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 2 (12.2).

Prerequisites for Using LOBs

Oracle Database includes all necessary resources for using LOBs in an application; however, there are some restrictions as described in the "LOB Rules and Restrictions" section.

Documentation Accessibility

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

Access to Oracle Support

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

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 Developer's Guide*
- *Oracle C++ Call Interface Developer's Guide*
- *Pro*C/C++ Developer's Guide*
- *Pro*COBOL Developer's Guide*
- *Oracle Database Developer'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*

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.
`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 Oracle Database

The following are the changes in *SecureFiles and Large Objects Developer's Guide* for Oracle Database 23ai.

- New Features in Release 23ai
- Deprecated Features in Release 23ai

New Features in Release 23ai

The following are the new features in the *SecureFiles and Large Objects Developer's Guide* for Oracle Database Release 23ai.

- Automatic SecureFiles Shrink
  Automatic SecureFiles Shrink automatically selects suitable SecureFiles LOB segments and shrinks the selected segment in the background.
- Distributed and Sharded Environments Support Additional Types of LOBs
  Earlier, persistent LOBs and temporary LOBs were supported in distributed and sharded environments.
- Estimate the Space Saved with Deduplication
  Before you enable deduplication, use the `GET_LOB_DEDUPLICATION_RATIO` function to estimate the space that you can save by enabling this feature for existing LOBs.
- Improved Performance of LOB Writes
  You can experience improved LOB read and write performance.
- Maximum Size of Inline LOBs is 8000
  Actual LOB values are stored either in the table row (inline) or outside of the table row (out-of-line).
- Migrate BasicFile LOBs Using the SecureFiles Migration Utility
  Use the SecureFiles Migration Utility to simplify the migration and compression of BasicFiles LOB segments to SecureFiles LOB segments.
- Rename LOB Segments
  From release 23ai onwards, you can use the `ALTER TABLE RENAME LOB` statement to rename LOB segments, and partitions, as well as subpartitions, in partitioned tables.
- Tune Performance for Parallel File System Operations
  Tune performance in environments that contain many PDBs and require multiple `DBMS_FS` requests to be processed in parallel.
- Value LOBs Optimize Reading LOB Values in a SQL Query
  Value LOBs, are a subset of Temporary LOBs, which are valid for a SQL fetch duration.

Automatic SecureFiles Shrink

Automatic SecureFiles Shrink automatically selects suitable SecureFiles LOB segments and shrinks the selected segment in the background.

With Automatic SecureFiles Shrink, the shrink operation happens transparently in small and gradual steps over time while allowing DDL and DML statements to execute concurrently. In the manual method, you must decide on which LOB segments to shrink using tools like

Segment Advisor, and use a DDL statement to execute the shrink operation. The manual method may not be feasible for very large LOB segments because it is time-consuming.

Automatic SecureFiles Shrink simplifies administrator duties and saves time due to the automation of this process. See Automatic SecureFiles Shrink.

Distributed and Sharded Environments Support Additional Types of LOBs

Earlier, persistent LOBs and temporary LOBs were supported in distributed and sharded environments.

You can now work with inline LOBs, value LOBs, and all temporary LOBs in distributed and sharded environments. Avail good performance, scalability, and garbage collection when you work with temporary LOBs. See Distributed LOBs.

Estimate the Space Saved with Deduplication

Before you enable deduplication, use the `GET_LOB_DEDUPLICATION_RATIO` function to estimate the space that you can save by enabling this feature for existing LOBs.

This enables you to take an informed decision to enable deduplication. See ALTER TABLE with Advanced LOB Deduplication.

Improved Performance of LOB Writes

You can experience improved LOB read and write performance.

The following enhancements improve the LOB read and write performance:

- Multiple LOBs in a single transaction are buffered simultaneously. This improves performance when you use switch between LOBs while writing within a single transaction.
- Various enhancements, such as acceleration of compressed LOB append and compression unit caching, improve the performance of reads and writes to compressed LOBs.
- The input-output buffer is resized based on the input data for large writes to LOBs with the NOCACHE option. This improves the performance for large direct writes, such as writes to file systems on DBFS and OFS.

Maximum Size of Inline LOBs is 8000

Actual LOB values are stored either in the table row (inline) or outside of the table row (out-of-line).

Now, the maximum size of the inline LOB is 8000. Earlier, the maximum size was 4000. This provides better input-output performance while processing LOB columns. You can experience the improved performance while running operations, such as full table scans, range scans, and DML. See Summary of CREATE TABLE LOB Storage Parameters for Securefile LOBs.

Migrate BasicFile LOBs Using the SecureFiles Migration Utility

Use the SecureFiles Migration Utility to simplify the migration and compression of BasicFiles LOB segments to SecureFiles LOB segments.

Earlier it was challenging to decide which BasicFile LOBs to migrate to SecureFile LOBs, and whether or not to compress the LOBs, especially considering that organizations often have

many databases, with a large numbers of schemas, tables, and segments. SecureFiles Migration Utility automates several steps that were earlier performed manually. It also generates several reports that help you decide which BasicFile LOBs you want to migrate and compress. This is the recommended method to migrate BasicFile LOB data to SecureFile storage. See Migrating LOBs with SecureFiles Migration Utility.

Rename LOB Segments

From release 23ai onwards, you can use the `ALTER TABLE RENAME LOB` statement to rename LOB segments, and partitions, as well as subpartitions, in partitioned tables.

Example 1 Command Syntax

```
ALTER TABLE RENAME LOB (<column_name>) <oldsegment_name> to <newsegment_name>;


ALTER TABLE RENAME LOB (<column_name>) partition <oldsegment_name> to
newsegment_name;


ALTER TABLE RENAME LOB (<column_name>) subpartition <oldsegment_name> to
newsegment_name;
```

Ensure that new name that you provide for the segment is unique within the database. In case of any conflicts, the database returns `ORA-64223` or `ORA-64223`. To check the names of the existing segments, you can query LOB views, such as `all_lobs`, `all_lobs_partition`, or `all_lob_subpartitions`.

See ALTER TABLE BNF.

Tune Performance for Parallel File System Operations

Tune performance in environments that contain many PDBs and require multiple `DBMS_FS` requests to be processed in parallel.

You can update the number `OFS_THREADS` to increase the number of `DBMS_FS` requests that are executed in parallel. This increases the number of worker threads executing the make, mount, unmount, and destroy operations on Oracle file systems in the Oracle database.

Increase in the value of `OFS_THREADS`, results in a significant reduction of time taken to execute parallel file system requests in environments that contain multiple PDBs. You can query the `V$OFS_THREADS` view to list all the running OFS threads and to retrieve details about the different OFS threads. See Views for OFS.

Value LOBs Optimize Reading LOB Values in a SQL Query

Value LOBs, are a subset of Temporary LOBs, which are valid for a SQL fetch duration.

Use Value LOBs to optimize reading LOB values in the context of a SQL query. Many applications use LOBs to store medium-sized objects, about a few mega-bytes in size, and just want to read the LOB value in the context of a SQL query.

Value LOBs are autonomous, read-only and more performant. A value LOB, in most instances, provides faster performance than a reference LOB. Oracle highly recommends that you use value LOBs if your application fetches a LOB for read purposes as part of a SQL query and consumes the LOB data before the next fetch is performed on the cursor.

A value LOB gets automatically freed when the next fetch for a cursor is performed. This prevents accumulation of temporary LOBs, which translates to better performance and scalability of the query. In the case of temporary LOBs, it has always been the user's responsibility to free the temporary LOB when their application is done processing it.

A SQL function for JSON can also return a value LOB. See Value LOBs.

Deprecated Features in Release 23ai

The following are the deprecated features in the *SecureFiles and Large Objects Developer's Guide* for Oracle Database Release 23ai.

- Deprecation of the mkstore Wallet Management Tool
- Deprecation of Enterprise User Security (EUS)
- Deprecation of Oracle JDBC Proprietary BLOB/CLOB Open and Close Methods

Deprecation of the mkstore Wallet Management Tool

The `mkstore` wallet management command line tool is deprecated with Oracle Database 23ai, and can be removed in a future release.

Deprecation of Enterprise User Security (EUS)

Enterprise User Security (EUS) is deprecated with Oracle Database 23ai.

Deprecation of Oracle JDBC Proprietary BLOB/CLOB Open and Close Methods

The Oracle JDBC methods `open()`, `close()`, and `isClosed()` in `OracleBlob`, `OracleClob`, and `OracleBfile` are deprecated for removal in Oracle Database 23ai.

