.open (BLOB.MODE_READWRITE);
```

Checking If the BLOB Is Open Using JDBC

To see if a `BLOB` is opened, your JDBC application can use the `isOpen` method defined in oracle.sql.BLOB. The return Boolean value indicates whether the `BLOB` has been previously opened or not. The `isOpen` method is defined as follows:

```
/**
 * Check whether the BLOB is opened.
 * @return true if the LOB is opened.
 */
 public boolean isOpen () throws SQLException
```

The usage example is:

```
BLOB blob = ...
// See if the BLOB is opened
boolean isOpen = blob.isOpen ();
```

Closing the BLOB Using JDBC

To close a BLOB, your JDBC application can use the close method defined in `oracle.sql.BLOB`. The close API is defined as follows:

```
/**
  * Close a previously opened BLOB.
  */
public void close () throws SQLException
```

The usage example is:

```
BLOB blob = ...
// close the BLOB
blob.close ();
```

JDBC: Opening and Closing CLOBs

Class `oracle.sql.CLOB` is the Oracle JDBC driver implementation of the standard JDBC `java.sql.Clob` interface. Table 13–46 lists the Oracle extension APIs in `oracle.sql.CLOB` to open and close `CLOB`s.

Table 13–46 JDBC: Opening and Closing CLOBs

Methods	Description
`public void open(int mode) throws SQLException`	Open the `CLOB`
`public boolean isOpen() throws SQLException`	See if the `CLOB` is opened
`public void close() throws SQLException`	Close the `CLOB`

Opening the CLOB Using JDBC

To open a `CLOB`, your JDBC application can use the open method defined in `oracle.sql.CLOB` class as follows:

```
/**
 * Open a CLOB in the indicated mode. Valid modes include MODE_READONLY,
 * and MODE_READWRITE. It is an error to open the same LOB twice.
 */
public void open (int mode) throws SQLException
```

The possible values of the mode parameter are:

```
public static final int MODE_READONLY
public static final int MODE_READWRITE
```

Each call to open opens the `CLOB`. For example,

```
CLOB clob = ...
clob.open (CLOB.MODE_READWRITE);
```

Checking If the CLOB Is Open Using JDBC

To see if a CLOB is opened, your JDBC application can use the isOpen method defined in oracle.sql.CLOB. The return Boolean value indicates whether the CLOB has been previously opened or not. The isOpen method is defined as follows:

```
/**
  * Check whether the CLOB is opened.
  * @return true if the LOB is opened.
  */
public boolean isOpen () throws SQLException
```

The usage example is:

```
CLOB clob = ...
 // See if the CLOB is opened
 boolean isOpen = clob.isOpen ();
```

Closing the CLOB Using JDBC

To close a CLOB, the JDBC application can use the close method defined in oracle.sql.CLOB. The close API is defined as follows:

```
/**
* Close a previously opened CLOB.
*/
public void close () throws SQLException
```

The usage example is:

```
CLOB clob = ...
// close the CLOB
clob.close ();
```

JDBC: Opening and Closing BFILEs

oracle.sql.BFILE class wraps the database BFILE object. Table 13–47 lists the Oracle extension APIs in oracle.sql.BFILE for opening and closing BFILEs.

Table 13–47 JDBC API Extensions for Opening and Closing BFILEs

Methods	Description
public void open() throws SQLException	Opens the BFILE
public void open(int mode) throws SQLException	Opens the BFILE
public boolean isOpen() throws SQLException	Checks if the BFILE is open
public void close() throws SQLException	Closes the BFILE

Opening BFILEs

To open a BFILE, your JDBC application can use the OPEN method defined in oracle.sql.BFILE class as follows:

```
/**
 * Open a external LOB in the read-only mode. It is an error
 * to open the same LOB twice.
 */
public void open () throws SQLException

/**
 * Open a external LOB in the indicated mode. Valid modes include
```

```
 * MODE_READONLY only. It is an error to open the same
 * LOB twice.
 */
public void open (int mode) throws SQLException
```

The only possible value of the mode parameter is:

```
public static final int MODE_READONLY
```

Each call to open opens the BFILE. For example,

```
BFILE bfile = ...
bfile.open ();
```

Checking If the BFILE Is Open

To see if a BFILE is opened, your JDBC application can use the isOpen method defined in oracle.sql.BFILE. The return Boolean value indicates whether the BFILE has been previously opened or not. The isOpen method is defined as follows:

```
/**
 * Check whether the BFILE is opened.
 * @return true if the LOB is opened.
 */
public boolean isOpen () throws SQLException
```

The usage example is:

```
BFILE bfile = ...
// See if the BFILE is opened
boolean isOpen = bfile.isOpen ();
```

Closing the BFILE

To close a BFILE, your JDBC application can use the close method defined in oracle.sql.BFILE. The close API is defined as follows:

```
/**
 * Close a previously opened BFILE.
*/
public void close () throws SQLException
```

The usage example is --

```
BFILE bfile = ...
// close the BFILE
bfile.close ();
```

Usage Example (OpenCloseLob.java)

```
/*
 * This sample shows how to open/close BLOB and CLOB.
 */

// You must import the java.sql package to use JDBC
import java.sql.*;

// You must import the oracle.sql package to use oracle.sql.BLOB
import oracle.sql.*;

class OpenCloseLob
{
```

```
  public static void main (String args [])
       throws SQLException
  {
    // Load the Oracle JDBC driver
    DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());

    String url = "jdbc:oracle:oci8:@";
    try {
      String url1 = System.getProperty("JDBC_URL");
      if (url1 != null)
        url = url1;
    } catch (Exception e) {
      // If there is any security exception, ignore it
      // and use the default
    }

    // Connect to the database
    Connection conn =
      DriverManager.getConnection (url, "scott", "password");
    // It is faster when auto commit is off
    conn.setAutoCommit (false);

    // Create a Statement
    Statement stmt = conn.createStatement ();

    try
    {
      stmt.execute ("drop table basic_lob_table");
    }
    catch (SQLException e)
    {
      // An exception could be raised here if the table did not exist.
    }

// Create a table containing a BLOB and a CLOB
stmt.execute ("create table basic_lob_table (x varchar2 (30), b blob, c clob)");

// Populate the table
stmt.execute (
    "insert into basic_lob_table values"
    + " ('one', '0101010101010101010101010101', 'onetwothreefour')");

    // Select the lobs
    ResultSet rset = stmt.executeQuery ("select * from basic_lob_table");
    while (rset.next ())
    {
      // Get the lobs
      BLOB blob = (BLOB) rset.getObject (2);
      CLOB clob = (CLOB) rset.getObject (3);

      // Open the lobs
      System.out.println ("Open the lobs");
      blob.open (BLOB.MODE_READWRITE);
      clob.open (CLOB.MODE_READWRITE);

      // Check if the lobs are opened
      System.out.println ("blob.isOpen()="+blob.isOpen());
      System.out.println ("clob.isOpen()="+clob.isOpen());

      // Close the lobs
```

```
      System.out.println ("Close the lobs");
      blob.close ();
      clob.close ();

      // Check if the lobs are opened
      System.out.println ("blob.isOpen()="+blob.isOpen());
      System.out.println ("clob.isOpen()="+clob.isOpen());
    }

    // Close the ResultSet
    rset.close ();

    // Close the Statement
    stmt.close ();

    // Close the connection
    conn.close ();
  }
}
```

Truncating LOBs Using JDBC

Oracle Database JDBC drivers contain APIs to truncate persistent LOBs. These APIs replace previous techniques that used `DBMS_LOB.trim()`.

JDBC: Truncating BLOBs

`oracle.sql.BLOB` class is Oracle JDBC driver implementation of the standard JDBC `java.sql.Blob` interface. Table 13–48 lists the Oracle extension API in `oracle.sql.BLOB` that truncates BLOBs.

Table 13–48 JDBC: Truncating BLOBs

Methods	Description
`public void truncate(long newlen) throws SQLException`	Truncates the `BLOB`

The truncate API is defined as follows:

```
/**
*Truncate the value of the BLOB to the length you specify in the newlen parameter.
 * @param newlen the new length of the BLOB.
 */
public void truncate (long newlen) throws SQLException
```

The `newlen` parameter specifies the new length of the `BLOB`.

JDBC: Truncating CLOBs

`oracle.sql.CLOB` class is the Oracle JDBC driver implementation of standard JDBC `java.sql.Clob` interface. Table 13–49 lists the Oracle extension API in `oracle.sql.CLOB` that truncates `CLOB`s.

Table 13–49 JDBC: Truncating CLOBs

Methods	Description
`public void truncate(long newlen) throws SQLException`	Truncates the `CLOB`

The truncate API is defined as follows:

```
/**
*Truncate the value of the CLOB to the length you specify in the newlen parameter.
 * @param newlen the new length of the CLOB.
 */
public void truncate (long newlen) throws SQLException
```

The `newlen` parameter specifies the new length of the `CLOB`.

See: "Trimming LOB Data" on page 22-34, for an example.

JDBC BLOB Streaming APIs

The JDBC interface provided with the database includes LOB streaming APIs that enable you to read from or write to a LOB at the requested position from a Java stream.

The `oracle.sql.BLOB` class implements the standard JDBC `java.sql.Blob` interface. Table 13–50 lists BLOB Streaming APIs.

Table 13–50 JDBC: BLOB Streaming APIs

Methods	Description
`public java.io.OutputStream setBinaryStream (long pos) throws SQLException`	JDBC 3.0: Retrieves a stream that can be used to write to the `BLOB` value that this `Blob` object represents, starting at position `pos`
`public java.io.InputStream getBinaryStream() throws SQLException`	JDBC 3.0: Retrieves a stream that can be used to read the `BLOB` value that this `Blob` object represents, starting at the beginning
`public java.io.InputStream getBinaryStream(long pos) throws SQLException`	Oracle extension: Retrieves a stream that can be used to read the `BLOB` value that this `Blob` object represents, starting at position `pos`

These APIs are defined as follows:

```
/**
 * Write to the BLOB from a stream at the requested position.
 *
 * @param pos is the position data to be put.
 * @return a output stream to write data to the BLOB
 */
public java.io.OutputStream setBinaryStream(long pos) throws SQLException

/**
 * Read from the BLOB as a stream at the requested position.
 *
 * @param pos is the position data to be read.
 * @return a output stream to write data to the BLOB
 */
public java.io.InputStream getBinaryStream(long pos) throws SQLException
```

JDBC CLOB Streaming APIs

The `oracle.sql.CLOB` class is the Oracle JDBC driver implementation of standard JDBC `java.sql.Clob` interface. Table 13–51 lists the `CLOB` streaming APIs.

Table 13–51 JDBC: CLOB Streaming APIs

Methods	Description
`public java.io.OutputStream setAsciiStream (long pos) throws SQLException`	JDBC 3.0: Retrieves a stream to be used to write ASCII characters to the `CLOB` value that this `Clob` object represents, starting at position `pos`
`public java.io.Writer setCharacterStream (long pos) throws SQLException`	JDBC 3.0: Retrieves a stream to be used to write Unicode characters to the `CLOB` value that this `Clob` object represents, starting, at position `pos`
`public java.io.InputStream getAsciiStream() throws SQLException`	JDBC 3.0: Retrieves a stream that can be used to read ASCII characters from the `CLOB` value that this `Clob` object represents, starting at the beginning
`public java.io.InputStream getAsciiStream(long pos) throws SQLException`	Oracle extension: Retrieves a stream that can be used to read ASCII characters from the `CLOB` value that this `Clob` object represents, starting at position `pos`
`public java.io.Reader getCharacterStream() throws SQLException`	JDBC 3.0: Retrieves a stream that can be used to read Unicode characters from the `CLOB` value that this `Clob` object represents, starting at the beginning
`public java.io.Reader getCharacterStream(long pos) throws SQLException`	Oracle extension: Retrieves a stream that can be used to read Unicode characters from the `CLOB` value that this `Clob` object represents, starting at position `pos`

These APIs are defined as follows:

```
/**
  * Write to the CLOB from a stream at the requested position.
  * @param pos is the position data to be put.
  * @return a output stream to write data to the CLOB
  */
public java.io.OutputStream setAsciiStream(long pos) throws SQLException

/**
     * Write to the CLOB from a stream at the requested position.
     * @param pos is the position data to be put.
     * @return a output stream to write data to the CLOB
     */
  public java.io.Writer setCharacterStream(long pos) throws SQLException

    /**
     * Read from the CLOB as a stream at the requested position.
     * @param pos is the position data to be put.
     * @return a output stream to write data to the CLOB
     */
  public java.io.InputStream getAsciiStream(long pos) throws SQLException

   /**
    * Read from the CLOB as a stream at the requested position.
    * @param pos is the position data to be put.
    * @return a output stream to write data to the CLOB
    */
   public java.io.Reader getCharacterStream(long pos) throws SQLException
```

BFILE Streaming APIs

`oracle.sql.BFILE` class wraps the database `BFILE`s. Table 13–52 lists the Oracle extension APIs in `oracle.sql.BFILE` that reads `BFILE` content from the requested position.

Table 13–52 JDBC: BFILE Streaming APIs

Methods	Description
`public java.io.InputStream getBinaryStream(long pos) throws SQLException`	Reads from the `BFILE` as a stream

These APIs are defined as follows:

```
/**
 * Read from the BLOB as a stream at the requested position.
 *
 * @param pos is the position data to be read.
 * @return a output stream to write data to the BLOB
 */
public java.io.InputStream getBinaryStream(long pos) throws SQLException
```

JDBC BFILE Streaming Example (NewStreamLob.java)

```
/*
 * This sample shows how to read/write BLOB and CLOB as streams.
 */

import java.io.*;

// You must import the java.sql package to use JDBC
import java.sql.*;

// You must import the oracle.sql package to use oracle.sql.BLOB
import oracle.sql.*;

class NewStreamLob
{
  public static void main (String args [])  throws Exception
  {
    // Load the Oracle JDBC driver
    DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());

    String url = "jdbc:oracle:oci8:@";
    try {
      String url1 = System.getProperty("JDBC_URL");
      if (url1 != null)
        url = url1;
    } catch (Exception e) {
      // If there is any security exception, ignore it
      // and use the default
    }

    // Connect to the database
    Connection conn =
      DriverManager.getConnection (url, "scott", "password");
    // It is faster when auto commit is off
    conn.setAutoCommit (false);
```

```
  // Create a Statement
  Statement stmt = conn.createStatement ();

  try
  {
    stmt.execute ("drop table basic_lob_table");
  }
  catch (SQLException e)
  {
    // An exception could be raised here if the table did not exist.
  }

  // Create a table containing a BLOB and a CLOB
  stmt.execute (
       "create table basic_lob_table"
       + "(x varchar2 (30), b blob, c clob)");

  // Populate the table
  stmt.execute (
       "insert into basic_lob_table values"
       + "('one', '0101010101010101010101010101', 'onetwothreefour')");

  System.out.println ("Dumping lobs");

  // Select the lobs
  ResultSet rset = stmt.executeQuery ("select * from basic_lob_table");
  while (rset.next ())
  {
    // Get the lobs
    BLOB blob = (BLOB) rset.getObject (2);
    CLOB clob = (CLOB) rset.getObject (3);

    // Print the lob contents
    dumpBlob (conn, blob, 1);
    dumpClob (conn, clob, 1);

    // Change the lob contents
    fillClob (conn, clob, 11, 50);
    fillBlob (conn, blob, 11, 50);
  }
  rset.close ();

  System.out.println ("Dumping lobs again");

  rset = stmt.executeQuery ("select * from basic_lob_table");
  while (rset.next ())
  {
    // Get the lobs
    BLOB blob = (BLOB) rset.getObject (2);
    CLOB clob = (CLOB) rset.getObject (3);

    // Print the lobs contents
    dumpBlob (conn, blob, 11);
    dumpClob (conn, clob, 11);
  }
  // Close all resources
  rset.close();
  stmt.close();
  conn.close();
}
```

```
// Utility function to dump Clob contents
static void dumpClob (Connection conn, CLOB clob, long offset)
  throws Exception
{
  // get character stream to retrieve clob data
  Reader instream = clob.getCharacterStream(offset);

  // create temporary buffer for read
  char[] buffer = new char[10];

  // length of characters read
  int length = 0;

  // fetch data
  while ((length = instream.read(buffer)) != -1)
  {
    System.out.print("Read " + length + " chars: ");

    for (int i=0; i<length; i++)
      System.out.print(buffer[i]);
    System.out.println();
  }

  // Close input stream
  instream.close();
}

// Utility function to dump Blob contents
static void dumpBlob (Connection conn, BLOB blob, long offset)
  throws Exception
{
  // Get binary output stream to retrieve blob data
  InputStream instream = blob.getBinaryStream(offset);
  // Create temporary buffer for read
  byte[] buffer = new byte[10];
  // length of bytes read
  int length = 0;
  // Fetch data
  while ((length = instream.read(buffer)) != -1)
  {
    System.out.print("Read " + length + " bytes: ");

    for (int i=0; i<length; i++)
      System.out.print(buffer[i]+" ");
    System.out.println();
  }

  // Close input stream
  instream.close();
}

// Utility function to put data in a Clob
static void fillClob (Connection conn, CLOB clob, long offset, long length)
  throws Exception
{
  Writer outstream = clob.setCharacterStream(offset);

  int i = 0;
  int chunk = 10;
```

```
    while (i < length)
    {
      outstream.write("aaaaaaaaaa", 0, chunk);

      i += chunk;
      if (length - i < chunk)
         chunk = (int) length - i;
    }
    outstream.close();
  }

  // Utility function to put data in a Blob
  static void fillBlob (Connection conn, BLOB blob, long offset, long length)
    throws Exception
  {
    OutputStream outstream = blob.setBinaryStream(offset);

    int i = 0;
    int chunk = 10;

    byte [] data = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 };

    while (i < length)
    {
      outstream.write(data, 0, chunk);

      i += chunk;
      if (length - i < chunk)
         chunk = (int) length - i;
    }
    outstream.close();
  }
}
```

JDBC and Empty LOBs

An empty `BLOB` can be created from the following API from `oracle.sql.BLOB`:

```
public static BLOB empty_lob () throws SQLException
```

Similarly, the following API from `oracle.sql.CLOB` creates an empty `CLOB`:

```
public static CLOB empty_lob () throws SQLException
```

Empty LOB instances are created by JDBC drivers without making database round trips. Empty LOBs can be used in the following cases:

- set APIs of `PreparedStatement`
- update APIs of updatable result set
- attribute value of STRUCTs
- element value of ARRAYs

> **Note:** Empty LOBs are special marker LOBs but not real LOB values.

JDBC applications cannot read or write to empty LOBs created from the preceding APIs. An ORA-17098 "Invalid empty lob operation" results if your application attempts to read/write to an empty LOB.

Oracle Provider for OLE DB (OraOLEDB)

Oracle Provider for OLE DB (OraOLEDB) offers high performance and efficient access to Oracle data for OLE DB and ADO developers. Developers programming with COM, C++, or any COM client can use OraOLEDB to access Oracle databases.

OraOLEDB is an OLE DB provider for Oracle. It offers high performance and efficient access to Oracle data including LOBs, and also allows updates to certain LOB types.

The following LOB types are supported by OraOLEDB:

- For Persistent LOBs:

  `READ/WRITE` through the rowset.

- For BFILEs:

  READ-ONLY through the rowset.

- Temporary LOBs:

  Are not supported through the rowset.

See Also: *Oracle Provider for OLE DB Developer's Guide for Microsoft Windows*

Overview of Oracle Data Provider for .NET (ODP.NET)

Oracle Data Provider for .NET (ODP.NET) is an implementation of a data provider for the Oracle database. ODP.NET uses Oracle native APIs to offer fast and reliable access to Oracle data and features from any .NET application. ODP.NET also uses and inherits classes and interfaces available in the Microsoft .NET Framework Class Library. The ODP.NET supports the following LOBs as native data types with .NET: `BLOB`, `CLOB`, `NCLOB`, and `BFILE`.

COM and .NET are complementary development technologies. Microsoft recommends that developers use the .NET Framework rather than COM for new development.

See Also: *Oracle Data Provider for .NET Developer's Guide for Microsoft Windows*

14

Performance Guidelines

This chapter contains these topics:

LOB Performance Guidelines

This section describes performance guidelines for applications that use LOB data types.

Chunk Size

A chunk is one or more Oracle blocks. You can specify the chunk size for the LOB when creating the table that contains the LOB. This corresponds to the data size used by Oracle Database when accessing or modifying the LOB value. Part of the chunk is used to store system-related information and the rest stores the LOB value. The API you are using has a function that returns the amount of space used in the LOB chunk to store the LOB value. In PL/SQL use `DBMS_LOB.GETCHUNKSIZE`. In OCI, use `OCILobGetChunkSize()`. For SecureFiles, the usable data area of the tablespace block size is returned.

Performance Guidelines for Small BasicFiles LOBs

If most LOBs in your database tables are small in size—8K bytes or less—and only a few rows have LOBs larger than 8K bytes, then use the following guidelines to maximize database performance:

- Use `ENABLE STORAGE IN ROW`.
- Set the `DB_BLOCK_SIZE` initialization parameter to 8K bytes and use a chunk size of 8K bytes.
- See "LOB Storage Parameters" on page 11-4 information on tuning other parameters such as `CACHE`, `PCTVERSION`, and `CHUNK` for the LOB segment.

General Performance Guidelines for BasicFiles LOBs

Use the following guidelines to achieve maximum performance with BasicFiles LOBs:

- When Possible, Read/Write Large Data Chunks at a Time:

Because LOBs are big, you can obtain the best performance by reading and writing large pieces of a LOB value at a time. This helps in several respects:

a. If accessing the LOB from the client side and the client is at a different node than the server, then large reads/writes reduce network overhead.

b. If using the `NOCACHE` option, then each small read/write incurs an I/O. Reading/writing large quantities of data reduces the I/O.

c. Writing to the LOB creates a new version of the LOB chunk. Therefore, writing small amounts at a time incurs the cost of a new version for each small write. If logging is on, then the chunk is also stored in the redo log.

- Use LOB Buffering to Read/Write Small Chunks of Data:

  If you must read or write small pieces of LOB data on the client, then use LOB buffering — see `OCILobEnableBuffering()`, `OCILobDisableBuffering()`, `OCILobFlushBuffer()`, `OCILobWrite2()`, `OCILobRead2()`. Basically, turn on LOB buffering before reading/writing small pieces of LOB data.

  See Also: "LOB Buffering Subsystem" on page 12-1 for more information on LOB buffering.

- Use OCILobRead2() and OCILobWrite2() with Callback:

  So that data is streamed to and from the LOB. Ensure the length of the entire write is set in the `amount` parameter on input. Whenever possible, read and write in *multiples* of the LOB *chunk* size.

- Use a Checkout/Check-in Model for LOBs:

  LOBs are optimized for the following operations:

  - SQL `UPDATE` which replaces the entire LOB value
  - Copy the entire LOB data to the client, modify the LOB data on the client side, copy the entire LOB data back to the database. This can be done using `OCILobRead2()` and `OCILobWrite2()` with streaming.

- Commit changes frequently.

Temporary LOB Performance Guidelines

In addition to the guidelines described earlier under "LOB Performance Guidelines" on LOB performance in general, here are some guidelines for using temporary LOBs:

- Use a separate temporary tablespace for temporary LOB storage instead of the default system tablespace

  This avoids device contention when copying data from persistent LOBs to temporary LOBs.

  If you use the newly provided enhanced SQL semantics functionality in your applications, then there are many more temporary LOBs created silently in SQL and PL/SQL than before. Ensure that ***temporary tablespace*** for storing these temporary LOBs is ***large enough*** for your applications. In particular, these temporary LOBs are silently created when you use the following:

  - SQL functions on LOBs
  - PL/SQL built-in character functions on LOBs
  - Variable assignments from `VARCHAR2/RAW` to `CLOBs/BLOBs`, respectively.

  - Perform a LONG-to-LOB migration

- In PL/SQL, use NOCOPY to pass temporary LOB parameters by reference whenever possible.

  Refer to the *Oracle Database PL/SQL Language Reference,* for more information on passing parameters by reference and parameter aliasing.

- Take advantage of buffer cache on temporary LOBs.

  Temporary LOBs created with the CACHE parameter set to true move through the buffer cache. Otherwise temporary LOBs are read directly from, and written directly to, disk.

- For optimal performance, temporary LOBs use reference on read, copy on write semantics. When a temporary LOB locator is assigned to another locator, the physical LOB data is not copied. Subsequent READ operations using either of the LOB locators refer to the same physical LOB data. On the first WRITE operation after the assignment, the physical LOB data is copied in order to preserve LOB value semantics, that is, to ensure that each locator points to a unique LOB value. This performance consideration mainly applies to the PL/SQL and OCI environments.

  In PL/SQL, reference on read, copy on write semantics are illustrated as follows:

```
LOCATOR1 BLOB;
LOCATOR2 BLOB;
DBMS_LOB.CREATETEMPORARY (LOCATOR1,TRUE,DBMS_LOB.SESSION);

-- LOB data is not copied in this assignment operation:
LOCATOR2 := LOCATOR;
-- These read operations refer to the same physical LOB copy:
DBMS_LOB.READ(LOCATOR1, ...);
DBMS_LOB.GETLENGTH(LOCATOR2, ...);

-- A physical copy of the LOB data is made on WRITE:
DBMS_LOB.WRITE(LOCATOR2, ...);
```

  In OCI, to ensure value semantics of LOB locators and data, `OCILobLocatorAssign()` is used to copy temporary LOB locators and the LOB Data. `OCILobLocatorAssign()` does not make a round trip to the server. The physical temporary LOB copy is made when LOB updates happen in the same round trip as the LOB update API as illustrated in the following:

```
OCILobLocator *LOC1;
OCILobLocator *LOC2;
OCILobCreateTemporary(... LOC1, ... TRUE,OCI_DURATION_SESSION);

/* No round-trip is incurred in the following call. */
OCILobLocatorAssign(... LOC1, LOC2);

/* Read operations refer to the same physical LOB copy. */
OCILobRead2(... LOC1 ...)

/* One round-trip is incurred to make a new copy of the
 * LOB data and to write to the new LOB copy.
 */
OCILobWrite2(... LOC1 ...)

/* LOC2 does not see the same LOB data as LOC1. */
```

```
OCILobRead2(... LOC2 ...)
```

If LOB value semantics are not intended, then you can use C pointers to achieve reference semantics as illustrated in the following:

```
OCILobLocator *LOC1;
OCILobLocator *LOC2;
OCILobCreateTemporary(... LOC1, ... TRUE,OCI_DURATION_SESSION);

/* Pointer is copied. LOC1 and LOC2 refer to the same LOB data. */
LOC2 = LOC1;

/* Write to LOC2. */
OCILobWrite2(...LOC2...)

/* LOC1 sees the change made to LOC2. */
OCILobRead2(...LOC1...)
```

- Use OCI_OBJECT mode for temporary LOBs

  To improve the performance of temporary LOBs on LOB assignment, use `OCI_OBJECT` mode for `OCILobLocatorAssign()`. In `OCI_OBJECT` mode, the database tries to minimize the number of deep copies to be done. Hence, after `OCILobLocatorAssign()` is done on a source temporary LOB in `OCI_OBJECT` mode, the source and the destination locators point to the same LOB until any modification is made through either LOB locator.

- Free up temporary LOBs returned from SQL queries and PL/SQL programs

  In PL/SQL, C (OCI), Java and other programmatic interfaces, SQL query results or PL/SQL program executions return temporary LOBs for operation/function calls on LOBs. For example:

  ```
  SELECT substr(CLOB_Column, 4001, 32000) FROM ...
  ```

  If the query is executed in PL/SQL, then the returned temporary LOBs are automatically freed at the end of a PL/SQL program block. You can also explicitly free the temporary LOBs at any time. In OCI and Java, the returned temporary LOB must be explicitly freed.

  Without proper deallocation of the temporary LOBs returned from SQL queries, temporary tablespace is filled and you may observe performance degradation.

Performance Considerations for SQL Semantics and LOBs

Be aware of the following performance issues when using SQL semantics with LOBs:

- Ensure that your temporary tablespace is large enough to accommodate LOBs stored out-of-line. Persistent LOBs that are greater than approximately 4000 bytes in size are stored outside of the LOB column.
- When possible, free unneeded temporary LOB instances. Unless you explicitly free a temporary LOB instance, the LOB remains in existence while your application is executing. More specifically, the instance exists while the scope in which the LOB was declared is executing.

  See Also: Chapter 16, "SQL Semantics and LOBs" for details on SQL semantics support for LOBs.

Moving Data to LOBs in a Threaded Environment

There are two possible procedures that you can use to move data to LOBs in a threaded environment, one of which should be avoided.

Recommended Procedure

Note:

- There is no requirement to create an empty LOB in this procedure.
- You can use the `RETURNING` clause as part of the `INSERT/UPDATE` statement to return a locked LOB locator. This eliminates the need for doing a `SELECT-FOR-UPDATE`, as mentioned in step 3.

The recommended procedure is as follows:

1. `INSERT` an empty LOB, `RETURNING` the LOB locator.
2. Move data into the LOB using this locator.
3. `COMMIT`. This releases the ROW locks and makes the LOB data persistent.

Alternatively, you can insert more than 4000 bytes of data directly for the LOB columns or LOB attributes.

Procedure to Avoid

The following sequence requires a new connection when using a threaded environment, adversely affects performance, and is not recommended:

1. Create an empty (non-`NULL`) LOB
2. Perform `INSERT` using the empty LOB
3. `SELECT-FOR-UPDATE` of the row just entered
4. Move data into the LOB
5. `COMMIT`. This releases the `ROW` locks and makes the LOB data persistent.

LOB Access Statistics

After Oracle Database 10*g* Release 2, three session-level statistics specific to LOBs are available to users: LOB reads, LOB writes, and LOB writes unaligned. Session statistics are accessible through the `V$MYSTAT`, `V$SESSTAT`, and `V$SYSSTAT` dynamic performance views. To query these views, the user must be granted the privileges `SELECT_CATALOG_ROLE`, `SELECT ON SYS.V_$MYSTAT` view, and `SELECT ON SYS.V_$STATNAME` view.

LOB reads is defined as the number of LOB API read operations performed in the session/system. A single LOB API read may correspond to multiple physical/logical disk block reads.

LOB writes is defined as the number of LOB API write operations performed in the session/system. A single LOB API write may correspond to multiple physical/logical disk block writes.

LOB writes unaligned is defined as the number of LOB API write operations whose start offset or buffer size is not aligned to the internal chunk size of the LOB. Writes aligned to chunk boundaries are the most efficient write operations. The internal

chunk size of a LOB is available through the LOB API (for example, using PL/SQL, by `DBMS_LOB.GETCHUNKSIZE()`).

The following simple example demonstrates how LOB session statistics are updated as the user performs read/write operations on LOBs.

It is important to note that session statistics are aggregated across operations to all LOBs accessed in a session; the statistics are not separated or categorized by objects (that is, table, column, segment, object numbers, and so on).

In these examples, you reconnect to the database for each demonstration to clear the `V$MYSTAT`. This enables you to see how the lob statistics change for the specific operation you are testing, without the potentially obscuring effect of past LOB operations within the same session.

> **See also:** *Oracle Database Reference*, appendix E, "Statistics Descriptions"

Example of Retrieving LOB Access Statistics

This example was created for retrieving LOB access statistics.

```
rem
rem Set up the user
rem

CONNECT / AS SYSDBA;
SET ECHO ON;
GRANT SELECT_CATALOG_ROLE TO pm;
GRANT SELECT ON sys.v_$mystat TO pm;
GRANT SELECT ON sys.v_$statname TO pm;

rem
rem Create a simplified view for statistics queries
rem

CONNECT pm;
SET ECHO ON;

DROP VIEW mylobstats;
CREATE VIEW mylobstats
AS
SELECT  SUBSTR(n.name,1,20) name,
        m.value             value
FROM    v$mystat    m,
        v$statname  n
WHERE   m.statistic# = n.statistic#
    AND n.name LIKE 'lob%';

rem
rem Create a test table
rem

DROP TABLE t;
CREATE TABLE t (i NUMBER, c CLOB)
    lob(c) STORE AS (DISABLE STORAGE IN ROW);

rem
rem Populate some data
rem
rem This should result in unaligned writes, one for
```

```
rem each row/lob populated.
rem

CONNECT pm
SELECT * FROM mylobstats;
INSERT INTO t VALUES (1, 'a');
INSERT INTO t VALUES (2, rpad('a',4000,'a'));
COMMIT;
SELECT * FROM mylobstats;

rem
rem Get the lob length
rem
rem Computing lob length does not read lob data, no change
rem in read/write stats.
rem

CONNECT pm;
SELECT * FROM mylobstats;
SELECT LENGTH(c) FROM t;
SELECT * FROM mylobstats;

rem
rem Read the lobs
rem
rem Lob reads are performed, one for each lob in the table.
rem

CONNECT pm;
SELECT * FROM mylobstats;
SELECT * FROM t;
SELECT * FROM mylobstats;

rem
rem Read and manipulate the lobs (through temporary lobs)
rem
rem The use of complex operators like "substr()" results in
rem the implicit creation and use of temporary lobs. operations
rem on temporary lobs also update lob statistics.
rem

CONNECT pm;
SELECT * FROM mylobstats;
SELECT substr(c, length(c), 1) FROM t;
SELECT substr(c, 1, 1) FROM t;
SELECT * FROM mylobstats;

rem
rem Perform some aligned overwrites
rem
rem Only lob write statistics are updated because both the
rem byte offset of the write, and the size of the buffer
rem being written are aligned on the lob chunksize.
rem

CONNECT pm;
SELECT * FROM mylobstats;
DECLARE
    loc     CLOB;
    buf     LONG;
```

```
    chunk   NUMBER;
BEGIN
    SELECT c INTO loc FROM t WHERE i = 1
        FOR UPDATE;

    chunk := DBMS_LOB.GETCHUNKSIZE(loc);
    buf   := rpad('b', chunk, 'b');

    -- aligned buffer length and offset
    DBMS_LOB.WRITE(loc, chunk, 1, buf);
    DBMS_LOB.WRITE(loc, chunk, 1+chunk, buf);
    COMMIT;
END;
/
SELECT * FROM mylobstats;

rem
rem Perform some unaligned overwrites
rem
rem Both lob write and lob unaligned write statistics are
rem updated because either one or both of the write byte offset
rem and buffer size are unaligned with the lob's chunksize.
rem

CONNECT pm;
SELECT * FROM mylobstats;
DECLARE
    loc CLOB;
    buf LONG;
BEGIN
    SELECT c INTO loc FROM t WHERE i = 1
        FOR UPDATE;

    buf := rpad('b', DBMS_LOB.GETCHUNKSIZE(loc), 'b');

    -- unaligned buffer length
    DBMS_LOB.WRITE(loc, DBMS_LOB.GETCHUNKSIZE(loc)-1, 1, buf);

    -- unaligned start offset
    DBMS_LOB.WRITE(loc, DBMS_LOB.GETCHUNKSIZE(loc), 2, buf);

    -- unaligned buffer length and start offset
    DBMS_LOB.WRITE(loc, DBMS_LOB.GETCHUNKSIZE(loc)-1, 2, buf);

    COMMIT;
END;
/
SELECT * FROM mylobstats;
DROP TABLE t;
DROP VIEW mylobstats;

CONNECT / AS SYSDBA
REVOKE SELECT_CATALOG_ROLE FROM pm;
REVOKE SELECT ON sys.v_$mystat FROM pm;
REVOKE SELECT ON sys.v_$statname FROM pm;

QUIT;
```

Part IV

SQL Access to LOBs

This part describes SQL semantics for LOBs supported in the SQL and PL/SQL environments.

This part contains these chapters:

15

DDL and DML Statements with LOBs

This chapter contains these topics:

> **See Also:** For guidelines on how to `INSERT` into a LOB when binds of more than 4000 bytes are involved, see the following sections in "Binds of All Sizes in INSERT and UPDATE Operations" on page 20-6.

Creating a Table Containing One or More LOB Columns

This section describes how to create a table containing one or more LOB columns.

When you use functions, `EMPTY_BLOB()` and `EMPTY_CLOB()`, the resulting LOB is initialized, but not populated with data. Also note that LOBs that are empty are not `NULL`.

> **See Also:**
>
> *Oracle Database SQL Language Reference* for a complete specification of syntax for using LOBs in `CREATE TABLE` and `ALTER TABLE` with:
>
> - `BLOB`, `CLOB`, `NCLOB` and `BFILE` columns
> - `EMPTY_BLOB` and `EMPTY_CLOB` functions
> - LOB storage clause for persistent LOB columns, and LOB attributes of embedded objects

Scenario

These examples use the following Sample Schemas:

- Human Resources (`HR`)
- Order Entry (`OE`)
- Product Media (`PM`)

Note that the HR and OE schemas must exist before the PM schema is created. For details on these schemas, refer to *Oracle Database Sample Schemas.*

> **Note:** Because you can use SQL DDL directly to create a table containing one or more LOB columns, it is not necessary to use the DBMS_LOB package.

```
/*  Setup script for creating Print_media,
    Online_media and associated structures
*/

DROP USER pm CASCADE;
DROP DIRECTORY ADPHOTO_DIR;
DROP DIRECTORY ADCOMPOSITE_DIR;
DROP DIRECTORY ADGRAPHIC_DIR;
DROP INDEX onlinemedia CASCADE CONSTRAINTS;
DROP INDEX printmedia CASCADE CONSTRAINTS;
DROP TABLE online_media CASCADE CONSTRAINTS;
DROP TABLE print_media CASCADE CONSTRAINTS;
DROP TYPE textdoc_typ;
DROP TYPE textdoc_tab;
DROP TYPE adheader_typ;
DROP TABLE adheader_typ;
CREATE USER pm identified by password;
GRANT CONNECT, RESOURCE to pm;

CREATE DIRECTORY ADPHOTO_DIR AS '/tmp/';
CREATE DIRECTORY ADCOMPOSITE_DIR AS '/tmp/';
CREATE DIRECTORY ADGRAPHIC_DIR AS '/tmp/';
CREATE DIRECTORY media_dir AS '/tmp/';
GRANT READ ON DIRECTORY ADPHOTO_DIR to pm;
GRANT READ ON DIRECTORY ADCOMPOSITE_DIR to pm;
GRANT READ ON DIRECTORY ADGRAPHIC_DIR to pm;
GRANT READ ON DIRECTORY media_dir to pm;

CONNECT pm/password  (or &pass);
COMMIT;

CREATE TABLE a_table (blob_col BLOB);

CREATE TYPE adheader_typ AS OBJECT (
   header_name    VARCHAR2(256),
   creation_date  DATE,
   header_text    VARCHAR(1024),
   logo           BLOB );

CREATE TYPE textdoc_typ AS OBJECT (
   document_typ   VARCHAR2(32),
   formatted_doc  BLOB);

CREATE TYPE Textdoc_ntab AS TABLE of textdoc_typ;

CREATE TABLE adheader_tab of adheader_typ (
Ad_finaltext DEFAULT EMPTY_CLOB(), CONSTRAINT
Take CHECK (Take IS NOT NULL),  DEFAULT NULL);

CREATE TABLE online_media
( product_id  NUMBER(6),
```

```
product_photo ORDSYS.ORDImage,
product_photo_signature ORDSYS.ORDImageSignature,
product_thumbnail ORDSYS.ORDImage,
product_video ORDSYS.ORDVideo,
product_audio ORDSYS.ORDAudio,
product_text CLOB,
product_testimonials ORDSYS.ORDDoc);

CREATE UNIQUE INDEX onlinemedia_pk
  ON online_media (product_id);

ALTER TABLE online_media
ADD (CONSTRAINT onlinemedia_pk
PRIMARY KEY (product_id), CONSTRAINT loc_c_id_fk
FOREIGN KEY (product_id)  REFERENCES oe.product_information(product_id)
);

CREATE TABLE print_media
(product_id NUMBER(6),
ad_id NUMBER(6),
ad_composite BLOB,
ad_sourcetext CLOB,
ad_finaltext CLOB,
ad_fktextn NCLOB,
ad_testdocs_ntab textdoc_tab,
ad_photo BLOB,
ad_graphic BFILE,
ad_header adheader_typ,
press_release LONG) NESTED TABLE ad_textdocs_ntab STORE AS textdocs_nestedtab;

CREATE UNIQUE INDEX printmedia_pk
  ON print_media (product_id, ad_id);

ALTER TABLE print_media
ADD (CONSTRAINT printmedia_pk
PRIMARY KEY (product_id, ad_id),
CONSTRAINT printmedia_fk FOREIGN KEY (product_id)
REFERENCES oe.product_information(product_id)
);
```

Creating a Nested Table Containing a LOB

This section describes how to create a nested table containing a LOB.

You must create the object type that contains the LOB attributes before you create a nested table based on that object type. In the example that follows, table `Print_media` contains nested table `ad_textdoc_ntab` that has type `textdoc_tab`. This type uses two LOB data types:

- `BFILE` - an advertisement graphic
- `CLOB` - an advertisement transcript

The actual embedding of the nested table is accomplished when the structure of the containing table is defined. In our example, this is effected by the `NESTED TABLE` statement when the `Print_media` table is created as shown in the following example:

```
/* Create type textdoc_typ as the base type
   for the nested table textdoc_ntab,
   where textdoc_ntab contains a LOB:
*/
```

```
CREATE TYPE textdoc_typ AS OBJECT
(
   document_typ     VARCHAR2(32),
   formatted_doc    BLOB
);
/

/* The type has been created. Now you need a */
/* nested table of that type to embed in */
/* table Print_media, so: */
CREATE TYPE textdoc_ntab AS TABLE of textdoc_typ;
/

CREATE TABLE textdoc_ntable (
   id NUMBER,
   ntab_col textdoc_ntab)
NESTED TABLE ntab_col STORE AS textdoc_nestedtab;

DROP TYPE textdoc_typ force;
DROP TYPE textdoc_ntab;
DROP TABLE textdoc_ntable;
```

See Also:

- "Creating a Table Containing One or More LOB Columns" on page 15-1
- *Oracle Database SQL Language Reference* for further information on `CREATE TABLE`

Inserting a Row by Selecting a LOB From Another Table

This section describes how to insert a row containing a LOB as SELECT.

Note: Persistent LOB types `BLOB`, `CLOB`, and `NCLOB`, use *copy semantics*, as opposed to *reference semantics* that apply to `BFILE`s. When a `BLOB`, `CLOB`, or `NCLOB` is copied from one row to another in the same table or a different table, the *actual* LOB value is copied, not just the LOB locator.

For LOBs, one of the advantages of using an object-relational approach is that you can define a type as a common template for related tables. For instance, it makes sense that both the tables that store archival material and working tables that use those libraries, share a common structure.

For example, assuming `Print_media` and `Online_media` have identical schemas. The statement creates a new LOB locator in table `Print_media`. `It also` copies the LOB data from `Online_media` to the location pointed to by the new LOB locator inserted in table `Print_media`.

The following code fragment is based on the fact that the table `Online_media` is of the same type as `Print_media` referenced by the `ad_textdocs_ntab` column of table `Print_media`. It inserts values into the library table, and then inserts this same data into `Print_media` by means of a `SELECT`.