Oracle is deprecating these methods, which are replaced with `openLob()`, `closeLob()` and `isClosedLob()`. The method `close()` conflicts with the interface type `java.lang.Autocloseable`. Removing the proprietary method `close()` makes it possible for `java.lang OracleBlob`, `java.lang.OracleClob`, and `java.lang.OracleBfile` to extend the Autocloseable interface at some future time. The `open()` and `isClosed()` methods will be removed and replaced in a future release to maintain rational names for these methods.

1

Introduction to Large Objects and SecureFiles

Large Objects are used to hold large amounts of data inside Oracle Database, SecureFiles provides performance comparable to file system performance, and DBFS provides file system interface to files stored in Oracle Database.

- What Are Large Objects?
  Large Objects (LOBs), SecureFiles LOBs, and Database File System (DBFS) work together with various database features to support application development.
- Where Should We Use LOBs?
  Large objects are suitable for semistructured and unstructured data.
- LOB Classifications
  LOBs store a variety of data such as audio, video, documents, and so on. Based on the type of data stored in the LOB or memory management mechanism used, there are different classifications.
- LOB Locator and LOB Value
  A LOB instance has a locator and a value. A LOB locator is a reference, or a *pointer*, to where the LOB value is physically stored. The LOB value is the data stored in the LOB.
- LOB Restrictions
  You have to keep a few restrictions in mind while working with LOB data.
- How to Navigate This Book
  This section elaborates how to navigate this book using a flow chart that provides information about the relevant chapters you must read for understanding various concepts or performing various tasks.

1.1 What Are Large Objects?

Large Objects (LOBs), SecureFiles LOBs, and Database File System (DBFS) work together with various database features to support application development.

Large Objects

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.

SecureFile LOBs

SecureFile LOBs are LOBs that are created in a tablespace managed with Automatic Segment Space Management (ASSM). SecureFiles is the default storage mechanism for LOBs in database tables. Oracle strongly recommends SecureFiles for storing and managing LOBs.

Database File System (DBFS)

Database File System (DBFS) provides a file system interface to files that are stored in an Oracle Database.

Files stored in an Oracle Database are usually stored as SecureFiles LOBs, and path names, directories, and other file system information is stored in the database tables. SecureFiles

LOBs is the default storage method for DBFS, but BasicFiles LOBs can be used in some situations.

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.

1.2 Where Should We Use LOBs?

Large objects are suitable for semistructured and unstructured data.

Large object features enable you to store the following types of data in the database and also in the operating system files that are accessed from the database.

- 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, 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.

Data unsuited for LOBs

- 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 suited for the object-relational features of the Oracle Database such as collections, references, and user-defined types.

Maximum Size of a LOB

The maximum permissible LOB size for your configuration depends on the block size setting of the tablespace. It is calculated as `(4 gigabytes - 1)*(space usable for data in the LOB block)`. For example, if a LOB is stored in a tablespace of block size 8K, then the approximate maximum LOB size is about 32 terabytes.

1.3 LOB Classifications

LOBs store a variety of data such as audio, video, documents, and so on. Based on the type of data stored in the LOB or memory management mechanism used, there are different classifications.

- Large Object Data Types
  Oracle Database provides a set of large object data types as SQL data types, where the term *LOB* generally refers to the set.
- Types of LOBs
  This section describes the three types of LOB data that Oracle supports.
- LOBs in Object Data Types
  Typically, there is no difference in the use of a LOB instance in a LOB column or in an object data type, as its member.

- Oracle Data Types Stored in LOBs
  Many data types provided with Oracle Database are stored as or created with LOB types.

1.3.1 Large Object Data Types

Oracle 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 related sections, also apply to the corresponding data types provided for other programmatic environments.

The following table describes each large object data type that the database supports and describes the kind of data that uses it.

Table 1-1 Types of Large Object Data

SQL Data Type	Description
`BLOB`	Binary Large Object Stores any kinds of data in binary format. Used for 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.
`BFILE`	External Binary File A binary file stored outside of the database in the host operating system file system, but accessible from database tables. `BFILE`s can be accessed from your application on a **read-only** basis. Use `BFILE`s 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.

1.3.2 Types of LOBs

This section describes the three types of LOB data that Oracle supports.

Persistent LOBs

A persistent LOB is a LOB instance that exists in a table row in the database. Persistent LOBs 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. Persistent LOBs can be of data types `BLOB`, `CLOB` and `NCLOB`.

Temporary LOBs

A temporary LOB instance is created when you instantiate a LOB only within the scope of your local application. Temporary LOBs are transient, just like other local variables in an application. A temporary LOB becomes persistent when you insert it into a table row. Temporary LOBs can be of data types `BLOB`, `CLOB` and `NCLOB`.

A Value LOB is a special kind of read-only temporary LOB with optimizations for better performance and manageability compared to a reference LOB. Many applications use LOBs to store medium-sized objects, about a few mega-bytes in size, and just want to read the LOB value in the context of a SQL query. Oracle recommends that you use Value LOBs for applications which use LOBs as a larger `VARCHAR` or `RAW` data type.

BFILEs

`BFILE`s are data objects stored in operating system files, outside the database tablespaces. Data stored in a table column of type `BFILE` is physically located in an operating system file, not in the database.

`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.

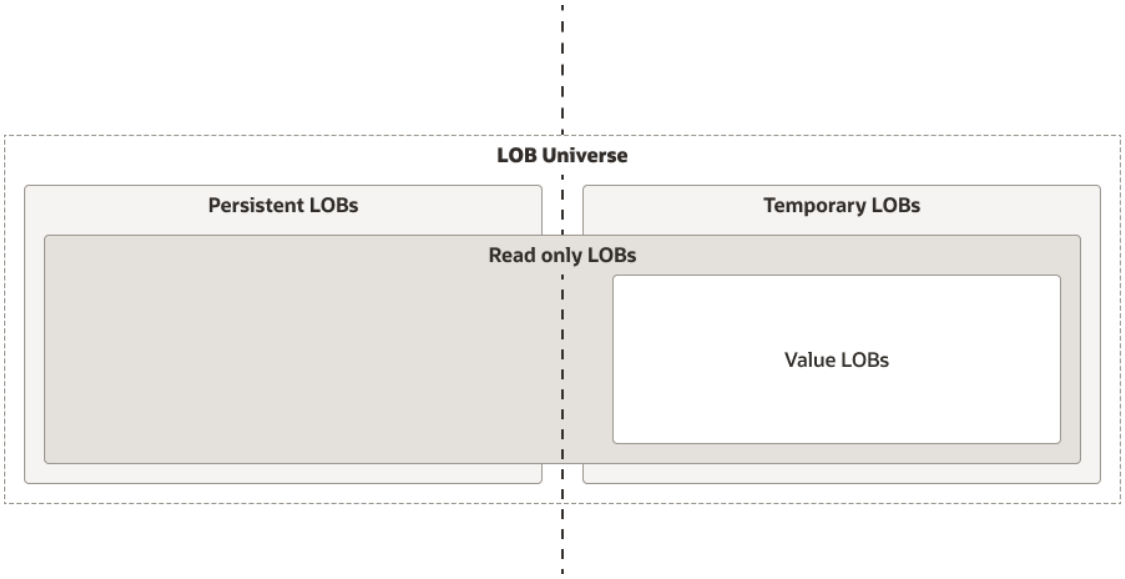
You typically use `BFILE`s to hold:

- 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:**
>
> All the information related to BFILEs is exclusively documented either in BFILEs or in Managing LOBs: Database Administration.

The following picture summarizes the relationship between different kinds of LOBs.



1.3.3 LOBs in Object Data Types

Typically, there is no difference in the use of a LOB instance in a LOB column or in an object data type, as its member.

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.

1.3.4 Oracle Data Types Stored in LOBs

Many data types provided with Oracle Database are stored as or created with LOB types.

The following list mentions a few data types that you can store with LOB types:

- `VARCHAR2` or `RAW` data types of size greater than 4000 bytes
- `JSON` data type
- `XMLType` stored as `BINARY XML` or `CLOB`
- `VARRAY` stored as LOB

1.4 LOB Locator and LOB Value

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

A LOB locator can be assigned to any LOB instance of the same type, such as `BLOB`, `CLOB`, `NCLOB`, or `BFILE`. 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.

There are two different techniques to access and modify LOBs:

- Using LOBs Without Locators
  LOBs can be used in many operations similar to how VARCHAR2 or RAW data types are used. Such LOB operations can be performed without the use of LOB locators.
- Using LOBs with Locators
  You can use the LOB locator to access and modify LOB values by passing the LOB locator to the LOB APIs supplied with the database. These operations support efficient piecewise read and write to LOBs.

1.4.1 Using LOBs Without Locators

LOBs can be used in many operations similar to how VARCHAR2 or RAW data types are used. Such LOB operations can be performed without the use of LOB locators.

LOB operations that are similar to `VARCHAR2` and `RAW` types include:

- SQL and PLSQL built-in functions and implicit assignments

> **See Also:**
>
> – SQL Semantics for LOBs
> – PL/SQL Semantics for LOBs

- Data interface on LOBs that enables you to insert or select entire LOB 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`. For example, in PLSQL you can insert a `VARCHAR2` buffer into a `CLOB` column, and in OCI you can bind a buffer of type `SQLT_CHAR` to a `CLOB` column.
  – Define an output buffer in your application that holds character data selected from a `CLOB` or `RAW` data selected from a `BLOB`. For example, in PLSQL you can select the `CLOB` output of a query into a `VARCHAR2` buffer, and in OCI you can define a `CLOB` query result item to a buffer of type `SQLT_CHAR`.

> **See Also:**
>
> Data Interface for LOBs

1.4.2 Using LOBs with Locators

You can use the LOB locator to access and modify LOB values by passing the LOB locator to the LOB APIs supplied with the database. These operations support efficient piecewise read and write to LOBs.

You should use this mode if your application needs to perform random or piecewise read or write calls to LOBs, which means it needs to specify the offset or amount of the operation to read or write a part of the LOB value.

> **See Also:**
>
> Locator Interface for LOBs

1.5 LOB Restrictions

You have to keep a few restrictions in mind while working with LOB data.

LOB columns are subject to the following rules and restrictions:

- You cannot specify a LOB as a primary key column.
- 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.

- Clusters cannot contain LOBs, either as key or nonkey columns.
- Even though compressed `VARRAY` data types are supported, they are less performant.
- 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
- 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. 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.
- 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 functional or domain index. In addition, Oracle Text lets you define an index on a `CLOB` column.
- In SQL Loader, a field read from a LOB cannot be used as an argument to a clause.
- Case-insensitive searches on `CLOB` columns often do not succeed. If you perform the following 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%';
```

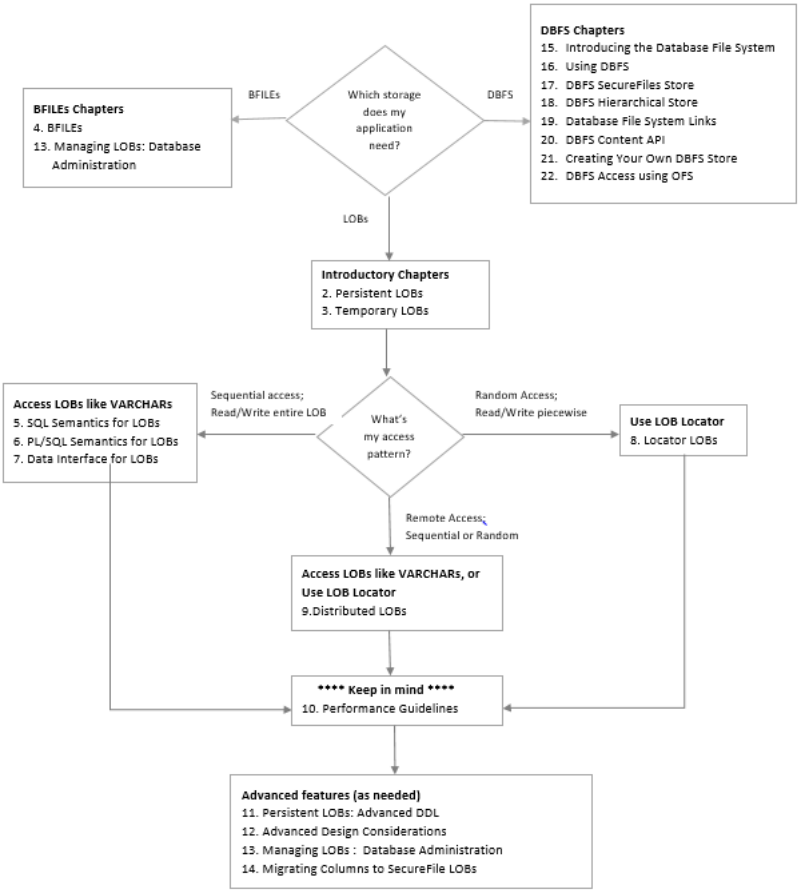
  The select fails without the `LOWER` function. You can perform case-insensitive searches with Oracle Text or the `DBMS_LOB.INSTR()` function.

See Also:

- Restrictions on SQL Operations on LOBs
- Guidelines and Restrictions for Implicit Conversions with LOBs
- Restrictions for Data Interface on Remote LOBs
- Restrictions when using remote LOB locators
- Restrictions on Mounted File Systems
- Restrictions on Types of Files Stored at DBFS Mount Points
- Restrictions on Index Organized Tables with LOB Columns
- Restrictions on Migrating LOBs with Data Pump

1.6 How to Navigate This Book

This section elaborates how to navigate this book using a flow chart that provides information about the relevant chapters you must read for understanding various concepts or performing various tasks.



2
Persistent LOBs

A persistent LOB is a LOB instance that exists in a table row in the database. Persistent LOBs can be stored as SecureFiles or BasicFiles.

The term LOB can represent LOBs of either SecureFiles or BasicFiles type, unless the storage type is explicitly indicated. It can be either by name for both storage types, or by reference to archiving or linking, which only applies to the SecureFiles storage type. Oracle strongly recommends SecureFiles for storing and managing LOBs.

SecureFiles LOB storage is the default in the `CREATE TABLE` statement, if no storage type is explicitly specified. All new LOB columns use SecureFiles LOB storage by default, which is the recommended method for storing and managing LOBs. SecureFiles LOB storage is designed to provide great performance and scalability to meet or exceed the performance of traditional network file system. However, you must use BasicFiles LOB storage for LOB storage in tablespaces that are not managed with Automatic Segment Space Management (ASSM). SecureFiles LOBs can only be created in tablespaces managed with Automatic Segment Space Management (ASSM).

- Creating a Table with LOB Columns
  You can use the `CREATE TABLE` statement or an `ALTER TABLE ADD` column statement to create a new LOB column. This section introduces basic DDL operations on LOBs to get you started quickly.
- Inserting and Updating LOB Values in Tables
  Oracle Database provides various methods to insert and update the data available in LOB columns of database tables.
- Selecting LOB Values from Tables
  You can select a LOB into a Character Buffer, a RAW Buffer, or a LOB variable for performing read and write operations.
- Performing DML and Query Operations on LOBs in Nested Tables
  This section describes the `INSERT`, `UPDATE`, and `SELECT` operations on LOBs in Nested Tables. To update LOBs in a nested table, you must lock the row containing the LOB explicitly.
- Performing Parallel DDL, Parallel DML (PDML), and Parallel Query (PQ) Operations on LOBs
  Oracle supports parallel execution of the following operations when performed on partitioned tables with SecureFiles LOBs or BasicFiles LOBs.
- Sharding with LOBs
  LOBs can be used in a sharded environment. This section discusses the interfaces to support LOBs in sharded tables.

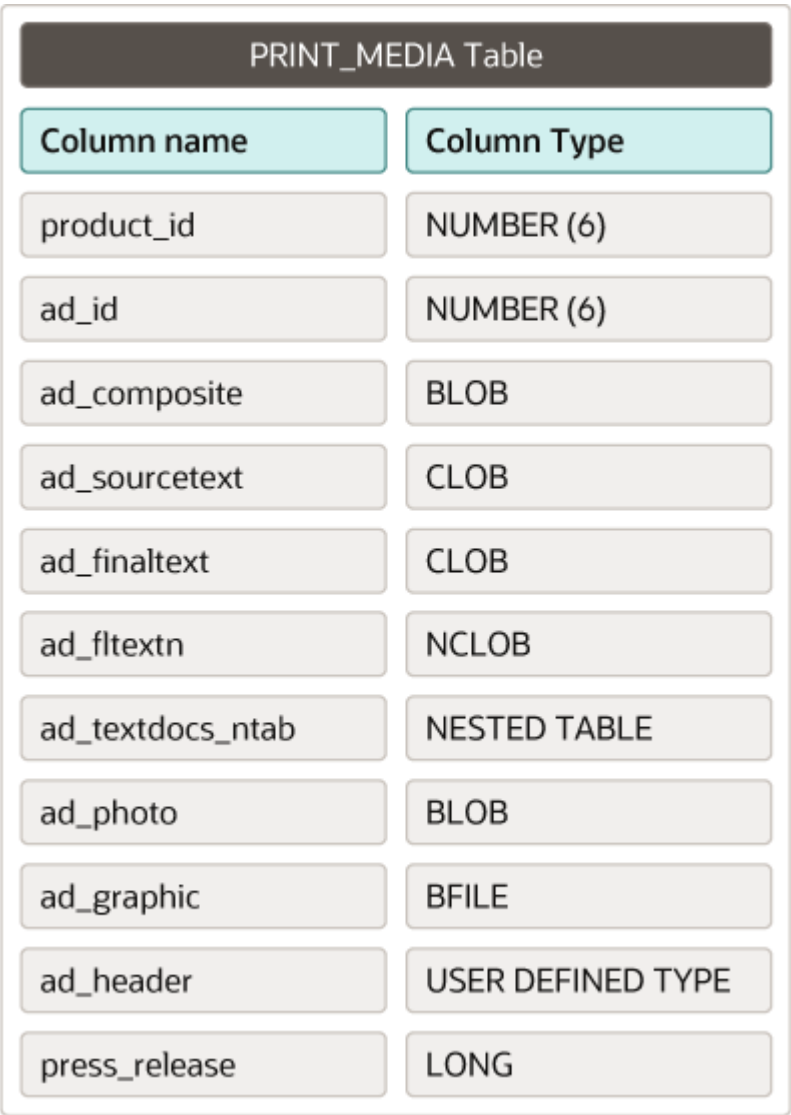
2.1 Creating a Table with LOB Columns

You can use the `CREATE TABLE` statement or an `ALTER TABLE ADD` column statement to create a new LOB column. This section introduces basic DDL operations on LOBs to get you started quickly.

Following is an example of creating a table with columns of various LOB types, including LOBs in Object Types and nested tables:

```
CREATE USER pm identified by password;
GRANT CONNECT, RESOURCE to pm IDENTIFIED BY pm;
CONNECT pm/pm

-- Create an object type with a LOB
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 a nested table type of Object type containing a LOB
CREATE TYPE Textdoc_ntab AS TABLE of textdoc_typ;

-- Create a table of Object type, and specify a default value for LOB column
CREATE TABLE adheader_tab of adheader_typ (
    logo DEFAULT EMPTY_BLOB(),
    CONSTRAINT header_name CHECK (header_name IS NOT NULL),
     header_text DEFAULT NULL);
-- Create a table with columns of different LOB types,
-- and of object type with LOBs, and nested table containing LOB
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_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);
```

Figure 2-1 print_media table



You can also perform advanced DDL operations, like the following, on LOBs:

- Specify LOB storage parameters: You can override the default LOB storage settings by specifying parameters like `SECUREFILE/BASICFILE`, `TABLESPACE` where the LOB data will be stored, `ENABLE/DISABLE STORAGE IN ROW`, `RETENTION`, caching, logging, etc. You can also specify SecureFile specific parameters like `COMPRESSION`, `DEDUPLICATION` and `ENCRYPTION`.
- Alter an existing LOB column: You can use the `ALTER TABLE MODIFY LOB` syntax to change any LOB storage parameters that don't require LOB data movement and the `ALTER TABLE MOVE LOB` syntax to change any LOB storage parameters that require LOB data movement.
- Create indexes on LOB columns: You can build a functional or a domain index on a LOB column. You cannot build a B-tree or bitmap index on a LOB column.
- Partition a table containing LOB columns: All partitioning schemes supported by Oracle are fully supported on LOBs.
- Use LOBs in Index-Organized tables.

> **See Also:**
>
> Persistent LOBs: Advanced DDL

2.2 Inserting and Updating LOB Values in Tables

Oracle Database provides various methods to insert and update the data available in LOB columns of database tables.

- Inserting and Updating with a Buffer
  You can insert a character string directly into a `CLOB` or `NCLOB` column. Similarly, you can insert a raw buffer into a `BLOB` column. This is the most efficient way to insert data into a LOB.
- Inserting and Updating by Selecting a LOB From Another Table
  You can insert into a LOB column of a table by selecting data from a LOB column of the same table or a different table. You can also insert data into a LOB column of a table by selecting a LOB returned by a SQL operator or a PL/SQL function.
- Inserting and Updating with a NULL or Empty LOB
  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.
- Inserting and Updating with a LOB Locator
  If you are using a Programmatic Interface, which has a LOB variable that was previously populated by a persistent or temporary LOB locator, then you can insert a row by initializing the LOB bind variable.

2.2.1 Inserting and Updating with a Buffer

You can insert a character string directly into a `CLOB` or `NCLOB` column. Similarly, you can insert a raw buffer into a `BLOB` column. This is the most efficient way to insert data into a LOB.

The following code snippet inserts a character string into a `CLOB` column:

```
/* Store records in the archive table Online_media: */
INSERT INTO Online_media (product_id, product_text) VALUES (3060, 'some text
about this CRT Monitor');
```

The following code snippet updates the value in a `CLOB` column with character buffer:

```
UPDATE Online_media set product_text = 'some other text' where product_id =
3060;
```

> **See Also:**
>
> Data Interface for LOBs for more information about `INSERT` and `UPDATE` operations

2.2.2 Inserting and Updating by Selecting a LOB From Another Table

You can insert into a LOB column of a table by selecting data from a LOB column of the same table or a different table. You can also insert data into a LOB column of a table by selecting a LOB returned by a SQL operator or a PL/SQL function.

Ensure that you meet the following conditions while selecting data from columns that are part of more than one table:

- The LOB data type is the same for both the columns in the tables
- Implicit conversion is allowed between the two LOB data types used in both the columns

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.

The following code snippet demonstrates inserting a LOB column from by selecting a LOB from another table. The columns `online_media.product_text` and `print_media.ad_sourcetext` are both `CLOB` types.

```
/* 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);

/* Insert values into Print_media by selecting a SQL function returning a
CLOB */
INSERT INTO Print_media (product_id, ad_id, ad_sourcetext)
(SELECT product_id, 11001, substr(product_text, 5) FROM Online_media WHERE
product_id = 3060);

/* 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;

/* Updating a row by selecting a SQL function returning a CLOB */

UPDATE Print_media SET ad_sourcetext = (SELECT substr(product_text, 5) FROM
online_media WHERE product_id = 3060);
WHERE product_id = 3060 AND ad_id = 11001;
```

The following code snippet demonstrates updating a LOB column from by selecting a LOB from another table.

```
/* 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;

/* Updating a row by selecting a SQL function returning a CLOB */
UPDATE Print_media SET ad_sourcetext = (SELECT substr(product_text, 5) FROM
online_media WHERE product_id = 3060)
WHERE product_id = 3060 AND ad_id = 11001;
```

> **See Also:**
>
> - *Oracle Database SQL Language Reference* for more information on `INSERT`
> - Performing Parallel DDL, Parallel DML (PDML), and Parallel Query (PQ) Operations on LOBs for information about how to make the `INSERT AS SELECT` operation run in parallel

2.2.3 Inserting and Updating with a NULL or Empty LOB

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.

Inserting a NULL LOB value

A persistent 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 scalar data types. To INSERT a NULL value into a LOB column, simply use a statement like:

```
INSERT INTO print_media(product_id, ad_id, ad_sourcetext) VALUES (1, 1, NULL);
```

This is useful in situations where you want to use a `SELECT` statement, such as the following, to determine whether or not the LOB holds a NULL value:

```
SELECT COUNT (*) FROM print_media WHERE ad_graphic IS NULL;
```

> **Caution:**
>
> You cannot call `DBMS_LOB` functions or LOB APIs in other Programmatic Interfaces on a NULL LOB, so you must then use a SQL `UPDATE` statement to reset the LOB column to a non-NULL (or empty) value.

Inserting an EMPTY LOB value

Before you can write data to a persistent LOB using an API like `DBMS_LOB.WRITE` or `OCILobWrite2`, the LOB column must be non-`NULL`, that is, it must contain a locator that points to an empty or a populated LOB value.

You can initialize a `BLOB` column value by using the `EMPTY_BLOB()` function as a default predicate. Similarly, a `CLOB` or `NCLOB` column value can be initialized by using the `EMPTY_CLOB()` function. Use the `RETURNING` clause in the `INSERT` and `UPDATE` statement, to minimize the number of round trips while writing the LOB using APIs.

Following PL/SQL block initializes a `CLOB` column with an empty LOB using the `EMPTY_CLOB()` function and also updates the LOB value in a column with an empty `CLOB` using the `EMPTY_CLOB()` function.