```
/* Store records in the archive table Online_media: */
INSERT INTO Online_media
   VALUES (3060, NULL, NULL, NULL, NULL,
           'some text about this CRT Monitor', NULL);
```

```
/* Insert values into Print_media by selecting from Online_media: */
INSERT INTO Print_media (product_id, ad_id, ad_sourcetext)
   (SELECT product_id, 11001, product_text
          FROM Online_media WHERE product_id = 3060);
```

See Also:

- *Oracle Database SQL Language Reference* for more information on `INSERT`
- *Oracle Database Sample Schemas* for a description of the PM Schema and the `Print_media` table used in this example

Inserting a LOB Value Into a Table

This section describes how to insert a LOB value using `EMPTY_CLOB()` or `EMPTY_BLOB()`.

Usage Notes

Here are guidelines for inserting LOBs:

Before Inserting Make the LOB Column Non-Null

Before you write data to a persistent LOB, make the LOB column non-`NULL`; that is, the LOB column must contain a locator that points to an empty or populated LOB value. You can initialize a `BLOB` column value by using the function `EMPTY_BLOB()` as a default predicate. Similarly, a `CLOB` or `NCLOB` column value can be initialized by using the function `EMPTY_CLOB()`.

You can also initialize a LOB column with a character or raw string less than 4000 bytes in size. For example:

```
INSERT INTO Print_media (product_id, ad_id, ad_sourcetext)
     VALUES (1, 1, 'This is a One Line Advertisement');
```

Note that you can also perform this initialization during the `CREATE TABLE` operation. See "Creating a Table Containing One or More LOB Columns" on page 15-1 for more information.

These functions are special functions in Oracle SQL, and are not part of the `DBMS_LOB` package.

```
/* In the new row of table Print_media,
   the columns ad_sourcetext and ad_fltextn are initialized using EMPTY_CLOB(),
   the columns ad_composite and ad_photo are initialized using EMPTY_BLOB(),
   the column formatted-doc in the nested table is initialized using
   EMPTY_BLOB(),
   the column logo in the column object is initialized using EMPTY_BLOB(): */
INSERT INTO Print_media
   VALUES (3060,11001, EMPTY_BLOB(), EMPTY_CLOB(),EMPTY_CLOB(),EMPTY_CLOB(),
   textdoc_tab(textdoc_typ ('HTML', EMPTY_BLOB())), EMPTY_BLOB(), NULL,
   adheader_typ('any header name', <any date>, 'ad header text goes here',
   EMPTY_BLOB()),
 'Press release goes here');
```

Inserting a Row by Initializing a LOB Locator Bind Variable

This section gives examples of how to insert a row by initializing a LOB locator bind variable.

Preconditions

Before you can insert a row using this technique, the following conditions must be met:

- The table containing the source row must exist.
- The destination table must exist.

For details on creating tables containing LOB columns, see "LOB Storage Parameters" on page 11-4.

Usage Notes

For guidelines on how to INSERT and UPDATE a row containing a LOB when binds of more than 4000 bytes are involved, see "Binds of All Sizes in INSERT and UPDATE Operations" on page 20-6.

Syntax

See the following syntax references for details on using this operation in each programmatic environment:

- SQL: *Oracle Database SQL Language Reference*, the `INSERT` statement
- C (OCI): *Oracle Call Interface Programmer's Guide* "Relational Functions"
- C++ (OCCI): *Oracle C++ Call Interface Programmer's Guide*
- COBOL (Pro*COBOL) *Pro*COBOL Programmer's Guide* for information on LOBs, usage notes on LOB Statements, and embedded SQL and precompiler directives — `INSERT`.
- C/C++ (Pro*C/C++): *Pro*C/C++ Programmer's Guide* `INSERT`
- Java (JDBC):*Oracle Database JDBC Developer's Guide* "Working With LOBs" — Creating and Populating a `BLOB` or `CLOB` Column

Examples

Examples for this use case are provided in the following programmatic environments:

- PL/SQL: Inserting a Row by Initializing a LOB Locator Bind Variable on page 15-6
- C (OCI): Inserting a Row by Initializing a LOB Locator Bind Variable on page 15-7
- C++ (OCCI): No example is provided with this release.
- COBOL (Pro*COBOL): Inserting a Row by Initializing a LOB Locator Bind Variable on page 15-8
- C/C++ (Pro*C/C++): Inserting a Row by Initializing a LOB Locator Bind Variable on page 15-8
- Java (JDBC): Inserting a Row by Initializing a LOB Locator Bind Variable on page 15-9

PL/SQL: Inserting a Row by Initializing a LOB Locator Bind Variable

```
/* This file is installed in the following path when you install */
/* the database: $ORACLE_HOME/rdbms/demo/lobs/plsql/linsert.sql */

/* inserting a row through an insert statement */

CREATE OR REPLACE PROCEDURE insertLOB_proc (Lob_loc IN BLOB) IS
```

```
BEGIN
  /* Insert the BLOB into the row */
  DBMS_OUTPUT.PUT_LINE('------------ LOB INSERT EXAMPLE ------------');
  INSERT INTO print_media (product_id, ad_id, ad_photo)
        values (3106, 60315, Lob_loc);
END;
/
```

C (OCI): Inserting a Row by Initializing a LOB Locator Bind Variable

```
/* This file is installed in the following path when you install */
/* the database: $ORACLE_HOME/rdbms/demo/lobs/oci/linsert.c */

/* Insert the Locator into table using Bind Variables. */
#include <oratypes.h>
#include <lobdemo.h>
void insertLOB_proc(OCILobLocator *Lob_loc, OCIEnv *envhp,
                    OCIError *errhp, OCISvcCtx *svchp, OCIStmt *stmthp)
{
  int            product_id;
  OCIBind       *bndhp3;
  OCIBind       *bndhp2;
  OCIBind       *bndhp1;
  text          *insstmt =
   (text *) "INSERT INTO Print_media (product_id, ad_id, ad_sourcetext) \
             VALUES (:1, :2, :3)";

  printf ("------------ OCI Lob Insert Demo --------------\n");
  /* Insert the locator into the Print_media table with product_id=3060 */
  product_id = (int)3060;

  /* Prepare the SQL statement */
  checkerr (errhp, OCIStmtPrepare(stmthp, errhp, insstmt, (ub4)
                                  strlen((char *) insstmt),
                                  (ub4) OCI_NTV_SYNTAX, (ub4)OCI_DEFAULT));

  /* Binds the bind positions */
  checkerr (errhp, OCIBindByPos(stmthp, &bndhp1, errhp, (ub4) 1,
                                (void *) &product_id, (sb4) sizeof(product_id),
                                SQLT_INT, (void *) 0, (ub2 *)0, (ub2 *)0,
                                (ub4) 0, (ub4 *) 0, (ub4) OCI_DEFAULT));

  checkerr (errhp, OCIBindByPos(stmthp, &bndhp1, errhp, (ub4) 2,
                                (void *) &product_id, (sb4) sizeof(product_id),
                                SQLT_INT, (void *) 0, (ub2 *)0, (ub2 *)0,
                                (ub4) 0, (ub4 *) 0, (ub4) OCI_DEFAULT));

  checkerr (errhp, OCIBindByPos(stmthp, &bndhp2, errhp, (ub4) 3,
                                (void *) &Lob_loc, (sb4) 0,  SQLT_CLOB,
                                (void *) 0, (ub2 *)0, (ub2 *)0,
                                (ub4) 0, (ub4 *) 0, (ub4) OCI_DEFAULT));

  /* Execute the SQL statement */
  checkerr (errhp, OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                                  (CONST OCISnapshot*) 0, (OCISnapshot*) 0,
                                  (ub4) OCI_DEFAULT));
}
```

COBOL (Pro*COBOL): Inserting a Row by Initializing a LOB Locator Bind Variable

```
* This file is installed in the following path when you install
* the database: $ORACLE_HOME/rdbms/demo/lobs/procob/linsert.pco

 IDENTIFICATION DIVISION.
 PROGRAM-ID. INSERT-LOB.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.

 01 BLOB1 SQL-BLOB.
 01  USERID   PIC X(11) VALUES "PM/password".
     EXEC SQL INCLUDE SQLCA END-EXEC.

 PROCEDURE DIVISION.
 INSERT-LOB.

     EXEC SQL WHENEVER SQLERROR DO PERFORM SQL-ERROR END-EXEC.
     EXEC SQL CONNECT :USERID END-EXEC.
* Initialize the BLOB locator
     EXEC SQL ALLOCATE :BLOB1 END-EXEC.
* Populate the LOB
     EXEC SQL WHENEVER NOT FOUND GOTO END-OF-BLOB END-EXEC.
     EXEC SQL
        SELECT AD_PHOTO INTO :BLOB1 FROM PRINT_MEDIA
         WHERE PRODUCT_ID = 2268 AND AD_ID = 21001 END-EXEC.

* Insert the value with PRODUCT_ID of 3060
     EXEC SQL
        INSERT INTO PRINT_MEDIA (PRODUCT_ID, AD_PHOTO)
           VALUES (3060, 11001, :BLOB1)END-EXEC.

* Free resources held by locator
 END-OF-BLOB.
     EXEC SQL WHENEVER NOT FOUND CONTINUE END-EXEC.
     EXEC SQL FREE :BLOB1 END-EXEC.
     EXEC SQL ROLLBACK WORK RELEASE END-EXEC.
     STOP RUN.

 SQL-ERROR.
     EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
     DISPLAY " ".
     DISPLAY "ORACLE ERROR DETECTED:".
     DISPLAY " ".
     DISPLAY SQLERRMC.
     EXEC SQL ROLLBACK WORK RELEASE END-EXEC.
     STOP RUN.
```

Note: For simplicity in demonstrating this feature, this example does not perform the password management techniques that a deployed system normally uses. In a production environment, follow the Oracle Database password management guidelines, and disable any sample accounts. See *Oracle Database Security Guide* for password management guidelines and other security recommendations.

C/C++ (Pro*C/C++): Inserting a Row by Initializing a LOB Locator Bind Variable

```
/* This file is installed in the following path when you install */
/* the database: $ORACLE_HOME/rdbms/demo/lobs/proc/linsert.pc */

#include <oci.h>
#include <stdio.h>
#include <sqlca.h>

void Sample_Error()
{
   EXEC SQL WHENEVER SQLERROR CONTINUE;
   printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
   EXEC SQL ROLLBACK WORK RELEASE;
   exit(1);
}

void insertUseBindVariable_proc(Rownum, Lob_loc)
   int Rownum, Rownum2;
   OCIBlobLocator *Lob_loc;
{
   EXEC SQL WHENEVER SQLERROR DO Sample_Error();
   EXEC SQL INSERT INTO Print_media (product_id, ad_id, ad_photo)
      VALUES (:Rownum, :Rownum2, :Lob_loc);
}
void insertBLOB_proc()
{
   OCIBlobLocator *Lob_loc;

   /* Initialize the BLOB Locator: */
   EXEC SQL ALLOCATE :Lob_loc;

   /* Select the LOB from the row where product_id = 2268 and ad_id=21001: */
   EXEC SQL SELECT ad_photo INTO :Lob_loc
      FROM Print_media WHERE product_id = 2268 AND ad_id = 21001;

   /* Insert into the row where product_id = 3106 and ad_id = 13001: */
   insertUseBindVariable_proc(3106, 13001, Lob_loc);

   /* Release resources held by the locator: */
   EXEC SQL FREE :Lob_loc;
}

void main()
{
   char *samp = "pm/password";
   EXEC SQL CONNECT :pm;
   insertBLOB_proc();
   EXEC SQL ROLLBACK WORK RELEASE;
}
```

Java (JDBC): Inserting a Row by Initializing a LOB Locator Bind Variable

```
/* This file is installed in the following path when you install */
/* the database: $ORACLE_HOME/rdbms/demo/lobs/java/linsert.java */

// Core JDBC classes:
import java.sql.DriverManager;
import java.sql.Connection;
import java.sql.Statement;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
```

```
import java.sql.SQLException;

// Oracle Specific JDBC classes:
import oracle.sql.*;
import oracle.jdbc.driver.*;

public class linsert
{
  public static void main (String args [])
       throws Exception
  {
    // Load the Oracle JDBC driver
    DriverManager.registerDriver (new oracle.jdbc.driver.OracleDriver ());
    // Connect to the database:
    Connection conn =
       DriverManager.getConnection ("jdbc:oracle:oci8:@", "pm", "password");

    // It's faster when auto commit is off:
    conn.setAutoCommit (false);

    // Create a Statement:
    Statement stmt = conn.createStatement ();
    try
    {
       ResultSet rset = stmt.executeQuery (
  "SELECT ad_photo FROM Print_media WHERE product_id = 3106 AND ad_id = 13001");
       if (rset.next())
       {
          // retrieve the LOB locator from the ResultSet
          BLOB adphoto_blob = ((OracleResultSet)rset).getBLOB (1);
          OraclePreparedStatement ops =
          (OraclePreparedStatement) conn.prepareStatement(
"INSERT INTO Print_media (product_id, ad_id, ad_photo) VALUES (2268, "
+ "21001, ?)");
          ops.setBlob(1, adphoto_blob);
          ops.execute();
          conn.commit();
          conn.close();
       }
    }
    catch (SQLException e)
    {
       e.printStackTrace();
    }
  }
}
```

Updating a LOB with EMPTY_CLOB() or EMPTY_BLOB()

This section describes how to UPDATE a LOB with `EMPTY_CLOB()` or `EMPTY_BLOB()`.

> **Note:** Performance improves when you update the LOB with the actual value, instead of using `EMPTY_CLOB()` or `EMPTY_BLOB()`.

Preconditions

Before you write data to a persistent LOB, make the LOB column non-`NULL`; that is, the LOB column must contain a locator that points to an empty or populated LOB value.

You can initialize a BLOB column value by using the function EMPTY_BLOB() as a default predicate. Similarly, a CLOB or NCLOB column value can be initialized by using the function EMPTY_CLOB().

You can also initialize a LOB column with a character or raw string less than 4000 bytes in size. For example:

```
UPDATE Print_media
          SET ad_sourcetext = 'This is a One Line Story'
          WHERE product_id = 2268;
```

You can perform this initialization during CREATE TABLE (see "Creating a Table Containing One or More LOB Columns" on page 15-1) or, as in this case, by means of an INSERT.

The following example shows a series of updates using the EMPTY_CLOB operation to different data types.

```
UPDATE Print_media SET ad_sourcetext = EMPTY_CLOB()
       WHERE product_id = 3060 AND ad_id = 11001;

UPDATE Print_media SET ad_fltextn = EMPTY_CLOB()
       WHERE product_id = 3060 AND ad_id = 11001;

UPDATE Print_media SET ad_photo = EMPTY_BLOB()
       WHERE product_id = 3060 AND ad_id = 11001;
```

See Also: SQL: *Oracle Database SQL Language Reference* for more information on UPDATE

Updating a Row by Selecting a LOB From Another Table

This section describes how to use the SQL UPDATE AS SELECT statement to update a row containing a LOB column by selecting a LOB from another table.

To use this technique, you must update by means of a reference. For example, the following code updates data from online_media:

```
Rem Updating a row by selecting a LOB from another table (persistent LOBs)

UPDATE Print_media SET ad_sourcetext =
   (SELECT * product_text FROM online_media WHERE product_id = 3060);
    WHERE product_id = 3060 AND ad_id = 11001;
```

16

SQL Semantics and LOBs

This chapter describes SQL semantics that are supported for LOBs. These techniques allow you to use LOBs directly in SQL code and provide an alternative to using LOB-specific APIs for some operations.

This chapter contains these topics:

- Using LOBs in SQL
- SQL Functions and Operators Supported for Use with LOBs
- Implicit Conversion of LOB Data Types in SQL
- Unsupported Use of LOBs in SQL
- VARCHAR2 and RAW Semantics for LOBs

  See Also: "Performance Considerations for SQL Semantics and LOBs" on page 14-4.

- Built-in Functions for Remote LOBs and BFILEs

Using LOBs in SQL

You can access `CLOB` and `NCLOB` data types using SQL `VARCHAR2` semantics, such as SQL string operators and functions. (`LENGTH` functions can be used with `BLOB` data types and `CLOB` and `NCLOB`s.) These techniques are beneficial in the following situations:

- When performing operations on LOBs that are relatively small in size (up to about 100K bytes).
- After migrating your database from `LONG` columns to LOB data types, any SQL string functions, contained in your existing PL/SQL application, continue to work after the migration.

SQL semantics are not recommended in the following situations:

- When you use advanced features such as random access and piece-wise fetch, you must use LOB APIs.
- When performing operations on LOBs that are relatively large in size (greater than 1MB) using SQL semantics can impact performance. Using the LOB APIs is recommended in this situation.

Note: SQL semantics are used with persistent and temporary LOBs. (SQL semantics do not apply to `BFILE` columns because `BFILE` is a read-only data type.)

SQL Functions and Operators Supported for Use with LOBs

Many SQL operators and functions that take VARCHAR2 columns as arguments also accept LOB columns. The following list summarizes which categories of SQL functions and operators are supported for use with LOBs. Details on individual functions and operators are given in Table 16–1.

The following categories of SQL functions and operators are supported for use with LOBs:

- Concatenation
- Comparison

  (Some comparison functions are not supported for use with LOBs.)
- Character functions
- Conversion

  (Some conversion functions are not supported for use with LOBs.)

The following categories of functions are not supported for use with LOBs:

- Aggregate functions

  Note that although pre-defined aggregate functions are not supported for use with LOBs, you can create user-defined aggregate functions to use with LOBs. See the *Oracle Database Data Cartridge Developer's Guide* for more information on user-defined aggregate functions.
- Unicode functions

Details on individual functions and operators are in Table 16–1, which lists SQL operators and functions that take VARCHAR2 types as operands or arguments, or return a VARCHAR2 value. The SQL column identifies the functions and operators that are supported for CLOB and NCLOB data types. (The LENGTH function is also supported for the BLOB data type.)

The DBMS_LOB PL/SQL package supplied with Oracle Database supports using LOBs with most of the functions listed in Table 16–1 as indicated in the PL/SQL column.

Note: Operators and functions with No indicated in the SQL column of Table 16–1 do not work in SQL queries used in PL/SQL blocks - even though some of these operators and functions are supported for use directly in PL/SQL code.

Implicit Conversion of CLOB to CHAR Types

Functions designated as CNV in the SQL or PL/SQL column of Table 16–1 are performed by converting the CLOB to a character data type, such as VARCHAR2. In the SQL environment, only the first 4K bytes of the CLOB are converted and used in the operation; in the PL/SQL environment, only the first 32K bytes of the CLOB are converted and used in the operation.

Table 16–1 SQL VARCHAR2 Functions and Operators on LOBs

Category	Operator / Function	SQL Example / Comments	SQL	PL/SQL
Concatenation	||, CONCAT()	Select clobCol || clobCol2 from tab;	Yes	Yes
Comparison	= , !=, >, >=, <, <=, <>, ^=	if clobCol=clobCol2 then...	No	Yes
Comparison	IN, NOT IN	if clobCol NOT IN (clob1, clob2, clob3) then...	No	Yes
Comparison	SOME, ANY, ALL	if clobCol < SOME (select clobCol2 from...) then...	No	N/A
Comparison	BETWEEN	if clobCol BETWEEN clobCol2 and clobCol3 then...	No	Yes
Comparison	LIKE [ESCAPE]	if clobCol LIKE '%pattern%' then...	Yes	Yes
Comparison	IS [NOT] NULL	where clobCol IS NOT NULL	Yes	Yes
Character Functions	INITCAP, NLS_INITCAP	select INITCAP(clobCol) from...	CNV	CNV
Character Functions	LOWER, NLS_LOWER, UPPER, NLS_UPPER	...where LOWER(clobCol1) = LOWER(clobCol2)	Yes	Yes
Character Functions	LPAD, RPAD	select RPAD(clobCol, 20, ' La') from...	Yes	Yes
Character Functions	TRIM, LTRIM, RTRIM	...where RTRIM(LTRIM(clobCol,'ab'), 'xy') = 'cd'	Yes	Yes
Character Functions	REPLACE	select REPLACE(clobCol, 'orig','new') from...	Yes	Yes
Character Functions	SOUNDEX	...where SOUNDEX(clobCOl) = SOUNDEX('SMYTHE')	CNV	CNV
Character Functions	SUBSTR	...where substr(clobCol, 1,4) = like 'THIS'	Yes	Yes
Character Functions	TRANSLATE	select TRANSLATE(clobCol, '123abc','NC') from...	CNV	CNV
Character Functions	ASCII	select ASCII(clobCol) from...	CNV	CNV
Character Functions	INSTR	...where instr(clobCol, 'book') = 11	Yes	Yes
Character Functions	LENGTH	...where length(clobCol) != 7;	Yes	Yes
Character Functions	NLSSORT	...where NLSSORT (clobCol,'NLS_SORT = German') > NLSSORT ('S','NLS_SORT = German')	CNV	CNV
Character Functions	INSTRB, SUBSTRB, LENGTHB	These functions are supported only for CLOBs that use single-byte character sets. (LENGTHB is supported for BLOBs and CLOBs.)	Yes	Yes

Table 16–1 (Cont.) SQL VARCHAR2 Functions and Operators on LOBs

Category	Operator / Function	SQL Example / Comments	SQL	PL/SQL
Character Functions - Regular Expressions	REGEXP_LIKE	This function searches a character column for a pattern. Use this function in the WHERE clause of a query to return rows matching the regular expression you specify. See the *Oracle Database SQL Language Reference* for syntax details on SQL functions for regular expressions. See the *Oracle Database Development Guide* for information on using regular expressions with the database.	Yes	Yes
Character Functions - Regular Expressions	REGEXP_REPLACE	This function searches for a pattern in a character column and replaces each occurrence of that pattern with the pattern you specify.	Yes	Yes
Character Functions - Regular Expressions	REGEXP_INSTR	This function searches a string for a given occurrence of a regular expression pattern. You specify which occurrence you want to find and the start position to search from. This function returns an integer indicating the position in the string where the match is found.	Yes	Yes
Character Functions - Regular Expressions	REGEXP_SUBSTR	This function returns the actual substring matching the regular expression pattern you specify.	Yes	Yes
Conversion	CHARTOROWID	CHARTOROWID(clobCol)	CNV	CNV
Conversion	COMPOSE	COMPOSE('string') Returns a Unicode string given a string in the data type CHAR, VARCHAR2, CLOB, NCHAR, NVARCHAR2, NCLOB. An o code point qualified by an umlaut code point is returned as the o-umlaut code point.	CNV	CNV
Conversion	DECOMPOSE	DECOMPOSE('str' [CANONICAL | COMPATIBILITY]) Valid for Unicode character arguments. Returns a Unicode string after decomposition in the same character set as the input. o-umlaut code point is returned as the o code point followed by the umlaut code point.	CNV	CNV
Conversion	HEXTORAW	HEXTORAW(CLOB)	No	CNV
Conversion	CONVERT	select CONVERT(clobCol,'WE8DEC','WE8HP') from...	Yes	CNV
Conversion	TO_DATE	TO_DATE(clobCol)	CNV	CNV
Conversion	TO_NUMBER	TO_NUMBER(clobCol)	CNV	CNV
Conversion	TO_TIMESTAMP	TO_TIMESTAMP(clobCol)	No	CNV
Conversion	TO_MULTI_BYTE TO_SINGLE_BYTE	TO_MULTI_BYTE(clobCol) TO_SINGLE_BYTE(clobCol)	CNV	CNV
Conversion	TO_CHAR	TO_CHAR(clobCol)	Yes	Yes
Conversion	TO_NCHAR	TO_NCHAR(clobCol)	Yes	Yes

Table 16–1 (Cont.) SQL VARCHAR2 Functions and Operators on LOBs

Category	Operator / Function	SQL Example / Comments	SQL	PL/SQL
Conversion	TO_LOB	INSERT INTO... SELECT TO_LOB(longCol)... Note that TO_LOB can only be used to create or insert into a table with LOB columns as SELECT FROM a table with a LONG column.	N/A	N/A
Conversion	TO_CLOB	TO_CLOB(varchar2Col)	Yes	Yes
Conversion	TO_NCLOB	TO_NCLOB(varchar2Clob)	Yes	Yes
Aggregate Functions	COUNT	select count(clobCol) from...	No	N/A
Aggregate Functions	MAX, MIN	select MAX(clobCol) from...	No	N/A
Aggregate Functions	GROUPING	select grouping(clobCol) from... group by cube (clobCol);	No	N/A
Other Functions	GREATEST, LEAST	select GREATEST (clobCol1, clobCol2) from...	No	CNV
Other Functions	DECODE	select DECODE(clobCol, condition1, value1, defaultValue) from...	CNV	CNV
Other Functions	NVL	select NVL(clobCol,'NULL') from...	Yes	Yes
Other Functions	DUMP	select DUMP(clobCol) from...	No	N/A
Other Functions	VSIZE	select VSIZE(clobCol) from...	No	N/A
Unicode	INSTR2, SUBSTR2, LENGTH2, LIKE2	These functions use UCS2 code point semantics.	No	CNV
Unicode	INSTR4, SUBSTR4, LENGTH4, LIKE4	These functions use UCS4 code point semantics.	No	CNV
Unicode	INSTRC, SUBSTRC, LENGTHC, LIKEC	These functions use complete character semantics.	No	CNV

CLOBs and NCLOBs Do Not Follow Session Collation Settings

Standard operators that operate on CLOBs and NCLOBs without first converting them to VARCHAR2 or NVARCHAR2, (those marked Yes in the SQL or PL/SQL columns of Table 16–1), do not behave linguistically, except for REGEXP functions. Binary comparison of the character data is performed irrespective of the NLS_COMP and NLS_SORT parameter settings.

The REGEXP functions listed below are the exceptions, where, if CLOB or NCLOB data is passed in, the linguistic comparison is similar to the comparison of VARCHAR2 and NVARCHAR2 values.

- REGEXP_LIKE
- REGEXP_REPLACE
- REGEXP_INSTR
- REGEXP_SUBSTR
- REGEXP_COUNT

UNICODE Support

Variations on the INSTR, SUBSTR, LENGTH, and LIKE functions are provided for Unicode support. (These variations are indicated as Unicode in the Category column of Table 16–1.)

See Also:

- *Oracle Database Globalization Support Guide*
- *Oracle Database Development Guide*
- *Oracle Database SQL Language Reference*
- *Oracle Database PL/SQL Packages and Types Reference*

for a detailed description on the usage of UNICODE functions.

Codepoint Semantics

Codepoint semantics of the INSTR, SUBSTR, LENGTH, and LIKE functions, described in Table 16–1, differ depending on the data type of the argument passed to the function. These functions use different codepoint semantics depending on whether the argument is a VARCHAR2 or a CLOB type as follows:

- When the argument is a CLOB, UCS2 codepoint semantics are used for all character sets.
- When the argument is a character type, such as VARCHAR2, the default codepoint semantics are used for the given character set:
  - UCS2 codepoint semantics are used for AL16UTF16 and UTF8 character sets.
  - UCS4 codepoint semantics are used for all other character sets, such as AL32UTF8.
- If you are storing character data in a CLOB or NCLOB, then note that the amount and offset parameters for any APIs that read or write data to the CLOB or NCLOB are specified in UCS2 codepoints. In some character sets, a full character consists one or more UCS2 codepoints called a surrogate pair. In this scenario, you must ensure that the amount or offset you specify does not cut into a full character. This avoids reading or writing a partial character.
- Starting from 10*g*, Oracle Database helps to detect half surrogate pair on read/write boundaries in such scenarios. In the case of read, the offset and amount is adjusted accordingly to avoid returning a half character, in which case the amount returned could be less than what is asked for. In the case of write, an error is raised to prevent from corrupting the existing data caused by overwriting a partial character in the destination CLOB or NCLOB.

Return Values for SQL Semantics on LOBs

The return type of a function or operator that takes a LOB or VARCHAR2 is the same as the data type of the argument passed to the function or operator.

Functions that take more than one argument, such as CONCAT, return a LOB data type if one or more arguments is a LOB. For example, CONCAT(CLOB, VARCHAR2) returns a CLOB.

See Also: *Oracle Database SQL Language Reference* for details on the CONCAT function and the concatenation operator (||).

A LOB instance is always accessed and manipulated through a LOB locator. This is also true for return values: SQL functions and operators return a LOB locator when the return value is a LOB instance.

Any LOB instance returned by a SQL function is a temporary LOB instance. LOB instances in tables (persistent LOBs) are not modified by SQL functions, even when the function is used in the SELECT list of a query.

LENGTH Return Value for LOBs

The return value of the LENGTH function differs depending on whether the argument passed is a LOB or a character string:

- If the input is a character string of length zero, then LENGTH returns NULL.
- For a CLOB of length zero, or an empty locator such as that returned by EMPTY_CLOB(), the LENGTH and DBMS_LOB.GETLENGTH functions return 0.

Implicit Conversion of LOB Data Types in SQL

Some LOB data types support implicit conversion and can be used in operations such as cross-type assignment and parameter passing. These conversions are processed at the SQL layer and can be performed in all client interfaces that use LOB types.

Implicit Conversion Between CLOB and NCLOB Data Types in SQL

The database enables you to perform operations such as cross-type assignment and cross-type parameter passing between CLOB and NCLOB data types. The database performs implicit conversions between these types when necessary to preserve properties such as character set formatting.

Note that, when implicit conversions occur, each character in the source LOB is changed to the character set of the destination LOB, if needed. In this situation, some degradation of performance may occur if the data size is large. When the character set of the destination and the source are the same, there is no degradation of performance.

After an implicit conversion between CLOB and NCLOB types, the destination LOB is implicitly created as a temporary LOB. This new temporary LOB is independent from the source LOB. If the implicit conversion occurs as part of a define operation in a SELECT statement, then any modifications to the destination LOB do not affect the persistent LOB in the table that the LOB was selected from as shown in the following example:

```
SQL> -- check lob length before update
SQL> select dbms_lob.getlength(ad_sourcetext) from Print_media
  2       where product_id=3106 and ad_id = 13001;

DBMS_LOB.GETLENGTH(AD_SOURCETEXT)
---------------------------------
         205

SQL>
SQL> declare
  2   clob1 clob;
  3   amt number:=10;
  4  BEGIN
  5    -- select a clob column into a clob, no implicit convesion
  6    SELECT ad_sourcetext INTO clob1 FROM Print_media
  7      WHERE product_id=3106 and ad_id=13001 FOR UPDATE;
```

```
  8
  9    dbms_lob.trim(clob1, amt); -- Trim the selected lob to 10 bytes
 10  END;
 11  /

PL/SQL procedure successfully completed.

SQL> -- Modification is performed on clob1 which points to the
SQL> -- clob column in the table
SQL> select dbms_lob.getlength(ad_sourcetext) from Print_media
  2       where product_id=3106 and ad_id = 13001;

DBMS_LOB.GETLENGTH(AD_SOURCETEXT)
---------------------------------
          10

SQL>
SQL> rollback;

Rollback complete.

SQL> -- check lob length before update
SQL> select dbms_lob.getlength(ad_sourcetext) from Print_media
  2       where product_id=3106 and ad_id = 13001;

DBMS_LOB.GETLENGTH(AD_SOURCETEXT)
---------------------------------
         205

SQL>
SQL> declare
  2   nclob1 nclob;
  3   amt number:=10;
  4  BEGIN
  5
  6    -- select a clob column into a nclob, implicit conversion occurs
  7    SELECT ad_sourcetext INTO nclob1 FROM Print_media
  8      WHERE product_id=3106 and ad_id=13001 FOR UPDATE;
  9
 10    dbms_lob.trim(nclob1, amt); -- Trim the selected lob to 10 bytes
 11  END;
 12  /

PL/SQL procedure successfully completed.