```
DECLARE
    c CLOB;
```

```
    amt INTEGER := 11;
    buf VARCHAR(11) := 'Hello there';
BEGIN
  /* Insert empty_clob() */
  INSERT INTO Print_media(product_id, ad_id, ad_sourcetext) VALUES (1, 1,
EMPTY_CLOB()) RETURNING ad_source INTO c;
  /* The following statement updates the persistent LOB directly */
  DBMS_LOB.WRITE(c, amt, 1, buf);

  /* Update column to an empty_clob() */
  UPDATE Print_media SET ad_sourcetext = EMPTY_CLOB() WHERE product_id = 2
AND ad_id = 2 RETURNING ad_source INTO c;
  /* The following statement updates the persistent LOB directly */
  DBMS_LOB.WRITE(c, amt, 1, buf);
END;
/
```

2.2.4 Inserting and Updating with a LOB Locator

If you are using a Programmatic Interface, which has a LOB variable that was previously populated by a persistent or temporary LOB locator, then you can insert a row by initializing the LOB bind variable.

You can populate a LOB variable with a persistent LOB or a temporary LOB by either selecting one out from the database using SQL or by creating a temporary LOB. This section provides information about how to achieve this in various programmatic environments.

- PL/SQL: Inserting a Row by Initializing a LOB Locator Bind Variable
  The following code snippet demonstrates how to insert a row by initializing a LOB locator bind variable using PL/SQL APIs.
- JDBC (Java): Inserting a Row by Initializing a LOB Locator Bind Variable
  The following code snippet demonstrates how to insert a row by initializing a LOB locator bind variable using JDBC APIs:
- OCI (C): Inserting a Row by Initializing a LOB Locator Bind Variable
  The following code snippet demonstrates how to insert a row by initializing a LOB locator bind variable using OCI APIs:
- Pro*C/C++ (C/C++): Inserting a Row by Initializing a LOB Locator Bind Variable
  The following code snippet demonstrates how to insert a row by initializing a LOB locator bind variable using Pro*C/C++ APIs:
- Pro*COBOL (COBOL): Inserting a Row by Initializing a LOB Locator Bind Variable
  The following code snippet demonstrates how to insert a row by initializing a LOB locator bind variable using Pro*COBOL APIs:

2.2.4.1 PL/SQL: Inserting a Row by Initializing a LOB Locator Bind Variable

The following code snippet demonstrates how to insert a row by initializing a LOB locator bind variable using PL/SQL APIs.

```
/* 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;
/
```

2.2.4.2 JDBC (Java): Inserting a Row by Initializing a LOB Locator Bind Variable

The following code snippet demonstrates how to insert a row by initializing a LOB locator bind variable using JDBC APIs:

```
// 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();
    }
  }
}
```

2.2.4.3 OCI (C): Inserting a Row by Initializing a LOB Locator Bind Variable

The following code snippet demonstrates how to insert a row by initializing a LOB locator bind variable using OCI APIs:

```
/* 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));
}
```

2.2.4.4 Pro*C/C++ (C/C++): Inserting a Row by Initializing a LOB Locator Bind Variable

The following code snippet demonstrates how to insert a row by initializing a LOB locator bind variable using Pro*C/C++ APIs:

```
#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;
}
```

2.2.4.5 Pro*COBOL (COBOL): Inserting a Row by Initializing a LOB Locator Bind Variable

The following code snippet demonstrates how to insert a row by initializing a LOB locator bind variable using Pro*COBOL APIs:

You can insert a row by initializing a LOB locator bind variable in COBOL (Pro*COBOL).

```
   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.
```

2.3 Selecting LOB Values from Tables

You can select a LOB into a Character Buffer, a RAW Buffer, or a LOB variable for performing read and write operations.

- Selecting a LOB into a Character Buffer or a Raw Buffer
  You can directly select a CLOB or NCLOB value into a character buffer or a BLOB value. This is called the Data Interface, and is the most efficient way for selecting from a LOB column.
- Selecting a LOB into a LOB Variable for Read Operations
  You can select a persistent or temporary LOB into a LOB variable, and then use APIs to perform various read operations on it.
- Selecting a LOB into a LOB Variable for Write Operations
  To perform a write operation using a LOB locator, you must lock the row in the table in order to prevent other database users from writing to the LOB during a transaction.

2.3.1 Selecting a LOB into a Character Buffer or a Raw Buffer

You can directly select a CLOB or NCLOB value into a character buffer or a BLOB value. This is called the Data Interface, and is the most efficient way for selecting from a LOB column.

> **See Also:**
>
> - Data Interface for LOBs
> - PL/SQL Semantics for LOBs

2.3.2 Selecting a LOB into a LOB Variable for Read Operations

You can select a persistent or temporary LOB into a LOB variable, and then use APIs to perform various read operations on it.

Following code selects a LOB Locator into a variable:

```
DECLARE
    perslob CLOB;
    templob CLOB;
    amt INTEGER := 11;
    buf VARCHAR(100);
BEGIN
    SELECT ad_source, substr(ad_source, 3) INTO perslob, templob FROM
Print_media WHERE product_id = 1 AND ad_id = 1;
    DBMS_LOB.READ(perslob, amt, buf);
    DBMS_LOB.READ(templob, amt, buf);
END;
/
```

> **See Also:**
>
> - A Selected Locator Becomes a Read-Consistent Locator
> - LOB Locators and Transaction Boundaries

2.3.3 Selecting a LOB into a LOB Variable for Write Operations

To perform a write operation using a LOB locator, you must lock the row in the table in order to prevent other database users from writing to the LOB during a transaction.

You can use one of the following mechanisms for this operation:

- Performing an `INSERT` or an `UPDATE` operation with a `RETURNING` clause.

> **See Also:**
>
> Inserting and Updating with a NULL or Empty LOB

- Performing a `SELECT` for an `UPDATE` operation. The following code snippet shows how to select a LOB value to perform a write operation using `UPDATE`.

```
DECLARE
    c CLOB;
    amt INTEGER := 9;
    buf VARCHAR(100) := 'New Value';
BEGIN
    SELECT ad_sourcetext INTO c FROM Print_media WHERE product_id = 1 AND
ad_id = 1 FOR UPDATE;
    DBMS_LOB.WRITE(c, amt, 1, buf);
END;
/
```

- Using an OCI `pin` or `lock` function in OCI programs.

2.4 Performing DML and Query Operations on LOBs in Nested Tables

This section describes the `INSERT`, `UPDATE`, and `SELECT` operations on LOBs in Nested Tables. To update LOBs in a nested table, you must lock the row containing the LOB explicitly.

To lock the row containing the LOB, you must specify the `FOR UPDATE` clause in the subquery prior to updating the LOB value. The following example shows how to perform DML and query operations on LOBs in nested tables.

> **Note:**
>
> Locking the row of a parent table does not lock the row of a nested table containing LOB columns.

Example 2-1 Performing DML and Query Operations on LOBs in Nested Tables

```
CONNECT pm/pm;


-------------------------------------------------------------------------
---------- Inserting LOBs in Nested Tables ------------------------------
-------------------------------------------------------------------------

-- INSERT a row into the NT column of print_media with actual data for lob
INSERT INTO print_media (product_id, ad_id, ad_textdocs_ntab)
VALUES
(1, 1, textdoc_tab(textdoc_typ('txt', to_blob('BABABABABABA')),
                   textdoc_typ('pdf', to_blob('AAAAAAAAAAAA'))));

-- INSERT a row into the NT column of print_media with empty_lob for the lob
INSERT INTO print_media (product_id, ad_id, ad_textdocs_ntab)
VALUES
(2, 2, textdoc_tab(textdoc_typ('txt', empty_blob()),
                   textdoc_typ('pdf', empty_blob())));
```

```
SET SERVEROUTPUT ON

--------------------------------------------------------------------------
---------- Read/Write LOBs in Nested Tables using locators --------------
--------------------------------------------------------------------------

-- INSERT-RETURNING, then write to the LOBs
DECLARE
  txt textdoc_tab;
BEGIN
  INSERT INTO print_media p(product_id, ad_id, ad_textdocs_ntab) VALUES
    (3, 3, textdoc_tab(textdoc_typ('txt', empty_blob()),
                       textdoc_typ('pdf', empty_blob())))
  RETURNING p.ad_textdocs_ntab into txt;

  for elem in 1 .. txt.count loop
    DBMS_LOB.WRITEAPPEND(txt(elem).formatted_doc, 2, hextoraw(elem||'FF'));
  end loop;
END;
/

SELECT ad_textdocs_ntab FROM print_media WHERE product_id = 3;

-- SELECT on NT lob, then read
DECLARE
  txt textdoc_tab;
  pos INTEGER;
  amt INTEGER;
  buf RAW(40);
BEGIN
  SELECT ad_textdocs_ntab INTO txt FROM print_media WHERE product_id = 1;

  for elem in 1 .. txt.count loop
    amt := 40;
    pos := 1;
    DBMS_LOB.READ(txt(elem).formatted_doc, amt, pos, buf);
    DBMS_OUTPUT.PUT_LINE(buf);
  end loop;
END;
/

-- SELECT for update on the NT lob, then write
DECLARE
  txt textdoc_tab;
  pos INTEGER;
  amt INTEGER;
  buf RAW(40);
BEGIN
  SELECT ad_textdocs_ntab INTO txt FROM print_media
  WHERE product_id = 1 FOR UPDATE;

  for elem in 1 .. txt.count loop
    DBMS_LOB.WRITEAPPEND(txt(elem).formatted_doc, 2, hextoraw(elem||'FF'));
  end loop;
END;
/
```

```
SELECT ad_textdocs_ntab FROM print_media WHERE product_id = 1;
```

2.5 Performing Parallel DDL, Parallel DML (PDML), and Parallel Query (PQ) Operations on LOBs

Oracle supports parallel execution of the following operations when performed on partitioned tables with SecureFiles LOBs or BasicFiles LOBs.

- `CREATE TABLE AS SELECT`
- `INSERT AS SELECT`
- Multitable `INSERT`
- `SELECT`
- `DELETE`
- `UPDATE`
- `MERGE` (conditional `UPDATE` and `INSERT`)
- `ALTER TABLE MOVE`
- SQL Loader
- Import/Export

Additionally, Oracle supports parallel execution of the following operations when performed on non-partitioned tables with only SecureFile LOBs:

- `CREATE TABLE AS SELECT`
- `INSERT AS SELECT`
- Multitable `INSERT`
- `SELECT`
- `DELETE`
- `UPDATE`
- `MERGE` (conditional `UPDATE` and `INSERT`)
- `ALTER TABLE MOVE`
- SQL Loader

Restrictions on parallel operations with LOBs

- Parallel insert direct load (PIDL) is disabled if a table also has a BasicFiles LOB column, in addition to a SecureFiles LOB column.
- PDML is disabled if LOB column is part of a constraint.
- PDML does not work when there are any domain indexes defined on the LOB column.
- Parallelism must be specified only for top-level non-partitioned tables.
- Use the `ALTER TABLE MOVE` statement with LOB storage clause, to change the storage properties of LOB columns instead of the `ALTER TABLE MODIFY` statement. The `ALTER TABLE MOVE` statement is more efficient because it executes in parallel and does not generate undo logs.

> **See Also:**
>
> *Oracle Database Administrator's Guide* section "Managing Processes for Parallel SQL Execution"
>
> *Oracle Database SQL Language Reference* section "ALTER TABLE"

2.6 Sharding with LOBs

LOBs can be used in a sharded environment. This section discusses the interfaces to support LOBs in sharded tables.

The following interfaces are supported:

- Query and DML statements
  - Cross shard queries involving LOBs are supported.
  - DML statements involving more than one shard are not supported. This behavior is similar to scalar columns.
  - DML statements involving a single shard are supported from coordinator.
  - Locator selected from a shard can be passed as bind value to the same shard.
- `OCILob`
  All non-BFILE related OCILob APIs in a sharding environment are supported, with some restrictions.

  On the coordinator, the `OCI_ATTR_LOB_REMOTE` attribute of a LOB descriptor returns `TRUE` if the LOB was obtained from a sharded table.

  Restrictions: For APIs that take two locators as input, `OCILobAppend`, `OCILobCompare` for example, both of the locators should be obtained from the same shard. If locators are from different shards an error is given.
- `DBMS_LOB`

  All non-BFILE related DBMS_LOB APIs in a sharding environment are supported, with some restrictions. On the coordinator, `DBMS_LOB.isremote` returns `TRUE` if the LOB was obtained from a sharded table.

  Restrictions: For APIs that take two locators as input, `DBMS_LOB.append` and `DBMS_LOB.compare` for example, both of the locators should be obtained from the same shard. If the locators are from different shards an error given.

> **See Also:**
>
> Sharded Tables

3
Temporary LOBs

Temporary LOBs are transient, just like other local variables in an application. This chapter discusses operations that are specific to temporary LOBs.

- Before You Begin
  Ensure that you go through the topics in this section before you start working with temporary LOBs.
- Temporary LOB APIs in Different Programmatic Interfaces
  This section lists the temporary LOB specific APIs in different Programmatic Interfaces.
- Transforming LOBs
  Value LOBs are a subset of read-only temporary LOBs, which are a subset of temporary LOBs.

3.3 Transforming LOBs

Value LOBs are a subset of read-only temporary LOBs, which are a subset of temporary LOBs.

The following table summarizes how you can transform different kinds of LOBs. The column on the left is the source LOB which you want to transform. The column headings are the target LOBs, the final state of the LOB after the transformation. For example, to transform read-only temporary LOBs to value LOBs, you can use `LOB_VALUE(lob_producing_plsql_function(...))`.

Source LOB	Value LOBs	Read-only Temporary LOBs	Temporary LOBs	Persistent LOBs
Value LOBs	Not applicable	• Pass from SQL to PLSQL. If passed from PLSQL out to JDBC, OCI etc, it stays a read-only temporary LOB. • Send to older JDBC, OCI etc clients	Not directly possible	Not directly possible
Read-only Temporary LOBs	`LOB_VALUE(lob_producing_plsql_function(...))`	Not applicable	Not directly possible	Not directly possible
Temporary LOBs	`LOB_VALUE(`*`temporary_lob`*`)`	Open in READ mode	Not applicable	Not directly possible
Persistent LOBs	• SELECT from column declared as QUERY AS VALUE • LOB_VALUE(persistent_lob)	Not directly possible	Use SQL operators, such as `to_clob()` or `substr()`	Not applicable

The following example shows how you can transform temporary LOBs and read-only temporary LOBs to value LOBs. Also, how you can transform a value LOB to a read-only temporary LOB.

```
DROP TABLE t;

CREATE TABLE t (c clob) lob(c) query as value;
INSERT INTO t VALUES ('I am a CLOB');

CREATE OR REPLACE FUNCTION Vbl2rdo (c clob) RETURN clob IS
BEGIN
    RETURN c;
END;

/
-- Transform value LOB to read-only temporary LOB
var tc clob;
BEGIN
    SELECT c INTO :tc FROM t;
END;
/

print :tc
SELECT Vbl2rdo(c) FROM t;

-- Transform read-only temporary LOB to value LOB
SELECT lob_value(:tc) FROM dual;

-- Transform temporary LOB to value LOB
SELECT lob_value(to_clob('I am a temporary LOB')) FROM dual;
```

The following example shows how you can transform a persistent LOB to a temporary LOB and a value LOB.