SQL> -- Modification to nclob1 does not affect the clob in the table,
SQL> -- because nclob1 is a independent temporary LOB

SQL> select dbms_lob.getlength(ad_sourcetext) from Print_media
  2       where product_id=3106 and ad_id = 13001;

DBMS_LOB.GETLENGTH(AD_SOURCETEXT)
---------------------------------
         205
```

See Also:

- "Implicit Conversions Between CLOB and VARCHAR2" on page 17-1 for information on PL/SQL semantics support for implicit conversions between CLOB and VARCHAR2 types.
- "Implicit Character Set Conversions with LOBs" on page 11-4 for more information on implicit character set conversions when loading LOBs from BILEs.
- *Oracle Database SQL Language Reference* for details on implicit conversions supported for all data types.

Unsupported Use of LOBs in SQL

Table 16–2 lists SQL operations that are not supported on LOB columns.

Table 16–2 Unsupported Usage of LOBs in SQL

SQL Operations Not Supported	Example of unsupported usage
SELECT DISTINCT	SELECT DISTINCT clobCol from...
SELECT clause ORDER BY	SELECT... ORDER BY clobCol
SELECT clause GROUP BY	SELECT avg(num) FROM... GROUP BY clobCol
UNION, INTERSECT, MINUS (Note that UNION ALL works for LOBs.)	SELECT clobCol1 from tab1 UNION SELECT clobCol2 from tab2;
Join queries	SELECT... FROM... WHERE tab1.clobCol = tab2.clobCol
Index columns	CREATE INDEX clobIndx ON tab(clobCol)...

VARCHAR2 and RAW Semantics for LOBs

The following semantics, used with VARCHAR2 and RAW data types, also apply to LOBs:

- Defining a CHAR buffer on a CLOB

  You can define a VARCHAR2 for a CLOB and RAW for a BLOB column. You can also define CLOB and BLOB types for VARCHAR2 and RAW columns.

- Selecting a CLOB column into a CHAR buffer or VARCHAR2

  If a CLOB column is selected into a VARCHAR2 variable, then data stored in the CLOB column is retrieved and put into the CHAR buffer. If the buffer is not large enough to contain all the CLOB data, then a truncation error is thrown and no data is written to the buffer. After successful completion of the SELECT operation, the VARCHAR2 variable holds as a regular character buffer.

  In contrast, when a CLOB column is selected into a local CLOB variable, the CLOB locator is fetched.

- Selecting a BLOB column into a RAW

  When a BLOB column is selected into a RAW variable, the BLOB data is copied into the RAW buffer. If the size of the BLOB exceeds the size of the buffer, then a truncation error is thrown and no data is written to the buffer.

LOBs Returned from SQL Functions

When a LOB is returned from a SQL function, the result returned is a temporary LOB. Your application should view the temporary LOB as local storage for the data returned from the SELECT operation as follows:

- In PL/SQL, the temporary LOB has the same lifetime (duration) as other local PL/SQL program variables. It can be passed to subsequent SQL or PL/SQL VARCHAR2 functions or queries as a PL/SQL local variable. The temporary LOB goes out of scope at the end of the program block at which time, the LOB is freed. These are the same semantics as those for PL/SQL VARCHAR2 variables. At any time, nonetheless, you can use a DBMS_LOB.FREETEMPORARY() call to release the resources taken by the local temporary LOBs.

  Note: If the SQL statement returns a LOB or a LOB is an OUT parameter for a PL/SQL function or procedure, you must test if it is a temporary LOB, and if it is, then free it after you are done with it.

- In OCI, the temporary LOBs returned from SQL queries are always in session duration, unless a user-defined duration is present, in which case, the temporary LOBs are in the user-defined duration.

  Caution: Ensure that your temporary tablespace is large enough to store all temporary LOB results returned from queries in your program(s).

The following example illustrates selecting out a CLOB column into a VARCHAR2 and returning the result as a CHAR buffer of declared size:

```
DECLARE
  vc1 VARCHAR2(32000);
  lb1 CLOB;
  lb2 CLOB;
BEGIN
  SELECT clobCol1 INTO vc1 FROM tab WHERE colID=1;
  -- lb1 is a temporary LOB
  SELECT clobCol2 || clobCol3 INTO lb1 FROM tab WHERE colID=2;

  lb2 := vc1|| lb1;
  -- lb2 is a still temporary LOB, so the persistent data in the database
  -- is not modified. An update is necessary to modify the table data.
  UPDATE tab SET clobCol1 = lb2 WHERE colID = 1;

DBMS_LOB.FREETEMPORARY(lb2); -- Free up the space taken by lb2

<... some more queries ...>

END; -- at the end of the block, lb1 is automatically freed
```

IS NULL and IS NOT NULL Usage with VARCHAR2s and CLOBs

You can use the IS NULL and IS NOT NULL operators with LOB columns. When used with LOBs, these operators determine whether a LOB locator is stored in the row.

Note: In the SQL 92 standard, a character string of length zero is distinct from a NULL string. The return value of IS NULL differs when you pass a LOB compared to a VARCHAR2:

- When you pass an initialized LOB of length zero to the IS NULL function, zero (FALSE) is returned. These semantics are compliant with the SQL standard.
- When you pass a VARCHAR2 of length zero to the IS NULL function, TRUE is returned.

WHERE Clause Usage with LOBs

SQL functions with LOBs as arguments, except functions that compare LOB values, are allowed in predicates of the WHERE clause. For example, the LENGTH function can be included in the predicate of the WHERE clause:

```
CREATE TABLE t (n NUMBER, c CLOB);
INSERT INTO t VALUES (1, 'abc');

SELECT * FROM t WHERE c IS NOT NULL;
SELECT * FROM t WHERE LENGTH(c) > 0;
SELECT * FROM t WHERE c LIKE '%a%';
SELECT * FROM t WHERE SUBSTR(c, 1, 2) LIKE '%b%';
SELECT * FROM t WHERE INSTR(c, 'b') = 2;
```

Built-in Functions for Remote LOBs and BFILEs

Whatever SQL built-in functions and user-defined functions that are supported on local LOBs and BFILEs are also supported on remote LOBs and BFILEs, as long as the final value returned by nested functions is not a LOB. This includes functions for remote persistent and temporary LOBs and for BFILEs.

Built-in SQL functions which are executed on a remote site can be part of any SQL statement, like SELECT, INSERT, UPDATE, and DELETE. For example:

```
SELECT LENGTH(ad_sourcetext) FROM print_media@remote_site -- CLOB
SELECT LENGTH(ad_fltextn) FROM print_media@remote_site;   -- NCLOB
SELECT LENGTH(ad_composite) FROM print_media@remote_site; -- BLOB
SELECT product_id from print_media@remote_site WHERE LENGTH(ad_sourcetext) > 3;

UPDATE print_media@remote_site SET product_id = 2 WHERE LENGTH(ad_sourcetext) > 3;

SELECT TO_CHAR(foo@dbs2(...)) FROM dual@dbs2;
-- where foo@dbs2 returns a temporary LOB
```

The SQL functions fall under the following (not necessarily exclusive) categories:

- SQL functions that are not supported on LOBs:

  These functions are relevant only for CLOBs: an example is DECODE.

  These functions **cannot be supported** on remote LOBs because they are not supported on local LOBs.

- Functions taking exactly one LOB argument (all other arguments are of other data types) and not returning a LOB:

  These functions are relevant only for CLOBs, NCLOBs, and BLOBs: an example is LENGTH and **it is supported**. For example:

```
SELECT LENGTH(ad_composite) FROM print_media@remote_site;
SELECT LENGTH(ad_header.logo) FROM print_media@remote_site; -- LOB in object

SELECT product_id from print_media@remote_site WHERE LENGTH(ad_sourcetext) > 3;
```

- Functions that return a LOB:

  All these functions are relevant only for CLOBs and NCLOBs. These functions may return the original LOB or produce a temporary LOB. These functions can be performed on the remote site, as long as the result returned to the local site is not a LOB.

  Functions returning a temporary LOB are: `REPLACE`, `SUBSTR`, `CONCAT`, `||`, `TRIM`, `LTRIM`, `RTRIM`, `LOWER`, `UPPER`, `NLS_LOWER`, `NLS_UPPER`, `LPAD`, and `RPAD`.

  Functions returning the original LOB locator are: `NVL`, `DECODE`, and `CASE`. Note that even though `DECODE` and `CASE` are not supported currently to operate on LOBs, they could operate on other data types and return a LOB.

  For example, **the following statements are supported:**

  ```
  SELECT TO_CHAR(CONCAT(ad_sourcetext, ad_sourcetext)) FROM
      print_media@remote_site;

  SELECT TO_CHAR(SUBSTR(ad_fltextnfs, 1, 3)) FROM
      print_media@remote_site;
  ```

  But the **following statements are not supported:**

  ```
  SELECT CONCAT(ad_sourcetext, ad_sourcetext) FROM
      print_media@remote_site;

  SELECT SUBSTR(ad_sourcetext, 1, 3) FROM print_media@remote_site;
  ```

- Functions that take in more than one LOB argument:

  These are: `INSTR`, `LIKE`, `REPLACE`, `CONCAT`, `||`, `SUBSTR`, `TRIM`, `LTRIM`, `RTRIM`, `LPAD`, and `RPAD`. All these functions are relevant only for `CLOB`s and `NCLOB`s.

  These functions are supported only if all the LOB arguments are in the same `dblink`, and the value returned is not a LOB. For example, **the following is supported:**

  ```
  SELECT TO_CHAR(CONCAT(ad_sourcetext, ad_sourcetext)) FROM
  print_media@remote_site; -- CLOB

  SELECT TO_CHAR(CONCAT(ad_fltextn, ad_fltextn)) FROM
  print_media@remote_site; -- NCLOB
  ```

  But the following is not supported:

  ```
  SELECT TO_CHAR(CONCAT(a.ad_sourcetext, b.ad_sourcetext)) FROM
  print_media@db1 a, print_media@db2 b WHERE a.product_id = b.product_id;
  ```

17

PL/SQL Semantics for LOBs

This chapter contains these topics:

PL/SQL Statements and Variables

In PL/SQL, semantic changes have been made.

> **Note:** The following discussions, concerning CLOBs and VARCHAR2s, also apply to BLOBs and RAWs, unless otherwise noted. In the text, BLOB and RAW are not explicitly mentioned.

PL/SQL semantics support is described in the following sections:

Implicit Conversions Between CLOB and VARCHAR2

Implicit conversions from CLOB to VARCHAR2 and from VARCHAR2 to CLOB data types are allowed in PL/SQL. These conversions enable you to perform the following operations in your application:

- CLOB columns can be selected into VARCHAR2 PL/SQL variables
- VARCHAR2 columns can be selected into CLOB variables
- Assignment and parameter passing between CLOBs and VARCHAR2s

Accessing a CLOB as a VARCHAR2 in PL/SQL

The following example illustrates the way CLOB data is accessed when the CLOBs are treated as VARCHAR2s:

```
declare
   myStoryBuf VARCHAR2(4001);
BEGIN
```

```
   SELECT ad_sourcetext INTO myStoryBuf FROM print_media WHERE ad_id = 12001;
   -- Display Story by printing myStoryBuf directly
END;
/
```

Assigning a CLOB to a VARCHAR2 in PL/SQL

```
declare
myLOB CLOB;
BEGIN
SELECT 'ABCDE' INTO myLOB FROM print_media WHERE ad_id = 11001;
-- myLOB is a temporary LOB.
-- Use myLOB as a lob locator
  DBMS_OUTPUT.PUT_LINE('Is temp? '||DBMS_LOB.ISTEMPORARY(myLOB));
END;
/
```

Explicit Conversion Functions

In SQL and PL/SQL, the following explicit conversion functions convert other data types to and from CLOB, NCLOB, and BLOB as part of the LONG-to-LOB migration:

- TO_CLOB(): Converting from VARCHAR2, NVARCHAR2, or NCLOB to a CLOB
- TO_NCLOB(): Converting from VARCHAR2, NVARCHAR2, or CLOB to an NCLOB
- TO_BLOB(): Converting from RAW to a BLOB
- TO_CHAR() converts a CLOB to a CHAR type. When you use this function to convert a character LOB into the database character set, if the LOB value to be converted is larger than the target type, then the database returns an error. Implicit conversions also raise an error if the LOB data does not fit.
- TO_NCHAR() converts an NCLOB to an NCHAR type. When you use this function to convert a character LOB into the national character set, if the LOB value to be converted is larger than the target type, then the database returns an error. Implicit conversions also raise an error if the LOB data does not fit.
- CAST does not directly support any of the LOB data types. When you use CAST to convert a CLOB value into a character data type, an NCLOB value into a national character data type, or a BLOB value into a RAW data type, the database implicitly converts the LOB value to character or raw data and then explicitly casts the resulting value into the target data type. If the resulting value is larger than the target type, then the database returns an error.

Other explicit conversion functions are not supported, such as, TO_NUMBER(), see Table 16–1, " SQL VARCHAR2 Functions and Operators on LOBs". Conversion function details are explained in Chapter 18, "Migrating Columns from LONGs to LOBs".'

Note that LOBs do not support duplicate LONG binds.

VARCHAR2 and CLOB in PL/SQL Built-In Functions

CLOB and VARCHAR2 are still two distinct types. But depending on the usage, a CLOB can be passed to SQL and PL/SQL VARCHAR2 built-in functions, used exactly like a VARCHAR2. Or the variable can be passed into DBMS_LOB APIs, acting like a LOB locator. Please see the following combined example,"CLOB Variables in PL/SQL".

PL/SQL VARCHAR2 functions and operators can take CLOBs as arguments or operands.

When the size of a VARCHAR2 variable is not large enough to contain the result from a function that returns a CLOB, or a SELECT on a CLOB column, an error is raised and no operation is performed. This is consistent with VARCHAR2 semantics.

CLOB Variables in PL/SQL

```
1 declare
2   myStory CLOB;
3   revisedStory CLOB;
4   myGist VARCHAR2(100);
5   revisedGist VARCHAR2(100);
6 BEGIN
7  -- select a CLOB column into a CLOB variable
8  SELECT Story INTO myStory FROM print_media WHERE product_id=10;
9  -- perform VARCHAR2 operations on a CLOB variable
10 revisedStory := UPPER(SUBSTR(myStory, 100, 1));
11 -- revisedStory is a temporary LOB
12 -- Concat a VARCHAR2 at the end of a CLOB
13 revisedStory := revisedStory || myGist;
14 -- The following statement raises an error because myStory is
15 -- longer than 100 bytes
16 myGist := myStory;
17 END;
```

Please note that in line 10 of "CLOB Variables in PL/SQL", a temporary CLOB is implicitly created and is pointed to by the revisedStory CLOB locator. In the current interface the line can be expanded as:

```
buffer VARCHAR2(32000)
DBMS_LOB.CREATETEMPORARY(revisedStory);
buffer := UPPER(DBMS_LOB.SUBSTR(myStory,100,1));
DBMS_LOB.WRITE(revisedStory,length(buffer),1, buffer);
```

In line 13, myGist is appended to the end of the temporary LOB, which has the same effect of:

```
DBMS_LOB.WRITEAPPEND(revisedStory, myGist, length(myGist));
```

In some occasions, implicitly created temporary LOBs in PL/SQL statements can change the representation of LOB locators previously defined. Consider the next example.

Change in Locator-Data Linkage

```
1 declare
2 myStory CLOB;
3 amt number:=100;
4 buffer VARCHAR2(100):='some data';
5 BEGIN
6 -- select a CLOB column into a CLOB variable
7 SELECT Story INTO myStory FROM print_media WHERE product_id=10;
8 DBMS_LOB.WRITE(myStory, amt, 1, buf);
9 -- write to the persistent LOB in the table
10
11 myStory:= UPPER(SUBSTR(myStory, 100, 1));
12 -- perform VARCHAR2 operations on a CLOB variable, temporary LOB created.
13 -- Changes are not reflected in the database table from this point on.
14
15 update print_media set Story = myStory WHERE product_id = 10;
16 -- an update is necessary to synchronize the data in the table.
17 END;
```

After line 7, myStory represents a persistent LOB in print_media.

The DBMS_LOB.WRITE call in line 8 directly writes the data to the table.

No UPDATE statement is necessary. Subsequently in line 11, a temporary LOB is created and assigned to myStory because myStory is now used like a local VARCHAR2 variable. The LOB locator myStory now points to the newly-created temporary LOB.

Therefore, modifications to myStory are no longer reflected in the database. To propagate the changes to the database table, an UPDATE statement becomes necessary now. Note again that for the previous persistent LOB, the UPDATE is not required.

Temporary LOBs created in a program block as a result of a SELECT or an assignment are freed automatically at the end of the PL/SQL block or function or procedure. You must also free the temporary LOBs that were created with DBMS_LOB.CREATETEMPORARY to reclaim system resources and temporary tablespace. Do this by calling DBMS_LOB.FREETEMPORARY on the CLOB variable.

> **Note:** If the SQL statement returns a LOB or a LOB is an OUT parameter for a PL/SQL function or procedure, you must test if it is a temporary LOB, and if it is, then free it after you are done with it.

Freeing Temporary LOBs Automatically and Manually

```
declare
   Story1 CLOB;
   Story2 CLOB;
   StoryCombined CLOB;
   StoryLower CLOB;
BEGIN
   SELECT Story INTO Story1 FROM print_media WHERE product_ID = 1;
   SELECT Story INTO Story2 FROM print_media WHERE product_ID = 2;
   StoryCombined := Story1 || Story2; -- StoryCombined is a temporary LOB
   -- Free the StoryCombined manually to free up space taken
   DBMS_LOB.FREETEMPORARY(StoryCombined);
   StoryLower := LOWER(Story1) || LOWER(Story2);
END; -- At the end of block, StoryLower is freed.
```

PL/SQL Functions for Remote LOBs and BFILEs

Built-in and user-defined PL/SQL functions that are executed on the remote site and operate on remote LOBs and BFILEs are allowed, as long as the final value returned by nested functions is not a LOB. Examples are:

```
SELECT product_id FROM print_media@dbs2 WHERE foo@dbs2(ad_sourcetext, 'aa') > 0;
-- foo is a user-define function returning a NUMBER

DELETE FROM print_media@dbs2 WHERE DBMS_LOB.GETLENGTH@dbs2(ad_graphic) = 0;
```

Restrictions on Remote User-Defined Functions

- The restrictions that apply to SQL functions apply here also.

> **See Also:** "Built-in Functions for Remote LOBs and BFILEs" on page 16-11

- A function in one dblink cannot operate on LOB data in another dblink.For example, **the following statement is not supported:**

```
SELECT a.product_id FROM print_media@dbs1 a, print_media@dbs2 b WHERE
   CONTAINS@dbs1(b.ad_sourcetext, 'aa') > 0;
```

- One query block cannot contain tables and functions at different dblinks. For example, **the following statement is not supported:**

```
SELECT a.product_id FROM print_media@dbs2 a, print_media@dbs3 b
    WHERE CONTAINS@dbs2(a.ad_sourcetext, 'aa') > 0 AND
    foo@dbs3(b.ad_sourcetext) > 0;
--  foo is a user-defined function in dbs3
```

- There is no support for performing remote LOB operations (that is, DBMS_LOB) from within PL/SQL, other than issuing SQL statements from PL/SQL.

Remote Functions in PL/SQL, OCI, and JDBC

All the SQL statements listed above work the same if they are executed from inside PL/SQL, OCI, and JDBC. No additional functionality is provided.

18

Migrating Columns from LONGs to LOBs

This chapter describes techniques for migrating tables that use LONG data types to LOB data types. This chapter contains these topics:

- Benefits of Migrating LONG Columns to LOB Columns
- Preconditions for Migrating LONG Columns to LOB Columns
- Using utldtree.sql to Determine how to Optimize the Application
- Converting Tables from LONG to LOB Data Types
- Migrating Applications from LONGs to LOBs

> **See Also:** The following chapters in this book describe support for LOB data types in various programming environments:
>
> - Chapter 16, "SQL Semantics and LOBs"
> - Chapter 17, "PL/SQL Semantics for LOBs"
> - Chapter 20, "Data Interface for Persistent LOBs"

Benefits of Migrating LONG Columns to LOB Columns

There are many benefits to migrating table columns from LONG data types to LOB data types.

> **Note:** You can use the techniques described in this chapter to do either of the following:
>
> - Convert columns of type LONG to either CLOB or NCLOB columns
> - Convert columns of type LONG RAW to BLOB type columns
>
> Unless otherwise noted, discussions in this chapter regarding LONG to LOB conversions apply to both of these data type conversions.

The following list compares the semantics of LONG and LOB data types in various application development scenarios:

- The number of LONG type columns is limited. Any given table can have a maximum of only one LONG type column. The number of LOB type columns in a table is not limited.
- You can use the data interface for LOBs to enable replication of tables that contain LONG or LONG RAW columns. Replication is allowed on LOB columns, but is not

supported for LONG and LONG RAW columns. The database omits columns containing LONG and LONG RAW data types from replicated tables.

If a table is replicated or has materialized views, and its LONG column is changed to LOB, then you may have to manually fix the replicas.

Caution: Oracle does not support converting LOBs into LONGs. Ensure that you have no requirement to maintain any column as a LONG before converting it into a LOB.

Preconditions for Migrating LONG Columns to LOB Columns

This section describes preconditions that must be met before converting a LONG column to a LOB column.

See Also: "Migrating Applications from LONGs to LOBs" on page 18-7 before converting your table to determine whether any limitations on LOB columns prevent you from converting to LOBs.

Dropping a Domain Index on a LONG Column Before Converting to a LOB

Any domain index on a LONG column must be dropped before converting the LONG column to LOB column. See "Indexes on Columns Converted from LONG to LOB Data Types" on page 18-8 for more information.

Preventing Generation of Redo Space on Tables Converted to LOB Data Types

Generation of redo space can cause performance problems during the process of converting LONG columns. Redo changes for the table are logged during the conversion process only if the table has LOGGING on.

Redo changes for the column being converted from LONG to LOB are logged only if the storage characteristics of the LOB column indicate LOGGING. The logging setting (LOGGING or NOLOGGING) for the LOB column is inherited from the tablespace in which the LOB is created.

To prevent generation of redo space during migration, do the following before migrating your table (syntax is in BNF):

1. `ALTER TABLE Long_tab NOLOGGING;`
2. `ALTER TABLE Long_tab MODIFY (long_col CLOB [DEFAULT <default_val>]) LOB (long_col) STORE AS (NOCACHE NOLOGGING);`

   Note that you must also specify NOCACHE when you specify NOLOGGING in the STORE AS clause.
3. `ALTER TABLE Long_tab MODIFY LOB (long_col) (CACHE);`
4. `ALTER TABLE Long_tab LOGGING;`
5. Make a backup of the tablespaces containing the table and the LOB column.

Using utldtree.sql to Determine how to Optimize the Application

You can use the utility, rdbms/admin/utldtree.sql, to determine which parts of your application require rewriting when you migrate your table from LONG to LOB column types. This utility enables you to recursively see all objects that are dependent

on a given object. For example, you can see all objects which depend on a table with a LONG column. You can only see objects for which you have permission.

Instructions on how to use utldtree.sql are documented in the file itself. Also, utldtree.sql is only needed for PL/SQL. For SQL and OCI, you have no requirement to change your applications.

Converting Tables from LONG to LOB Data Types

This section describes the following techniques for migrating existing tables from LONG to LOB data types:

- "Using ALTER TABLE to Convert LONG Columns to LOB Columns" on page 18-3
- "Copying a LONG to a LOB Column Using the TO_LOB Operator" on page 18-4
- "Online Redefinition of Tables with LONG Columns" on page 18-5 where high availability is critical
- "Using Oracle Data Pump to Migrate a Database" on page 18-7 when you can convert using this utility

Using ALTER TABLE to Convert LONG Columns to LOB Columns

You can use the ALTER TABLE statement in SQL to convert a LONG column to a LOB column. To do so, use the following syntax:

```
ALTER TABLE [<schema>.]<table_name>
   MODIFY ( <long_column_name> { CLOB | BLOB | NCLOB }
  [DEFAULT <default_value>]
) [LOB_storage_clause];
```

For example, if you had a table that was created as follows:

```
CREATE TABLE Long_tab (id NUMBER, long_col LONG);
```

then you can change the column long_col in table Long_tab to data type CLOB using following ALTER TABLE statement:

```
ALTER TABLE Long_tab MODIFY ( long_col CLOB );
```

Note: The ALTER TABLE statement copies the contents of the table into a new space, and frees the old space at the end of the operation. This temporarily doubles the space requirements.

Note that when using the ALTER TABLE statement to convert a LONG column to a LOB column, only the following options are allowed:

- DEFAULT which enables you to specify a default value for the LOB column.
- The *LOB_storage_clause*, which enables you to specify the LOB storage characteristics for the converted column, can be specified in the MODIFY clause.

Other ALTER TABLE options are not allowed when converting a LONG column to a LOB type column.

Migration Issues

General issues concerning migration include the following:

- All constraints of your previous LONG columns are maintained for the new LOB columns. The only constraint allowed on LONG columns are NULL and NOT NULL. To alter the constraints for these columns, or alter any other columns or properties of this table, you have to do so in a subsequent ALTER TABLE statement.
- If you do not specify a default value, then the default value for the LONG column becomes the default value of the LOB column.
- Most of the existing triggers on your table are still usable, however UPDATE OF triggers can cause issues. See "Migrating Applications from LONGs to LOBs" on page 18-7 for more details.

Copying a LONG to a LOB Column Using the TO_LOB Operator

If you do not want to use ALTER TABLE, as described earlier in this section, then you can use the TO_LOB operator on a LONG column to copy it to a LOB column. You can use the CREATE TABLE AS SELECT statement or the INSERT AS SELECT statement with the TO_LOB operator to copy data from a LONG column to a CLOB or NCLOB column, or from a LONG RAW column to a BLOB column. For example, if you have a table with a LONG column that was created as follows:

```
CREATE TABLE Long_tab (id NUMBER, long_col LONG);
```

then you can do the following to copy the column to a LOB column:

```
CREATE TABLE Lob_tab (id NUMBER, clob_col CLOB);
INSERT INTO Lob_tab SELECT id, TO_LOB(long_col) FROM long_tab;
COMMIT;
```

If the INSERT returns an error (because of lack of undo space), then you can incrementally migrate LONG data to the LOB column using the WHERE clause. After you ensure that the data is accurately copied, you can drop the original table and create a view or synonym for the new table using one of the following sequences:

```
DROP TABLE Long_tab;
CREATE VIEW Long_tab (id, long_col) AS SELECT * from Lob_tab;
```

or

```
DROP TABLE Long_tab;
CREATE SYNONYM Long_tab FOR Lob_tab;
```

This series of operations is equivalent to changing the data type of the column Long_col of table Long_tab from LONG to CLOB. With this technique, you have to re-create any constraints, triggers, grants and indexes on the new table.

Use of the TO_LOB operator is subject to the following limitations:

- You can use TO_LOB to copy data to a LOB column, but not to a LOB attribute of an object type.
- You cannot use TO_LOB with a remote table. For example, the following statements do not work:

  ```
  INSERT INTO tb1@dblink (lob_col) SELECT TO_LOB(long_col) FROM tb2;
  INSERT INTO tb1 (lob_col) SELECT TO_LOB(long_col) FROM tb2@dblink;
  CREATE TABLE tb1 AS SELECT TO_LOB(long_col) FROM tb2@dblink;
  ```

- The TO_LOB operator cannot be used in the CREATE TABLE AS SELECT statement to convert a LONG or LONG RAW column to a LOB column when creating an index organized table.

To work around this limitation, create the index organized table, and then do an INSERT AS SELECT of the LONG or LONG RAW column using the TO_LOB operator.

- You cannot use TO_LOB inside any PL/SQL block.

Online Redefinition of Tables with LONG Columns

Tables with LONG and LONG RAW columns can be migrated using online table redefinition. This technique is suitable for migrating LONG columns in database tables where high availability is critical.

To use this technique, you must convert LONG columns to LOB types during the redefinition process as follows:

- Any LONG column must be converted to a CLOB or NCLOB column.
- Any LONG RAW column must be converted to a BLOB column.

This conversion is performed using the TO_LOB() operator in the column mapping of the DBMS_REDEFINITION.START_REDEF_TABLE() procedure.

Note: You cannot perform online redefinition of tables with LONG or LONG RAW columns unless you convert the columns to LOB types as described in this section.

General tasks involved in the online redefinition process are given in the following list. Issues specific to converting LONG and LONG RAW columns are called out. See the related documentation referenced at the end of this section for additional details on the online redefinition process that are not described here.

- Create an empty interim table. This table holds the migrated data when the redefinition process is done. In the interim table:
  - Define a CLOB or NCLOB column for each LONG column in the original table that you are migrating.
  - Define a BLOB column for each LONG RAW column in the original table that you are migrating.
- Start the redefinition process. To do so, call DBMS_REDEFINITION.START_REDEF_TABLE and pass the column mapping using the TO_LOB operator as follows:

```
DBMS_REDEFINITION.START_REDEF_TABLE(
    'schema_name',
    'original_table',
    'interim_table',
    'TO_LOB(long_col_name) lob_col_name',
    'options_flag',
    'orderby_cols');
```

  where *long_col_name* is the name of the LONG or LONG RAW column that you are converting in the original table and *lob_col_name* is the name of the LOB column in the interim table. This LOB column holds the converted data.
- Call the DBMS_REDEFINITION.COPY_TABLE_DEPENDENTS procedure as described in the related documentation.
- Call the DBMS_REDEFINITION.FINISH_REDEF_TABLE procedure as described in the related documentation.

Parallel Online Redefinition

On a system with sufficient resources for parallel execution, redefinition of a `LONG` column to a LOB column can be executed in parallel under the following conditions:

In the case where the destination table is non-partitioned:

- The segment used to store the LOB column in the destination table belongs to a locally managed tablespace with Automatic Segment Space Management (ASSM) enabled, which is now the default.
- There is a simple mapping from one `LONG` column to one LOB column, and the destination table has only one LOB column.

In the case where the destination table is partitioned, the normal methods for parallel execution for partitioning apply. When the destination table is partitioned, then online redefinition is executed in parallel.

Example of Online Redefinition

The following example demonstrates online redefinition with LOB columns.

```
REM Grant privileges required for online redefinition.
GRANT execute ON DBMS_REDEFINITION TO pm;
GRANT ALTER ANY TABLE TO pm;
GRANT DROP ANY TABLE TO pm;
GRANT LOCK ANY TABLE TO pm;
GRANT CREATE ANY TABLE TO pm;
GRANT SELECT ANY TABLE TO pm;

REM Privileges required to perform cloning of dependent objects.
GRANT CREATE ANY TRIGGER TO pm;
GRANT CREATE ANY INDEX TO pm;

connect pm/passwd

drop table cust;
create table cust(c_id   number primary key,
                  c_zip  number,
                  c_name varchar(30) default null,
                  c_long long
                  );
insert into cust values(1, 94065, 'hhh', 'ttt');

-- Creating Interim Table
-- There is no requirement to specify constraints because they are
-- copied over from the original table.
create table cust_int(c_id   number not null,
                  c_zip  number,
                  c_name varchar(30) default null,
                  c_long clob
                  );

declare
 col_mapping varchar2(1000);
BEGIN
--  map all the columns in the interim table to the original table
 col_mapping :=
               'c_id            c_id  , '||
               'c_zip           c_zip , '||
               'c_name          c_name, '||
               'to_lob(c_long)  c_long';
```

```
dbms_redefinition.start_redef_table('pm', 'cust', 'cust_int', col_mapping);
END;
/

declare
 error_count pls_integer := 0;
BEGIN
  dbms_redefinition.copy_table_dependents('pm', 'cust', 'cust_int',
                                          1, true, true, true, false,
                                          error_count);

  dbms_output.put_line('errors := ' || to_char(error_count));
END;
/

exec  dbms_redefinition.finish_redef_table('pm', 'cust', 'cust_int');

-- Drop the interim table
drop table cust_int;

desc cust;

-- The following insert statement fails. This illustrates
-- that the primary key constraint on the c_id column is
-- preserved after migration.

insert into cust values(1, 94065, 'hhh', 'ttt');

select * from cust;
```

See Also: The following related documentation provides additional details on the redefinition process described earlier in this section:

- *Oracle Database Administrator's Guide* gives detailed procedures and examples of redefining tables online.
- *Oracle Database PL/SQL Packages and Types Reference* includes information on syntax and other details on usage of procedures in the `DBMS_REDEFINITION` package.

Using Oracle Data Pump to Migrate a Database

If you are exporting data as part of a migration to a new database, create a table on the destination database with LOB columns and Data Pump calls the LONG-to-LOB function implicitly.

For details on using Oracle Data Pump, refer to *Oracle Database Utilities*.

Migrating Applications from LONGs to LOBs

This section discusses differences between `LONG` and LOB data types that may impact your application migration plans or require you to modify your application.

Most APIs that work with LONG data types in the PL/SQL and OCI environments are enhanced to also work with LOB data types. These APIs are collectively referred to as the *data interface for persistent LOBs*, or simply the *data interface*. Among other things, the data interface provides the following benefits:

- Changes needed are minimal in PL/SQL and OCI applications that use tables with columns converted from LONG to LOB data types.
- You can work with LOB data types in your application without having to deal with LOB locators.

See Also:

- Chapter 20, "Data Interface for Persistent LOBs" for details on PL/SQL and OCI APIs included in the data interface.
- Chapter 16, "SQL Semantics and LOBs" for details on SQL syntax supported for LOB data types.
- Chapter 17, "PL/SQL Semantics for LOBs" for details on PL/SQL syntax supported for LOB data types.

LOB Columns Are Not Allowed in Clustered Tables

LOB columns are not allowed in clustered tables, whereas LONGs are allowed. If a table is a part of a cluster, then any LONG or LONG RAW column cannot be changed to a LOB column.

LOB Columns Are Not Allowed in AFTER UPDATE OF Triggers

You cannot have LOB columns in the UPDATE OF list of an AFTER UPDATE OF trigger. LONG columns are allowed in such triggers. For example, the following create trigger statement is not valid:

```
CREATE TABLE t(lobcol CLOB);
CREATE TRIGGER trig AFTER UPDATE OF lobcol ON t ...;
```

All other triggers work on LOB columns.

Indexes on Columns Converted from LONG to LOB Data Types

Indexes on any column of the table being migrated must be manually rebuilt after converting any LONG column to a LOB column. This includes function-based indexes.

Any function-based index on a LONG column is unusable during the conversion process and must be rebuilt after converting. Application code that uses function-based indexing should work without modification after converting.

Note that, any domain indexes on a LONG column must be dropped before converting the LONG column to LOB column. You can rebuild the domain index after converting.

To rebuild an index after converting, use the following steps:

1. Select the index from your original table as follows:

```
SELECT index_name FROM user_indexes WHERE table_name='LONG_TAB';
```

Note: The table name must be capitalized in this query.

2. For the selected index, use the command:

```
ALTER INDEX <index> REBUILD
```

Empty LOBs Compared to NULL and Zero Length LONGs

A LOB column can hold an *empty* LOB. An empty LOB is a LOB locator that is fully initialized, but not populated with data. Because LONG data types do not use locators, the *empty* concept does not apply to LONG data types.

Both LOB column values and LONG column values, inserted with an initial value of NULL or an empty string literal, have a NULL value. Therefore, application code that uses NULL or zero-length values in a LONG column functions exactly the same after you convert the column to a LOB type column.

In contrast, a LOB initialized to empty has a non-NULL value as illustrated in the following example:

```
CREATE TABLE long_tab(id NUMBER, long_col LONG);
CREATE TABLE lob_tab(id NUMBER, lob_col CLOB);

INSERT INTO long_tab values(1, NULL);

REM     A zero length string inserts a NULL into the LONG column:
INSERT INTO long_tab values(1, '');

INSERT INTO lob_tab values(1, NULL);

REM     A zero length string inserts a NULL into the LOB column:
INSERT INTO lob_tab values(1, '');

REM     Inserting an empty LOB inserts a non-NULL value:
INSERT INTO lob_tab values(1, empty_clob());

DROP TABLE long_tab;
DROP TABLE lob_tab;
```

Overloading with Anchored Types

For applications using anchored types, some overloaded variables resolve to different targets during the conversion to LOBs. For example, given the procedure p overloaded with specifications 1 and 2:

```
procedure p(l long) is ...;       -- (specification 1)
procedure p(c clob) is ...;       -- (specification 2)
```

and the procedure call:

```
declare
     var  longtab.longcol%type;
   BEGIN
     ...
   p(var);
     ...
END;
```

Prior to migrating from LONG to LOB columns, this call would resolve to specification 1. Once longtab is migrated to LOB columns this call resolves to specification 2. Note that this would also be true if the parameter type in specification 1 were a CHAR, VARCHAR2, RAW, LONG RAW.

If you have migrated you tables from LONG columns to LOB columns, then you must manually examine your applications and determine whether overloaded procedures must be changed.

Some applications that included overloaded procedures with LOB arguments before migrating may still break. This includes applications that do not use LONG anchored types. For example, given the following specifications (1 and 2) and procedure call for procedure p:

```
procedure p(n number) is ...;       -- (1)
procedure p(c clob) is ...;         -- (2)

p('123');                -- procedure call
```

Before migrating, the only conversion allowed was CHAR to NUMBER, so specification 1 would be chosen. After migrating, both conversions are allowed, so the call is ambiguous and raises an overloading error.

Some Implicit Conversions Are Not Supported for LOB Data Types

PL/SQL permits implicit conversion from NUMBER, DATE, ROW_ID, BINARY_INTEGER, and PLS_INTEGER data types to a LONG; however, implicit conversion from these data types to a LOB is not allowed.

If your application uses these implicit conversions, then you have to explicitly convert these types using the TO_CHAR operator for character data or the TO_RAW operator for binary data. For example, if your application has an assignment operation such as:

```
number_var := long_var;  -- The RHS is a LOB variable after converting.
```

then you must modify your code as follows:

```
number_var := TO_CHAR(long_var);
-- Assuming that long_var is of type CLOB after conversion
```

The following conversions are not supported for LOB types:

- BLOB to VARCHAR2, CHAR, or LONG
- CLOB to RAW or LONG RAW

This applies to all operations where implicit conversion takes place. For example if you have a SELECT statement in your application as follows:

```
SELECT long_raw_column INTO my_varchar2 VARIABLE FROM my_table
```

and long_raw_column is a BLOB after converting your table, then the SELECT statement produces an error. To make this conversion work, you must use the TO_RAW operator to explicitly convert the BLOB to a RAW as follows:

```
SELECT TO_RAW(long_raw_column) INTO my_varchar2 VARIABLE FROM my_table
```

The same holds for selecting a CLOB into a RAW variable, or for assignments of CLOB to RAW and BLOB to VARCHAR2.

Part V

Using LOB APIs

This part provides details on using LOB APIs in supported environments. Examples of LOB API usage are given.

This part contains these chapters:

- Chapter 19, "Operations Specific to Persistent and Temporary LOBs"
- Chapter 20, "Data Interface for Persistent LOBs"
- Chapter 22, "Using LOB APIs"
- Chapter 21, "LOB APIs for BFILE Operations"

19

Operations Specific to Persistent and Temporary LOBs

This chapter discusses LOB operations that differ between persistent and temporary LOB instances. This chapter contains these topics:

- Persistent LOB Operations
- Temporary LOB Operations
- Creating Persistent and Temporary LOBs in PL/SQL
- Freeing Temporary LOBs in OCI

See Also:

- Chapter 22, "Using LOB APIs" gives details and examples of API usage for LOB APIs that can be used with either temporary or persistent LOBs.
- Chapter 21, "LOB APIs for BFILE Operations" gives details and examples for usage of LOB APIs that operate on BFILEs.

Persistent LOB Operations

This section describes operations that apply only to persistent LOBs.

Inserting a LOB into a Table

You can insert LOB instances into persistent LOB columns using any of the methods described in Chapter 15, "DDL and DML Statements with LOBs".

Selecting a LOB from a Table

You can select a persistent LOB from a table just as you would any other data type. In the following example, persistent LOB instances of different types are selected into PL/SQL variables.

```
declare
  blob1 BLOB;
  blob2 BLOB;
  clob1 CLOB;
  nclob1 NCLOB;
BEGIN
  SELECT ad_photo INTO blob1 FROM print_media WHERE Product_id = 2268
        FOR UPDATE;
```

```
  SELECT ad_photo INTO blob2 FROM print_media WHERE Product_id = 3106;

  SELECT ad_sourcetext INTO clob1 FROM Print_media
      WHERE product_id=3106 and ad_id=13001 FOR UPDATE;

  SELECT ad_fltextn INTO nclob1 FROM Print_media
      WHERE product_id=3060 and ad_id=11001 FOR UPDATE;

END;
/
show errors;
```

Temporary LOB Operations

This section describes operations that apply only to temporary LOB instances.

Creating and Freeing a Temporary LOB

To create a temporary LOB instance, you must declare a variable of the given LOB data type and pass the variable to the `CREATETEMPORARY` API. The temporary LOB instance exists in your application until it goes out of scope, your session terminates, or you explicitly free the instance. Freeing a temporary LOB instance is recommended to free system resources.

The following example demonstrates how to create and free a temporary LOB in the PL/SQL environment using the DBMS_LOB package.

```
declare
  blob1 BLOB;
  blob2 BLOB;
  clob1 CLOB;
  nclob1 NCLOB;
BEGIN
  -- create temp LOBs
  DBMS_LOB.CREATETEMPORARY(blob1,TRUE, DBMS_LOB.SESSION);
  DBMS_LOB.CREATETEMPORARY(blob2,TRUE, DBMS_LOB.SESSION);
  DBMS_LOB.CREATETEMPORARY(clob1,TRUE, DBMS_LOB.SESSION);
  DBMS_LOB.CREATETEMPORARY(nclob1,TRUE, DBMS_LOB.SESSION);

  -- fill with data
  writeDataToLOB_proc(blob1);
  writeDataToLOB_proc(blob2);

  -- CHAR->LOB conversion
  clob1 := 'abcde';
  nclob1 := TO_NCLOB(clob1);

  -- Other APIs
  call_lob_apis(blob1, blob2, clob1, nclob1);

  -- free temp LOBs
  DBMS_LOB.FREETEMPORARY(blob1);
  DBMS_LOB.FREETEMPORARY(blob2);
  DBMS_LOB.FREETEMPORARY(clob1);
  DBMS_LOB.FREETEMPORARY(nclob1);

END;
/
show errors;
```

Creating Persistent and Temporary LOBs in PL/SQL

The code example that follows illustrates how to create persistent and temporary LOBs in PL/SQL. This code is in the demonstration file:

```
$ORACLE_HOME/rdbms/demo/lobs/plsql/lobdemo.sql
```

This demonstration file also calls procedures in separate PL/SQL files that illustrate usage of other LOB APIs. For a list of these files and links to more information about related LOB APIs, see "PL/SQL LOB Demonstration Files" on page A-1.