```
DROP TABLE t2;

CREATE TABLE t2 (c CLOB);

INSERT INTO t2 VALUES ('I am a CLOB');

-- Transform persistent LOB to value LOB
SELECT Lob_value(c) FROM t2;

-- Transform persistent LOB to temporary LOB
SELECT To_clob(c) FROM t2;
```

3.1 Before You Begin

Ensure that you go through the topics in this section before you start working with temporary LOBs.

- Creating Temporary LOBs
  This section describes how a temporary LOB gets created or generated in a client program.

- Handling Temporary LOBs on the Client Side
  You must consider the aspects discussed in this section while handling the temporary LOBs that are generated by the client programs.

3.1.1 Creating Temporary LOBs

This section describes how a temporary LOB gets created or generated in a client program.

You can create temporary LOB instances in one of the following ways:

- Declare a variable of the given LOB data type and pass it to the temporary LOB creation API. For example, in PL/SQL it is `DBMS_LOB.CREATETEMPORARY`, and in OCI it is `OCILobCreateTemporary()`.
- Invoke a SQL or PL/SQL built-in function that produces a temporary LOB, for example, the `SUBSTR` function.
- Invoke a PL/SQL stored procedure or function that returns a temporary LOB as an `OUT` bind variable or a return value.

The temporary LOB instance exists in your application until it goes out of scope, your session terminates, or you explicitly free the instance.

Temporary LOBs reside in either the PGA memory or the temporary tablespace, depending on their size. Ensure that the PGA memory and the temporary tablespace have space that is large enough for the temporary LOBs used by your application.

> **Note:**
>
> - Oracle highly recommends that you release the temporary LOB instances to free the system resources. Failure to do so may cause accumulation of temporary LOBs and can considerably slow down your system.
> - Starting with Oracle Database Release 21c, you do not need to check whether a LOB is temporary or persistent before releasing the temporary LOB. If you call the `DBMS_LOB.FREETEMPORARY` procedure or the `OCILobFreeTemporary()` function on a LOB, it will perform either of the following operations:
>   - For a temporary LOB, it will release the LOB.
>   - For a persistent LOB, it will do nothing (no-op).

> **See Also:**
>
> Performance Guidelines

3.1.2 Handling Temporary LOBs on the Client Side

You must consider the aspects discussed in this section while handling the temporary LOBs that are generated by the client programs.

Preventing Temporary LOB Accumulation

Every time a client program such as JDBC or OCI obtains a LOB locator from SQL or PL/SQL, and you suspect that it is producing a temporary LOB, then free the LOB as soon as your application has consumed the LOB. If you do not free the temporary LOB, then it will lead to accumulation of temporary LOBs, which can considerably slow down your system.

> **Note:**
>
> A temporary LOB duration is always upgraded to `SESSION`, when it is shipped to the client side.

For example, to prevent temporary LOB accumulation, an OCI application must call the `OCILobFreeTemporary()` function in the following scenarios:

- After getting a locator from a define during a `SELECT` statement or an `OUT` bind variable from a PL/SQL procedure or function. It is desirable that you free the temporary LOB as soon as you finish performing the required operations on it. If not, then you must free it before reusing the variable for fetching the next row or for another purpose.
- Before performing a pointer assignment, like `<var1 = var2>`, free the old temporary LOB in the variable `<var1>`.

LOB Assignment
You must take special care when assigning the `OCILobLocator` pointers in an OCI program while using the assignment (=) operator. Pointer assignments create a shallow copy of the LOB. After the pointer assignment, the source and the target LOBs point to the same copy of data. This means that if you call the `OCILobFreeTemporary()` function on either one of them, then both variables will point to non-existent LOBs.

These semantics are different from using the LOB APIs, such as the `OCILobLocatorAssign()` function to perform assignments. When you use these APIs, the locators logically point to independent copies of data after assignment. This means that eventually the `OCILobFreeTemporary()` function must be called on each LOB descriptor separately, so that it frees all LOBs involved in the operation.

For temporary LOBs, before performing pointer assignments, you must ensure that you free any temporary LOB in the target LOB locator by calling the `OCIFreeTemporary()` function. In contrast, when the `OCILobLocatorAssign()` function is used, the original temporary LOB in the target LOB locator variable, if any, is freed automatically before the assignment happens.

3.2 Temporary LOB APIs in Different Programmatic Interfaces

This section lists the temporary LOB specific APIs in different Programmatic Interfaces.

Most of the examples in the following sections use the `print_media` table. Following is the structure of the `print_media` table.

PRINT_MEDIA Table

Column name	Column Type
product_id	NUMBER (6)
ad_id	NUMBER (6)
ad_composite	BLOB
ad_sourcetext	CLOB
ad_finaltext	CLOB
ad_fltextn	NCLOB
ad_textdocs_ntab	NESTED TABLE
ad_photo	BLOB
ad_graphic	BFILE
ad_header	USER DEFINED TYPE
press_release	LONG

- PL/SQL APIs for Temporary LOBs
  This section describes the PL/SQL APIs used with temporary LOBs.
- JDBC API for Temporary LOBs
  This section describes the PL/SQL APIs used with temporary LOBs.
- OCI APIs for Temporary LOBs
  This section describes the OCI APIs used with temporary LOBs.
- ODP.NET API for Temporary LOBs
  This section describes the ODP.NET APIs used with temporary LOBs.
- Pro*C/C++ and Pro*COBOL APIs for Temporary LOBs
  This section describes the Pro*C/C++ and Pro*COBOL APIs for Temporary LOBs.

See Also:

Comparing the LOB Interfaces

3.2.1 PL/SQL APIs for Temporary LOBs

This section describes the PL/SQL APIs used with temporary LOBs.

> **See Also:**
>
> DBMS_LOB

Table 3-1 DBMS_LOB Functions and Procedures for 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

Example 3-1 PL/SQL API for Temporary LOBs

```
DECLARE
  blob1 BLOB;
  clob1 CLOB;
  clob2 CLOB;
  nclob1 NCLOB;
BEGIN
  -- create a temp LOB using CREATETEMPORARY and fill it with data
  DBMS_LOB.CREATETEMPORARY(blob1,TRUE, DBMS_LOB.SESSION);
  writeDataToLOB_proc(blob1);

  -- create a temp LOB using SQL built-in function
  SELECT substr(ad_sourcetext, 5) INTO clob1 FROM print_media WHERE
product_id=1 AND ad_id=1;

  -- create a temp LOB using a PLSQL built-in function
  nclob1 := TO_NCLOB(clob1);

  -- create a temp LOB using a PLSQL procedure. Assume foo creates a temp lob
and it's parameter is IN/OUT
  foo(clob2);

  -- Other APIs
  CALL_LOB_APIS(blob1, clob1, clob2, nclob1);

  -- free temp LOBs
  DBMS_LOB.FREETEMPORARY(blob1);
  DBMS_LOB.FREETEMPORARY(clob1);
  DBMS_LOB.FREETEMPORARY(clob2);
  DBMS_LOB.FREETEMPORARY(nclob1);

END;
/
show errors;
```

3.2.2 JDBC API for Temporary LOBs

This section describes the PL/SQL APIs used with temporary LOBs.

> **See Also:**
>
> Working with LOBs and BFILEs

Table 3-2 jdbc.sql.Clob and java.sql.Blob APIs for Temporary LOBs

Methods	Description
`createTemporary`	Creates a temporary LOB
`isTemporary`	Checks if a LOB locator refers to a temporary LOB
`freeTemporary`	Frees a temporary LOB

Example 3-2 JDBC API for Temporary LOBs

```
public class listempc
{
  public static void main (String args [])
    throws Exception
  {
    Connection conn = LobDemoConnectionFactory.getConnection();

    // SELECT TEMPORARY LOB USING SQL
    Statement stmt = conn.createStatement ();
    ResultSet rset = stmt.executeQuery
          ("SELECT SUBSTR(ad_sourcetext, 5) FROM Print_media WHERE product_id
= 3106 AND ad_id = 1");
    if (rset.next())
    {
      Clob clob = rset.getClob (1);
      System.out.println("Is lob temporary: " + ((CLOB)clob).isTemporary());

      call_other_apis_to_read_write_from_lob(clob);
      clob.free();
    }
    stmt.close();

    // CREATE TEMPORARY LOB VIA API
    Clob clob = conn.createClob();

    System.out.println( "Is clob temporary: " +
((oracle.jdbc.OracleClob)clob).isTemporary());

    call_other_apis_to_read_write_from_lob(clob);

     // ALWAYS FREE THE TEMPORARY LOB WHEN DONE WITH IT
    clob.free();

    conn.close();
```

```
  }
}
```

3.2.3 OCI APIs for Temporary LOBs

This section describes the OCI APIs used with temporary LOBs.

> **See Also:**
>
> LOB and BFILE Operations

Table 3-3 OCI APIs for Temporary LOBs

Function / Procedure	Description
`OCILobCreateTemporary()`	Creates a Temporary LOB
`OCILobIsTemporary()`	Checks if a LOB locator refers to a temporary LOB
`OCILobFreeTemporary()`	Frees a temporary LOB

Example 3-3 OCI APIs for Temporary LOBs