```
--------------------------------------------------------------------------------
------------------------  Persistent LOB operations ------------------------
--------------------------------------------------------------------------------

declare
  blob1 BLOB;
  blob2 BLOB;
  clob1 CLOB;
  nclob1 NCLOB;
BEGIN
  SELECT ad_photo INTO blob1 FROM print_media WHERE Product_id = 2268
        FOR UPDATE;
  SELECT ad_photo INTO blob2 FROM print_media WHERE Product_id = 3106;

  SELECT ad_sourcetext INTO clob1 FROM Print_media
      WHERE product_id=3106 and ad_id=13001 FOR UPDATE;

  SELECT ad_fltextn INTO nclob1 FROM Print_media
      WHERE product_id=3060 and ad_id=11001 FOR UPDATE;

  call_lob_apis(blob1, blob2, clob1, nclob1);
  rollback;
END;
/
show errors;

--------------------------------------------------------------------------------
------------------------  Temporary LOB operations ------------------------
--------------------------------------------------------------------------------

declare
  blob1 BLOB;
  blob2 BLOB;
  clob1 CLOB;
  nclob1 NCLOB;
BEGIN
  -- create temp LOBs
  DBMS_LOB.CREATETEMPORARY(blob1,TRUE, DBMS_LOB.SESSION);
  DBMS_LOB.CREATETEMPORARY(blob2,TRUE, DBMS_LOB.SESSION);
  DBMS_LOB.CREATETEMPORARY(clob1,TRUE, DBMS_LOB.SESSION);
  DBMS_LOB.CREATETEMPORARY(nclob1,TRUE, DBMS_LOB.SESSION);

  -- fill with data
  writeDataToLOB_proc(blob1);
  writeDataToLOB_proc(blob2);

  -- CHAR->LOB conversion
  clob1 := 'abcde';
  nclob1 := TO_NCLOB(clob1);
```

```
  -- Other APIs
  call_lob_apis(blob1, blob2, clob1, nclob1);

  -- free temp LOBs
  DBMS_LOB.FREETEMPORARY(blob1);
  DBMS_LOB.FREETEMPORARY(blob2);
  DBMS_LOB.FREETEMPORARY(clob1);
  DBMS_LOB.FREETEMPORARY(nclob1);

END;
/
show errors;
```

Freeing Temporary LOBs in OCI

Any time that your OCI program obtains a LOB locator from SQL or PL/SQL, check that the locator is temporary. If it is, free the locator when your application is finished with it. The locator can be from a define during a select or an out bind. A temporary LOB duration is always upgraded to session when it is shipped to the client side. The application must do the following before the locator is overwritten by the locator of the next row:

```
OCILobIsTemporary(env, err, locator, is_temporary);
if(is_temporary)
    OCILobFreeTemporary(svc, err, locator);
```

> **See Also:** *Oracle Call Interface Programmer's Guide* chapter 16, section "LOB Functions."

20

Data Interface for Persistent LOBs

This chapter contains these topics:

Overview of the Data Interface for Persistent LOBs

The data interface for persistent LOBs includes a set of Java, PL/SQL, and OCI APIs that are extended to work with LOB data types. These APIs, originally designed for use with legacy data types such as `LONG`, `LONG RAW`, and `VARCHAR2`, can also be used with the corresponding LOB data types shown in Table 20–1 and Table 20–2. These tables show the legacy data types in the *bind or define type* column and the corresponding supported LOB data type in the *LOB column type* column. You can use the data interface for LOBs to store and manipulate character data and binary data in a LOB column just as if it were stored in the corresponding legacy data type.

Note: The data interface works for LOB columns and LOBs that are attributes of objects. In this chapter *LOB columns* means LOB columns and LOB attributes.

You can use array bind and define interfaces to insert and select multiple rows in one round-trip.

For simplicity, this chapter focuses on character data types; however, the same concepts apply to the full set of character and binary data types listed in Table 20–1 and Table 20–2. `CLOB` also means `NCLOB` in these tables.

Table 20–1 Corresponding LONG and LOB Data Types in SQL and PL/SQL

Bind or Define Type	LOB Column Type	Used For Storing
`CHAR`	`CLOB`	Character data
`LONG`	`CLOB`	Character data
`VARCHAR2`	`CLOB`	Character data

Table 20–1 (Cont.) Corresponding LONG and LOB Data Types in SQL and PL/SQL

Bind or Define Type	LOB Column Type	Used For Storing
LONG RAW	BLOB	Binary data
RAW	BLOB	Binary data

Table 20–2 Corresponding LONG and LOB Data Types in OCI

Bind or Define Type	LOB Column Type	Used For Storing
SQLT_AFC(n)	CLOB	Character data
SQLT_CHR	CLOB	Character data
SQLT_LNG	CLOB	Character data
SQLT_VCS	CLOB	Character data
SQLT_BIN	BLOB	Binary data
SQLT_LBI	BLOB	Binary data
SQLT_LVB	BLOB	Binary data

Benefits of Using the Data Interface for Persistent LOBs

Using the data interface for persistent LOBs has the following benefits:

- If your application uses LONG data types, then you can use the same application with LOB data types with little or no modification of your existing application required. To do so, just convert LONG audiotape columns in your tables to LOB audiotape columns as discussed in Chapter 18, "Migrating Columns from LONGs to LOBs".
- Performance is better for OCI applications that use sequential access techniques. A piecewise INSERT or fetch using the data interface has comparable performance to using OCI functions like OCILobRead2() and OCILobWrite2(). Because the data interface allows more than 4K bytes of data to be inserted into a LOB in a single OCI call, a round-trip to the server is saved.
- You can read LOB data in one OCIStmtFetch() call, instead of fetching the LOB locator first and then calling OCILobRead2(). This improves performance when you want to read LOB data starting at the beginning.
- You can use array bind and define interfaces to insert and select multiple rows with LOBs in one round trip.

Using the Data Interface for Persistent LOBs in PL/SQL

The data interface enables you to use LONG and LOB data types listed in Table 20–1 to perform the following operations in PL/SQL:

- INSERT or UPDATE character data stored in datatypes such as VARCHAR2, CHAR, or LONG into a CLOB column.
- INSERT or UPDATE binary data stored in datatypes such as RAW or LONG RAW into a BLOB column.
- Use the SELECT statement on CLOB columns to select data into a character buffer variable such as CHAR, LONG, or VARCHAR2.

- Use the SELECT statement on BLOB columns to select data into a binary buffer variable such as RAW and LONG RAW.
- Make cross-type assignments (implicit type conversions) between CLOB and VARCHAR2, CHAR, or LONG variables.
- Make cross-type assignments (implicit type conversions) between BLOB and RAW or LONG RAW variables.
- Pass LOB datatypes to functions defined to accept LONG datatypes or pass LONG datatypes to functions defined to accept LOB datatypes. For example, you can pass a CLOB instance to a function defined to accept another character type, such as VARCHAR2, CHAR, or LONG.
- Use CLOBs with other PL/SQL functions and operators that accept VARCHAR2 arguments such as INSTR and SUBSTR. See "Passing CLOBs to SQL and PL/SQL Built-In Functions" on page 20-4 for a complete list.

Note: When using the data interface for LOBs with the SELECT statement in PL/SQL, you cannot specify the amount you want to read. You can only specify the buffer length of your buffer. If your buffer length is smaller than the LOB data length, then the database throws an exception.

See Also:

- Chapter 16, "SQL Semantics and LOBs" for details on LOB support in SQL statements
- "Some Implicit Conversions Are Not Supported for LOB Data Types" on page 18-10

Guidelines for Accessing LOB Columns Using the Data Interface in SQL and PL/SQL

This section describes techniques you use to access LOB columns or attributes using the data interface for persistent LOBs.

Data from CLOB and BLOB columns or attributes can be referenced by regular SQL statements, such as INSERT, UPDATE, and SELECT.

There is no piecewise INSERT, UPDATE, or fetch routine in PL/SQL. Therefore, the amount of data that can be accessed from a LOB column or attribute is limited by the maximum character buffer size. PL/SQL supports character buffer sizes up to 32KB - 1 (32767 bytes). For this reason, only LOBs less than 32K bytes in size can be accessed by PL/SQL applications using the data interface for persistent LOBs.

If you must access more than 32KB -1 using the data interface, then you must make OCI calls from the PL/SQL code to use the APIs for piece-wise insert and fetch.

Use the following guidelines for using the data interface to access LOB columns or attributes:

- INSERT operations

  You can INSERT into tables containing LOB columns or attributes using regular INSERT statements in the VALUES clause. The field of the LOB column can be a literal, a character datatype, a binary datatype, or a LOB locator.

- UPDATE operations

LOB columns or attributes can be updated as a whole by UPDATE... SET statements. In the SET clause, the new value can be a literal, a character datatype, a binary datatype, or a LOB locator.

- 4000 byte limit on hexadecimal to raw and raw to hexadecimal conversions

  The database does not do implicit hexadecimal to RAW or RAW to hexadecimal conversions on data that is more than 4000 bytes in size. You cannot bind a buffer of character data to a binary datatype column, and you cannot bind a buffer of binary data to a character datatype column if the buffer is over 4000 bytes in size. Attempting to do so results in your column data being truncated at 4000 bytes.

  For example, you cannot bind a VARCHAR2 buffer to a LONG RAW or a BLOB column if the buffer is more than 4000 bytes in size. Similarly, you cannot bind a RAW buffer to a LONG or a CLOB column if the buffer is more than 4000 bytes in size.

- SELECT operations

  LOB columns or attributes can be selected into character or binary buffers in PL/SQL. If the LOB column or attribute is longer than the buffer size, then an exception is raised without filling the buffer with any data. LOB columns or attributes can also be selected into LOB locators.

Implicit Assignment and Parameter Passing

Implicit assignment and parameter passing are supported for LOB columns. For the data types listed in Table 20–1 and Table 20–2, you can pass or assign: any character type to any other character type, or any binary type to any other binary type using the data interface for persistent LOBs.

Implicit assignment works for variables declared explicitly and for variables declared by referencing an existing column type using the %TYPE attribute as show in the following example. This example assumes that column long_col in table t has been migrated from a LONG to a CLOB column.

```
CREATE TABLE t (long_col LONG); -- Alter this table to change LONG column to LOB
DECLARE
   a VARCHAR2(100);
   b t.long_col%type; -- This variable changes from LONG to CLOB
BEGIN
   SELECT * INTO b FROM t;
   a := b;  -- This changes from "VARCHAR2 := LONG to VARCHAR2 := CLOB
   b := a;  -- This changes from "LONG := VARCHAR2 to CLOB := VARCHAR2
END;
```

Implicit parameter passing is allowed between functions and procedures. For example, you can pass a CLOB to a function or procedure where the formal parameter is defined as a VARCHAR2.

Note: The assigning a VARCHAR2 buffer to a LOB variable is somewhat less efficient than assigning a VARCHAR2 to a LONG variable because the former involves creating a temporary LOB. Therefore, PL/SQL users experience a slight deterioration in the performance of their applications.

Passing CLOBs to SQL and PL/SQL Built-In Functions

Implicit parameter passing is also supported for built-in PL/SQL functions that accept character data. For example, INSTR can accept a CLOB and other character data.

Any SQL or PL/SQL built-in function that accepts a VARCHAR2 can accept a CLOB as an argument. Similarly, a VARCHAR2 variable can be passed to any DBMS_LOB API for any parameter that takes a LOB locator.

See Also: Chapter 16, "SQL Semantics and LOBs"

Explicit Conversion Functions

In PL/SQL, the following explicit conversion functions convert other data types to CLOB and BLOB datatypes as follows:

- TO_CLOB() converts LONG, VARCHAR2, and CHAR to CLOB
- TO_BLOB() converts LONG RAW and RAW to BLOB

Also note that the conversion function TO_CHAR() can convert a CLOB to a CHAR type.

Calling PL/SQL and C Procedures from SQL

When a PL/SQL or C procedure is called from SQL, buffers with more than 4000 bytes of data are not allowed.

Calling PL/SQL and C Procedures from PL/SQL

You can call a PL/SQL or C procedure from PL/SQL. You can pass a CLOB as an actual parameter where CHR is the formal parameter, or vice versa. The same holds for BLOBs and RAWs.

One example of when these cases can arise is when either the formal or the actual parameter is an anchored type, that is, the variable is declared using the *table_name.column_name*%type syntax.

PL/SQL procedures or functions can accept a CLOB or a VARCHAR2 as a formal parameter. For example the PL/SQL procedure could be one of the following:

- When the formal parameter is a CLOB:

```
CREATE OR REPLACE PROCEDURE get_lob(table_name IN VARCHAR2, lob INOUT
CLOB) AS
   ...
BEGIN
  ...
END;
/
```

- When the formal parameter is a VARCHAR2:

```
CREATE OR REPLACE PROCEDURE get_lob(table_name IN VARCHAR2, lob INOUT
VARCHAR2) AS
   ...
BEGIN
  ...
END;
/
```

The calling function could be of any of the following types:

- When the actual parameter is a CHR:

```
create procedure ...
declare
c VARCHAR2[200];
BEGIN
```

```
  get_lob('table_name', c);
END;
```

- When the actual parameter is a CLOB:

```
create procedure ...
declare
c CLOB;
BEGIN
  get_lob('table_name', c);
END;
```

Binds of All Sizes in INSERT and UPDATE Operations

Binds of all sizes are supported for INSERT and UPDATE operations on LOB columns. Multiple binds of any size are allowed in a single INSERT or UPDATE statement.

> **Note:** When you create a table, the length of the default value you specify for any LOB column is restricted to 4000 bytes.

4000 Byte Limit on Results of a SQL Operator

If you bind more than 4000 bytes of data to a BLOB or a CLOB, and the data consists of a SQL operator, then Oracle Database limits the size of the result to at most 4000 bytes.

The following statement inserts only 4000 bytes because the result of LPAD is limited to 4000 bytes:

```
INSERT INTO print_media (ad_sourcetext) VALUES (lpad('a', 5000, 'a'));
```

The following statement inserts only 2000 bytes because the result of LPAD is limited to 4000 bytes, and the implicit hexadecimal to raw conversion converts it to 2000 bytes of RAW data:

```
INSERT INTO print_media (ad_photo) VALUES (lpad('a', 5000, 'a'));
```

Example of 4000 Byte Result Limit of a SQL Operator

The following example illustrates how the result for SQL operators is limited to 4000 bytes.

```
/* The following command inserts only 4000 bytes because the result of
 * LPAD is limited to 4000 bytes */
INSERT INTO print_media(product_id, ad_id, ad_sourcetext)
  VALUES (2004, 5, lpad('a', 5000, 'a'));
SELECT LENGTH(ad_sourcetext) FROM print_media
  WHERE product_id=2004 AND ad_id=5;
ROLLBACK;

/* The following command inserts only 2000 bytes because the result of
 * LPAD is limited to 4000 bytes, and the implicit hex to raw conversion
 * converts it to 2000 bytes of RAW data. */
INSERT INTO print_media(product_id, ad_id, ad_composite)
  VALUES (2004, 5, lpad('a', 5000, 'a'));
SELECT LENGTH(ad_composite) from print_media
  WHERE product_id=2004 AND ad_id=5;
ROLLBAACK;
```

Restrictions on Binds of More Than 4000 Bytes

The following lists the restrictions for binds of more than 4000 bytes:

- If a table has both `LONG` and LOB columns, then you can bind more than 4000 bytes of data to either the `LONG` or LOB columns, but not both in the same statement.
- In an `INSERT AS SELECT` operation, binding of any length data to LOB columns is not allowed.

Parallel DML (PDML) Support for LOBs

Oracle supports parallel execution of most of the following DML operations when performed on partitioned tables with SecureFiles LOBs or BasicFiles LOBs, and non-partitioned tables with SecureFiles LOBs only:

- `INSERT`
- `INSERT AS SELECT`
- `CREATE TABLE AS SELECT`
- `DELETE`
- `UPDATE`
- `MERGE` (conditional `UPDATE` and `INSERT`)
- Multi-table `INSERT`
- SQL Loader
- Import/Export

Starting with release 12*c*, enhanced support for parallel DML includes the following:

- LOB columns stored as SecureFiles LOBs in non-partitioned tables. (Previous releases already included partitioned tables)
- Direct load support for SecureFiles LOB columns that have context index defined on them.

Restrictions

- Parallel insert direct load (PIDL) is disabled if a table also has a BasicFiles LOB column, in addition to a SecureFiles LOB column.
- Some domain index implementations may limit load distribution and degrade performance due to their design.
- Parallelism must be specified only for top-level non-partitioned tables.

> **See Also:** *Oracle Database Administrator's Guide* section "Managing Processes for Parallel SQL Execution"

Example: PL/SQL - Using Binds of More Than 4000 Bytes in INSERT and UPDATE

```
DECLARE
  bigtext VARCHAR2(32767);
  smalltext VARCHAR2(2000);
  bigraw RAW (32767);
BEGIN
  bigtext := LPAD('a', 32767, 'a');
  smalltext := LPAD('a', 2000, 'a');
  bigraw := utl_raw.cast_to_raw (bigtext);
```

```
  /* Multiple long binds for LOB columns are allowed for INSERT: */
  INSERT INTO print_media(product_id, ad_id, ad_sourcetext, ad_composite)
    VALUES (2004, 1, bigtext, bigraw);

  /* Single long bind for LOB columns is allowed for INSERT: */
  INSERT INTO print_media (product_id, ad_id, ad_sourcetext)
    VALUES (2005, 2, smalltext);

  bigtext := LPAD('b', 32767, 'b');
  smalltext := LPAD('b', 20, 'a');
  bigraw := utl_raw.cast_to_raw (bigtext);

  /* Multiple long binds for LOB columns are allowed for UPDATE: */
  UPDATE print_media SET ad_sourcetext = bigtext, ad_composite = bigraw,
    ad_finaltext = smalltext;

  /* Single long bind for LOB columns is allowed for UPDATE: */
  UPDATE print_media SET ad_sourcetext = smalltext, ad_finaltext = bigtext;

  /* The following is NOT allowed because we are trying to insert more than
     4000 bytes of data in a LONG and a LOB column: */
  INSERT INTO print_media(product_id, ad_id, ad_sourcetext, press_release)
    VALUES (2030, 3, bigtext, bigtext);

  /* Insert of data into LOB attribute is allowed */
  INSERT INTO print_media(product_id, ad_id, ad_header)
     VALUES (2049, 4, adheader_typ(null, null, null, bigraw));

  /* The following is not allowed because we try to perform INSERT AS
     SELECT data INTO LOB */
  INSERT INTO print_media(product_id, ad_id, ad_sourcetext)
    SELECT 2056, 5, bigtext FROM dual;

END;
/
```

Using the Data Interface for LOBs with INSERT, UPDATE, and SELECT Operations

`INSERT` and `UPDATE` statements on LOBs are used in the same way as on `LONG`s. For example:

```
DECLARE
  ad_buffer VARCHAR2(100);
BEGIN
  INSERT INTO print_media(product_id, ad_id, ad_sourcetext)
    VALUES(2004, 5, 'Source for advertisement 1');
  UPDATE print_media SET ad_sourcetext= 'Source for advertisement 2'
    WHERE product_id=2004 AND ad_id=5;
  /* This retrieves the LOB column if it is up to 100 bytes, otherwise it
   * raises an exception */
  SELECT ad_sourcetext INTO ad_buffer FROM print_media
    WHERE product_id=2004 AND ad_id=5;
END;
/
```

Using the Data Interface for LOBs in Assignments and Parameter Passing

The data interface for LOBs enables implicit assignment and parameter passing as shown in the following example:

```
CREATE TABLE t (clob_col CLOB, blob_col BLOB);
INSERT INTO t VALUES('abcdefg', 'aaaaaa');

DECLARE
  var_buf VARCHAR2(100);
  clob_buf CLOB;
  raw_buf RAW(100);
  blob_buf BLOB;
BEGIN
  SELECT * INTO clob_buf, blob_buf FROM t;
  var_buf := clob_buf;
  clob_buf:= var_buf;
  raw_buf := blob_buf;
  blob_buf := raw_buf;
END;
/

CREATE OR REPLACE PROCEDURE FOO ( a IN OUT CLOB) IS
BEGIN
  -- Any procedure body
  a := 'abc';
END;
/

CREATE OR REPLACE PROCEDURE BAR (b IN OUT VARCHAR2) IS
BEGIN
  -- Any procedure body
  b := 'xyz';
END;
/

DECLARE
  a VARCHAR2(100) := '1234567';
  b CLOB;
BEGIN
  FOO(a);
  SELECT clob_col INTO b FROM t;
  BAR(b);
END;
/
```

Using the Data Interface for LOBs with PL/SQL Built-In Functions

This example illustrates the use of CLOBs in PL/SQL built-in functions, using the data interface for LOBs:

```
DECLARE
  my_ad CLOB;
  revised_ad CLOB;
  myGist VARCHAR2(100):= 'This is my gist.';
  revisedGist VARCHAR2(100);
BEGIN
  INSERT INTO print_media (product_id, ad_id, ad_sourcetext)
    VALUES (2004, 5, 'Source for advertisement 1');

  -- select a CLOB column into a CLOB variable
```

```
  SELECT ad_sourcetext INTO my_ad FROM print_media
    WHERE product_id=2004 AND ad_id=5;

  -- perform VARCHAR2 operations on a CLOB variable
  revised_ad := UPPER(SUBSTR(my_ad, 1, 20));

  -- revised_ad is a temporary LOB
  -- Concat a VARCHAR2 at the end of a CLOB
  revised_ad := revised_ad || myGist;

  -- The following statement raises an error if my_ad is
  -- longer than 100 bytes
  myGist := my_ad;
END;
/
```

Using the Data Interface for Persistent LOBs in OCI

This section discusses OCI functions included in the data interface for persistent LOBs. These OCI functions work for LOB datatypes exactly the same way as they do for `LONG` datatypes. Using these functions, you can perform `INSERT`, `UPDATE`, fetch, bind, and define operations in OCI on LOBs using the same techniques you would use on other datatypes that store character or binary data.

Note: You can use array bind and define interfaces to insert and select multiple rows with LOBs in one round trip.

See Also: *Oracle Call Interface Programmer's Guide*, section "Runtime Data Allocation and Piecewise Operations in OCI"

Binding LOB Datatypes in OCI

You can bind LOB datatypes in the following operations:

- Regular, piecewise, and callback binds for `INSERT` and `UPDATE` operations
- Array binds for `INSERT` and `UPDATE` operations
- Parameter passing across PL/SQL and OCI boundaries

Piecewise operations can be performed by polling or by providing a callback. To support these operations, the following OCI functions accept the `LONG` and LOB data types listed in Table 20–2.

- `OCIBindByName()` and `OCIBindByPos()`

  These functions create an association between a program variable and a placeholder in the SQL statement or a PL/SQL block for `INSERT` and `UPDATE` operations.

- `OCIBindDynamic()`

  You use this call to register callbacks for dynamic data allocation for `INSERT` and `UPDATE` operations

- `OCIStmtGetPieceInfo()` and `OCIStmtSetPieceInfo()`

  These calls are used to get or set piece information for piecewise operations.

Defining LOB Datatypes in OCI

The data interface for persistent LOBs allows the following OCI functions to accept the LONG and LOB data types listed in Table 20–2.

- `OCIDefineByPos()`

  This call associates an item in a `SELECT` list with the type and output data buffer.

- `OCIDefineDynamic()`

  This call registers user callbacks for `SELECT` operations if the `OCI_DYNAMIC_FETCH` mode was selected in `OCIDefineByPos()` function call.

When you use these functions with LOB types, the LOB data, and not the locator, is selected into your buffer. Note that in OCI, you cannot specify the amount you want to read using the data interface for LOBs. You can only specify the buffer length of your buffer. The database only reads whatever amount fits into your buffer and the data is truncated.

Using Multibyte Character Sets in OCI with the Data Interface for LOBs

When the client character set is in a multibyte format, functions included in the data interface operate the same way with LOB datatypes as they do for LONG datatypes as follows:

- For a *piecewise* fetch in a multibyte character set, a multibyte character could be cut in the middle, with some bytes at the end of one buffer and remaining bytes in the next buffer.
- For a *regular* fetch, if the buffer cannot hold all bytes of the last character, then Oracle returns as many bytes as fit into the buffer, hence returning partial characters.

Using OCI Functions to Perform INSERT or UPDATE on LOB Columns

This section discusses the various techniques you can use to perform `INSERT` or `UPDATE` operations on LOB columns or attributes using the data interface. The operations described in this section assume that you have initialized the OCI environment and allocated all necessary handles.

Simple INSERTs or UPDATEs in One Piece

To perform simple `INSERT` or `UPDATE` operations in one piece using the data interface for persistent LOBs, perform the following steps:

1. Call `OCIStmtPrepare()` to prepare the statement in `OCI_DEFAULT` mode.
2. Call `OCIBindByName()` or `OCIBindbyPos()` in `OCI_DEFAULT` mode to bind a placeholder for LOB as character data or binary data.
3. Call `OCIStmtExecute()` to do the actual `INSERT` or `UPDATE` operation.

Using Piecewise INSERTs and UPDATEs with Polling

To perform piecewise `INSERT` or `UPDATE` operations with polling using the data interface for persistent LOBs, do the following steps:

1. Call `OCIStmtPrepare()` to prepare the statement in `OCI_DEFAULT` mode.
2. Call `OCIBindByName()` or `OCIBindbyPos()` in `OCI_DATA_AT_EXEC` mode to bind a LOB as character data or binary data.

3. Call OCIStmtExecute() in default mode. Do each of the following in a loop while the value returned from OCIStmtExecute() is OCI_NEED_DATA. Terminate your loop when the value returned from OCIStmtExecute() is OCI_SUCCESS.
   - Call OCIStmtGetPieceInfo() to retrieve information about the piece to be inserted.
   - Call OCIStmtSetPieceInfo() to set information about piece to be inserted.

Piecewise INSERTs and UPDATEs with Callback

To perform piecewise INSERT or UPDATE operations with callback using the data interface for persistent LOBs, do the following steps:

1. Call OCIStmtPrepare() to prepare the statement in OCI_DEFAULT mode.
2. Call OCIBindByName() or OCIBindbyPos() in OCI_DATA_AT_EXEC mode to bind a placeholder for the LOB column as character data or binary data.
3. Call OCIBindDynamic() to specify the callback.
4. Call OCIStmtExecute() in default mode.

Array INSERT and UPDATE Operations

To perform array INSERT or UPDATE operations using the data interface for persistent LOBs, use any of the techniques discussed in this section in conjunction with OCIBindArrayOfStruct(), or by specifying the number of iterations (*iter*), with *iter* value greater than 1, in the OCIStmtExecute() call.

Using the Data Interface to Fetch LOB Data in OCI

This section discusses techniques you can use to fetch data from LOB columns or attributes in OCI using the data interface for persistent LOBs.

Simple Fetch in One Piece

To perform a simple fetch operation on LOBs in one piece using the data interface for persistent LOBs, do the following:

1. Call OCIStmtPrepare() to prepare the SELECT statement in OCI_DEFAULT mode.
2. Call OCIDefineByPos() to define a select list position in OCI_DEFAULT mode to define a LOB as character data or binary data.
3. Call OCIStmtExecute() to run the SELECT statement.
4. Call OCIStmtFetch() to do the actual fetch.

Piecewise Fetch with Polling

To perform a piecewise fetch operation on a LOB column with polling using the data interface for LOBs, do the following steps:

1. Call OCIStmtPrepare() to prepare the SELECT statement in OCI_DEFAULT mode.
2. Call OCIDefinebyPos() to define a select list position in OCI_DYNAMIC_FETCH mode to define the LOB column as character data or binary data.
3. Call OCIStmtExecute() to run the SELECT statement.
4. Call OCIStmtFetch() in default mode. Do each of the following in a loop while the value returned from OCIStmtFetch() is OCI_NEED_DATA. Terminate your loop when the value returned from OCIStmtFetch() is OCI_SUCCESS.

- Call OCIStmtGetPieceInfo() to retrieve information about the piece to be fetched.
- Call OCIStmtSetPieceInfo() to set information about piece to be fetched.

Piecewise with Callback

To perform a piecewise fetch operation on a LOB column with callback using the data interface for persistent LOBs, do the following:

1. Call OCIStmtPrepare() to prepare the statement in OCI_DEFAULT mode.
2. Call OCIDefinebyPos() to define a select list position in OCI_DYNAMIC_FETCH mode to define the LOB column as character data or binary data.
3. Call OCIStmtExecute() to run the SELECT statement.
4. Call OCIDefineDynamic() to specify the callback.
5. Call OCIStmtFetch() in default mode.

Array Fetch

To perform an array fetch in OCI using the data interface for persistent LOBs, use any of the techniques discussed in this section in conjunction with OCIDefineArrayOfStruct(), or by specifying the number of iterations (*iter*), with the value of *iter* greater than 1, in the OCIStmtExecute() call.

PL/SQL and C Binds from OCI

When you call a PL/SQL procedure from OCI, and have an IN or OUT or IN OUT bind, you should be able to:

- Bind a variable as SQLT_CHR or SQLT_LNG where the formal parameter of the PL/SQL procedure is SQLT_CLOB, or
- Bind a variable as SQLT_BIN or SQLT_LBI where the formal parameter is SQLT_BLOB

The following two cases work:

Calling PL/SQL Out-binds in the "begin foo(:1); end;" Manner

Here is an example of calling PL/SQL out-binds in the "begin foo(:1); end;" Manner:

```
text *sqlstmt = (text *)"BEGIN get_lob(:c); END; " ;
```

Calling PL/SQL Out-binds in the "call foo(:1);" Manner

Here is an example of calling PL/SQL out-binds in the "call foo(:1);" manner:

```
text *sqlstmt = (text *)"CALL get_lob(:c);" ;
```

In both these cases, the rest of the program has these statements:

```
OCIStmtPrepare(stmthp, errhp, sqlstmt, (ub4)strlen((char *)sqlstmt),
               (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
   curlen = 0;
OCIBindByName(stmthp, &bndhp[3], errhp,
        (text *) ":c", (sb4) strlen((char *) ":c"),
        (dvoid *) buf5, (sb4) LONGLEN, SQLT_CHR,
        (dvoid *) 0, (ub2 *) 0, (ub2 *) 0,
        (ub4) 1, (ub4 *) &curlen, (ub4) OCI_DATA_AT_EXEC);
```

The PL/SQL procedure, `get_lob()`, is as follows:

```
procedure get_lob(c INOUT CLOB) is  -- This might have been column%type
  BEGIN
  ... /* The procedure body could be in PL/SQL or C*/
  END;
```

Example: C (OCI) - Binds of More than 4000 Bytes for INSERT and UPDATE

```
void insert3()
{
/* Insert of data into LOB attributes is allowed. */
   ub1 buffer[8000];
   text *insert_sql = (text *)"INSERT INTO Print_media (ad_header) \
               VALUES (adheader_typ(NULL, NULL, NULL,:1))";
   OCIStmtPrepare(stmthp, errhp, insert_sql, strlen((char*)insert_sql),
            (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
   OCIBindByPos(stmthp, &bindhp[0], errhp, 1, (dvoid *)buffer, 2000,
             SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
   OCIStmtExecute(svchp, stmthp, errhp, 1, 0, (const OCISnapshot*) 0,
                  (OCISnapshot*)0, OCI_DEFAULT);
}
```

Using the Data Interface for LOBs in PL/SQL Binds from OCI on LOBs

The data interface for LOBs allows LOB PL/SQL binds from OCI to work as follows. When you call a PL/SQL procedure from OCI, and have an `IN` or `OUT` or `IN OUT` bind, you should be able to bind a variable as `SQLT_CHR`, where the formal parameter of the PL/SQL procedure is `SQLT_CLOB`.

> **Note:** C procedures are wrapped inside a PL/SQL stub, so the OCI application always calls the PL/SQL stub.

For the OCI calling program, the following are likely cases:

Calling PL/SQL Out-binds in the "begin foo(:1); end;" Manner

For example:

```
text *sqlstmt = (text *)"BEGIN PKG1.P5 (:c); END; " ;
```

Calling PL/SQL Out-binds in the "call foo(:1);" Manner

For example:

```
text *sqlstmt = (text *)"CALL PKG1.P5( :c );" ;
```

In both these cases, the rest of the program is as follows:

```
   OCIStmtPrepare(stmthp, errhp, sqlstmt, (ub4)strlen((char *)sqlstmt),
            (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
   curlen = 0;

   OCIBindByName(stmthp, &bndhp[3], errhp,
            (text *) ":c4", (sb4) strlen((char *) ":c"),
            (dvoid *) buf5, (sb4) LONGLEN, SQLT_CHR,
            (dvoid *) 0, (ub2 *) 0, (ub2 *) 0,
            (ub4) 1, (ub4 *) &curlen, (ub4) OCI_DATA_AT_EXEC);
```

```
    OCIStmtExecute(svchp, stmthp, errhp,(ub4) 0,(ub4) 0, (const OCISnapshot*) 0,
            (OCISnapshot*) 0,(ub4) OCI_DEFAULT);
```

The PL/SQL procedure `PKG1.P5` is as follows:

```
   CREATE OR REPLACE PACKAGE BODY pkg1 AS
     ...
   procedure p5 (c OUT CLOB) is
     -- This might have been table%rowtype (so it   is CLOB now)
   BEGIN
     ...
   END p5;

END pkg1;
```

Binding LONG Data for LOB Columns in Binds Greater Than 4000 Bytes

The following example illustrates binding character data for a LOB column:

```
void simple_insert()
{
  word buflen;
  text buf[5000];
  text *insstmt = (text *) "INSERT INTO Print_media(Product_id, Ad_id,\
                  Ad_sourcetext) VALUES (2004, 1, :SRCTXT)";

  OCIStmtPrepare(stmthp, errhp, insstmt, (ub4)strlen((char *)insstmt),
                (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);

  OCIBindByName(stmthp, &bndhp[0], errhp,
                (text *) ":SRCTXT", (sb4) strlen((char *) ":SRCTXT"),
                (dvoid *) buf, (sb4) sizeof(buf), SQLT_CHR,
                (dvoid *) 0, (ub2 *) 0, (ub2 *) 0,
                (ub4) 0, (ub4 *) 0, (ub4) OCI_DEFAULT);

  memset((void *)buf, (int)'A', (size_t)5000);
  OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                 (const OCISnapshot*) 0, (OCISnapshot*) 0, (ub4) OCI_DEFAULT);
}
```

Binding LONG Data to LOB Columns Using Piecewise INSERT with Polling

The following example illustrates using piecewise `INSERT` with polling using the data interface for LOBs.

```
void piecewise_insert()
{
  text *sqlstmt = (text *)"INSERT INTO Print_media(Product_id, Ad_id,\
                  Ad_sourcetext) VALUES (:1, :2, :3)";
  ub2 rcode;
  ub1 piece, i;
  word product_id = 2004;
  word ad_id = 2;
  ub4 buflen;
  char buf[5000];

  OCIStmtPrepare(stmthp, errhp, sqlstmt, (ub4)strlen((char *)sqlstmt),
                 (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
  OCIBindByPos(stmthp, &bndhp[0], errhp, (ub4) 1,
               (dvoid *) &product_id, (sb4) sizeof(product_id), SQLT_INT,
               (dvoid *) 0, (ub2 *)0, (ub2 *)0,
```

```
              (ub4) 0, (ub4 *) 0, (ub4) OCI_DEFAULT);
  OCIBindByPos(stmthp, &bndhp[1], errhp, (ub4) 2,
              (dvoid *) &ad_id, (sb4) sizeof(ad_id), SQLT_INT,
              (dvoid *) 0, (ub2 *)0, (ub2 *)0,
              (ub4) 0, (ub4 *) 0, (ub4) OCI_DEFAULT);
  OCIBindByPos(stmthp, &bndhp[2], errhp, (ub4) 3,
              (dvoid *) 0, (sb4) 15000, SQLT_LNG,
              (dvoid *) 0, (ub2 *)0, (ub2 *)0,
              (ub4) 0, (ub4 *) 0, (ub4) OCI_DATA_AT_EXEC);

  i = 0;
  while (1)
  {
    i++;
    retval = OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                            (CONST OCISnapshot*) 0, (OCISnapshot*) 0,
                            (ub4) OCI_DEFAULT);
    switch(retval)
    {
    case OCI_NEED_DATA:
      memset((void *)buf, (int)'A'+i, (size_t)5000);
      buflen = 5000;
      if (i == 1) piece = OCI_FIRST_PIECE;
      else if (i == 3) piece = OCI_LAST_PIECE;
      else piece = OCI_NEXT_PIECE;

      if (OCIStmtSetPieceInfo((dvoid *)bndhp[2],
                              (ub4)OCI_HTYPE_BIND, errhp, (dvoid *)buf,
                              &buflen, piece, (dvoid *) 0, &rcode))
        {
          printf("ERROR: OCIStmtSetPieceInfo: %d \n", retval);
          break;
        }

      break;
    case OCI_SUCCESS:
      break;
    default:
      printf( "oci exec returned %d \n", retval);
      report_error(errhp);
      retval = OCI_SUCCESS;
    } /* end switch */
    if (retval == OCI_SUCCESS)
      break;
  } /* end while(1) */
}
```

Binding LONG Data to LOB Columns Using Piecewise INSERT with Callback

The following example illustrates binding `LONG` data to LOB columns using a piecewise `INSERT` with callback:

```
void callback_insert()
{
  word buflen = 15000;
  word product_id = 2004;
  word ad_id = 3;
  text *sqlstmt = (text *) "INSERT INTO Print_media(Product_id, Ad_id,\
                  Ad_sourcetext) VALUES (:1, :2, :3)";
  word pos = 3;
```

```
  OCIStmtPrepare(stmthp, errhp, sqlstmt, (ub4)strlen((char *)sqlstmt),
                 (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT)

  OCIBindByPos(stmthp, &bndhp[0], errhp, (ub4) 1,
               (dvoid *) &product_id, (sb4) sizeof(product_id), SQLT_INT,
               (dvoid *) 0, (ub2 *)0, (ub2 *)0,
               (ub4) 0, (ub4 *) 0, (ub4) OCI_DEFAULT);
  OCIBindByPos(stmthp, &bndhp[1], errhp, (ub4) 2,
               (dvoid *) &ad_id, (sb4) sizeof(ad_id), SQLT_INT,
               (dvoid *) 0, (ub2 *)0, (ub2 *)0,
               (ub4) 0, (ub4 *) 0, (ub4) OCI_DEFAULT);
  OCIBindByPos(stmthp, &bndhp[2], errhp, (ub4) 3,
               (dvoid *) 0, (sb4) buflen, SQLT_CHR,
               (dvoid *) 0, (ub2 *)0, (ub2 *)0,
               (ub4) 0, (ub4 *) 0, (ub4) OCI_DATA_AT_EXEC);

  OCIBindDynamic(bndhp[2], errhp, (dvoid *) (dvoid *) &pos,
                 insert_cbk, (dvoid *) 0, (OCICallbackOutBind) 0);

  OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                 (const OCISnapshot*) 0, (OCISnapshot*) 0,
                 (ub4) OCI_DEFAULT);
} /* end insert_data() */

/* Inbind callback to specify input data. */
static sb4 insert_cbk(dvoid *ctxp, OCIBind *bindp, ub4 iter, ub4 index,
                      dvoid **bufpp, ub4 *alenpp, ub1 *piecep, dvoid **indpp)
{
  static int a = 0;
  word   j;
  ub4    inpos = *((ub4 *)ctxp);
  char   buf[5000];

  switch(inpos)
  {
  case 3:
    memset((void *)buf, (int) 'A'+a, (size_t) 5000);
    *bufpp = (dvoid *) buf;
    *alenpp = 5000 ;
    a++;
    break;
  default: printf("ERROR: invalid position number: %d\n", inpos);
  }

  *indpp = (dvoid *) 0;
  *piecep = OCI_ONE_PIECE;
  if (inpos == 3)
  {
    if (a<=1)
    {
      *piecep = OCI_FIRST_PIECE;
      printf("Insert callback: 1st piece\n");
    }
    else if (a<3)
    {
      *piecep = OCI_NEXT_PIECE;
      printf("Insert callback: %d'th piece\n", a);
    }
    else {
```

```
      *piecep = OCI_LAST_PIECE;
      printf("Insert callback: %d'th piece\n", a);
      a = 0;
    }
  }
  return OCI_CONTINUE;
}
```

Binding LONG Data to LOB Columns Using an Array INSERT

The following example illustrates binding character data for LOB columns using an array `INSERT` operation:

```
void array_insert()
{
  ub4 i;
  word buflen;
  word arrbuf1[5];
  word arrbuf2[5];
  text arrbuf3[5][5000];
  text *insstmt = (text *)"INSERT INTO Print_media(Product_id, Ad_id,\
                  Ad_sourcetext) VALUES (:PID, :AID, :SRCTXT)";

  OCIStmtPrepare(stmthp, errhp, insstmt,
                 (ub4)strlen((char *)insstmt), (ub4) OCI_NTV_SYNTAX,
                 (ub4) OCI_DEFAULT);

  OCIBindByName(stmthp, &bndhp[0], errhp,
                (text *) ":PID", (sb4) strlen((char *) ":PID"),
                (dvoid *) &arrbuf1[0], (sb4) sizeof(arrbuf1[0]), SQLT_INT,
                (dvoid *) 0, (ub2 *)0, (ub2 *) 0,
                (ub4) 0, (ub4 *) 0, (ub4) OCI_DEFAULT);

  OCIBindByName(stmthp, &bndhp[1], errhp,
                (text *) ":AID", (sb4) strlen((char *) ":AID"),
                (dvoid *) &arrbuf2[0], (sb4) sizeof(arrbuf2[0]), SQLT_INT,
                (dvoid *) 0, (ub2 *)0, (ub2 *) 0,
                (ub4) 0, (ub4 *) 0, (ub4) OCI_DEFAULT);

  OCIBindByName(stmthp, &bndhp[2], errhp,
                (text *) ":SRCTXT", (sb4) strlen((char *) ":SRCTXT"),
                (dvoid *) arrbuf3[0], (sb4) sizeof(arrbuf3[0]), SQLT_CHR,
                (dvoid *) 0, (ub2 *) 0, (ub2 *) 0,
                (ub4) 0, (ub4 *) 0, (ub4) OCI_DEFAULT);

  OCIBindArrayOfStruct(bndhp[0], errhp sizeof(arrbuf1[0]),
                       indsk, rlsk, rcsk);
  OCIBindArrayOfStruct(bndhp[1], errhp, sizeof(arrbuf2[0]),
                       indsk, rlsk, rcsk);
  OCIBindArrayOfStruct(bndhp[2], errhp, sizeof(arrbuf3[0]),
                       indsk, rlsk, rcsk);

  for (i=0; i<5; i++)
  {
    arrbuf1[i] = 2004;
    arrbuf2[i] = i+4;
    memset((void *)arrbuf3[i], (int)'A'+i, (size_t)5000);
  }
  OCIStmtExecute(svchp, stmthp, errhp, (ub4) 5, (ub4) 0,
                 (const OCISnapshot*) 0, (OCISnapshot*) 0,
```

```
                (ub4) OCI_DEFAULT);

}
```

Selecting a LOB Column into a LONG Buffer Using a Simple Fetch

The following example illustrates selecting a LOB column using a simple fetch:

```
void simple_fetch()
{
  word retval;
  text buf[15000];
  text *selstmt = (text *) "SELECT Ad_sourcetext FROM Print_media WHERE\
                 Product_id = 2004";

  OCIStmtPrepare(stmthp, errhp, selstmt, (ub4)strlen((char *)selstmt),
                (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);

  retval = OCIStmtExecute(svchp, stmthp, errhp, (ub4) 0, (ub4) 0,
                          (const OCISnapshot*) 0, (OCISnapshot*) 0,
                          (ub4) OCI_DEFAULT);
  while (retval == OCI_SUCCESS || retval == OCI_SUCCESS_WITH_INFO)
  {
    OCIDefineByPos(stmthp, &defhp, errhp, (ub4) 1, (dvoid *) buf,
                   (sb4) sizeof(buf), (ub2) SQLT_CHR, (dvoid *) 0,
                   (ub2 *) 0, (ub2 *) 0, (ub4) OCI_DEFAULT);
    retval = OCIStmtFetch(stmthp, errhp, (ub4) 1,
                          (ub4) OCI_FETCH_NEXT, (ub4) OCI_DEFAULT);
    if (retval == OCI_SUCCESS || retval == OCI_SUCCESS_WITH_INFO)
      printf("buf = %.*s\n", 15000, buf);
  }
}
```

Selecting a LOB Column into a LONG Buffer Using Piecewise Fetch with Polling

The following example illustrates selecting a LOB column into a `LONG` buffer using a piecewise fetch with polling:

```
void piecewise_fetch()
{
  text buf[15000];
  ub4 buflen=5000;
  word retval;
  text *selstmt = (text *) "SELECT Ad_sourcetext FROM Print_media
                 WHERE Product_id = 2004 AND Ad_id = 2";

  OCIStmtPrepare(stmthp, errhp, selstmt,
                (ub4) strlen((char *)selstmt),
                (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);

  OCIDefineByPos(stmthp, &dfnhp, errhp, (ub4) 1,
                (dvoid *) NULL, (sb4) 100000, SQLT_LNG,
                (dvoid *) 0, (ub2 *) 0,
                (ub2 *) 0, (ub4) OCI_DYNAMIC_FETCH);

  retval = OCIStmtExecute(svchp, stmthp, errhp, (ub4) 0, (ub4) 0,
                          (CONST OCISnapshot*) 0, (OCISnapshot*) 0,
                          (ub4) OCI_DEFAULT);

  retval = OCIStmtFetch(stmthp, errhp, (ub4) 1 ,
```

```
                        (ub2) OCI_FETCH_NEXT, (ub4) OCI_DEFAULT);

  while (retval != OCI_NO_DATA && retval != OCI_SUCCESS)
  {
    ub1 piece;
    ub4 iter;
    ub4 idx;

    genclr((void *)buf, 5000);
    switch(retval)
    {
    case OCI_NEED_DATA:
      OCIStmtGetPieceInfo(stmthp, errhp, &hdlptr, &hdltype,
                          &in_out, &iter, &idx, &piece);
      buflen = 5000;
      OCIStmtSetPieceInfo(hdlptr, hdltype, errhp,
                          (dvoid *) buf, &buflen, piece,
                          (CONST dvoid *) &indp1, (ub2 *) 0);
      retval = OCI_NEED_DATA;
      break;
    default:
      printf("ERROR: piece-wise fetching, %d\n", retval);
      return;
    } /* end switch */
    retval = OCIStmtFetch(stmthp, errhp, (ub4) 1 ,
                          (ub2) OCI_FETCH_NEXT, (ub4) OCI_DEFAULT);
    printf("Data : %.5000s\n", buf);
  } /* end while */
}
```

Selecting a LOB Column into a LONG Buffer Using Piecewise Fetch with Callback

The following example illustrates selecting a `LONG` column into a LOB buffer when using a piecewise fetch with callback:

```
char buf[5000];
void callback_fetch()
{
  word outpos = 1;
  text *sqlstmt = (text *) "SELECT Ad_sourcetext FROM Print_media WHERE
                  Product_id = 2004 AND Ad_id = 3";

  OCIStmtPrepare(stmthp, errhp, sqlstmt, (ub4)strlen((char *)sqlstmt),
                 (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
  OCIDefineByPos(stmthp, &dfnhp[0], errhp, (ub4) 1,
                 (dvoid *) 0, (sb4)3 * sizeof(buf), SQLT_CHR,
                 (dvoid *) 0, (ub2 *)0, (ub2 *)0,
                 (ub4) OCI_DYNAMIC_FETCH);

  OCIDefineDynamic(dfnhp[0], errhp, (dvoid *) &outpos,
                   (OCICallbackDefine) fetch_cbk);

  OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                 (const OCISnapshot*) 0, (OCISnapshot*) 0,
                 (ub4) OCI_DEFAULT);
  buf[ 4999 ] = '\0';
  printf("Select callback: Last piece: %s\n", buf);
}

/* ------------------------------------------------------------ */
```

```
/* Fetch callback to specify buffers. */
/* ------------------------------------------------------------- */
static sb4 fetch_cbk(dvoid *ctxp, OCIDefine *dfnhp, ub4 iter, dvoid **bufpp,
                     ub4 **alenpp, ub1 *piecep, dvoid **indpp, ub2 **rcpp)
{
  static int a = 0;
  ub4 outpos = *((ub4 *)ctxp);
  ub4 len = 5000;
  switch(outpos)
  {
  case 1:
    a ++;
    *bufpp = (dvoid *) buf;
    *alenpp = &len;
    break;
  default:
    *bufpp = (dvoid *) 0;
    *alenpp = (ub4 *) 0;
    printf("ERROR: invalid position number: %d\n", outpos);
  }
  *indpp = (dvoid *) 0;
  *rcpp = (ub2 *) 0;

  buf[len] = '\0';
  if (a<=1)
  {
    *piecep = OCI_FIRST_PIECE;
    printf("Select callback: 0th piece\n");
  }
  else if (a<3)
  {
    *piecep = OCI_NEXT_PIECE;
    printf("Select callback: %d'th piece: %s\n", a-1, buf);
  }
  else {
    *piecep = OCI_LAST_PIECE;
    printf("Select callback: %d'th piece: %s\n", a-1, buf);
    a = 0;
  }
  return OCI_CONTINUE;
}
```

Selecting a LOB Column into a LONG Buffer Using an Array Fetch

The following example illustrates selecting a LOB column into a `LONG` buffer using an array fetch:

```
void array_fetch()
{
  word i;
  text arrbuf[5][5000];
  text *selstmt = (text *) "SELECT Ad_sourcetext FROM Print_media WHERE
                   Product_id = 2004 AND Ad_id >=4";

  OCIStmtPrepare(stmthp, errhp, selstmt, (ub4)strlen((char *)selstmt),
                 (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);

  OCIStmtExecute(svchp, stmthp, errhp, (ub4) 0, (ub4) 0,
                 (const OCISnapshot*) 0, (OCISnapshot*) 0, (ub4) OCI_DEFAULT);
```

```
  OCIDefineByPos(stmthp, &defhp1, errhp, (ub4) 1,
                  (dvoid *) arrbuf[0], (sb4) sizeof(arrbuf[0]),
                  (ub2) SQLT_CHR, (dvoid *) 0,
                  (ub2 *) 0, (ub2 *) 0, (ub4) OCI_DEFAULT);

  OCIDefineArrayOfStruct(dfnhp1, errhp, sizeof(arrbuf[0]), indsk,
                         rlsk, rcsk);

  retval = OCIStmtFetch(stmthp, errhp, (ub4) 5,
                        (ub4) OCI_FETCH_NEXT, (ub4) OCI_DEFAULT);
  if (retval == OCI_SUCCESS || retval == OCI_SUCCESS_WITH_INFO)
  {
     printf("%.5000s\n", arrbuf[0]);
     printf("%.5000s\n", arrbuf[1]);
     printf("%.5000s\n", arrbuf[2]);
     printf("%.5000s\n", arrbuf[3]);
     printf("%.5000s\n", arrbuf[4]);
  }
}
```

Using the Data Interface for Persistent LOBs in Java

You can also read and write `CLOB` and `BLOB` data using the same streaming mechanism as for `LONG` and `LONG RAW` data. To read, use `defineColumnType(nn, Types.LONGVARCHAR)` or `defineColumnType(nn, Types.LONGVARBINARY)` on the column. This produces a direct stream on the data as if it is a `LONG` or `LONG RAW` column. For input in a `PreparedStatement`, you may use `setBinaryStream()`, `setCharacterStream()`, or `setAsciiStream()` for a parameter which is a `BLOB` or `CLOB`. These methods use the stream interface to create a LOB in the database from the data in the stream. Both of these techniques reduce database round trips and may result in improved performance in some cases. See the Javadoc on stream data for the significant restrictions which apply, at `http://www.oracle.com/technology/`.

Refer to the following in the *JDBC Developer's Guide and Reference*:

See Also:

- *Oracle Database JDBC Developer's Guide*, "Working with LOBs and BFILEs", section "Data Interface for LOBs"
- *Oracle Database JDBC Developer's Guide*, "JDBC Standards Support"

Using the Data Interface with Remote LOBs

The data interface for insert, update, and select of remote LOBs (access over a `dblink`) is supported after Oracle Database 10*g* Release 2. The examples in the following sections are for the `print_media` table created in two schemas: `dbs1` and `dbs2`. The `CLOB` column of that table used in the examples shown is `ad_finaltext`. The examples to be given for PL/SQL, OCI, and Java use binds and defines for this one column, but multiple columns can also be accessed. Here is the functionality supported and its limitations:

- You can define a `CLOB` as `CHAR` or `NCHAR` and an `NCLOB` as CHAR or NCHAR. `CLOB` and `NCLOB` can be defined as a `LONG`. A `BLOB` can be defined as a `RAW` or a `LONG RAW`.
- Array binds and defines are supported.

See Also: "Remote Data Interface Example in PL/SQL" on page 20-23 and the sections following it.

Non-Supported Syntax

- Queries involving more than one database are not supported:

```
SELECT t1.lobcol, a2.lobcol FROM t1, t2.lobcol@dbs2 a2 WHERE
LENGTH(t1.lobcol) = LENGTH(a2.lobcol);
```

  Neither is this query (in a PL/SQL block):

```
SELECT t1.lobcol INTO varchar_buf1 FROM t1@dbs1
UNION ALL
SELECT t2.lobcol INTO varchar_buf2 FROM t2@dbs2;
```

- Only binds and defines for data going into remote persistent LOB columns are supported, so that parameter passing in PL/SQL where CHAR data is bound or defined for remote LOBs is not allowed because this could produce a remote temporary LOB, which are not supported. These statements all produce errors:

```
SELECT foo() INTO varchar_buf FROM table1@dbs2; -- foo returns a LOB

SELECT foo()@dbs INTO char_val FROM DUAL; -- foo returns a LOB

SELECT XMLType().getclobval INTO varchar_buf FROM table1@dbs2;
```

- If the remote object is a view such as

```
CREATE VIEW v AS SELECT foo() a FROM ... ; -- foo returns a LOB
/* The local database then tries to get the CLOB data and returns an error */
SELECT a INTO varchar_buf FROM v@dbs2;
```

  This returns an error because it produces a remote temporary LOB, which is not supported.

- RETURNING INTO does not support implicit conversions between CHAR and CLOB.
- PL/SQL parameter passing is not allowed where the actual argument is a LOB type and the remote argument is a VARCHAR2, NVARCHAR2, CHAR, NCHAR, or RAW.

Remote Data Interface Example in PL/SQL

The data interface only supports data of size less than 32KB in PL/SQL. The following snippet shows a PL/SQL example:

```
CONNECT pm
declare
  my_ad varchar(6000) := lpad('b', 6000, 'b');
BEGIN
  INSERT INTO print_media@dbs2(product_id, ad_id, ad_finaltext)
       VALUES (10000, 10, my_ad);
  -- Reset the buffer value
  my_ad := 'a';
  SELECT ad_finaltext INTO my_ad FROM print_media@dbs2
       WHERE product_id = 10000;
END;
/
```

If ad_finaltext were a BLOB column instead of a CLOB, my_ad has to be of type RAW. If the LOB is greater than 32KB - 1 in size, then PL/SQL raises a truncation error and the contents of the buffer are undefined.

Remote Data Interface Example in OCI

The data interface only supports data of size less than 2 GBytes (the maximum value possible of a variable declared as sb4) for OCI. The following pseudocode can be enhanced to be a part of an OCI program:

```
...
text *sql = (text *)"insert into print_media@dbs2
                     (product_id, ad_id, ad_finaltext)
                     values (:1, :2, :3)";
OCIStmtPrepare(...);
OCIBindByPos(...); /* Bind data for positions 1 and 2
                      * which are independent of LOB */
OCIBindByPos(stmthp, &bndhp[2], errhp, (ub4) 3,
             (dvoid *) charbuf1, (sb4) len_charbuf1, SQLT_CHR,
             (dvoid *) 0, (ub2 *)0, (ub2 *)0, 0, 0, OCI_DEFAULT);
OCIStmtExecute(...);

...

text *sql = (text *)"select ad_finaltext from print_media@dbs2
                     where product_id = 10000";
OCIStmtPrepare(...);
OCIDefineByPos(stmthp, &dfnhp[2], errhp, (ub4) 1,
             (dvoid *) charbuf2, (sb4) len_charbuf2, SQLT_CHR,
             (dvoid *) 0, (ub2 *)0, (ub2 *)0, OCI_DEFAULT);
OCIStmtExecute(...);
...
```

If ad_finaltext were a BLOB instead of a CLOB, then you bind and define using type SQLT_BIN. If the LOB is greater than 2GB - 1 in size, then OCI raises a truncation error and the contents of the buffer are undefined.

Remote Data Interface Examples in JDBC

The following code snippets works with all three JDBC drivers (OCI, Thin, and kprb in the database):

Bind:

This is for the non-streaming mode:

```
...
String sql = "insert into print_media@dbs2 (product_id, ad_id, ad_final_text)" +
             " values (:1, :2, :3)";
    PreparedStatement pstmt = conn.prepareStatement(sql);
    pstmt.setInt( 1, 2 );
    pstmt.setInt( 2, 20);
    pstmt.setString( 3, "Java string" );
    int rows = pstmt.executeUpdate();
...
```

For the streaming mode, the same code as the preceding works, except that the setString() statement is replaced by one of the following:

```
pstmt.setCharacterStream( 3, new LabeledReader(), 1000000 );
pstmt.setAsciiStream( 3, new LabeledAsciiInputStream(), 1000000 );
```

Here, LabeledReader() and LabeledAsciiInputStream() produce character and ASCII streams respectively. If ad_finaltext were a BLOB column instead of a CLOB, then the preceding example works if the bind is of type RAW:

```
pstmt.setBytes( 3, <some byte[] array> );

pstmt.setBinaryStream( 3, new LabeledInputStream(), 1000000 );
```

Here, `LabeledInputStream()` produces a binary stream.

Define:

For non-streaming mode:

```
OracleStatement stmt = (OracleStatement)(conn.createStatement());
  stmt.defineColumnType( 1, Types.VARCHAR );
  ResultSet rst = stmt.executeQuery("select ad_finaltext from print_media@dbs2" );
  while( rst.next() )
     {
       String s = rst.getString( 1 );
       System.out.println( s );
     }
```

For streaming mode:

```
OracleStatement stmt = (OracleStatement)(conn.createStatement());
  stmt.defineColumnType( 1, Types.LONGVARCHAR );
  ResultSet rst = stmt.executeQuery("select ad_finaltext from print_media@dbs2" );
  while( rst.next() )
     {
       Reader reader = rst.getCharacterStream( 1 );
       while( reader.ready() )
       {
         System.out.print( (char)(reader.next()) );
       }
       System.out.println();
     }
```

If `ad_finaltext` were a `BLOB` column instead of a `CLOB`, then the preceding examples work if the define is of type `LONGVARBINARY`:

```
...
   OracleStatement stmt = (OracleStatement)conn.createStatement();

   stmt.defineColumnType( 1, Types.INTEGER );
   stmt.defineColumnType( 2, Types.LONGVARBINARY );

   ResultSet rset = stmt.executeQuery("SELECT ID, LOBCOL FROM LOBTAB@MYSELF");

   while(rset.next())
    {
     /* using getBytes() */
     /*
     byte[] b = rset.getBytes("LOBCOL");
     System.out.println("ID: " + rset.getInt("ID") + "  length: " + b.length);
     */

        /* using getBinaryStream() */
        InputStream byte_stream = rset.getBinaryStream("LOBCOL");
        byte [] b = new byte [100000];
        int b_len = byte_stream.read(b);
        System.out.println("ID: " + rset.getInt("ID") + "  length: " + b_len);

        byte_stream.close();
    }
...
```

See Also: *Oracle Database JDBC Developer's Guide*

21

LOB APIs for BFILE Operations

This chapter describes APIs for operations that use BFILEs. APIs covered in this chapter are listed in Table 21–1.

The following information is given for each operation described in this chapter:

- **Usage Notes** provide implementation guidelines such as information specific to a given programmatic environment or data type.
- **Syntax** refers you to the syntax reference documentation for each supported programmatic environment.
- **Examples** describe any setup tasks necessary to run the examples given. Demonstration files listed are available in subdirectories under `$ORACLE_HOME/rdbms/demo/lobs/` named `plsql`, `oci`, `vb`, and `java`. The driver program `lobdemo.sql` is in `/plsql` and the driver program `lobdemo.c` is in `/oci`.

> **Note:** LOB APIs do not support loading data into BFILEs. See "Using SQL*Loader to Load LOBs" on page 3-1 for details on techniques for loading data into BFILEs.

This chapter contains these topics:

Supported Environments for BFILE APIs

Table 21–1, " Environments Supported for BFILE APIs" indicates which programmatic environments are supported for the APIs discussed in this chapter. The first column describes the operation that the API performs. The remaining columns indicate with Yes or No whether the API is supported in PL/SQL, OCI, COBOL, Pro*C/C++, and JDBC.

Table 21–1 Environments Supported for BFILE APIs

Operation	PL/SQL	OCI	COBOL	Pro*C/C++	JDBC
Inserting a Row Containing a BFILE on page 21-24	Yes	Yes	Yes	Yes	Yes
Loading a LOB with BFILE Data on page 21-10	Yes	Yes	Yes	Yes	Yes
Opening a BFILE with FILEOPEN on page 21-12	Yes	Yes	No	No	Yes
Opening a BFILE with OPEN on page 21-11	Yes	Yes	Yes	Yes	Yes
Determining Whether a BFILE Is Open Using ISOPEN on page 21-13	Yes	Yes	Yes	Yes	Yes
Determining Whether a BFILE Is Open with FILEISOPEN on page 21-14	Yes	Yes	No	No	Yes
Displaying BFILE Data on page 21-14	Yes	Yes	Yes	Yes	Yes
Reading Data from a BFILE on page 21-15	Yes	Yes	Yes	Yes	Yes
Reading a Portion of BFILE Data Using SUBSTR on page 21-16	Yes	No	Yes	Yes	Yes
Comparing All or Parts of Two BFILES on page 21-17	Yes	No	Yes	Yes	Yes
Checking If a Pattern Exists in a BFILE Using INSTR on page 21-18	Yes	No	Yes	Yes	Yes
Determining Whether a BFILE Exists on page 21-18	Yes	Yes	Yes	Yes	Yes
Getting the Length of a BFILE on page 21-19	Yes	Yes	Yes	Yes	Yes
Assigning a BFILE Locator on page 21-19	Yes	Yes	Yes	Yes	Yes
Getting Directory Object Name and File Name of a BFILE on page 21-20	Yes	Yes	Yes	Yes	Yes

Table 21–1 (Cont.) Environments Supported for BFILE APIs

Operation	PL/SQL	OCI	COBOL	Pro*C/C++	JDBC
Updating a BFILE by Initializing a BFILE Locator on page 21-21	Yes	Yes	Yes	Yes	Yes
Closing a BFILE with FILECLOSE on page 21-21	Yes	Yes	No	No	Yes
Closing a BFILE with CLOSE on page 21-22	Yes	Yes	Yes	Yes	Yes
Closing All Open BFILEs with FILECLOSEALL on page 21-23	Yes	Yes	Yes	Yes	Yes

Accessing BFILEs

To access BFILEs use one of the following interfaces:

- OCI (Oracle Call Interface)
- PL/SQL (DBMS_LOB package)
- Precompilers, such as Pro*C/C++ and Pro*COBOL
- Java (JDBC)

See Also: Chapter 13, "Overview of Supplied LOB APIs" for information about supported environments for accessing BFILEs.

Directory Objects

The DIRECTORY object facilitates administering access and usage of BFILE data types. A DIRECTORY object specifies a *logical alias name* for a physical directory on the database server file system under which the file to be accessed is located. You can access a file in the server file system only if granted the required access privilege on DIRECTORY object. You can also use Oracle Enterprise Manager Cloud Control to manage DIRECTORY objects.

See Also:

- CREATE DIRECTORY in *Oracle Database SQL Language Reference*
- See *Oracle Database Administrator's Guide* for the description of Oracle Enterprise Manager Cloud Control

Initializing a BFILE Locator

The DIRECTORY object also provides the flexibility to manage the locations of the files, instead of forcing you to hard-code the absolute path names of physical files in your applications. A directory object name is used in conjunction with the BFILENAME function, in SQL and PL/SQL, or the OCILobFileSetName() in OCI, for initializing a BFILE locator.

Note: The database does not verify that the directory and path name you specify actually exist. You should take care to specify a valid directory in your operating system. If your operating system uses case-sensitive path names, then be sure you specify the directory in the correct format. There is no requirement to specify a terminating slash (for example, `/tmp/` is not necessary, simply use `/tmp`).

Directory specifications cannot contain ".." anywhere in the path (for example, `/abc/def/hij..`).

How to Associate Operating System Files with a BFILE

To associate an operating system file to a `BFILE`, first create a `DIRECTORY` object which is an alias for the full path name to the operating system file.

To associate existing operating system files with relevant database records of a particular table use Oracle SQL DML (Data Manipulation Language). For example:

- Use `INSERT` to initialize a `BFILE` column to point to an existing file in the server file system.
- Use `UPDATE` to change the reference target of the `BFILE`.
- Initialize a `BFILE` to `NULL` and then update it later to refer to an operating system file using the `BFILENAME` function.
- OCI users can also use `OCILobFileSetName()` to initialize a `BFILE` locator variable that is then used in the `VALUES` clause of an `INSERT` statement.

Directory Example

The following statements associate the files `Image1.gif` and `image2.gif` with records having `key_value` of 21 and 22 respectively. '`IMG`' is a `DIRECTORY` object that represents the physical directory under which `Image1.gif` and `image2.gif` are stored.

You may be required to set up data structures similar to the following for certain examples to work:

```
CREATE TABLE Lob_table (
   Key_value NUMBER NOT NULL,
   F_lob BFILE)
   INSERT INTO Lob_table VALUES
      (21,  BFILENAME('IMG', 'Image1.gif'));
   INSERT INTO Lob_table VALUES
      (22, BFILENAME('IMG', 'image2.gif'));
```

The following `UPDATE` statement changes the target file to `image3.gif` for the row with `key_value` of 22.

```
   UPDATE Lob_table SET f_lob = BFILENAME('IMG', 'image3.gif')
       WHERE Key_value = 22;
```

Note: The database does not expand environment variables specified in the DIRECTORY object or file name of a BFILE locator. For example, specifying:

```
BFILENAME('WORK_DIR', '$MY_FILE')
```

where MY_FILE, an environment variable defined in the operating system, is not valid.

BFILENAME and Initialization

BFILENAME is a built-in function that you use to initialize a BFILE column to point to an external file.

Once physical files are associated with records using SQL DML, subsequent read operations on the BFILE can be performed using PL/SQL DBMS_LOB package and OCI. However, these files are read-only when accessed through BFILEs, and so they cannot be updated or deleted through BFILEs.

As a consequence of the reference-based semantics for BFILEs, it is possible to have multiple BFILE columns in the same record or different records referring to the same file. For example, the following UPDATE statements set the BFILE column of the row with key_value = 21 in lob_table to point to the same file as the row with key_value = 22.

```
UPDATE lob_table
   SET f_lob = (SELECT f_lob FROM lob_table WHERE key_value = 22)
      WHERE key_value = 21;
```

Think of BFILENAME in terms of initialization — it can initialize the value for the following:

- BFILE column
- BFILE (automatic) variable declared inside a PL/SQL module

Characteristics of the BFILE Data Type

Using the BFILE data type has the following advantages:

- If your need for a particular BFILE is temporary and limited within the module on which you are working, then you can use the BFILE related APIs on the variable without ever having to associate this with a column in the database.
- Because you are not forced to create a BFILE column in a server side table, initialize this column value, and then retrieve this column value using a SELECT, you save a round-trip to the server.

For more information, refer to the example given for DBMS_LOB.LOADFROMFILE (see "Loading a LOB with BFILE Data" on page 21-10).

The OCI counterpart for BFILENAME is OCILobFileSetName(), which can be used in a similar fashion.

DIRECTORY Name Specification

You must have CREATE ANY DIRECTORY system privilege to create directories. Path names cannot contain two dots (".."). The naming convention for DIRECTORY objects is the same as that for tables and indexes. That is, normal identifiers are interpreted in

uppercase, but delimited identifiers are interpreted as is. For example, the following statement:

```
CREATE OR REPLACE DIRECTORY scott_dir AS '/usr/home/scott';
```

creates or redefines a DIRECTORY object whose name is 'SCOTT_DIR' (in uppercase). But if a delimited identifier is used for the DIRECTORY name, as shown in the following statement

```
CREATE DIRECTORY "Mary_Dir" AS '/usr/home/mary';
```

then the directory object name is 'Mary_Dir'. Use 'SCOTT_DIR' and 'Mary_Dir' when calling BFILENAME. For example:

```
BFILENAME('SCOTT_DIR', 'afile')
BFILENAME('Mary_Dir', 'afile')
```

On Windows Platforms

On Windows platforms the directory names are case-insensitive. Therefore the following two statements refer to the same directory:

```
CREATE DIRECTORY "big_cap_dir" AS "g:\data\source";

CREATE DIRECTORY "small_cap_dir" AS "G:\DATA\SOURCE";
```

BFILE Security

This section introduces the BFILE security model and associated SQL statements. The main SQL statements associated with BFILE security are:

- SQL DDL: CREATE and REPLACE or ALTER a DIRECTORY object
- SQL DML: GRANT and REVOKE the READ system and object privileges on DIRECTORY objects

Ownership and Privileges

The DIRECTORY object is a *system owned* object. For more information on system owned objects, see *Oracle Database SQL Language Reference*. Oracle Database supports two new system privileges, which are granted only to DBA:

- CREATE ANY DIRECTORY — for creating or altering the DIRECTORY object creation
- DROP ANY DIRECTORY — for deleting the DIRECTORY object

Read Permission on a DIRECTORY Object

READ permission on the DIRECTORY object enables you to read files located under that directory. The creator of the DIRECTORY object automatically earns the READ privilege.

If you have been granted the READ permission with GRANT option, then you may in turn grant this privilege to other users/roles and add them to your privilege domains.

Note: The READ permission is defined only on the DIRECTORY *object*, not on individual files. Hence there is no way to assign different privileges to files in the same directory.

The physical directory that it represents may or may not have the corresponding operating system privileges (*read* in this case) for the Oracle Server process.

It is the responsibility of the DBA to ensure the following:

- That the physical directory exists
- *Read* permission for the Oracle Server process is enabled on the file, the directory, and the path leading to it
- The directory remains available, and *read* permission remains enabled, for the entire duration of file access by database users

The privilege just implies that as far as the Oracle Server is concerned, you may read from files in the directory. These privileges are checked and enforced by the PL/SQL `DBMS_LOB` package and OCI APIs at the time of the actual file operations.

> **Caution: Because `CREATE ANY DIRECTORY` and `DROP ANY DIRECTORY` privileges potentially expose the server file system to all database users, the DBA should be prudent in granting these privileges to normal database users to prevent security breach.**

SQL DDL for BFILE Security

Refer to the *Oracle Database SQL Language Reference* for information about the following SQL DDL statements that create, replace, and drop `DIRECTORY` objects:

- `CREATE DIRECTORY`
- `DROP DIRECTORY`

SQL DML for BFILE Security

Refer to the *Oracle Database SQL Language Reference* for information about the following SQL DML statements that provide security for `BFILE`s:

- `GRANT` (system privilege)
- `GRANT` (object privilege)
- `REVOKE` (system privilege)
- `REVOKE` (object privilege)
- `AUDIT` (new statements)
- `AUDIT` (schema objects)

Catalog Views on Directories

Catalog views are provided for DIRECTORY objects to enable users to view object names and corresponding paths and privileges. Supported views are:

- `ALL_DIRECTORIES` (`OWNER`, `DIRECTORY_NAME`, `DIRECTORY_PATH`)

  This view describes all directories accessible to the user.

- `DBA_DIRECTORIES`(`OWNER`, `DIRECTORY_NAME`, `DIRECTORY_PATH`)

  This view describes all directories specified for the entire database.

Guidelines for DIRECTORY Usage

The main goal of the DIRECTORY feature is to enable a simple, flexible, non-intrusive, yet secure mechanism for the DBA to manage access to large files in the server file system. But to realize this goal, it is very important that the DBA follow these guidelines when using DIRECTORY objects:

- Do not map a DIRECTORY object to a data file directory. A DIRECTORY object should not be mapped to physical directories that contain Oracle data files, control files, log files, and other system files. Tampering with these files (accidental or otherwise) could corrupt the database or the server operating system.
- Only the DBA should have system privileges. The system privileges such as CREATE ANY DIRECTORY (granted to the DBA initially) should be used carefully and not granted to other users indiscriminately. In most cases, only the database administrator should have these privileges.
- Use caution when granting the DIRECTORY privilege. Privileges on DIRECTORY objects should be granted to different users carefully. The same holds for the use of the WITH GRANT OPTION clause when granting privileges to users.
- Do not drop or replace DIRECTORY objects when database is in operation. DIRECTORY objects should not be arbitrarily dropped or replaced when the database is in operation. If this were to happen, then operations *from all sessions* on all files associated with this DIRECTORY object fail. Further, if a DROP or REPLACE command is executed before these files could be successfully closed, then the references to these files are lost in the programs, and system resources associated with these files are not be released until the session(s) is shut down.

  The only recourse left to PL/SQL users, for example, is to either run a program block that calls DBMS_LOB.FILECLOSEALL and restart their file operations, or exit their sessions altogether. Hence, it is imperative that you use these commands with prudence, and preferably during maintenance downtimes.
- Use caution when revoking a user's privilege on DIRECTORY objects. Revoking a user's privilege on a DIRECTORY object using the REVOKE statement causes all subsequent operations on dependent files from the user's session to fail. Either you must re-acquire the privileges to close the file, or run a FILECLOSEALL in the session and restart the file operations.

In general, using DIRECTORY objects for managing file access is an extension of system administration work at the operating system level. With some planning, files can be logically organized into suitable directories that have READ privileges for the Oracle process.

DIRECTORY objects can be created with READ privileges that map to these physical directories, and specific database users granted access to these directories.

BFILEs in Shared Server (Multithreaded Server) Mode

The database does not support session migration for BFILE data types in shared server (multithreaded server) mode. This implies that operations on open BFILE instances can persist beyond the end of a call to a shared server.

In shared server sessions, BFILE operations are bound to one shared server, they cannot migrate from one server to another.

External LOB (BFILE) Locators

For BFILEs, the value is stored in a server-side operating system file; in other words, external to the database. The BFILE locator that refers to that file is stored in the row.

When Two Rows in a BFILE Table Refer to the Same File

If a BFILE locator variable that is used in a DBMS_LOB.FILEOPEN (for example L1) is assigned to another locator variable, (for example L2), then both L1 and L2 point to the same file. This means that two rows in a table with a BFILE column can refer to the same file or to two distinct files — a fact that the canny developer might turn to advantage, but which could well be a pitfall for the unwary.

BFILE Locator Variable

A BFILE locator variable operates like any other automatic variable. With respect to file operations, it operates like a *file descriptor* available as part of the standard input/output library of most conventional programming languages. This implies that once you define and initialize a BFILE locator, and open the file pointed to by this locator, all subsequent operations until the closure of this file must be done from within the same program block using this locator or local copies of this locator.

Guidelines for BFILEs

Note the following guidelines when working with BFILEs:

- Open and close a file from the same program block at same nesting level. The BFILE locator variable can be used, just as any scalar, as a parameter to other procedures, member methods, or external function callouts. However, it is recommended that you open and close a file from the same program block at the same nesting level.
- Set the BFILE value before flushing the object to the database. If an object contains a BFILE, then you must set the BFILE value before flushing the object to the database, thereby inserting a new row. In other words, you must call OCILobFileSetName() after OCIObjectNew() and before OCIObjectFlush().
- Indicate the DIRECTORY object name and file name before inserting or updating of a BFILE. It is an error to insert or update a BFILE without indicating a DIRECTORY object name and file name.

  This rule also applies to users using an OCI bind variable for a BFILE in an insert or update statement. The OCI bind variable must be initialized with a DIRECTORY object name and file name before issuing the insert or update statement.
- Initialize BFILE Before insert or update

  Note: OCISetAttr() does not allow the user to set a BFILE locator to NULL.

- Before using SQL to insert or update a row with a BFILE, you must initialize the BFILE to one of the following:
  - NULL (not possible if using an OCI bind variable)
  - A DIRECTORY object name and file name
- A path name cannot contain two dots ("..") anywhere in its specification. A file name cannot start with two dots.

Loading a LOB with BFILE Data

This section describes how to load a LOB with data from a BFILE.

> **See Also:** Table 21–1, " Environments Supported for BFILE APIs" on page 21-2, for a list of operations on BFILEs and APIs provided for each programmatic environment.
>
> *Oracle Database JDBC Developer's Guide* for details of working with BFILE functions in this chapter.

Preconditions

The following preconditions must exist before calling this procedure:

- The source BFILE instance must exist.
- The destination LOB instance must exist.

Usage Notes

> **Note:** The LOADBLOBFROMFILE and LOADCLOBFROMFILE procedures implement the functionality of this procedure and provide improved features for loading binary data and character data. The improved procedures are available in the PL/SQL environment only. When possible, using one of the improved procedures is recommended. See "Loading a BLOB with Data from a BFILE" on page 22-7 and "Loading a CLOB or NCLOB with Data from a BFILE" on page 22-8 for more information.

Character Set Conversion

In using OCI, or any of the programmatic environments that access OCI functionality, character set conversions are *implicitly* performed when translating from one character set to another.

BFILE to CLOB or NCLOB: Converting From Binary Data to a Character Set

When you use the DBMS_LOB.LOADFROMFILE procedure to populate a CLOB or NCLOB, you are populating the LOB with binary data from the BFILE. *No implicit translation* is performed from binary data to a character set. For this reason, you should use the LOADCLOBFROMFILE procedure when loading text (see Loading a CLOB or NCLOB with Data from a BFILE on page 22-8).

> **See Also:** *Oracle Database Globalization Support Guide* for character set conversion issues.

Amount Parameter

Note the following with respect to the amount parameter:

- DBMS_LOB.LOADFROMFILE

  If you want to load the entire BFILE, then pass the constant DBMS_LOB.LOBMAXSIZE. If you pass any other value, then it must be less than or equal to the size of the BFILE.

- OCILobLoadFromFile()

If you want to load the entire BFILE, then you can pass the constant UB4MAXVAL. If you pass any other value, then it must be less than or equal to the size of the BFILE.

- OCILobLoadFromFile2()

  If you want to load the entire BFILE, then you can pass the constant UB8MAXVAL. If you pass any other value, then it must be less than or equal to the size of the BFILE.

  See Also: Table 22–2, " Maximum LOB Size for Load from File Operations" on page 22-6 for details on the maximum value of the amount parameter.

Syntax

Use the following syntax references for each programmatic environment:

- PL/SQL (DBMS_LOB) *(Oracle Database PL/SQL Packages and Types Reference):* "DBMS_LOB" — LOADFROMFILE
- C (OCI): *Oracle Call Interface Programmer's Guide)*: Chapter 7, "LOB and File Operations", for usage notes and examples. Chapter 16, "LOB Functions" — OCILobLoadFromFile2().
- COBOL (Pro*COBOL) *Pro*COBOL Programmer's Guide* for information on LOBs, usage notes on LOB Statements, embedded SQL, and LOB LOAD precompiler directives.
- C/C++ (Pro*C/C++) (*Pro*C/C++ Programmer's Guide)*: "Large Objects (LOBs)", "LOB Statements" "Embedded SQL Statements and Directives"— LOB LOAD.
- Java (JDBC) *Oracle Database JDBC Developer's Guide)*: "Working With LOBs and BFILEs" — Working with BFILEs.

Examples

Examples are provided in the following programmatic environments:

- PL/SQL (DBMS_LOB): floaddat.sql
- OCI: floaddat.c
- Java (JDBC): No example.

Opening a BFILE with OPEN

This section describes how to open a BFILE using the OPEN function.

Note: You can also open a BFILE using the FILEOPEN function; however, using the OPEN function is recommended for new development. Using the FILEOPEN function is described in Opening a BFILE with FILEOPEN on page 21-12.

See Also: Table 21–1, " Environments Supported for BFILE APIs" on page 21-2, for a list of operations on BFILEs and APIs provided for each programmatic environment.

Syntax

Use the following syntax references for each programmatic environment:

- PL/SQL(DBMS_LOB) *(Oracle Database PL/SQL Packages and Types Reference): "DBMS_LOB"* — OPEN
- C (OCI) (*Oracle Call Interface Programmer's Guide*): Chapter 7, "LOB and File Operations", for usage notes. Chapter 16, section "LOB Functions" — `OCILobOpen()`, `OCILobClose()`.
- COBOL (Pro*COBOL) *Pro*COBOL Programmer's Guide* for information on LOBs, usage notes on LOB statements, and embedded SQL and precompiler directives — LOB OPEN.
- C/C++ (Pro*C/C++) (*Pro*C/C++ Programmer's Guide*): "Large Objects (LOBs)", "LOB Statements", "Embedded SQL Statements and Directives" — LOB OPEN.
- Java (JDBC) *(Oracle Database JDBC Developer's Guide):* "Working With LOBs and BFILEs" — Working with BFILEs.

Scenario

These examples open an image in operating system file `ADPHOTO_DIR`.

Examples

Examples are provided in the following programmatic environments:

- PL/SQL(DBMS_LOB): `fopen.sql`
- OCI: `fopen.c`
- Java (JDBC): `fopen.java`

Opening a BFILE with FILEOPEN

This section describes how to open a `BFILE` using the `FILEOPEN` function.

Note: The `FILEOPEN` function is not recommended for new application development. The `OPEN` function is recommended for new development. See "Opening a BFILE with OPEN" on page 21-11

See Also: Table 21–1, " Environments Supported for BFILE APIs" on page 21-2, for a list of operations on `BFILE`s and APIs provided for each programmatic environment.

Usage Notes for Opening a BFILE

While you can continue to use the older `FILEOPEN` form, Oracle *strongly recommends* that you switch to using `OPEN`, because this facilitates future extensibility.

Syntax

Use the following syntax references for each programmatic environment:

- PL/SQL (DBMS_LOB) *(Oracle Database PL/SQL Packages and Types Reference):* "DBMS_LOB" — FILEOPEN, FILECLOSE
- C (OCI) (*Oracle Call Interface Programmer's Guide*): Chapter 7, "LOB and File Operations, for usage notes. Chapter 16, section "LOB Functions" — `OCILobFileOpen()`, `OCILobFileClose()`, `OCILobFileSetName()`.

- COBOL (Pro*COBOL): A syntax reference is not applicable in this release.
- C/C++ (Pro*C/C++): A syntax reference is not applicable in this release.
- Java (JDBC) (*Oracle Database JDBC Developer's Guide)*: "Working With LOBs and BFILEs" — Working with BFILEs.

Scenario for Opening a BFILE

These examples open `keyboard_logo.jpg` in `DIRECTORY` object `MEDIA_DIR`.

Examples

Examples are provided in the following programmatic environments:

- PL/SQL (DBMS_LOB): `ffilopen.sql`
- OCI: `ffilopen.c`
- Java (JDBC): `ffilopen.java`

Determining Whether a BFILE Is Open Using ISOPEN

This section describes how to determine whether a `BFILE` is open using ISOPEN.

Note: This function (`ISOPEN`) is recommended for new application development. The older `FILEISOPEN` function, described in "Determining Whether a BFILE Is Open with FILEISOPEN" on page 21-14, is not recommended for new development.

See Also: Table 21–1, " Environments Supported for BFILE APIs" on page 21-2, for a list of operations on `BFILE`s and APIs provided for each programmatic environment.

Syntax

Use the following syntax references for each programmatic environment:

- PL/SQL (DBMS_LOB) *(Oracle Database PL/SQL Packages and Types Reference):* "DBMS_LOB" — ISOPEN
- C (OCI) (*Oracle Call Interface Programmer's Guide)*: Chapter 7, "LOB and File Operations" for usage notes. Chapter 16, section "LOB Functions" — `OCILobFileIsOpen()`.
- COBOL (Pro*COBOL) *Pro*COBOL Programmer's Guide* for information on LOBs, usage notes on LOB Statements, and embedded SQL and precompiler directives — LOB DESCRIBE ... ISOPEN.
- C/C++ (Pro*C/C++) (*Pro*C/C++ Programmer's Guide)*: "Large Objects (LOBs)", "LOB Statements", "Large Objects (LOBs)", "LOB Statements", "Embedded SQL Statements and Directives" — LOB DESCRIBE ... ISOPEN
- Java (JDBC) *(Oracle Database JDBC Developer's Guide):* "Working With LOBs and BFILEs" — Working with BFILEs.

Examples

Examples are provided in the following programmatic environments:

- PL/SQL (DBMS_LOB): `fisopen.sql`

- OCI: `fisopen.c`
- Java (JDBC): `fisopen.java`

Determining Whether a BFILE Is Open with FILEISOPEN

This section describes how to determine whether a `BFILE` is OPEN using the FILEISOPEN function.

Note: The `FILEISOPEN` function is not recommended for new application development. The `ISOPEN` function is recommended for new development. See Determining Whether a BFILE Is Open Using ISOPEN on page 21-13

See Also: Table 21–1, " Environments Supported for BFILE APIs" on page 21-2, for a list of operations on `BFILE`s and APIs provided for each programmatic environment.

Usage Notes

While you can continue to use the older `FILEISOPEN` form, Oracle *strongly recommends* that you switch to using `ISOPEN`, because this facilitates future extensibility.

Syntax

Use the following syntax references for each programmatic environment:

- PL/SQL(DBMS_LOB) *(Oracle Database PL/SQL Packages and Types Reference):* "DBMS_LOB" — FILEISOPEN
- C (OCI) (*Oracle Call Interface Programmer's Guide)*: Chapter 7, "LOB and File Operations" for usage notes. Chapter 16, section "LOB Functions" — `OCILobFileIsOpen()`.
- COBOL (Pro*COBOL): A syntax reference is not applicable in this release.
- C/C++ (Pro*C/C++): A syntax reference is not applicable in this release.
- Java (JDBC) (*Oracle Database JDBC Developer's Guide)*: "Working With LOBs and BFILEs" — Working with BFILEs.

Scenario

These examples query whether a `BFILE` associated with `ad_graphic` is open.

Examples

Examples are provided in the following programmatic environments:

- PL/SQL(DBMS_LOB): `ffisopen.sql`
- OCI: `ffisopen.c`
- Java (JDBC): `ffisopen.java`

Displaying BFILE Data

This section describes how to display `BFILE` data.

> **See Also:** Table 21–1, " Environments Supported for BFILE APIs" on page 21-2, for a list of operations on BFILEs and APIs provided for each programmatic environment.

Syntax

Use the following syntax references for each programmatic environment:

- PL/SQL (DBMS_LOB) *(Oracle Database PL/SQL Packages and Types Reference):* "DBMS_LOB" — READ. Chapter 29, "DBMS_OUTPUT" - PUT_LINE
- C (OCI) (*Oracle Call Interface Programmer's Guide)*: Chapter 7, "LOB and File Operations" for usage notes. Chapter 16, section "LOB Functions" — `OCILobFileOpen()`, `OCILobRead2()`.
- COBOL (Pro*COBOL) *Pro*COBOL Programmer's Guide* for information on LOBs, usage notes on LOB Statements, and embedded SQL and precompiler directives — LOB READ, DISPLAY.
- C/C++ (Pro*C/C++) (*Pro*C/C++ Programmer's Guide)*: "Large Objects (LOBs)", "LOB Statements" — READ
- Java (JDBC) (*Oracle Database JDBC Developer's Guide)*: Chapter 7, "Working With LOBs and BFILEs" — Working with BFILEs.

Examples

Examples are provided in these programmatic environments:

- PL/SQL (DBMS_LOB): `fdisplay.sql`
- OCI: `fdisplay.c`
- Java (JDBC): `fdisplay.java`

Reading Data from a BFILE

This section describes how to read data from a BFILE.

> **See Also:** Table 21–1, " Environments Supported for BFILE APIs" on page 21-2, for a list of operations on BFILEs and APIs provided for each programmatic environment.

Usage Notes

Note the following when using this operation.

Streaming Read in OCI

The most efficient way to read large amounts of BFILE data is by `OCILobRead2()` with the streaming mechanism enabled, and using polling or callback. To do so, specify the starting point of the read using the `offset` parameter as follows:

```
ub8  char_amt =  0;
ub8  byte_amt =  0;
ub4  offset = 1000;

OCILobRead2(svchp, errhp, locp, &byte_amt, &char_amt, offset, bufp, bufl,
            OCI_ONE_PIECE, 0, 0, 0, 0);
```

When using *polling mode*, be sure to look at the value of the `byte_amt` parameter after each `OCILobRead2()` call to see how many bytes were read into the buffer, because the buffer may not be entirely full.

When using *callbacks*, the `lenp` parameter, which is input to the callback, indicates how many bytes are filled in the buffer. Be sure to check the `lenp` parameter during your callback processing because the entire buffer may not be filled with data (see the *Oracle Call Interface Programmer's Guide*.)

Amount Parameter

- When calling `DBMS_LOB.READ`, the amount parameter can be larger than the size of the data; however, the amount parameter should be less than or equal to the size of the buffer. In PL/SQL, the buffer size is limited to 32K.
- When calling `OCILobRead2()`, you can pass a value of 0 (zero) for the `byte_amt` parameter to read to the end of the `BFILE`.

Syntax

Use the following syntax references for each programmatic environment:

- PL/SQL (DBMS_LOB) *(Oracle Database PL/SQL Packages and Types Reference):* "DBMS_LOB" — READ
- C (OCI) (*Oracle Call Interface Programmer's Guide)*: Chapter 7, "LOB and File Operations" for usage notes. Chapter 16, section "LOB Functions" — `OCILobRead2()`.
- COBOL (Pro*COBOL) *Pro*COBOL Programmer's Guide* for information on LOBs, usage notes on LOB Statements, and embedded SQL and precompiler directives — LOB READ.
- C/C++ (Pro*C/C++) (*Pro*C/C++ Programmer's Guide)*: "Large Objects (LOBs)", "LOB Statements", "Embedded SQL Statements and Directives" — LOB READ
- Java (JDBC) *(Oracle Database JDBC Developer's Guide):* Chapter 7, "Working With LOBs and BFILEs" — Working with BFILEs.

Examples

Examples are provided in the following programmatic environments:

- PL/SQL (DBMS_LOB): `fread.sql`
- OCI: `fread.c`
- Java (JDBC): `fread.java`

Reading a Portion of BFILE Data Using SUBSTR

This section describes how to read portion of `BFILE` data using `SUBSTR`.

> **See Also:** Table 21–1, " Environments Supported for BFILE APIs" on page 21-2, for a list of operations on `BFILE`s and APIs provided for each programmatic environment.

Syntax

Use the following syntax references for each programmatic environment:

- PL/SQL (DBMS_LOB) *(Oracle Database PL/SQL Packages and Types Reference):* "DBMS_LOB" — SUBSTR

- OCI: A syntax reference is not applicable in this release.
- COBOL (Pro*COBOL) *Pro*COBOL Programmer's Guide* for information on LOBs, usage notes on LOB Statements, and embedded SQL and precompiler directives — LOB OPEN, LOB CLOSE. See PL/SQL DBMS_LOB.SUBSTR.
- C/C++ (Pro*C/C++) (*Pro*C/C++ Programmer's Guide)*: "Large Objects (LOBs)", "LOB Statements", "Embedded SQL Statements and Directives" — LOB OPEN. See also PL/SQL DBMS_LOB.SUBSTR
- Java (JDBC) *(Oracle Database JDBC Developer's Guide)*: Chapter 7, "Working With LOBs and BFILEs" — Working with BFILEs.

Examples

Examples are provided in these programmatic environments:

- PL/SQL (DBMS_LOB): `freadprt.sql`
- C (OCI): No example is provided with this release.
- Java (JDBC): `freadprt.java`

Comparing All or Parts of Two BFILES

This section describes how to compare all or parts of two `BFILE`s.

> **See Also:** Table 21–1, " Environments Supported for BFILE APIs" on page 21-2, for a list of operations on `BFILE`s and APIs provided for each programmatic environment.

Syntax

Use the following syntax references for each programmatic environment:

- PL/SQL(DBMS_LOB) *(Oracle Database PL/SQL Packages and Types Reference):* "DBMS_LOB" — COMPARE
- C (OCI): A syntax reference is not applicable in this release.
- COBOL (Pro*COBOL) *Pro*COBOL Programmer's Guide* for information on LOBs, usage notes on LOB Statements, and embedded SQL and precompiler directives — LOB OPEN. See PL/SQL DBMS_LOB.COMPARE.
- C/C++ (Pro*C/C++) (*Pro*C/C++ Programmer's Guide)*: "Large Objects (LOBs)", "LOB Statements", "Embedded SQL Statements and Directives" — LOB OPEN. See PL/SQL DBMS_LOB.COMPARE.
- Java (JDBC) *(Oracle Database JDBC Developer's Guide):* "Working With LOBs and BFILEs" — Working with BFILEs.

Examples

Examples are provided in these programmatic environments:

- PL/SQL(DBMS_LOB): `fcompare.sql`
- OCI: No example is provided with this release.
- Java (JDBC): `fcompare.java`

Checking If a Pattern Exists in a BFILE Using INSTR

This section describes how to determine whether a pattern exists in a `BFILE` using `INSTR`.

> **See Also:** Table 21–1, " Environments Supported for BFILE APIs" on page 21-2, for a list of operations on `BFILE`s and APIs provided for each programmatic environment.

Syntax

Use the following syntax references for each programmatic environment:

- PL/SQL (DBMS_LOB) *(Oracle Database PL/SQL Packages and Types Reference):* "DBMS_LOB" — INSTR
- C (OCI): A syntax reference is not applicable in this release.
- COBOL (Pro*COBOL) *Pro*COBOL Programmer's Guide* for information on LOBs, usage notes on LOB Statements, and embedded SQL and precompiler directives — LOB OPEN. See PL/SQL DBMS_LOB.INSTR.
- C/C++ (Pro*C/C++) (*Pro*C/C++ Programmer's Guide)*: "Large Objects (LOBs)", "LOB Statements", "Embedded SQL Statements and Directives" — LOB OPEN. See PL/SQL DBMS_LOB.INSTR.
- Java (JDBC) *(Oracle Database JDBC Developer's Guide):*"Working With LOBs and BFILEs" — Working with BFILEs.

Examples

These examples are provided in the following programmatic environments:

- PL/SQL (DBMS_LOB): `fpattern.sql`
- OCI: No example is provided with this release.
- Java (JDBC): `fpattern.java`

Determining Whether a BFILE Exists

This procedure determines whether a `BFILE` locator points to a valid `BFILE` instance.

> **See Also:** Table 21–1, " Environments Supported for BFILE APIs" on page 21-2, for a list of operations on `BFILE`s and APIs provided for each programmatic environment.

Syntax

Use the following syntax references for each programmatic environment:

- PL/SQL (DBMS_LOB) *Oracle Database PL/SQL Packages and Types Reference):* "DBMS_LOB" — FILEEXISTS
- C (OCI) *Oracle Call Interface Programmer's Guide)*: Chapter 7, "LOB and File Operations" for usage notes. Chapter 16, section "LOB Functions" — `OCILobFileExists()`.
- COBOL (Pro*COBOL) *Pro*COBOL Programmer's Guide* for information on LOBs, usage notes on LOB Statements, and embedded SQL and precompiler directives — LOB DESCRIBE ... FILEEXISTS.

- C/C++ (Pro*C/C++) *Pro*C/C++ Programmer's Guide)*: "Large Objects (LOBs)", "LOB Statements", "Embedded SQL Statements and Directives" — LOB DESCRIBE ... GET FILEEXISTS
- Java (JDBC) *Oracle Database JDBC Developer's Guide:* "Working With LOBs and BFILEs" — Working with BFILEs.

Examples

The examples are provided in the following programmatic environments:

- PL/SQL (DBMS_LOB): `fexists.sql`
- OCI: `fexists.c`
- Java (JDBC): `fexists.java`

Getting the Length of a BFILE

This section describes how to get the length of a `BFILE`.

> **See Also:** Table 21–1, " Environments Supported for BFILE APIs" on page 21-2, for a list of operations on `BFILE`s and APIs provided for each programmatic environment.

Syntax

Use the following syntax references for each programmatic environment:

- PL/SQL (DBMS_LOB) *(Oracle Database PL/SQL Packages and Types Reference):* "DBMS_LOB" — GETLENGTH
- C (OCI) (*Oracle Call Interface Programmer's Guide)*: Chapter 7, "LOB and File Operations". Chapter 16, section "LOB Functions" — `OCILobGetLength2()`.
- COBOL (Pro*COBOL) *Pro*COBOL Programmer's Guide* for information on LOBs, usage notes on LOB Statements, and embedded SQL and precompiler directives — LOB DESCRIBE ... GET LENGTH INTO ...
- C/C++ (Pro*C/C++) (*Pro*C/C++ Programmer's Guide)*: "Large Objects (LOBs)", "LOB Statements", "Embedded SQL Statements and Directives" — LOB DESCRIBE ... GET LENGTH INTO ...
- Java (JDBC) *Oracle Database JDBC Developer's Guide*: "Working With LOBs and BFILEs" — Working with BFILEs.

Examples

The examples are provided in these programmatic environments:

- PL/SQL (DBMS_LOB): `flength.sql`
- OCI: `flength.c`
- Java (JDBC): `flength.java`

Assigning a BFILE Locator

This section describes how to assign one `BFILE` locator to another.

> **See Also:** Table 21–1, " Environments Supported for BFILE APIs" on page 21-2, for a list of operations on BFILEs and APIs provided for each programmatic environment.

Syntax

Use the following syntax references for each programmatic environment:

- SQL *(Oracle Database SQL Language Reference):* Chapter 7, "SQL Statements" — CREATE PROCEDURE
- PL/SQL (DBMS_LOB): Refer to Chapter 12, "Advanced Design Considerations" of this manual for information on assigning one lob locator to another.
- C (OCI) (*Oracle Call Interface Programmer's Guide)*: Chapter 7, "LOB and File Operations" for usage notes. Chapter 16, section "LOB Functions" — `OCILobLocatorAssign()`.
- COBOL (Pro*COBOL) *Pro*COBOL Programmer's Guide* for information on LOBs, usage notes on LOB Statements, and embedded SQL and precompiler directives — LOB ASSIGN
- C/C++ (Pro*C/C++) (*Pro*C/C++ Programmer's Guide)*: "Large Objects (LOBs)", "LOB Statements", "Embedded SQL Statements and Directives" — LOB ASSIGN
- Java (JDBC*) Oracle Database JDBC Developer's Guide:* "Working With LOBs and BFILEs" — Working with BFILEs.

Examples

The examples are provided in the following programmatic environments:

- PL/SQL (DBMS_LOB): `fcopyloc.sql`
- OCI: `fcopyloc.c`
- Java (JDBC): `fcopyloc.java`

Getting Directory Object Name and File Name of a BFILE

This section describes how to get the `DIRECTORY` object name and file name of a `BFILE`.

> **See Also:** Table 21–1, " Environments Supported for BFILE APIs" on page 21-2, for a list of operations on BFILEs and APIs provided for each programmatic environment.

Syntax

Use the following syntax references for each programmatic environment:

- PL/SQL (DBMS_LOB) *(Oracle Database PL/SQL Packages and Types Reference):* "DBMS_LOB" — FILEGETNAME
- C (OCI) (*Oracle Call Interface Programmer's Guide)*: Chapter 7, "LOB and File Operations" for usage notes. Chapter 16, section "LOB Functions" — `OCILobFileGetName()`.
- COBOL (Pro*COBOL) *Pro*COBOL Programmer's Guide* for information on LOBs, usage notes on LOB Statements, and embedded SQL and precompiler directives — LOB DESCRIBE ... GET DIRECTORY ...

- C/C++ (Pro*C/C++) (*Pro*C/C++ Programmer's Guide)*: "Large Objects (LOBs)", "LOB Statements", "Embedded SQL Statements and Directives" — LOB DESCRIBE ... GET DIRECTORY ...
- Java (JDBC) *Oracle Database JDBC Developer's Guide:* "Working With LOBs and BFILEs" — Working with BFILEs.

Examples

Examples of this procedure are provided in the following programmatic environments:

- PL/SQL (DBMS_LOB): `fgetdir.sql`
- OCI: `fgetdir.c`
- Java (JDBC): `fgetdir.java`

Updating a BFILE by Initializing a BFILE Locator

This section describes how to update a `BFILE` by initializing a `BFILE` locator.

> **See Also:** Table 21–1, " Environments Supported for BFILE APIs" on page 21-2, for a list of operations on `BFILE`s and APIs provided for each programmatic environment.

Syntax

Use the following syntax references for each programmatic environment:

- PL/SQL (DBMS_LOB): See the *(Oracle Database SQL Language Reference),* Chapter 7, "SQL Statements" — UPDATE
- C (OCI) (*Oracle Call Interface Programmer's Guide)*: Chapter 7, "LOB and File Operations" for usage notes. Chapter 16, section "LOB Functions" — `OCILobFileSetName()`.
- COBOL (Pro*COBOL) *Pro*COBOL Programmer's Guide* for information on LOBs, usage notes on LOB Statements, and embedded SQL and precompiler directives — ALLOCATE. See also *(Oracle Database SQL Language Reference), Chapter 7, "SQL Statements"* — UPDATE
- C/C++ (Pro*C/C++) (*Pro*C/C++ Programmer's Guide)*: "Large Objects (LOBs)", "LOB Statements", "Embedded SQL Statements and Directives". See also *(Oracle Database SQL Language Reference), Chapter 7, "SQL Statements"* — UPDATE
- Java (JDBC) *Oracle Database JDBC Developer's Guide:* "Working With LOBs and BFILEs" — Working with BFILEs.

Examples

- PL/SQL (DBMS_LOB): `fupdate.sql`
- OCI: `fupdate.c`
- Java (JDBC): `fupdate.java`

Closing a BFILE with FILECLOSE

This section describes how to close a `BFILE` with `FILECLOSE`.

Note: This function (FILECLOSE) is not recommended for new development. For new development, use the CLOSE function instead. See "Closing a BFILE with CLOSE" on page 21-22 for more information.

See Also: Table 21–1, " Environments Supported for BFILE APIs" on page 21-2, for a list of operations on BFILEs and APIs provided for each programmatic environment.

Syntax

Use the following syntax references for each programmatic environment:

- PL/SQL (DBMS_LOB)*(Oracle Database PL/SQL Packages and Types Reference):* "DBMS_LOB" — FILEOPEN, FILECLOSE
- C (OCI) (*Oracle Call Interface Programmer's Guide)*: Chapter 7, "LOB and File Operations" for usage notes. Chapter 16, section "LOB Functions" — OCILobFileClose().
- COBOL (Pro*COBOL): A syntax reference is not applicable in this release.
- C/C++ (Pro*C/C++): A syntax reference is not applicable in this release.
- Java (JDBC*) Oracle Database JDBC Developer's Guide:* "Working With LOBs and BFILEs" — Working with BFILEs.

Examples

- PL/SQL (DBMS_LOB): fclose_f.sql
- OCI: fclose_f.c
- Java (JDBC): fclose_f.java

Closing a BFILE with CLOSE

This section describes how to close a BFILE with the CLOSE function.

Note: This function (CLOSE) is recommended for new application development. The older FILECLOSE function, is not recommended for new development.

See Also: Table 21–1, " Environments Supported for BFILE APIs" on page 21-2, for a list of operations on BFILEs and APIs provided for each programmatic environment.

Usage Notes

Opening and closing a BFILE is mandatory. You must close the instance later in the session.

See Also:

- Opening a BFILE with OPEN on page 21-11
- Determining Whether a BFILE Is Open Using ISOPEN on page 21-13

Syntax

Use the following syntax references for each programmatic environment:

- PL/SQL (DBMS_LOB) *(Oracle Database PL/SQL Packages and Types Reference):* "DBMS_LOB" — CLOSE
- C (OCI) (*Oracle Call Interface Programmer's Guide*): Chapter 7, "LOB and File Operations" for usage notes. Chapter 16, section "LOB Functions" — `OCILobClose()`.
- COBOL (Pro*COBOL) *Pro*COBOL Programmer's Guide* for information on LOBs, usage notes on LOB Statements, and embedded SQL and precompiler directives — LOB CLOSE
- C/C++ (Pro*C/C++) (*Pro*C/C++ Programmer's Guide*): "Large Objects (LOBs)", "LOB Statements", "Embedded SQL Statements and Directives" — LOB CLOSE
- Java (JDBC) *Oracle Database JDBC Developer's Guide:* "Working With LOBs and BFILEs" — Working with BFILEs.

Examples

- PL/SQL (DBMS_LOB): `fclose_c.sql`
- OCI: `fclose_c.c`
- Java (JDBC): `fclose_c.java`

Closing All Open BFILEs with FILECLOSEALL

This section describes how to close all open `BFILE`s.

You are responsible for closing any `BFILE` instances before your program terminates. For example, you must close any open `BFILE` instance before the termination of a PL/SQL block or OCI program.

You must close open BFILE instances even in cases where an exception or unexpected termination of your application occurs. In these cases, if a `BFILE` instance is not closed, then it is still considered open by the database. Ensure that your exception handling strategy does not allow BFILE instances to remain open in these situations.

See Also:

- Table 21–1, " Environments Supported for BFILE APIs" on page 21-2, for a list of operations on `BFILE`s and APIs provided for each programmatic environment.
- "Setting Maximum Number of Open BFILEs" on page 3-5

Syntax

Use the following syntax references for each programmatic environment:

- PL/SQL (DBMS_LOB) *(Oracle Database PL/SQL Packages and Types Reference):* "DBMS_LOB" — FILECLOSEALL
- C (OCI) (*Oracle Call Interface Programmer's Guide*): Chapter 7, "LOB and File Operations" for usage notes. Chapter 16, section "LOB Functions" — `OCILobFileCloseAll()`.
- COBOL (Pro*COBOL) *Pro*COBOL Programmer's Guide* for information on LOBs, usage notes on LOB Statements, and embedded SQL and precompiler directives — LOB FILE CLOSE ALL

- C/C++ (Pro*C/C++) (*Pro*C/C++ Programmer's Guide)*: "Large Objects (LOBs)", "LOB Statements", "Embedded SQL Statements and Directives" — LOB FILE CLOSE ALL
- Java (JDBC) *Oracle Database JDBC Developer's Guide:* Chapter 7, "Working With LOBs and BFILEs" — Working with BFILEs.

Examples

- PL/SQL (DBMS_LOB): `fclosea.sql`
- OCI: `fclosea.c`
- Java (JDBC): `fclosea.java`

Inserting a Row Containing a BFILE

This section describes how to insert a row containing a `BFILE` by initializing a `BFILE` locator.

See Also:

- Table 21–1, " Environments Supported for BFILE APIs" on page 21-2, for a list of operations on `BFILE`s and APIs provided for each programmatic environment.

Usage Notes

You must initialize the `BFILE` locator bind variable to `NULL` or a `DIRECTORY` object and file name before issuing the `INSERT` statement.

Syntax

See the following syntax references for each programmatic environment:

- SQL(*Oracle Database SQL Language Reference*, Chapter 7 "SQL Statements" — INSERT
- C (OCI) *Oracle Call Interface Programmer's Guide*: Chapter 7, "LOB and File Operations".
- COBOL (Pro*COBOL) *Pro*COBOL Programmer's Guide* for information on LOBs, usage notes on LOB Statements, embedded SQL, and precompiler directives. See also *Oracle Database SQL Language Reference*, for related information on the SQL INSERT statement.
- C/C++ (Pro*C/C++) *Pro*C/C++ Programmer's Guide*: "Large Objects (LOBs)", "LOB Statements", "Embedded SQL Statements and Directives" — LOB FILE SET. See also (*Oracle Database SQL Language Reference*), Chapter 7 "SQL Statements" — INSERT
- Java (JDBC) *Oracle Database JDBC Developer's Guide*: "Working With LOBs and BFILEs" — Working with BFILEs.

Examples

- PL/SQL (DBMS_LOB): `finsert.sql`
- OCI: `finsert.c`
- Java (JDBC): `finsert.java`

22

Using LOB APIs

This chapter describes APIs that perform operations on BLOB, CLOB, and NCLOB data types. The operations given in this chapter can be used with either persistent or temporary LOB instances. Note that operations in this chapter do not apply to BFILEs. APIs covered in this chapter are listed in Table 22–1.

See Also:

- Chapter 19, "Operations Specific to Persistent and Temporary LOBs" for information on how to create temporary and persistent LOB instances and other operations specific to temporary or persistent LOBs.
- Chapter 21, "LOB APIs for BFILE Operations" for information on operations specific to BFILE instances.

The following information is given for each operation described in this chapter:

- **Preconditions** describe dependencies that must be met and conditions that must exist before calling each operation.
- **Usage Notes** provide implementation guidelines such as information specific to a given programmatic environment or data type.
- **Syntax** refers you to the syntax reference documentation for each supported programmatic environment.
- **Examples** describe any setup tasks necessary to run the examples given. Demonstration files listed are available in subdirectories under $ORACLE_HOME/rdbms/demo/lobs/ named plsql, oci, vb, and java. The driver program lobdemo.sql is in /plsql and the driver program lobdemo.c is in /oci.

This chapter contains these topics:

Supported Environments

Table 22–1, " Environments Supported for LOB APIs" indicates which programmatic environments are supported for the APIs discussed in this chapter. The first column describes the operation that the API performs. The remaining columns indicate with Yes or No whether the API is supported in PL/SQL, OCI, OCCI, COBOL, Pro*C/C++, and JDBC.

Table 22–1 Environments Supported for LOB APIs

Operation	PL/SQL	OCI	OCCI	COBOL	Pro*C/C++	JDBC
Appending One LOB to Another on page 22-3	Yes	Yes	No	Yes	Yes	Yes
Determining Character Set Form on page 22-4	No	Yes	No	No	No	No
Determining Character Set ID on page 22-5	No	Yes	No	No	No	No
Determining Chunk Size, See: Writing Data to a LOB on page 22-26	Yes	Yes	Yes	Yes	Yes	Yes
Comparing All or Part of Two LOBs on page 22-20	Yes	No	No	Yes	Yes	Yes
Converting a BLOB to a CLOB on page 22-39	Yes	No	No	No	No	No

Table 22–1 (Cont.) Environments Supported for LOB APIs

Operation	PL/SQL	OCI	OCCI	COBOL	Pro*C/C++	JDBC
Converting a CLOB to a BLOB on page 22-39	Yes	No	No	No	No	No
Copying a LOB Locator on page 22-23	Yes	Yes	No	Yes	Yes	Yes
Copying All or Part of One LOB to Another LOB on page 22-22	Yes	Yes	No	Yes	Yes	Yes
Disabling LOB Buffering on page 22-37	No	Yes	No	Yes	Yes	No
Displaying LOB Data on page 22-11	Yes	Yes	No	Yes	Yes	Yes
Enabling LOB Buffering on page 22-35	No	No	No	Yes	Yes	No
Equality: Checking If One LOB Locator Is Equal to Another on page 22-24	No	Yes	No	No	Yes	Yes
Erasing Part of a LOB on page 22-34	Yes	Yes	No	Yes	Yes	Yes
Flushing the Buffer on page 22-36	No	Yes	No	Yes	Yes	No
Determining Whether LOB Locator Is Initialized on page 22-24	No	Yes	No	No	Yes	No
Length: Determining the Length of a LOB on page 22-22	Yes	Yes	No	Yes	Yes	Yes
Loading a LOB with Data from a BFILE on page 22-5	Yes	Yes	No	Yes	Yes	Yes
Loading a BLOB with Data from a BFILE on page 22-7	Yes	No	No	No	No	No
Loading a CLOB or NCLOB with Data from a BFILE on page 22-8	Yes	No	No	No	No	No
LOB Array Read on page 22-13	No	Yes	No	No	No	No
LOB Array Write on page 22-28	No	Yes	No	No	No	No
Opening Persistent LOBs with the OPEN and CLOSE Interfaces on page 12-9	Yes	Yes	Yes	Yes	Yes	Yes
Open: Determining Whether a LOB is Open on page 22-10	Yes	Yes	Yes	Yes	Yes	Yes
Patterns: Checking for Patterns in a LOB Using INSTR on page 22-21	Yes	No	No	Yes	Yes	Yes
Reading a Portion of a LOB (SUBSTR) on page 22-20	Yes	No	No	Yes	Yes	Yes
Reading Data from a LOB on page 22-12	Yes	Yes	No	Yes	Yes	Yes
Storage Limit, Determining: Maximum Storage Limit for Terabyte-Size LOBs on page 12-22	Yes	No	No	No	No	No
Trimming LOB Data on page 22-34	Yes	Yes	No	Yes	Yes	Yes
WriteNoAppend, see Appending to a LOB on page 22-25.	No	No	No	No	No	No
Writing Data to a LOB on page 22-26	Yes	Yes	Yes	Yes	Yes	Yes

Appending One LOB to Another

This operation appends one LOB instance to another.

Preconditions

Before you can append one LOB to another, the following conditions must be met:

- Two LOB instances must exist.
- Both instances must be of the same type, for example both `BLOB` or both `CLOB` types.
- You can pass any combination of persistent or temporary LOB instances to this operation.

Usage Notes

Persistent LOBs: You must lock the row you are selecting the LOB from prior to updating a LOB value if you are using the PL/SQL `DBMS_LOB` Package or OCI. While the SQL `INSERT` and `UPDATE` statements implicitly lock the row, locking the row can be done explicitly using the SQL `SELECT FOR UPDATE` statement in SQL and PL/SQL programs, or by using an OCI `pin` or `lock` function in OCI programs. For more details on the state of the locator after an update, refer to "Example of Updating LOBs Through Updated Locators" on page 12-12.

Syntax

See the following syntax references for each programmatic environment:

- PL/SQL (DBMS_LOB Package): *Oracle Database PL/SQL Packages and Types Reference* "DBMS_LOB" — APPEND
- C (OCI): *Oracle Call Interface Programmer's Guide* "LOB Functions" — OCILobAppend()
- C++ (OCCI): *Oracle C++ Call Interface Programmer's Guide*
- COBOL (Pro*COBOL) *Pro*COBOL Programmer's Guide* for information on LOBs, usage notes on LOB Statements, and embedded SQL and precompiler directives — LOB APPEND.
- C/C++ (Pro*C/C++): *Pro*C/C++ Programmer's Guide* for information on embedded SQL statements and directives — LOB APPEND
- Java (JDBC): *Oracle Database JDBC Developer's Guide* for information on creating and populating LOB columns in Java.

Examples

To run the following examples, you must create two LOB instances and pass them when you call the given append operation. Creating a LOB instance is described in Chapter 19, "Operations Specific to Persistent and Temporary LOBs".

Examples for this use case are provided in the following programmatic environments:

- PL/SQL (DBMS_LOB Package): `lappend.sql`
- OCI: `lappend.c`
- Java (JDBC): `lappend.java`

Determining Character Set Form

This section describes how to get the character set form of a LOB instance.

Syntax

Use the following syntax references for each programmatic environment:

- PL/SQL (DBMS_LOB Package): There is no applicable syntax reference for this operation.
- C (OCI): *Oracle Call Interface Programmer's Guide* "LOB Functions" — OCILobCharSetForm()
- C++ (OCCI): *Oracle C++ Call Interface Programmer's Guide*
- COBOL (Pro*COBOL): There is no applicable syntax reference for this operation
- C/C++ (Pro*C/C++): There is no applicable syntax reference for this operation.
- Java (JDBC): There is no applicable syntax reference for this operation.

Example

The example demonstrates how to determine the character set form of the foreign language text (`ad_fltextn`).

This functionality is currently available only in OCI:

- OCI: `lgetchfm.c`

Determining Character Set ID

This section describes how to determine the character set ID.

Syntax

Use the following syntax references for each programmatic environment:

- PL/SQL (DBMS_LOB Package): There is no applicable syntax reference for this operation.
- C (OCI): *Oracle Call Interface Programmer's Guide* "Relational Functions" — LOB Functions, OCILobCharSetId()
- C++ (OCCI): *Oracle C++ Call Interface Programmer's Guide*
- COBOL (Pro*COBOL): There is no applicable syntax reference for this operation.
- C/C++ (Pro*C/C++): There is no applicable syntax reference for this operation
- Java (JDBC): There is no applicable syntax reference for this operation.

Example

This functionality is currently available only in OCI:

- OCI: `lgetchar.c`

Loading a LOB with Data from a BFILE

This operation loads a LOB with data from a `BFILE`. This procedure can be used to load data into any persistent or temporary LOB instance of any LOB data type.

See Also:

- The `LOADBLOBFROMFILE` and `LOADCLOBFROMFILE` procedures implement the functionality of this procedure and provide improved features for loading binary data and character data. (These improved procedures are available in the PL/SQL environment only.) When possible, using one of the improved procedures is recommended. See "Loading a BLOB with Data from a BFILE" on page 22-7 and "Loading a CLOB or NCLOB with Data from a BFILE" on page 22-8 for more information.
- As an alternative to this operation, you can use SQL*Loader to load persistent LOBs with data directly from a file in the file system. See "Using SQL*Loader to Load LOBs" on page 3-1 for more information.

Preconditions

Before you can load a LOB with data from a `BFILE`, the following conditions must be met:

- The `BFILE` must exist.
- The target LOB instance must exist.

Usage Notes

Note the following issues regarding this operation.

Use LOADCLOBFROMFILE When Loading Character Data

When you use the `DBMS_LOB.LOADFROMFILE` procedure to load a `CLOB` or `NCLOB` instance, you are loading the LOB with binary data from the `BFILE` and no implicit character set conversion is performed. For this reason, using the `DBMS_LOB.LOADCLOBFROMFILE` procedure is recommended when loading character data, see Loading a CLOB or NCLOB with Data from a BFILE on page 22-8 for more information.

Specifying Amount of BFILE Data to Load

The value you pass for the amount parameter to functions listed in Table 22–2 must be one of the following:

- An amount less than or equal to the actual size (in bytes) of the `BFILE` you are loading.
- The maximum allowable LOB size (in bytes). Passing this value, loads the entire `BFILE`. You can use this technique to load the entire `BFILE` without determining the size of the `BFILE` before loading. To get the maximum allowable LOB size, use the technique described in Table 22–2.

Table 22–2 Maximum LOB Size for Load from File Operations

Environment	Function	To pass maximum LOB size, get value of:
`DBMS_LOB`	`DBMS_LOB.LOADBLOBFROMFILE`	`DBMS_LOB.LOBMAXSIZE`
`DBMS_LOB`	`DBMS_LOB.LOADCLOBFROMFILE`	`DBMS_LOB.LOBMAXSIZE`
`OCI`	`OCILobLoadFromFile2()` (For LOBs of any size.)	`UB8MAXVAL`

Table 22–2 (Cont.) Maximum LOB Size for Load from File Operations

Environment	Function	To pass maximum LOB size, get value of:
OCI	OCILobLoadFromFile()	UB4MAXVAL
	(For LOBs less than 4 gigabytes in size.)	

Syntax

See the following syntax references for details on using this operation in each programmatic environment:

- PL/SQL (DBMS_LOB Package): *Oracle Database PL/SQL Packages and Types Reference* "DBMS_LOB" — LOADFROMFILE.
- C (OCI): *Oracle Call Interface Programmer's Guide* "LOB Functions" — OCILobLoadFromFile().
- C++ (OCCI): *Oracle C++ Call Interface Programmer's Guide*
- COBOL (Pro*COBOL) *Pro*COBOL Programmer's Guide* for information on LOBs, usage notes on LOB statements, and embedded SQL and precompiler directives — `LOB LOAD`, `LOB OPEN`, `LOB CLOSE`.
- C/C++ (Pro*C/C++): *Pro*C/C++ Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — `LOB LOAD`
- Java (JDBC): *Oracle Database JDBC Developer's Guide* Chapter 7, "Working With LOBs" — Creating and Populating a `BLOB` or `CLOB` Column.

Examples

Examples are provided in the following programmatic environments:

- PL/SQL (DBMS_LOB Package): `lloaddat.sql`
- OCI: `lloaddat.c`
- Java (JDBC): `lloaddat.java`

Loading a BLOB with Data from a BFILE

This procedure loads a `BLOB` with data from a `BFILE`. This procedure can be used to load data into any persistent or temporary `BLOB` instance.

See Also:

- "Loading a LOB with Data from a BFILE" on page 22-5
- To load character data, use `DBMS_LOB.LOADCLOBFROMFILE`. See "Loading a CLOB or NCLOB with Data from a BFILE" on page 22-8 for more information.
- As an alternative to this operation, you can use SQL*Loader to load persistent LOBs with data directly from a file in the file system. See "Using SQL*Loader to Load LOBs" on page 3-1 for more information.

Preconditions

The following conditions must be met before calling this procedure:

- The target `BLOB` instance must exist.

- The source BFILE must exist.
- You must open the BFILE. (After calling this procedure, you must close the BFILE at some point.)

Usage Notes

Note the following with respect to this operation:

New Offsets Returned

Using DBMS_LOB.LOADBLOBFROMFILE to load binary data into a BLOB achieves the same result as using DBMS_LOB.LOADFROMFILE, but also returns the new offsets of BLOB.

Specifying Amount of BFILE Data to Load

The value you pass for the amount parameter to the DBMS_LOB.LOADBLOBFROMFILE function must be one of the following:

- An amount less than or equal to the actual size (in bytes) of the BFILE you are loading.
- The maximum allowable LOB size: DBMS_LOB.LOBMAXSIZE. Passing this value causes the function to load the entire BFILE. This is a useful technique for loading the entire BFILE without introspecting the size of the BFILE.

See Also: Table 22–2, " Maximum LOB Size for Load from File Operations"

Syntax

See *Oracle Database PL/SQL Packages and Types Reference*, "DBMS_LOB" — LOADBLOBFROMFILE procedure for syntax details on this procedure.

Examples

This example is available in PL/SQL only. This API is not provided in other programmatic environments. The online file is lldblobf.sql. This example illustrates:

- How to use LOADBLOBFROMFILE to load the entire BFILE without getting its length first.
- How to use the return value of the offsets to calculate the actual amount loaded.

Loading a CLOB or NCLOB with Data from a BFILE

This procedure loads a CLOB or NCLOB with character data from a BFILE. This procedure can be used to load data into a persistent or temporary CLOB or NCLOB instance.

See Also:

- "Loading a LOB with Data from a BFILE" on page 22-5
- To load binary data, use DBMS_LOB.LOADBLOBFROMFILE. See "Loading a BLOB with Data from a BFILE" on page 22-7 for more information.
- As an alternative to this operation, you can use SQL*Loader to load persistent LOBs with data directly from a file in the file system. See "Using SQL*Loader to Load LOBs" on page 3-1 for more information.

Preconditions

The following conditions must be met before calling this procedure:

- The target CLOB or NCLOB instance must exist.
- The source BFILE must exist.
- You must open the BFILE. (After calling this procedure, you must close the BFILE at some point.)

Usage Notes

You can specify the character set id of the BFILE when calling this procedure. Doing so, ensures that the character set is properly converted from the BFILE data character set to the destination CLOB or NCLOB character set.

Specifying Amount of BFILE Data to Load

The value you pass for the amount parameter to the DBMS_LOB.LOADCLOBFROMFILE function must be one of the following:

- An amount less than or equal to the actual size (in characters) of the BFILE data you are loading.
- The maximum allowable LOB size: DBMS_LOB.LOBMAXSIZE

  Passing this value causes the function to load the entire BFILE. This is a useful technique for loading the entire BFILE without introspecting the size of the BFILE.

Syntax

See *Oracle Database PL/SQL Packages and Types Reference*, "DBMS_LOB" — LOADCLOBFROMFILE procedure for syntax details on this procedure.

Examples

The following examples illustrate different techniques for using this API:

- "PL/SQL: Loading Character Data from a BFILE into a LOB"
- "PL/SQL: Loading Segments of Character Data into Different LOBs"

PL/SQL: Loading Character Data from a BFILE into a LOB

The following example illustrates:

- How to use default csid (0).
- How to load the entire file without calling getlength for the BFILE.
- How to find out the actual amount loaded using return offsets.

This example assumes that ad_source is a BFILE in UTF8 character set format and the database character set is UTF8. The online file is lldclobf.sql.

PL/SQL: Loading Segments of Character Data into Different LOBs

The following example illustrates:

- How to get the character set ID from the character set name using the NLS_CHARSET_ID function.
- How to load a stream of data from a single BFILE into different LOBs using the returned offset value and the language context lang_ctx.

- How to read a warning message.

This example assumes that `ad_file_ext_01` is a `BFILE` in `JA16TSTSET` format and the database national character set is `AL16UTF16`. The online file is `lldclobs.sql`.

Determining Whether a LOB is Open

This operation determines whether a LOB is open.

Preconditions

The LOB instance must exist before executing this procedure.

Usage Notes

When a LOB is open, it must be closed at some point later in the session.

Syntax

Use the following syntax references for each programmatic environment:

- PL/SQL (DBMS_LOB Package): *Oracle Database PL/SQL Packages and Types Reference* "DBMS_LOB" — `OPEN`, `ISOPEN`.
- C (OCI): *Oracle Call Interface Programmer's Guide* "LOB Functions" — `OCILobIsOpen()`.
- C++ (OCCI): *Oracle C++ Call Interface Programmer's Guide*
- COBOL (Pro*COBOL) *Pro*COBOL Programmer's Guide* for information on LOBs, usage notes on LOB Statements, and embedded SQL and precompiler directives — LOB DESCRIBE.
- C/C++ (Pro*C/C++): *Pro*C/C++ Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — `LOB DESCRIBE ... ISOPEN ...`
- Java (JDBC): *Oracle Database JDBC Developer's Guide*, "Working With LOBs" — Creating and Populating a `BLOB` or `CLOB` Column.

Examples

Examples are provided in the following programmatic environments:

- PL/SQL (DBMS_LOB Package): `lisopen.sql`
- OCI: `lisopen.c`
- C++ (OCCI): No example is provided with this release.
- Java (JDBC): `lisopen.java`

Java (JDBC): Checking If a LOB Is Open

Here is how to check a `BLOB` or a `CLOB`.

Checking If a CLOB Is Open

To see if a `CLOB` is open, your JDBC application can use the `isOpen` method defined in `oracle.sql.CLOB`. The return Boolean value indicates whether the `CLOB` has been previously opened or not. The `isOpen` method is defined as follows:

```
/**
  * Check whether the CLOB is opened.
  * @return true if the LOB is opened.
```

```
  */
public boolean isOpen () throws SQLException
```

The usage example is:

```
CLOB clob = ...
 // See if the CLOB is opened
 boolean isOpen = clob.isOpen ();
...
```

Checking If a BLOB Is Open

To see if a `BLOB` is open, your JDBC application can use the `isOpen` method defined in `oracle.sql.BLOB`. The return Boolean value indicates whether the `BLOB` has been previously opened or not. The `isOpen` method is defined as follows:

```
/**
 * Check whether the BLOB is opened.
 * @return true if the LOB is opened.
 */
 public boolean isOpen () throws SQLException
```

The usage example is:

```
BLOB blob = ...
// See if the BLOB is opened
boolean isOpen = blob.isOpen ();
...
```

Displaying LOB Data

This section describes APIs that allow you to read LOB data. You can use this operation to read LOB data into a buffer. This is useful if your application requires displaying large amounts of LOB data or streaming data operations.

Usage Notes

Note the following when using these APIs.

Streaming Mechanism

The most efficient way to read large amounts of LOB data is to use `OCILobRead2()` with the streaming mechanism enabled.

Amount Parameter

The value you pass for the amount parameter is restricted for the APIs described in Table 22–3.

Table 22–3 Maximum LOB Size for Amount Parameter

Environment	Function	Value of amount parameter is limited to:
DBMS_LOB	`DBMS_LOB.READ`	The size of the buffer, 32Kbytes.
OCI	`OCILobRead()`	`UB4MAXVAL`
	(For LOBs less than 4 gigabytes in size.)	Specifying this amount reads the entire file.

Table 22–3 (Cont.) Maximum LOB Size for Amount Parameter

Environment	Function	Value of amount parameter is limited to:
OCI	`OCILobRead2()`	`UB8MAXVAL`
	(For LOBs of any size.)	Specifying this amount reads the entire file.

Syntax

Use the following syntax references for each programmatic environment:

- PL/SQL (DBMS_LOB Package): *Oracle Database PL/SQL Packages and Types Reference* "DBMS_LOB" — OPEN, READ, CLOSE.
- C (OCI): *Oracle Call Interface Programmer's Guide* "LOB Functions" —, OCILobOpen(), OCILobRead2(), OCILobClose().
- C++ (OCCI): *Oracle C++ Call Interface Programmer's Guide*
- COBOL (Pro*COBOL) *Pro*COBOL Programmer's Guide* for information on LOBs, usage notes on LOB statements, and embedded SQL and precompiler directives — LOB READ.
- C/C++ (Pro*C/C++): *Pro*C/C++ Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — `LOB READ`
- Java (JDBC): *Oracle Database JDBC Developer's Guide*, "Working With LOBs" — Creating and Populating a `BLOB` or `CLOB` Column.

Examples

Examples are provided in the following programmatic environments:

- PL/SQL (DBMS_LOB Package): `ldisplay.sql`
- OCI: `ldisplay.c`
- C++ (OCCI): No example is provided in this release.
- Java (JDBC): `ldisplay.java`

Reading Data from a LOB

This section describes how to read data from LOBs using `OCILobRead2()`.

Usage Notes

Note the following when using this operation.

Streaming Read in OCI

The most efficient way to read large amounts of LOB data is to use `OCILobRead2()` with the streaming mechanism enabled using polling or callback. To do so, specify the starting point of the read using the `offset` parameter as follows:

```
ub8  char_amt =  0;
ub8  byte_amt =  0;
ub4  offset = 1000;

OCILobRead2(svchp, errhp, locp, &byte_amt, &char_amt, offset, bufp, bufl,
            OCI_ONE_PIECE, 0, 0, 0, 0);
```

When using *polling mode*, be sure to look at the value of the `byte_amt` parameter after each `OCILobRead2()` call to see how many bytes were read into the buffer because the buffer may not be entirely full.

When using *callbacks*, the `lenp` parameter, which is input to the callback, indicates how many bytes are filled in the buffer. Be sure to check the `lenp` parameter during your callback processing because the entire buffer may not be filled with data (see the *Oracle Call Interface Programmer's Guide*.)

Chunk Size

A chunk is one or more Oracle blocks. You can specify the chunk size for the BasicFiles LOB when creating the table that contains the LOB. This corresponds to the data size used by Oracle Database when accessing or modifying the LOB value. Part of the chunk is used to store system-related information and the rest stores the LOB value. The API you are using has a function that returns the amount of space used in the LOB chunk to store the LOB value. In PL/SQL use `DBMS_LOB.GETCHUNKSIZE`. In OCI, use `OCILobGetChunkSize()`. For SecureFiles, `CHUNK` is an advisory size and is provided for backward compatibility purposes.

To improve performance, you may run `write` requests using a multiple of the value returned by one of these functions. The reason for this is that you are using the same unit that the Oracle database uses when reading data from disk. If it is appropriate for your application, then you should batch reads until you have enough for an entire chunk instead of issuing several LOB read calls that operate on the same LOB chunk.

Syntax

Use the following syntax references for each programmatic environment:

- PL/SQL (DBMS_LOB Package): *Oracle Database PL/SQL Packages and Types Reference* "DBMS_LOB" — `OPEN`, `GETCHUNKSIZE`, `READ`, `CLOSE`.
- C (OCI): *Oracle Call Interface Programmer's Guide* "LOB Functions" — `OCILobOpen()`, `OCILobRead2()`, `OCILobClose()`.
- C++ (OCCI): *Oracle C++ Call Interface Programmer's Guide*
- COBOL (Pro*COBOL) *Pro*COBOL Programmer's Guide* for information on LOBs, usage notes on LOB statements, and embedded SQL and precompiler directives — `LOB READ`.
- C/C++ (Pro*C/C++): *Pro*C/C++ Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — LOB READ
- Java (JDBC): *Oracle Database JDBC Developer's Guide* Chapter 7, "Working With LOBs" — Creating and Populating a `BLOB` or `CLOB` Column.

Examples

Examples are provided in the following programmatic environments:

- PL/SQL (DBMS_LOB Package): `lread.sql`
- OCI: `lread.c`
- Java (JDBC): `lread.java`

LOB Array Read

This section describes how to read LOB data for multiple locators in one round trip, using `OCILobArrayRead()`.

Usage Notes

This function improves performance in reading LOBs in the size range less than about 512 Kilobytes. For an OCI application example, assume that the program has a prepared SQL statement such as:

```
SELECT lob1 FROM lob_table for UPDATE;
```

where `lob1` is the LOB column and `lob_array` is an array of define variables corresponding to a LOB column:

```
OCILobLocator * lob_array[10];

...
 for (i=0; i<10, i++)        /* initialize array of locators */
    lob_array[i] = OCIDescriptorAlloc(..., OCI_DTYPE_LOB, ...);

...

OCIDefineByPos(..., 1, (dvoid *) lob_array, ... SQLT_CLOB, ...);

/* Execute the statement with iters = 10 to do an array fetch of 10 locators. */
OCIStmtExecute ( <service context>, <statement handle>, <error handle>,
                 10,    /* iters  */
                 0,     /* row offset */
                 NULL,  /* snapshot IN */
                 NULL,  /* snapshot out */
                 OCI_DEFAULT /* mode */);
...

  ub4 array_iter = 10;
  char  *bufp[10];
  oraub8 bufl[10];
  oraub8 char_amtp[10];
  oraub8 offset[10];

 for (i=0; i<10; i++)
  {
    bufp[i] = (char *)malloc(1000);
    bufl[i] = 1000;
    offset[i] = 1;
    char_amtp[i] = 1000;  /* Single byte fixed width char set. */
  }

/* Read the 1st 1000 characters for all 10 locators in one
 * round trip. Note that offset and amount need not be
 * same for all the locators. */

OCILobArrayRead(<service context>, <error handle>,
                &array_iter, /* array size */
                lob_array,   /* array of locators */
                NULL,        /* array of byte amounts */
                char_amtp,   /* array of char amounts */
                offset,      /* array of offsets */
       (void **)bufp,        /* array of read buffers */
                bufl,        /* array of buffer lengths */
                OCI_ONE_PIECE,  /* piece information */
                NULL,           /* callback context */
                NULL,           /* callback function */
                0,              /* character set ID - default */
                SQLCS_IMPLICIT);/* character set form */
```

```
 ...

for (i=0; i<10; i++)
  {
    /* Fill bufp[i] buffers with data to be written */
    strncpy (bufp[i], "Test Data------", 15);
    bufl[i] = 1000;
    offset[i] = 50;
    char_amtp[i] = 15;  /* Single byte fixed width char set. */
  }

/* Write the 15 characters from offset 50 to all 10
 * locators in one round trip. Note that offset and
 * amount need not be same for all the locators. */
 */

OCILobArrayWrite(<service context>, <error handle>,
                  &array_iter, /* array size */
                  lob_array,   /* array of locators */
                  NULL,        /* array of byte amounts */
                  char_amtp,   /* array of char amounts */
                  offset,      /* array of offsets */
             (void **)bufp,    /* array of read buffers */
                  bufl,        /* array of buffer lengths */
                  OCI_ONE_PIECE,  /* piece information */
                  NULL,           /* callback context */
                  NULL,           /* callback function */
                  0,              /* character set ID - default */
                  SQLCS_IMPLICIT);/* character set form */
...
```

Streaming Support

LOB array APIs can be used to read/write LOB data in multiple pieces. This can be done by using polling method or a callback function.

Here data is read/written in multiple pieces sequentially for the array of locators. For polling, the API would return to the application after reading/writing each piece with the `array_iter` parameter (OUT) indicating the index of the locator for which data is read/written. With a callback, the function is called after reading/writing each piece with `array_iter` as IN parameter.

Note that:

- It is possible to read/write data for a few of the locators in one piece and read/write data for other locators in multiple pieces. Data is read/written in one piece for locators which have sufficient buffer lengths to accommodate the whole data to be read/written.
- Your application can use different amount value and buffer lengths for each locator.
- Your application can pass zero as the amount value for one or more locators indicating pure streaming for those locators. In the case of reading, LOB data is read to the end for those locators. For writing, data is written until `OCI_LAST_PIECE` is specified for those locators.

LOB Array Read in Polling Mode

The following example reads 10Kbytes of data for each of 10 locators with 1Kbyte buffer size. Each locator needs 10 pieces to read the complete data. `OCILobArrayRead()` must be called 100 (10*10) times to fetch all the data.

First we call `OCILobArrayRead()` with `OCI_FIRST_PIECE` as `piece` parameter. This call returns the first 1K piece for the first locator.

Next `OCILobArrayRead()` is called in a loop until the application finishes reading all the pieces for the locators and returns `OCI_SUCCESS`. In this example it loops 99 times returning the pieces for the locators sequentially.

```
/* Fetch the locators */
...

     /* array_iter parameter indicates the number of locators in the array read.
      * It is an IN parameter for the 1st call in polling and is ignored as IN
      * parameter for subsequent calls. As OUT parameter it indicates the locator
      * index for which the piece is read.
      */

     ub4    array_iter = 10;
     char  *bufp[10];
     oraub8 bufl[10];
     oraub8 char_amtp[10];
     oraub8 offset[10];
     sword  st;

     for (i=0; i<10; i++)
     {
       bufp[i] = (char *)malloc(1000);
       bufl[i] = 1000;
       offset[i] = 1;
       char_amtp[i] = 10000;       /* Single byte fixed width char set. */
     }

     st =  OCILobArrayRead(<service context>, <error handle>,
                         &array_iter, /* array size */
                         lob_array, /* array of locators */
                         NULL,      /* array of byte amounts */
                         char_amtp, /* array of char amounts */
                         offset,    /* array of offsets */
                 (void **)bufp,     /* array of read buffers */
                         bufl,      /* array of buffer lengths */
                         OCI_FIRST_PIECE, /* piece information */
                         NULL,           /* callback context */
                         NULL,           /* callback function */
                         0,              /* character set ID - default */
                         SQLCS_IMPLICIT); /* character set form */

     /* First piece for the first locator is read here.
      * bufp[0]          => Buffer pointer into which data is read.
      * char_amtp[0 ]    => Number of characters read in current buffer
      *
      */

     While ( st == OCI_NEED_DATA)
     {
          st =  OCILobArrayRead(<service context>, <error handle>,
                          &array_iter, /* array size */
                          lob_array, /* array of locators */
```

```
                          NULL,      /* array of byte amounts */
                          char_amtp, /* array of char amounts */
                          offset,    /* array of offsets */
                    (void **)bufp,      /* array of read buffers */
                          bufl,      /* array of buffer lengths */
                          OCI_NEXT_PIECE, /* piece information */
                          NULL,           /* callback context */
                          NULL,           /* callback function */
                          0,              /* character set ID - default */
                          SQLCS_IMPLICIT);

        /* array_iter returns the index of the current array element for which
         * data is read. for example, aray_iter = 1 implies first locator,
         * array_iter = 2 implies second locator and so on.
         *
         * lob_array[ array_iter - 1]=> Lob locator for which data is read.
         * bufp[array_iter - 1]      => Buffer pointer into which data is read.
         * char_amtp[array_iter - 1] => Number of characters read in current buffer
         */

...
         /* Consume the data here */
...
     }
```

LOB Array Read with Callback

The following example reads 10Kbytes of data for each of 10 locators with 1Kbyte buffer size. Each locator needs 10 pieces to read all the data. The callback function is called 100 (10*10) times to return the pieces sequentially.

```
/* Fetch the locators */
...
     ub4    array_iter = 10;
     char  *bufp[10];
     oraub8 bufl[10];
     oraub8 char_amtp[10];
     oraub8 offset[10];
     sword  st;

     for (i=0; i<10; i++)
     {
       bufp[i] = (char *)malloc(1000);
       bufl[i] = 1000;
       offset[i] = 1;
       char_amtp[i] = 10000;       /* Single byte fixed width char set. */
      }

      st =  OCILobArrayRead(<service context>, <error handle>,
                        &array_iter, /* array size */
                        lob_array,   /* array of locators */
                        NULL,        /* array of byte amounts */
                        char_amtp,   /* array of char amounts */
                        offset,      /* array of offsets */
                 (void **)bufp,        /* array of read buffers */
                        bufl,        /* array of buffer lengths */
                        OCI_FIRST_PIECE,  /* piece information */
                        ctx,              /* callback context */
                        cbk_read_lob,     /* callback function */
                        0,                /* character set ID - default */
                        SQLCS_IMPLICIT);
```

```
...
/* Callback function for LOB array read. */
sb4 cbk_read_lob(dvoid *ctxp, ub4 array_iter, CONST dvoid *bufxp, oraub8 len,
                 ub1 piece, dvoid **changed_bufpp, oraub8 *changed_lenp)
{
   static ub4 piece_count = 0;
   piece_count++;
   switch (piece)
   {
    case OCI_LAST_PIECE:
      /*--- buffer processing code goes here ---*/
(void) printf("callback read the %d th piece(last piece) for %dth locator \n\n",
                piece_count, array_iter );
      piece_count = 0;
      break;
    case OCI_FIRST_PIECE:
      /*--- buffer processing code goes here ---*/
      (void) printf("callback read the 1st piece for %dth locator\n",
                    array_iter);
    /* --Optional code to set changed_bufpp and changed_lenp if the buffer needs
          to be changed dynamically --*/
      break;
    case OCI_NEXT_PIECE:
      /*--- buffer processing code goes here ---*/
      (void) printf("callback read the %d th piece for %dth locator\n",
                    piece_count, array_iter);
      /* --Optional code to set changed_bufpp and changed_lenp if the  buffer
           must be changed dynamically --*/
      break;
      default:
      (void) printf("callback read error: unkown piece = %d.\n", piece);
      return OCI_ERROR;
   }
    return OCI_CONTINUE;
}
...
```

Polling LOB Array Read

The next example is polling LOB data in `OCILobArrayRead()` with variable `amtp`, `bufl`, and `offset`.

```
/* Fetch the locators */
...

     ub4    array_iter = 10;
     char  *bufp[10];
     oraub8 bufl[10];
     oraub8 char_amtp[10];
     oraub8 offset[10];
     sword  st;

     for (i=0; i<10; i++)
     {
       bufp[i] = (char *)malloc(1000);
       bufl[i] = 1000;
       offset[i] = 1;
       char_amtp[i] = 10000;       /* Single byte fixed width char set. */
     }

     /* For 3rd locator read data in 500 bytes piece from offset 101. Amount
```

```
 * is 2000, that is, total number of pieces is 2000/500 = 4.
 */
offset[2] = 101; bufl[2] = 500; char_amtp[2] = 2000;

/* For 6th locator read data in 100 bytes piece from offset 51. Amount
 * is 0 indicating pure polling, that is, data is read till the end of
 * the LOB is reached.
 */
offset[5] = 51;  bufl[5] = 100; char_amtp[5] = 0;

/* For 8th locator read 100 bytes of data in one piece. Note amount
 * is less than buffer length indicating single piece read.
 */
offset[7] = 61;  bufl[7] = 200; char_amtp[7] = 100;

st =  OCILobArrayRead(<service context>, <error handle>,
                    &array_iter, /* array size */
                    lob_array, /* array of locators */
                    NULL,      /* array of byte amounts */
                    char_amtp, /* array of char amounts */
                    offset,    /* array of offsets */
           (void **)bufp,      /* array of read buffers */
                    bufl,      /* array of buffer lengths */
                    OCI_FIRST_PIECE, /* piece information */
                    NULL,           /* callback context */
                    NULL,           /* callback function */
                    0,              /* character set ID - default */
                    SQLCS_IMPLICIT); /* character set form */

/* First piece for the first locator is read here.
 * bufp[0]          => Buffer pointer into which data is read.
 * char_amtp[0 ]    => Number of characters read in current buffer
 *
 */

while ( st == OCI_NEED_DATA)
{
     st =  OCILobArrayRead(<service context>, <error handle>,
                     &array_iter, /* array size */
                     lob_array, /* array of locators */
                     NULL,      /* array of byte amounts */
                     char_amtp, /* array of char amounts */
                     offset,    /* array of offsets */
            (void **)bufp,      /* array of read buffers */
                     bufl,      /* array of buffer lengths */
                     OCI_NEXT_PIECE, /* piece information */
                     NULL,           /* callback context */
                     NULL,           /* callback function */
                     0,              /* character set ID - default */
                     SQLCS_IMPLICIT);

  /* array_iter returns the index of the current array element for which
   * data is read. for example, aray_iter = 1 implies first locator,
   * array_iter = 2 implies second locator and so on.
   *
   * lob_array[ array_iter - 1]=> Lob locator for which data is read.
   * bufp[array_iter - 1]      => Buffer pointer into which data is read.
   * char_amtp[array_iter - 1]=>Number of characters read in current buffer
   */
```

```
...
        /* Consume the data here */
...
     }
```

Syntax

Use the following syntax references for the OCI programmatic environment:

C (OCI): *Oracle Call Interface Programmer's Guide* "LOB Functions" — `OCILobArrayRead()`.

Example

An example is provided in the following programmatic environment:

OCI: lreadarr.c

Reading a Portion of a LOB (SUBSTR)

This section describes how to read a portion of a LOB using SUBSTR.

Syntax

Use the following syntax references for each programmatic environment:

- PL/SQL (DBMS_LOB Package): *Oracle Database PL/SQL Packages and Types Reference* "DBMS_LOB" — SUBSTR, OPEN, CLOSE
- C (OCI): There is no applicable syntax reference for this use case.
- C++ (OCCI): *Oracle C++ Call Interface Programmer's Guide*
- COBOL (Pro*COBOL) *Pro*COBOL Programmer's Guide* for information on LOBs, usage notes on LOB Statements, and embedded SQL and precompiler directives — `ALLOCATE, LOB OPEN, LOB READ, LOB CLOSE`.
- C/C++ (Pro*C/C++): *Pro*C/C++ Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — LOB READ. See PL/SQL `DBMS_LOB.SUBSTR`.
- Java (JDBC): *Oracle Database JDBC Developer's Guide*, "Working With LOBs" — Creating and Populating a `BLOB` or `CLOB` Column.

Examples

Examples are provided in the following programmatic environments:

- PL/SQL (DBMS_LOB Package): `lsubstr.sql`
- OCI: No example is provided with this release.
- C++ (OCCI): No example is provided with this release.
- Java (JDBC): `lsubstr.java`

Comparing All or Part of Two LOBs

This section describes how to compare all or part of two LOBs.

Syntax

Use the following syntax references for each programmatic environment:

- PL/SQL (DBMS_LOB Package): *Oracle Database PL/SQL Packages and Types Reference* "DBMS_LOB" — COMPARE.
- C (OCI): There is no applicable syntax reference for this use case.
- C++ (OCCI): *Oracle C++ Call Interface Programmer's Guide*
- COBOL (Pro*COBOL) *Pro*COBOL Programmer's Guide* or information on LOBs, usage notes on LOB Statements, and embedded SQL and precompiler directives — EXECUTE. Also reference PL/SQL DBMS_LOB.COMPARE.
- C/C++ (Pro*C/C++): *Pro*C/C++ Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — EXECUTE. Also reference PL/SQL `DBMS_LOB.COMPARE`.
- Java (JDBC): *Oracle Database JDBC Developer's Guide*, "Working With LOBs" — Creating and Populating a `BLOB` or `CLOB` Column.

Examples

Examples are provided in the following programmatic environments:

- PL/SQL (DBMS_LOB Package): `lcompare.sql`
- C (OCI): No example is provided with this release.
- C++ (OCCI): No example is provided with this release.
- Java (JDBC): `lcompare.java`

Patterns: Checking for Patterns in a LOB Using INSTR

This section describes how to see if a pattern exists in a LOB using `INSTR`.

Syntax

Use the following syntax references for each programmatic environment:

- PL/SQL (DBMS_LOB Package): *Oracle Database PL/SQL Packages and Types Reference* "DBMS_LOB" — INSTR.
- C (OCI): There is no applicable syntax reference for this use case.
- C++ (OCCI): *Oracle C++ Call Interface Programmer's Guide*
- COBOL (Pro*COBOL) *Pro*COBOL Programmer's Guide* for information on LOBs, usage notes on LOB statements, and embedded SQL and precompiler directives — EXECUTE. Also reference PL/SQL `DBMS_LOB.INSTR`.
- C/C++ (Pro*C/C++): *Pro*C/C++ Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — EXECUTE. Also reference PL/SQL `DBMS_LOB.INSTR`.
- Java (JDBC): *Oracle Database JDBC Developer's Guide* Chapter 7, "Working With LOBs" — Creating and Populating a `BLOB` or `CLOB` Column.

Examples

Examples are provided in the following programmatic environments:

- PL/SQL (DBMS_LOB Package): `linstr.sql`
- C (OCI): No example is provided with this release.
- C++ (OCCI): No example is provided with this release.
- Java (JDBC): `linstr.java`

Length: Determining the Length of a LOB

This section describes how to determine the length of a LOB.

Syntax

Use the following syntax references for each programmatic environment:

- PL/SQL (DBMS_LOB Package): *Oracle Database PL/SQL Packages and Types Reference* "DBMS_LOB" — GETLENGTH
- C (OCI): *Oracle Call Interface Programmer's Guide* "LOB Functions" — `OCILobGetLength2()`.
- C++ (OCCI): *Oracle C++ Call Interface Programmer's Guide*
- COBOL (Pro*COBOL) *Pro*COBOL Programmer's Guide* for information on LOBs, usage notes on LOB Statements, and embedded SQL and precompiler directives — LOB DESCRIBE.
- C/C++ (Pro*C/C++): *Pro*C/C++ Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — LOB DESCRIBE ... GET LENGTH...
- Java (JDBC): *Oracle Database JDBC Developer's Guide*, "Working With LOBs" — Creating and Populating a `BLOB` or `CLOB` Column.

Examples

Examples are provided in the following programmatic environments:

- PL/SQL (DBMS_LOB Package) `llength.sql`
- OCI: `llength.c`
- C++ (OCCI): No example is provided with this release.
- Java (JDBC): `llength.java`

Copying All or Part of One LOB to Another LOB

This section describes how to copy all or part of a LOB to another LOB. These APIs copy an amount of data you specify from a source LOB to a destination LOB.

Usage Notes

Note the following issues when using this API.

Specifying Amount of Data to Copy

The value you pass for the `amount` parameter to the `DBMS_LOB.COPY` function must be one of the following:

- An amount less than or equal to the actual size of the data you are loading.
- The maximum allowable LOB size: `DBMS_LOB.LOBMAXSIZE`. Passing this value causes the function to read the entire LOB. This is a useful technique for reading the entire LOB without introspecting the size of the LOB.

Note that for character data, the amount is specified in characters, while for binary data, the amount is specified in bytes.

Locking the Row Prior to Updating

If you plan to update a LOB value, then you must lock the row containing the LOB prior to updating. While the SQL `INSERT` and `UPDATE` statements implicitly lock the

row, locking is done explicitly by means of a SQL `SELECT FOR UPDATE` statement in SQL and PL/SQL programs, or by using an `OCI pin` or `lock` function in OCI programs.

For more details on the state of the locator after an update, refer to "Example of Updating LOBs Through Updated Locators" on page 12-12.

Syntax

See the following syntax references for each programmatic environment:

- PL/SQL (DBMS_LOB Package): *Oracle Database PL/SQL Packages and Types Reference* "DBMS_LOB" — COPY
- C (OCI): *Oracle Call Interface Programmer's Guide* "LOB Functions" — OCILobCopy2
- C++ (OCCI): *Oracle C++ Call Interface Programmer's Guide*
- COBOL (Pro*COBOL) *Pro*COBOL Programmer's Guide* for information on LOBs, usage notes on LOB Statements, and embedded SQL and precompiler directives — `LOB COPY`. Also reference PL/SQL `DBMS_LOB.COPY`.
- C/C++ (Pro*C/C++): *Pro*C/C++ Programmer's Guide* for information on embedded SQL statements and directives — `LOB COPY`
- Java (JDBC): *Oracle Database JDBC Developer's Guide*, "Working With LOBs" — Creating and Populating a `BLOB` or `CLOB` Column.

Examples

Examples are provided in the following programmatic environments:

- PL/SQL (DBMS_LOB Package): `lcopy.sql`
- OCI: `lcopy.c`
- Java (JDBC): `lcopy.java`

Copying a LOB Locator

This section describes how to copy a LOB locator. Note that different locators may point to the same or different data, or to current or outdated data.

> **Note:** To assign one LOB to another using PL/SQL, use the `:=` operator. This is discussed in more detail in "Read-Consistent Locators" on page 12-10.

Syntax

Use the following syntax references for each programmatic environment:

- PL/SQL (DBMS_LOB Package): Refer to "Read-Consistent Locators" on page 12-10 for information on assigning one lob locator to another.
- C (OCI): *Oracle Call Interface Programmer's Guide* "LOB Functions" — `OCILobAssign()`, `OCILobIsEqual()`.
- C++ (OCCI): *Oracle C++ Call Interface Programmer's Guide*
- COBOL (Pro*COBOL) *Pro*COBOL Programmer's Guide* for information on LOBs, usage notes on LOB Statements, and embedded SQL and precompiler directives — ALLOCATE, LOB ASSIGN.

- C/C++ (Pro*C/C++): *Pro*C/C++ Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — SELECT, LOB ASSIGN
- Java (JDBC): *Oracle Database JDBC Developer's Guide* Chapter 7, "Working With LOBs" — Creating and Populating a `BLOB` or `CLOB` Column.

Examples

Examples are provided in the following programmatic environments:

- PL/SQL (DBMS_LOB Package): `lcopyloc.sql`
- OCI: `lcopyloc.c`
- C++ (OCCI): No example is provided with this release.
- Java (JDBC): `lcopyloc.java`

Equality: Checking If One LOB Locator Is Equal to Another

This section describes how to determine whether one LOB locator is equal to another. If two locators are equal, then this means that they refer to the same version of the LOB data.

See Also:

- Table 22–1, " Environments Supported for LOB APIs" on page 22-2
- "Read-Consistent Locators" on page 12-10

Syntax

Use the following syntax references for each programmatic environment:

- C (OCI): *Oracle Call Interface Programmer's Guide* "LOB Functions" — `OCILobAssign()`, `OCILobIsEqual()`.
- C++ (OCCI): *Oracle C++ Call Interface Programmer's Guide*
- COBOL (Pro*COBOL): There is no applicable syntax reference for this use case.
- C/C++ (Pro*C/C++): *Pro*C/C++ Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — `LOB ASSIGN`
- Java (JDBC): *Oracle Database JDBC Developer's Guide*, "Working With LOBs" — Creating and Populating a `BLOB` or `CLOB` Column.

Examples

Examples are provided in the following programmatic environments:

- PL/SQL: No example is provided with this release.
- OCI: `lequal.c`
- C++ (OCCI): No example is provided with this release.
- Java (JDBC): `lequal.java`

Determining Whether LOB Locator Is Initialized

This section describes how to determine whether a LOB locator is initialized.

See Also: Table 22–1, " Environments Supported for LOB APIs" on page 22-2

Syntax

Use the following syntax references for each programmatic environment:

- PL/SQL (DBMS_LOB Package): There is no applicable syntax reference for this use case.
- C (OCI): *Oracle Call Interface Programmer's Guide* "LOB Functions" — `OCILobLocatorIsInit()`.
- C++ (OCCI): *Oracle C++ Call Interface Programmer's Guide*
- COBOL (Pro*COBOL): There is no applicable syntax reference for this use case.
- C/C++ (Pro*C/C++) *Pro*C/C++ Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives". See C(OCI), `OCILobLocatorIsInit()`.
- Java (JDBC): There is no applicable syntax reference for this use case.

Examples

Examples are provided in the following programmatic environments:

- PL/SQL (DBMS_LOB Package): No example is provided with this release.
- OCI: `linit.c`
- C (OCCI)): No example is provided with this release.
- Java (JDBC): No example is provided with this release.

Appending to a LOB

This section describes how to write-append the contents of a buffer to a LOB.

See Also: Table 22–1, " Environments Supported for LOB APIs" on page 22-2

Usage Notes

Note the following issues regarding usage of this API.

Writing Singly or Piecewise

The `writeappend` operation writes a buffer to the end of a LOB.

For OCI, the buffer can be written to the LOB in a single piece with this call; alternatively, it can be rendered piecewise using callbacks or a standard polling method.

Writing Piecewise: When to Use Callbacks or Polling

If the value of the piece parameter is `OCI_FIRST_PIECE`, then data must be provided through callbacks or polling.

- If a callback function is defined in the `cbfp` parameter, then this callback function is called to get the next piece after a piece is written to the pipe. Each piece is written from `bufp`.
- If no callback function is defined, then `OCILobWriteAppend2()` returns the `OCI_NEED_DATA` error code. The application must call `OCILobWriteAppend2()` again to

write more pieces of the LOB. In this mode, the buffer pointer and the length can be different in each call if the pieces are of different sizes and from different locations. A piece value of `OCI_LAST_PIECE` terminates the piecewise write.

Locking the Row Prior to Updating Prior to updating a LOB value using the PL/SQL `DBMS_LOB` package or the OCI, you must lock the row containing the LOB. While the SQL `INSERT` and `UPDATE` statements implicitly lock the row, locking is done explicitly by means of an SQL `SELECT FOR UPDATE` statement in SQL and PL/SQL programs, or by using an `OCI pin` or `lock` function in OCI programs.

For more details on the state of the locator after an update, refer to "Example of Updating LOBs Through Updated Locators" on page 12-12.

Syntax

Use the following syntax references for each programmatic environment:

- PL/SQL (DBMS_LOB Package): *Oracle Database PL/SQL Packages and Types Reference "DBMS_LOB"* — `WRITEAPPEND`
- C (OCI): *Oracle Call Interface Programmer's Guide* "LOB Functions" — OCILobWriteAppend2()
- C++ (OCCI): *Oracle C++ Call Interface Programmer's Guide*
- COBOL (Pro*COBOL) *Pro*COBOL Programmer's Guide* for information on LOBs, usage notes on LOB statements, and embedded SQL and precompiler directives — `LOB WRITE APPEND`.
- C/C++ (Pro*C/C++): *Pro*C/C++ Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — LOB WRITE APPEND
- Java (JDBC): *Oracle Database JDBC Developer's Guide* Chapter 7, "Working With LOBs" — Creating and Populating a `BLOB` or `CLOB` Column.

Examples

Examples are provided in the following programmatic environments:

- PL/SQL (DBMS_LOB Package): `lwriteap.sql`
- OCI: `lwriteap.c`
- C++ (OCCI): No example is provided with this release.
- Java (JDBC): `lwriteap.java`

Writing Data to a LOB

This section describes how to write the contents of a buffer to a LOB.

See Also:

- Table 22–1, " Environments Supported for LOB APIs" on page 22-2
- Reading Data from a LOB on page 22-12

Usage Notes

Note the following issues regarding usage of this API.

Stream Write

The most efficient way to write large amounts of LOB data is to use `OCILobWrite2()` with the streaming mechanism enabled, and using polling or a callback. If you know how much data is written to the LOB, then specify that amount when calling `OCILobWrite2()`. This ensures that LOB data on the disk is contiguous. Apart from being spatially efficient, the contiguous structure of the LOB data makes reads and writes in subsequent operations faster.

Chunk Size

A chunk is one or more Oracle blocks. You can specify the chunk size for the LOB when creating the table that contains the LOB. This corresponds to the data size used by Oracle Database when accessing or modifying the LOB value. Part of the chunk is used to store system-related information and the rest stores the LOB value. The API you are using has a function that returns the amount of space used in the LOB chunk to store the LOB value. In PL/SQL use `DBMS_LOB.GETCHUNKSIZE`. In OCI, use `OCILobGetChunkSize()`.

Use a Multiple of the Returned Value to Improve Write Performance

To improve performance, run write requests using a multiple of the value returned by one of these functions. The reason for this is that the LOB chunk is versioned for every `write` operation. If all `writes` are done on a chunk basis, then no extra or excess versioning is incurred or duplicated. If it is appropriate for your application, then you should batch writes until you have enough for an entire chunk instead of issuing several LOB write calls that operate on the same LOB chunk.

Locking the Row Prior to Updating

Prior to updating a LOB value using the PL/SQL `DBMS_LOB` Package or OCI, you must lock the row containing the LOB. While the SQL `INSERT` and `UPDATE` statements implicitly lock the row, locking is done explicitly by means of a SQL `SELECT FOR UPDATE` statement in SQL and PL/SQL programs, or by using an `OCI pin` or `lock` function in OCI programs.

For more details on the state of the locator after an update, refer to "Example of Updating LOBs Through Updated Locators" on page 12-12.

Using DBMS_LOB.WRITE to Write Data to a BLOB

When you are passing a hexadecimal string to DBMS_LOB.WRITE() to write data to a BLOB, use the following guidelines:

- The `amount` parameter should be <= the buffer `length` parameter
- The `length` of the buffer should be ((`amount`*2) - 1). This guideline exists because the two characters of the string are seen as one hexadecimal character (and an implicit hexadecimal-to-raw conversion takes place), that is, every two bytes of the string are converted to one raw byte.

The following example is *correct*:

```
declare
   blob_loc  BLOB;
   rawbuf RAW(10);
   an_offset INTEGER := 1;
   an_amount BINARY_INTEGER := 10;
BEGIN
   select blob_col into blob_loc from a_table
where id = 1;
   rawbuf := '1234567890123456789';
```

```
   dbms_lob.write(blob_loc, an_amount, an_offset,
rawbuf);
   commit;
END;
```

Replacing the value for `an_amount` in the previous example with the following values, yields error message, ora_21560:

```
    an_amount BINARY_INTEGER := 11;
```

or

```
    an_amount BINARY_INTEGER := 19;
```

Syntax

Use the following syntax references for each programmatic environment:

- PL/SQL (DBMS_LOB Package): *Oracle Database PL/SQL Packages and Types Reference "DBMS_LOB"* — WRITE
- C (OCI): *Oracle Call Interface Programmer's Guide* "LOB Functions" — OCILobWrite2().
- C++ (OCCI): *Oracle C++ Call Interface Programmer's Guide*
- COBOL (Pro*COBOL) *Pro*COBOL Programmer's Guide* for information on LOBs, usage notes on LOB statements, and embedded SQL and precompiler directives — `LOB WRITE`.
- C/C++ (Pro*C/C++): *Pro*C/C++ Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — `LOB` WRITE
- Java (JDBC): *Oracle Database JDBC Developer's Guide* Chapter 7, "Working With LOBs" — Creating and Populating a `BLOB` or `CLOB` Column.

Examples

Examples are provided in the following programmatic environments:

- PL/SQL (DBMS_LOB Package): `lwrite.sql`
- OCI: `lwrite.c`
- Java (JDBC): `lwrite.java`

LOB Array Write

This section describes how to write LOB data for multiple locators in one round trip, using `OCILobArrayWrite()`.

Usage Notes

> **See Also:** "LOB Array Read" on page 22-13 for examples of array read/write.

LOB Array Write in Polling Mode

The following example writes 10Kbytes of data for each of `10` locators with a 1K buffer size. `OCILobArrayWrite()` has to be called `100` (`10` times `10`) times to write all the data. The function is used in a similar manner to `OCILobWrite2()`.

```
/* Fetch the locators */
```

```
...

/* array_iter parameter indicates the number of locators in the array read.
 * It is an IN parameter for the 1st call in polling and is ignored as IN
 * parameter for subsequent calls. As an OUT parameter it indicates the locator
 * index for which the piece is written.
 */

ub4    array_iter = 10;
char  *bufp[10];
oraub8 bufl[10];
oraub8 char_amtp[10];
oraub8 offset[10];
sword  st;
int    i, j;

for (i=0; i<10; i++)
{
  bufp[i] = (char *)malloc(1000);
  bufl[i] = 1000;
  /* Fill bufp here. */
...
  offset[i] = 1;
  char_amtp[i] = 10000;       /* Single byte fixed width char set. */
}

for (i  = 1; i <= 10; i++)
{
 /* Fill up bufp[i-1] here.  The first piece for ith locator would be written from
    bufp[i-1] */
...
    st =  OCILobArrayWrite(<service context>, <error handle>,
                        &array_iter, /* array size */
                        lob_array,   /* array of locators */
                        NULL,        /* array of byte amounts */
                        char_amtp,   /* array of char amounts */
                        offset,      /* array of offsets */
                (void **)bufp,       /* array of write buffers */
                        bufl,        /* array of buffer lengths */
                        OCI_FIRST_PIECE, /* piece information */
                        NULL,            /* callback context */
                        NULL,            /* callback function */
                        0,               /* character set ID - default */
                        SQLCS_IMPLICIT); /* character set form */

 for ( j = 2; j < 10; j++)
 {
 /* Fill up bufp[i-1] here.  The jth piece for ith locator would be written from
    bufp[i-1] */
...
 st =  OCILobArrayWrite(<service context>, <error handle>,
                         &array_iter, /* array size */
                         lob_array,   /* array of locators */
                         NULL,        /* array of byte amounts */
                         char_amtp,   /* array of char amounts */
                         offset,      /* array of offsets */
                  (void **)bufp,      /* array of write buffers */
                         bufl,        /* array of buffer lengths */
                         OCI_NEXT_PIECE, /* piece information */
                         NULL,           /* callback context */
```

```
                        NULL,           /* callback function */
                        0,              /* character set ID - default */
                        SQLCS_IMPLICIT);

    /* array_iter returns the index of the current array element for which
     * data is being written. for example, aray_iter = 1 implies first locator,
     * array_iter = 2 implies second locator and so on. Here i = array_iter.
     *
     * lob_array[ array_iter - 1] => Lob locator for which data is written.
     * bufp[array_iter - 1]       => Buffer pointer from which data is written.
     * char_amtp[ array_iter - 1] => Number of characters written in
     * the piece just written
     */
}

/* Fill up bufp[i-1] here.  The last piece for ith locator would be written from
   bufp[i -1] */
...
 st =  OCILobArrayWrite(<service context>, <error handle>,
                        &array_iter, /* array size */
                        lob_array,   /* array of locators */
                        NULL,        /* array of byte amounts */
                        char_amtp,   /* array of char amounts */
                        offset,      /* array of offsets */
                (void **)bufp,       /* array of write buffers */
                        bufl,        /* array of buffer lengths */
                        OCI_LAST_PIECE,  /* piece information */
                        NULL,            /* callback context */
                        NULL,            /* callback function */
                        0,               /* character set ID - default */
                        SQLCS_IMPLICIT);
}

...
```

LOB Array Write with Callback

The following example writes 10Kbytes of data for each of 10 locators with a 1K buffer size. A total of 100 pieces must be written (10 pieces for each locator). The first piece is provided by the `OCILobArrayWrite()` call. The callback function is called 99 times to get the data for subsequent pieces to be written.

```
/* Fetch the locators */
...

    ub4    array_iter = 10;
    char  *bufp[10];
    oraub8 bufl[10];
    oraub8 char_amtp[10];
    oraub8 offset[10];
    sword  st;

    for (i=0; i<10; i++)
    {
      bufp[i] = (char *)malloc(1000);
      bufl[i] = 1000;
      offset[i] = 1;
      char_amtp[i] = 10000;       /* Single byte fixed width char set. */
    }

 st =  OCILobArrayWrite(<service context>, <error handle>,
```

```
                        &array_iter, /* array size */
                        lob_array,   /* array of locators */
                        NULL,        /* array of byte amounts */
                        char_amtp,   /* array of char amounts */
                        offset,      /* array of offsets */
                (void **)bufp,       /* array of write buffers */
                        bufl,        /* array of buffer lengths */
                        OCI_FIRST_PIECE,  /* piece information */
                        ctx,              /* callback context */
                        cbk_write_lob     /* callback function */
                        0,                /* character set ID - default */
                        SQLCS_IMPLICIT);

...

/* Callback function for LOB array write. */
sb4 cbk_write_lob(dvoid *ctxp, ub4 array_iter, dvoid *bufxp, oraub8 *lenp,
                  ub1 *piecep, ub1 *changed_bufpp, oraub8 *changed_lenp)
{
 static ub4 piece_count = 0;
 piece_count++;

 printf (" %dth piece written  for %dth locator \n\n", piece_count, array_iter);

 /*-- code to fill bufxp with data goes here. *lenp should reflect the  size and
  *   should be less than or equal to MAXBUFLEN -- */
 /* --Optional code to set changed_bufpp and changed_lenp if the buffer must
  *   be changed dynamically --*/

  if (this is the last data buffer for current locator)
     *piecep = OCI_LAST_PIECE;
  else if (this is the first data buffer for the next locator)
     *piecep = OCI_FIRST_PIECE;
     piece_count = 0;
  else
     *piecep = OCI_NEXT_PIECE;

     return OCI_CONTINUE;
    }
...
```

Polling LOB Data in Array Write

The next example is polling LOB data in `OCILobArrayWrite()` with variable `amtp`, `bufl`, and `offset`.

```
/* Fetch the locators */
...

ub4    array_iter = 10;
char  *bufp[10];
oraub8 bufl[10];
oraub8 char_amtp[10];
oraub8 offset[10];
sword  st;
int    i, j;
int piece_count;

for (i=0; i<10; i++)
{
  bufp[i] = (char *)malloc(1000);
```

```
  bufl[i] = 1000;
  /* Fill bufp here. */
...
  offset[i] = 1;
  char_amtp[i] = 10000;       /* Single byte fixed width char set. */
}

     /* For 3rd locator write data in 500 bytes piece from offset 101. Amount
      * is 2000, that is, total number of pieces is 2000/500 = 4.
      */
     offset[2] = 101; bufl[2] = 500; char_amtp[2] = 2000;

     /* For 6th locator write data in 100 bytes piece from offset 51. Amount
      * is 0 indicating pure polling, that is, data is written
      * till OCI_LAST_PIECE
      */
     offset[5] = 51;  bufl[5] = 100; char_amtp[5] = 0;

     /* For 8th locator write 100 bytes of data in one piece. Note amount
      * is less than buffer length indicating single piece write.
      */
     offset[7] = 61;  bufl[7] = 200; char_amtp[7] = 100;

for (i  = 1; i <= 10; i++)
{
 /* Fill up bufp[i-1] here.  The first piece for ith locator would be written from
    bufp[i-1] */
...
    /* Calculate number of pieces that must be written */
    piece_count = char_amtp[i-1]/bufl[i-1];

    /* Single piece case */
    if (char_amtp[i-1] <= bufl[i-1])
      piece_count = 1;

    /* Zero amount indicates pure polling. So we can write as many
     * pieces as needed. Let us write 50 pieces.
     */
    if (char_amtp[i-1] == 0)
      piece_count = 50;

    st =  OCILobArrayWrite(<service context>, <error handle>,
                      &array_iter, /* array size */
                      lob_array,   /* array of locators */
                      NULL,        /* array of byte amounts */
                      char_amtp,   /* array of char amounts */
                      offset,      /* array of offsets */
              (void **)bufp,       /* array of write buffers */
                      bufl,        /* array of buffer lengths */
                      OCI_FIRST_PIECE, /* piece information */
                      NULL,            /* callback context */
                      NULL,            /* callback function */
                      0,               /* character set ID - default */
                      SQLCS_IMPLICIT); /* character set form */

 for ( j = 2; j < piece_count; j++)
 {
   /* Fill up bufp[i-1] here. The jth piece for ith locator would be written
    * from bufp[i-1] */
...
```

```
   st =  OCILobArrayWrite(<service context>, <error handle>,
                          &array_iter, /* array size */
                          lob_array,   /* array of locators */
                          NULL,        /* array of byte amounts */
                          char_amtp,   /* array of char amounts */
                          offset,      /* array of offsets */
                  (void **)bufp,       /* array of write buffers */
                          bufl,        /* array of buffer lengths */
                          OCI_NEXT_PIECE, /* piece information */
                          NULL,           /* callback context */
                          NULL,           /* callback function */
                          0,              /* character set ID - default */
                          SQLCS_IMPLICIT);

    /* array_iter returns the index of the current array element for which
     * data is being written. for example, aray_iter = 1 implies first locator,
     * array_iter = 2 implies second locator and so on. Here i = array_iter.
     *
     * lob_array[ array_iter - 1] => Lob locator for which data is written.
     * bufp[array_iter - 1]       => Buffer pointer from which data is written.
     * char_amtp[ array_iter - 1] => Number of characters written in
     * the piece just written
     */
}

/* Fill up bufp[i-1] here.  The last piece for ith locator would be written from
 * bufp[i -1] */
...

/* If piece_count is 1 it is a single piece write. */
if (piece_count[i] != 1)
  st =  OCILobArrayWrite(<service context>, <error handle>,
                          &array_iter, /* array size */
                          lob_array,   /* array of locators */
                          NULL,        /* array of byte amounts */
                          char_amtp,   /* array of char amounts */
                          offset,      /* array of offsets */
                  (void **)bufp,       /* array of write buffers */
                          bufl,        /* array of buffer lengths */
                          OCI_LAST_PIECE,  /* piece information */
                          NULL,            /* callback context */
                          NULL,            /* callback function */
                          0,               /* character set ID - default */
                          SQLCS_IMPLICIT);
}

...
```

Syntax

Use the following syntax references for the OCI programmatic environment:

C (OCI): *Oracle Call Interface Programmer's Guide* "LOB Functions" — `OCILobArrayWrite()`.

Example

An example is provided in the following programmatic environment:

OCI: lwritearr.c

Trimming LOB Data

This section describes how to trim a LOB to the size you specify.

> **See Also:** Table 22–1, " Environments Supported for LOB APIs" on page 22-2

Usage Notes

Note the following issues regarding usage of this API.

Locking the Row Prior to Updating

Prior to updating a LOB value using the PL/SQL `DBMS_LOB` Package, or OCI, you must lock the row containing the LOB. While the SQL `INSERT` and `UPDATE` statements implicitly lock the row, locking is done explicitly by means of:

- A `SELECT FOR UPDATE` statement in SQL and PL/SQL programs.
- An `OCI pin` or `lock` function in OCI programs.

For more details on the state of the locator after an update, refer to "Example of Updating LOBs Through Updated Locators" on page 12-12.

Syntax

Use the following syntax references for each programmatic environment:

- PL/SQL (DBMS_LOB Package): *Oracle Database PL/SQL Packages and Types Reference "DBMS_LOB"* — `TRIM`
- C (OCI): *Oracle Call Interface Programmer's Guide* "LOB Functions" — OCILobTrim2().
- C++ (OCCI): *Oracle C++ Call Interface Programmer's Guide*
- COBOL (Pro*COBOL) *Pro*COBOL Programmer's Guide* for information on LOBs, usage notes on LOB statements, and embedded SQL and precompiler directives — `LOB TRIM`.
- C/C++ (Pro*C/C++): *Pro*C/C++ Programmer's Guide* Appendix F, "Embedded SQL and Precompiler Directives" — `LOB TRIM`
- Java (JDBC): *Oracle Database JDBC Developer's Guide* Chapter 7, "Working With LOBs" — Creating and Populating a `BLOB` or `CLOB` Column.

Examples

Examples are provided in the following programmatic environments:

- PL/SQL (DBMS_LOB Package): `ltrim.sql`
- OCI: `ltrim.c`
- C++ (OCCI): No example is provided with this release.
- Java (JDBC): `ltrim.java`

Erasing Part of a LOB

This section describes how to erase part of a LOB.

> **See Also:** Table 22–1, " Environments Supported for LOB APIs" on page 22-2

Usage Notes

Note the following issues regarding usage of this API.

Locking the Row Prior to Updating

Prior to updating a LOB value using the PL/SQL `DBMS_LOB` Package or OCI, you must lock the row containing the LOB. While `INSERT` and `UPDATE` statements implicitly lock the row, locking is done explicitly by means of a `SELECT FOR UPDATE` statement in SQL and PL/SQL programs, or by using the `OCI pin` or `lock` function in OCI programs.

For more details on the state of the locator after an update, refer to"Example of Updating LOBs Through Updated Locators" on page 12-12.

Syntax

Use the following syntax references for each programmatic environment:

- PL/SQL (DBMS_LOB Package): *Oracle Database PL/SQL Packages and Types Reference* "DBMS_LOB" — ERASE
- C (OCI): *Oracle Call Interface Programmer's Guide* "LOB Functions" — `OCILobErase2()`.
- C++ (OCCI): *Oracle C++ Call Interface Programmer's Guide*
- COBOL (Pro*COBOL) *Pro*COBOL Programmer's Guide* for information on LOBs, usage notes on LOB statements, and embedded SQL and precompiler directives — `LOB ERASE`.
- C/C++ (Pro*C/C++): *Pro*C/C++ Programmer's Guide* Appendix F, "Embedded SQL and Precompiler Directives" — `LOB ERASE`
- Java (JDBC): *Oracle Database JDBC Developer's Guide*, "Working With LOBs" — Creating and Populating a `BLOB` or `CLOB` Column.

Examples

Examples are provided in the following programmatic environments:

- PL/SQL (DBMS_LOB Package): `lerase.sql`
- OCI: `lerase.c`
- C++ (OCCI): No example is provided with this release.
- Java (JDBC): `lerase.java`

Enabling LOB Buffering

This section describes how to enable LOB buffering.

> **See Also:** Table 22–1, " Environments Supported for LOB APIs" on page 22-2

Usage Notes

Enable LOB buffering when you are performing a small read or write of data. Once you have completed these tasks, you must disable buffering before you can continue with any other LOB operations.

Note:

- You must flush the buffer in order to make your modifications persistent.
- Do not enable buffering for the stream read and write involved in checkin and checkout.

For more information, refer to "LOB Buffering Subsystem" on page 12-1.

Syntax

Use the following syntax references for each programmatic environment:

- PL/SQL: This API is not available in any supplied PL/SQL packages.
- C (OCI): *Oracle Call Interface Programmer's Guide* "LOB Functions" — `OCILobEnableBuffering()`, `OCILobDisableBuffering()`, `OCIFlushBuffer()`.
- C++ (OCCI): *Oracle C++ Call Interface Programmer's Guide*
- COBOL (Pro*COBOL) *Pro*COBOL Programmer's Guide* for information on LOBs, usage notes on LOB statements, and embedded SQL and precompiler directives — LOB ENABLE BUFFERING.
- C/C++ (Pro*C/C++): *Pro*C/C++ Programmer's Guide* Appendix F, "Embedded SQL and Precompiler Directives" — `LOB ENABLE BUFFERING`
- Java (JDBC): There is no applicable syntax reference for this use case.

Examples

Examples are provided in the following programmatic environments:

- PL/SQL: No example is provided.
- C (OCI): No example is provided with this release. Using this API is similar to that described in the example, "Disabling LOB Buffering" on page 22-37.
- C++ (OCCI): No example is provided with this release.
- Java (JDBC): No example provided.

Flushing the Buffer

This section describes how to flush the LOB buffer.

See Also: Table 22–1, " Environments Supported for LOB APIs" on page 22-2

Usage Notes

Enable buffering when performing a small read or write of data. Once you have completed these tasks, you must disable buffering before you can continue with any other LOB operations.

Notes:

- You must flush the buffer in order to make your modifications persistent.
- Do not enable buffering for the stream read and write involved in checkin and checkout.

For more information, refer to "LOB Buffering Subsystem" on page 12-1.

Syntax

Use the following syntax references for each programmatic environment:

- PL/SQL (DBMS_LOB Package): There is no applicable syntax reference for this use case.
- C (OCI): *Oracle Call Interface Programmer's Guide* "LOB Functions" — OCILobEnableBuffering(), OCILobDisableBuffering(), OCIFlushBuffer().
- C++ (OCCI): *Oracle C++ Call Interface Programmer's Guide*
- COBOL (Pro*COBOL) *Pro*COBOL Programmer's Guide* for information on LOBs, usage notes on LOB statements, and embedded SQL and precompiler directives — LOB FLUSH BUFFER.
- C/C++ (Pro*C/C++): *Pro*C/C++ Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — LOB FLUSH BUFFER.
- Java (JDBC): There is no applicable syntax reference for this use case.

Examples

Examples are provided in the following programmatic environments:

- PL/SQL (DBMS_LOB Package): No example is provided with this release.
- C (OCI): No example is provided with this release. Using this API is similar to that described in the example, "Disabling LOB Buffering" on page 22-37.
- C++ (OCCI): No example is provided with this release.
- Java (JDBC): No example is provided with this release.

Disabling LOB Buffering

This section describes how to disable LOB buffering.

See Also: Table 22–1, " Environments Supported for LOB APIs" on page 22-2

Usage Notes

Enable buffering when performing a small read or write of data. Once you have completed these tasks, you must disable buffering before you can continue with any other LOB operations.

Note:

- You must flush the buffer in order to make your modifications persistent.
- Do not enable buffering for the stream read and write involved in checkin and checkout.

For more information, refer to "LOB Buffering Subsystem" on page 12-1

Syntax

Use the following syntax references for each programmatic environment:

- PL/SQL (DBMS_LOB Package)*:* There is no applicable syntax reference for this use case.
- C (OCI): *Oracle Call Interface Programmer's Guide* "LOB Functions" — `OCILobEnableBuffering()`, `OCILobDisableBuffering()`, `OCIFlushBuffer()`.
- C++ (OCCI): *Oracle C++ Call Interface Programmer's Guide*
- COBOL (Pro*COBOL) *Pro*COBOL Programmer's Guide* for information on LOBs, usage notes on LOB statements, and embedded SQL and precompiler directives — LOB DISABLE BUFFER.
- C/C++ (Pro*C/C++): *Pro*C/C++ Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — LOB DISABLE BUFFER
- Java (JDBC): There is no applicable syntax reference for this use case.

Examples

Examples are provided in the following programmatic environments:

- PL/SQL (DBMS_LOB Package): No example is provided with this release.
- OCI: `ldisbuf.c`
- C++ (OCCI): No example is provided with this release.
- Java (JDBC): No example is provided with this release.

Determining Whether a LOB instance Is Temporary

This section describes how to determine whether a LOB instance is temporary.

See Also: Table 22–1, " Environments Supported for LOB APIs" on page 22-2

Syntax

Use the following syntax references for each programmatic environment:

- PL/SQL (DBMS_LOB): *Oracle Database PL/SQL Packages and Types Reference* "DBMS_LOB" — ISTEMPORARY, FREETEMPORARY
- C (OCI): *Oracle Call Interface Programmer's Guide* "LOB Functions" — OCILobIsTemporary().
- COBOL (Pro*COBOL) *Pro*COBOL Programmer's Guide* for information on LOBs, usage notes on LOB statements, and embedded SQL and precompiler directives — LOB DESCRIBE, ISTEMPORARY.

- C/C++ (Pro*C/C++): *Pro*C/C++ Programmer's Guide* Appendix F, "Embedded SQL Statements and Directives" — LOB DESCRIBE...ISTEMPORARY
- Java (JDBC): *Oracle Database JDBC Developer's Guide*, "Working With LOBs" — Creating and Populating a `BLOB` or `CLOB` Column.

Examples

Examples are provided in the following programmatic environments:

- PL/SQL (DBMS_LOB Package): `listemp.sql`
- OCI: `listemp.c`

Java (JDBC): Determining Whether a BLOB Is Temporary

To see if a `BLOB` is temporary, the JDBC application can either use the `isTemporary` instance method to determine whether the current `BLOB` object is temporary, or pass the `BLOB` object to the static `isTemporary` method to determine whether the specified `BLOB` object is temporary. These two methods are defined in`listempb.java`.

This JDBC API replaces previous work-arounds that use `DBMS_LOB.isTemporary()`.

To determine whether a `CLOB` is temporary, the JDBC application can either use the `isTemporary` instance method to determine whether the current `CLOB` object is temporary, or pass the `CLOB` object to the static `isTemporary` method. These two methods are defined in `listempc.java`.

Converting a BLOB to a CLOB

You can convert a `BLOB` instance to a `CLOB` using the PL/SQL procedure `DBMS_LOB.CONVERTTOCLOB`. This technique is convenient if you have character data stored in binary format that you want to store in a `CLOB`. You specify the character set of the binary data when calling this procedure. See *Oracle Database PL/SQL Packages and Types Reference* for details on syntax and usage of this procedure.

Converting a CLOB to a BLOB

You can convert a `CLOB` instance to a `BLOB` instance using the PL/SQL procedure `DBMS_LOB.CONVERTTOBLOB`. This technique is a convenient way to convert character data to binary data using LOB APIs. See *Oracle Database PL/SQL Packages and Types Reference* for details on syntax and usage of this procedure.

Ensuring Read Consistency

This script can be used to ensure that hot backups can be taken of tables that have `NOLOGGING` or `FILESYSTEM_LIKE_LOGGING` LOBs and have a known recovery point with no read inconsistencies:

```
ALTER DATABASE FORCE LOGGING;
SELECT CHECKPOINT_CHANGE# FROM V$DATABASE;  --Start SCN
```

SCN (System Change Number) is a stamp that defines a version of the database at the time that a transaction is committed.

Perform the backup.

Run the next script:

```
ALTER SYSTEM CHECKPOINT GLOBAL;
```

```
SELECT CHECKPOINT_CHANGE# FROM V$DATABASE;  --End SCN
ALTER DATABASE NO FORCE LOGGING;
```

Back up the archive logs generated by the database. At the minimum, archive logs between start SCN and end SCN (including both SCN points) must be backed up.

To restore to a point with no read inconsistency, restore to end SCN as your incomplete recovery point. If recovery is done to an SCN after end SCN, there can be read inconsistency in the `NOLOGGING` LOBs.

For SecureFiles, if a read inconsistency is found during media recovery, the database treats the inconsistent blocks as holes and fills `BLOB`s with `0`'s and `CLOB`s with fill characters.

A

LOB Demonstration Files

This appendix describes files distributed with the database that demonstrate how LOBs are used in supported programmatic environments. This appendix contains these topics:

- PL/SQL LOB Demonstration Files
- OCI LOB Demonstration Files
- Java LOB Demonstration Files

PL/SQL LOB Demonstration Files

The following table lists PL/SQL demonstration files. These files are installed in `$ORACLE_HOME/rdbms/demo/lobs/plsql/`. A driver program, `lobdemo.sql`, that calls these files is found in the same directory.

Table A–1 PL/SQL Demonstration Examples

File Name	Description	Usage Information
`fclose_c.sql`	Closing a `BFILE` with `CLOSE`	Closing a BFILE with CLOSE on page 21-22
`fclose_f.sql`	Closing a `BFILE` with `FILECLOSE`	Closing a BFILE with FILECLOSE on page 21-21
`fclosea.sql`	Closing all open `BFILE`s	Closing All Open BFILEs with FILECLOSEALL on page 21-23
`fcompare.sql`	Comparing all or parts of two `BFILE`s	Comparing All or Parts of Two BFILES on page 21-17
`fcopyloc.sql`	Copying a LOB locator for a `BFILE`	Assigning a BFILE Locator on page 21-19
`fdisplay.sql`	Displaying `BFILE` data	Displaying BFILE Data on page 21-14
`fexists.sql`	Checking if a `BFILE` exists	Determining Whether a BFILE Exists on page 21-18
`ffilopen.sql`	Opening a `BFILE` with `FILEOPEN`	Opening a BFILE with FILEOPEN on page 21-12
`ffisopen.sql`	Checking if the `BFILE` is `OPEN` with `FILEISOPEN`	Determining Whether a BFILE Is Open with FILEISOPEN on page 21-14
`fgetdir.sql`	Getting the directory object name and filename of a `BFILE`	Getting Directory Object Name and File Name of a BFILE on page 21-20
`finsert.sql`	Inserting row containing a `BFILE` by initializing a `BFILE` locator	Inserting a Row Containing a BFILE on page 21-24

Table A–1 (Cont.) PL/SQL Demonstration Examples

File Name	Description	Usage Information
fisopen.sql	Checking if the BFILE is open with ISOPEN	Determining Whether a BFILE Is Open Using ISOPEN on page 21-13
flength.sql	Getting the length of a BFILE	Getting the Length of a BFILE on page 21-19
floadlob.sql	Loading a LOB with BFILE data	Loading a LOB with BFILE Data on page 21-10
fopen.sql	Opening a BFILE with OPEN	Opening a BFILE with OPEN on page 21-11
fpattern.sql	Checking if a pattern exists in a BFILE using instr	Checking If a Pattern Exists in a BFILE Using INSTR on page 21-18
fread.sql	Reading data from a BFILE	Reading Data from a BFILE on page 21-15
freadprt.sql	Reading portion of a BFILE data using substr	Reading a Portion of BFILE Data Using SUBSTR on page 21-16
fupdate.sql	Updating a BFILE by initializing a BFILE locator	Updating a BFILE by Initializing a BFILE Locator on page 21-21
lappend.sql	Appending one LOB to another	Appending One LOB to Another on page 22-3
lcompare.sql	Comparing all or part of LOB	Comparing All or Part of Two LOBs on page 22-20
lcopy.sql	Copying all or part of a LOB to another LOB	Copying All or Part of One LOB to Another LOB on page 22-22
lcopyloc.sql	Copying a LOB locator	Copying All or Part of One LOB to Another LOB on page 22-22
ldisplay.sql	Displaying LOB data	Displaying LOB Data on page 22-11
lerase.sql	Erasing part of a LOB	Erasing Part of a LOB on page 22-34
linsert.sql	Inserting a row by initializing LOB locator bind variable	Inserting a Row by Initializing a LOB Locator Bind Variable on page 15-5
linstr.sql	Seeing if pattern exists in LOB (instr)	Patterns: Checking for Patterns in a LOB Using INSTR on page 22-21
lisopen.sql	Seeing if LOB is open	Determining Whether a LOB is Open on page 22-10
listemp.sql	Seeing if LOB is temporary	Determining Whether a LOB instance Is Temporary on page 22-38
lldblobf.sql	Using DBMS_LOB.LOADBLOBFROMFILE to load a BLOB with data from a BFILE	Loading a BLOB with Data from a BFILE on page 22-7
lldclobf.sql	Using DBMS_LOB.LOADCLOBFROMFILE to load a CLOB or NCLOB with data from a BFILE	Loading a CLOB or NCLOB with Data from a BFILE on page 22-8
lldclobs.sql	Using DBMS_LOB.LOADCLOBFROMFILE to load segments of a stream of data from a BFILE into different CLOBs	Loading a CLOB or NCLOB with Data from a BFILE on page 22-8
llength.sql	Getting the length of a LOB	Length: Determining the Length of a LOB on page 22-22
lloaddat.sql	Loading a LOB with BFILE data	Loading a LOB with Data from a BFILE on page 22-5

Table A–1 (Cont.) PL/SQL Demonstration Examples

File Name	Description	Usage Information
lobuse.sql	Examples of LOB API usage.	Creating Persistent and Temporary LOBs in PL/SQL on page 19-3
lread.sql	Reading data from LOB	Reading Data from a LOB on page 22-12
lsubstr.sql	Reading portion of LOB (substr)	Reading a Portion of a LOB (SUBSTR) on page 22-20
ltrim.sql	Trimming LOB data	Trimming LOB Data on page 22-34
lwrite.sql	Writing data to a LOB	Writing Data to a LOB on page 22-26
lwriteap.sql	Writing to the end of LOB (write append)	Appending to a LOB on page 22-25

OCI LOB Demonstration Files

The following table lists OCI demonstration files. These files are installed in $ORACLE_HOME/rdbms/demo/lobs/oci/. A driver program, lobdemo.c, that calls these files is found in the same directory, as is the header file lobdemo.h.

Table A–2 OCI Demonstration Examples

File Name	Description	Usage Information
fclose_c.c	Closing a BFILE with CLOSE	Closing a BFILE with CLOSE on page 21-22
fclose_f.c	Closing a BFILE with FILECLOSE	Closing a BFILE with FILECLOSE on page 21-21
fclosea.c	Closing all open BFILEs	Closing All Open BFILEs with FILECLOSEALL on page 21-23
fcopyloc.c	Copying a LOB locator for a BFILE	Assigning a BFILE Locator on page 21-19
fdisplay.c	Displaying BFILE data	Displaying BFILE Data on page 21-14
fexists.c	Checking if a BFILE exists	Determining Whether a BFILE Exists on page 21-18
ffilopen.c	Opening a BFILE with FILEOPEN	Opening a BFILE with FILEOPEN on page 21-12
ffisopen.c	Checking if the BFILE is OPEN with FILEISOPEN	Determining Whether a BFILE Is Open with FILEISOPEN on page 21-14
fgetdir.c	Getting the directory object name and filename of a BFILE	Getting Directory Object Name and File Name of a BFILE on page 21-20
finsert.c	Inserting row containing a BFILE by initializing a BFILE locator	Inserting a Row Containing a BFILE on page 21-24
fisopen.c	Checking if the BFILE is open with ISOPEN	Determining Whether a BFILE Is Open Using ISOPEN on page 21-13
flength.c	Getting the length of a BFILE	Getting the Length of a BFILE on page 21-19
floadlob.c	Loading a LOB with BFILE data	Loading a LOB with BFILE Data on page 21-10
fopen.c	Opening a BFILE with OPEN	Opening a BFILE with OPEN on page 21-11
fread.c	Reading data from a BFILE	Reading Data from a BFILE on page 21-15

Table A–2 (Cont.) OCI Demonstration Examples

File Name	Description	Usage Information
fupdate.c	Updating a BFILE by initializing a BFILE locator	Updating a BFILE by Initializing a BFILE Locator on page 21-21
lappend.c	Appending one LOB to another	Appending One LOB to Another on page 22-3
lcopy.c	Copying all or part of a LOB to another LOB	Copying All or Part of One LOB to Another LOB on page 22-22
lcopyloc.c	Copying a LOB locator	Copying All or Part of One LOB to Another LOB on page 22-22
ldisbuf.c	Disabling LOB buffering (persistent LOBs)	Disabling LOB Buffering on page 22-37
ldisplay.c	Displaying LOB data	Displaying LOB Data on page 22-11
lequal.c	Seeing if one LOB locator is equal to another	Equality: Checking If One LOB Locator Is Equal to Another on page 22-24
lerase.c	Erasing part of a LOB	Erasing Part of a LOB on page 22-34
lgetchar.c	Getting character set id	Determining Character Set ID on page 22-5
lgetchfm.c	Getting character set form of the foreign language ad text, ad_fltextn	Determining Character Set Form on page 22-4
linit.c	Seeing if a LOB locator is initialized	Determining Whether LOB Locator Is Initialized on page 22-24
linsert.c	Inserting a row by initializing LOB locator bind variable	Inserting a Row by Initializing a LOB Locator Bind Variable on page 15-5
lisopen.c	Seeing if LOB is open	Determining Whether a LOB is Open on page 22-10
listemp.c	Seeing if LOB is temporary	Determining Whether a LOB instance Is Temporary on page 22-38
llength.c	Getting the length of a LOB	Length: Determining the Length of a LOB on page 22-22
lloaddat.c	Loading a LOB with BFILE data	Loading a LOB with Data from a BFILE on page 22-5
lread.c	Reading data from LOB	Reading Data from a LOB on page 22-12
lreadarr.c	Reading data from an array of LOB locators	LOB Array Read on page 22-13
ltrim.c	Trimming LOB data	Trimming LOB Data on page 22-34
lwrite.c	Writing data to a LOB	Writing Data to a LOB on page 22-26
lwritearr.c	Writing data into an array of LOB locators	LOB Array Write on page 22-28
lwriteap.c	Writing to the end of LOB (write append)	Appending to a LOB on page 22-25

Java LOB Demonstration Files

The following table lists Java demonstration files. These files are installed in $ORACLE_HOME/rdbms/demo/lobs/java/.

Table A–3 Java Demonstration Examples

File Name	Description	Usage Information
`Readme.txt`	-	See *Oracle Database JDBC Developer's Guide* for information on setting up your system to be able to compile and run JDBC programs with the Oracle Driver
LobDemoConnectionFactory.java	-	As written `LobDemoConnectionFactory` uses the JDBC OCI driver with a local connection. You should edit the URL `"jdbc:oracle:oci8:@"` to match your setup. Again see *Oracle Database JDBC Developer's Guide*.
`fclose_c.java`	Closing a `BFILE` with `CLOSE`	Closing a BFILE with CLOSE on page 21-22
`fclose_f.java`	Closing a `BFILE` with `FILECLOSE`	Closing a BFILE with FILECLOSE on page 21-21
`fclosea.java`	Closing all open `BFILE`s	Closing All Open BFILEs with FILECLOSEALL on page 21-23
`fcompare.java`	Comparing all or parts of two `BFILE`s	Comparing All or Parts of Two BFILES on page 21-17
`fexists.java`	Checking if a `BFILE` exists	Determining Whether a BFILE Exists on page 21-18
`ffilopen.java`	Opening a `BFILE` with `FILEOPEN`	Opening a BFILE with FILEOPEN on page 21-12
`ffisopen.java`	Checking if the `BFILE` is OPEN with `FILEISOPEN`	Determining Whether a BFILE Is Open with FILEISOPEN on page 21-14
`fgetdir.java`	Getting the directory object name and filename of a `BFILE`	Getting Directory Object Name and File Name of a BFILE on page 21-20
`finsert.java`	Inserting row containing a `BFILE` by initializing a `BFILE` locator	Inserting a Row Containing a BFILE on page 21-24
`fisopen.java`	Checking if the `BFILE` is open with `ISOPEN`	Determining Whether a BFILE Is Open Using ISOPEN on page 21-13
`flength.java`	Getting the length of a `BFILE`	Getting the Length of a BFILE on page 21-19
`fopen.java`	Opening a `BFILE` with `OPEN`	Opening a BFILE with OPEN on page 21-11
`fpattern.java`	Checking if a pattern exists in a `BFILE` using `instr`	Checking If a Pattern Exists in a BFILE Using INSTR on page 21-18
`fread.java`	Reading data from a `BFILE`	Reading Data from a BFILE on page 21-15
`fupdate.java`	Updating a `BFILE` by initializing a `BFILE` locator	Updating a BFILE by Initializing a BFILE Locator on page 21-21
`lappend.java`	Appending one LOB to another	Appending One LOB to Another on page 22-3
`lcompare.java`	Comparing all or part of LOB	Comparing All or Part of Two LOBs on page 22-20
`lcopy.java`	Copying all or part of a LOB to another LOB	Copying All or Part of One LOB to Another LOB on page 22-22

Table A–3 (Cont.) Java Demonstration Examples

File Name	Description	Usage Information
`lerase.java`	Erasing part of a LOB	Erasing Part of a LOB on page 22-34
`linsert.java`	Inserting a row by initializing LOB locator bind variable	Inserting a Row by Initializing a LOB Locator Bind Variable on page 15-5
`linstr.java`	Seeing if pattern exists in LOB (`instr`)	Patterns: Checking for Patterns in a LOB Using INSTR on page 22-21
`lisopen.java`	Seeing if LOB is open	Determining Whether a LOB is Open on page 22-10
`listempb.java`	Seeing if LOB is temporary	Determining Whether a LOB instance Is Temporary on page 22-38
`listempc.java`	Seeing if LOB is temporary	Determining Whether a LOB instance Is Temporary on page 22-38
`llength.java`	Getting the length of a LOB	Length: Determining the Length of a LOB on page 22-22
`lloaddat.java`	Loading a LOB with BFILE data	Loading a LOB with Data from a BFILE on page 22-5
`lread.java`	Reading data from LOB	Reading Data from a LOB on page 22-12
`lsubstr.java`	Reading portion of LOB (`substr`)	Reading a Portion of a LOB (SUBSTR) on page 22-20
`ltrim.java`	Trimming LOB data	Trimming LOB Data on page 22-34
`lwrite.java`	Writing data to a LOB	Writing Data to a LOB on page 22-26
`lwriteap.java`	Writing to the end of LOB (write append)	Appending to a LOB on page 22-25

Glossary

BFILE

A Large Object datatype that is a binary file residing in the file system, outside of the database data files and tablespace. Note that the `BFILE` datatype is also referred to as an *external LOB* in some documentation.

Binary Large Object (BLOB)

A Large Object datatype that has content consisting of binary data and is typically used to hold unstructured data. The `BLOB` datatype is included in the category **Persistent LOBs** because it resides in the database.

BLOB

See Binary Large Object (BLOB).

Character Large Object (CLOB)

The LOB data type that has content consisting of character data in the database character set. A CLOB can be indexed and searched by the Oracle Text search engine.

CLOB

See Character Large Object (CLOB).

deduplication

Deduplication enables Oracle Database to automatically detect duplicate LOB data and conserve space by only storing one copy (if storage parameter is `SECUREFILE`).

DBFS

The Database Filesystem, which is visible to end-users as the client-side API (`dbms_dbfs_content`).

DBFS Link

Database File System Links (DBFS Links) are references from SecureFiles LOBs to data stored outside the segment where the SecureFiles LOB resides.

external LOB

A Large Object datatype that is stored outside of the database tablespace. The `BFILE` datatype is the only external LOB datatype. See also `BFILE`.

internal persistent LOB

A large object (LOB) that is stored in the database in a `BLOB/CLOB/NCLOB` column.

introspect

To examine attributes or value of an object.

Large Objects (LOBs)

Large Objects include the following SQL datatypes: `BLOB`, `CLOB`, `NCLOB`, and `BFILE`. These datatypes are designed for storing data that is large in size. See also `BFILE`, Binary Large Object, Character Large Object, and National Character Large Object.

LOB

See Large Objects (LOBs)

LOB attribute

A large object datatype that is a field of an object datatype. For example a `CLOB` field of an object type.

LOB value

The actual data stored by the Large Object. For example, if a `BLOB` stores a picture, then the value of the `BLOB` is the data that makes up the image.

mount point

The path where the Database File System is mounted. Note that all file systems owned by the database user are seen at the mount point.

National Character Large Object

The LOB data type that has content consisting of Unicode character data in the database national character set. An `NCLOB` can be indexed and searched by the Oracle Text search engine.

NCLOB

See National Character Large Object.

persistent LOB

A `BLOB`, `CLOB`, or `NCLOB` that is stored in the database. A persistent LOB instance can be selected out of a table and used within the scope of your application. The ACID (atomic, consistent, isolated, durable) properties of the instance are maintained just as for any other column type. Persistent LOBs are sometimes also referred to as *internal persistent LOBs* or just, *internal LOBs*.

A persistent LOB can exist as a field of an object data type and an instance in a LOB-type column. For example a `CLOB` attribute of an instance of type `object`.

See also *temporary LOB* and *external LOB*.

SECUREFILE

LOB storage parameter that allows deduplication, encryption, and compression. The opposite parameter, that does not allow these features, is `BASICFILE`.

SPI

The DBFS Store Provider Interface, visible to end-users as the server-side SPI (`dbms_dbfs_content_spi`).

Store

A unified content repository, visible to the DBFS, and managed by a single store provider. The store itself may be a single relational table, a collection of tables, or even

a collection of relational and non-relational entities (e.g., hierarchical stores like tapes and the cloud, elements inside an XML file, components of HDF-style documents, and so on.

Store Provider

An entity, embodied as a P L/SQL package, that implements the DBFS SPI.

tablespace

A database storage unit that groups related logical structures together.

temporary LOB

A `BLOB`, `CLOB`, or `NCLOB` that is accessible and persists only within the application scope in which it is declared. A temporary LOB does not exist in database tables.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z