```
void temp_lob_operations()
{
  OCILobLocator *temp_clob1;
  OCILobLocator *temp_clob2;
  OCIStmt       *stmhp = (OCIStmt *) 0;
  OCIDefine     *dfnhp1;
  ub1            bufp[BUFLEN];
  ub4            amtp = 0;
  ub8            bamtp = 0;
  ub8            camtp = 0;
  ub2            retl1, rcode1;
  sb4            ind_ptr1 = 0;
  boolean        istemp = FALSE;
  char          *sel_stmt = "SELECT SUBSTR(ad_sourcetext, 5) FROM Print_media
WHERE product_id = 3106 AND ad_id = 1";

  /* allocate lob descriptors */
  checkerr(errhp, OCIDescriptorAlloc((dvoid *) envhp, (dvoid **) &temp_clob1,
                                     (ub4) OCI_DTYPE_LOB, (size_t) 0,
                                     (dvoid **) 0));
  checkerr(errhp, OCIDescriptorAlloc((dvoid *) envhp, (dvoid **) &temp_clob2,
                                     (ub4) OCI_DTYPE_LOB, (size_t) 0,
                                     (dvoid **) 0));

  /* statement handle */
  checkerr(errhp, OCIHandleAlloc( (dvoid *)envhp, (dvoid **) &stmhp,
                   (ub4) OCI_HTYPE_STMT, (size_t) 0, (dvoid **) 0));
  checkerr(errhp, OCIHandleAlloc( (dvoid *)stmhp, (dvoid **) &dfnhp1,
                   (ub4) OCI_HTYPE_DEFINE, (size_t) 0, (dvoid **) 0));

  /*-------------------- SELECT TEMPORARY LOB USING SQL
------------------------*/
```

```
  checkerr(errhp, OCIStmtPrepare(stmhp, errhp, (text *) sel_stmt,
          (ub4) strlen(sel_stmt), OCI_NTV_SYNTAX, OCI_DEFAULT));

  checkerr(errhp, OCIDefineByPos(stmhp, &dfnhp1, errhp, (ub4) 1, &temp_clob1,
                  (sb4) -1, SQLT_CLOB, &ind_ptr1, &retl1, &rcode1,
                  (ub4) OCI_DEFAULT));

  checkerr(errhp, OCIStmtExecute(svchp, stmhp, errhp, (ub4) 0, (ub4) 0,
                     (OCISnapshot *) NULL, (OCISnapshot *) NULL,
OCI_DEFAULT));
  checkerr(errhp, OCIStmtFetch(stmhp, errhp, 1, OCI_FETCH_NEXT, OCI_DEFAULT));

  checkerr(errhp, OCILobWriteAppend2(svchp, errhp, temp_clob1,
           (oraub8 *)&bamtp, (oraub8 *) &camtp, bufp, (oraub8)BUFLEN,
            OCI_ONE_PIECE, (dvoid*)0, (OCICallbackLobWrite2)0, (ub2)0,
            (ub1)SQLCS_IMPLICIT));

  /*--------------------- CREATE TEMPORARY LOB USING API
-------------------------*/
  checkerr(errhp, OCILobCreateTemporary(svchp, errhp, temp_clob2,
                  (ub2) 0, OCI_DEFAULT, OCI_TEMP_CLOB,
                  FALSE, OCI_DURATION_SESSION));

  /* write into bufp */
  strcpy((char *)bufp, (const char *)"Demo program for testing temp lobs");
  bamtp = amtp = (ub4) strlen((char *)bufp);

  /* write bufp contents to temp lob */
  checkerr(errhp, OCILobWrite2(svchp, errhp, temp_clob2, &amtp, 1,
                  (dvoid *)bufp, (ub4)bamtp , OCI_ONE_PIECE, (dvoid *)0,
                  (OCICallbackLobWrite) 0, (ub2) 0, (ub1) SQLCS_IMPLICIT));

  /*--------------------- ALWAYS FREE TEMPORARY LOBS -------------------------
*/
  checkerr(errhp, OCILobIsTemporary(envhp, errhp, temp_clob1, &istemp));
  if (istemp)
    checkerr(errhp, OCILobFreeTemporary(svchp, errhp, temp_clob1));

  checkerr(errhp, OCILobIsTemporary(envhp, errhp, temp_clob2, &istemp));
  if (istemp)
    checkerr(errhp, OCILobFreeTemporary(svchp, errhp, temp_clob2));

 /* Free lob descriptors */
  checkerr(errhp, OCIDescriptorFree ((dvoid *)temp_clob1, (ub4)
OCI_DTYPE_LOB));
  checkerr(errhp, OCIDescriptorFree ((dvoid *)temp_clob2, (ub4)
OCI_DTYPE_LOB));
}
```

3.2.4 ODP.NET API for Temporary LOBs

This section describes the ODP.NET APIs used with temporary LOBs.

> **See Also:**
>
> Temporary LOBs

Table 3-4 ODP.NET methods for Temporary LOBs in the OracleClob and OracleBlob Classes

Methods	Description
`Add()`	Creates a temporary LOB
`IsTemporary()`	Checks if a LOB locator refers to a temporary LOB
`Dispose()` or `Close()`	Frees a temporary LOB

3.2.5 Pro*C/C++ and Pro*COBOL APIs for Temporary LOBs

This section describes the Pro*C/C++ and Pro*COBOL APIs for Temporary LOBs.

> **See Also:**
>
> - Pro*C/C++ Programmer's Guide
> - Pro*COBOL Programmer's Guide

Table 3-5 Pro*C/C++ and Pro*COBOL APIs for Temporary LOBs

Statement	Description
`CREATE TEMPORARY`	Creates a Temporary LOB
`DESCRIBE [ISTEMPORARY]`	Checks if a LOB locator refers to a temporary LOB
`FREE TEMPORARY`	Frees a temporary LOB

4
Value LOBs

Value LOBs, are a subset of Temporary LOBs, that are autonomous, read-only and more performant.

4.1 About Value LOBs

Use Persistent and Temporary LOBs, henceforth referred to as Reference LOBs, for applications which require reads and writes on the LOB.

Many applications use LOBs to store medium-sized objects, about a few mega-bytes in size, and just want to read the LOB value in the context of a SQL query. Oracle recommends that you use Value LOBs for applications which use LOBs as a larger `VARCHAR` or `RAW` data type.

Value LOBs have the following characteristics:

- A Value LOB is a special kind of read-only temporary LOB with optimizations for better performance and manageability compared to a reference LOB.
- All LOB read APIs are supported on value LOBs, but none of the write APIs are supported. LOB read APIs allow the data to be read piecewise and support random access by allowing the user to specify the amount and the offset of the operation.
- A value LOB gets automatically freed when the next fetch for a cursor is performed. In the case of temporary LOB, it has always been the user's responsibility to free the temporary LOB when their application is done processing it.

Several SQL operators such as `substr`, `to_clob()` create temporary LOBs, but it is the responsibility of the user to free these temporary LOBs. If you don't free the temporary LOBs, it can considerably slow down your system. With Value LOBs, a user can use LOBs similar to the `VARCHAR32k` data type without the responsibility to free them. Oracle Database automatically frees the LOBs as soon as the next set of rows are fetched in SQL. Since the concept of fetch duration exists only for SQL, value LOBs exist only in the context of a SQL query.

- A value LOB can be arbitrary sized, similar to a reference LOB.

  As all LOB read APIs are supported on value LOBs, there is no limit on the LOB size. However, value LOBs are best suited for documents of size up to a few megabytes, which can be prefetched to the client.

- A value LOB, in most instances, has faster performance than a reference LOB. Oracle highly recommends that you use value LOBs if your application fetches a LOB for read purposes as part of a SQL query and consumes the LOB data before the next fetch is performed on the cursor.

- PL/SQL doesn't have the concept of value LOBs.

  Value LOBs exist in the SQL query, as well as on the client side programmatic interfaces, such as JDBC, OCI, and ODP.NET. In PL/SQL, a temporary LOB is automatically freed when a user overwrites the variable containing that LOB. Hence any value LOBs passed from SQL to PL/SQL are converted to read-only temporary LOBs, and have the duration of the variable that holds the LOB. For more information, see PL/SQL APIs for Value LOBs.

4.2 When to Use Value LOBs

Many applications use LOBs to store medium-sized objects, about a few mega-bytes in size, and just want to read the LOB value in the context of a SQL query.

Oracle recommends that you use Value LOBs for applications which use LOBs as a larger `VARCHAR` or `RAW` data type. All the LOB reads are performed before the next set of SQL rows are fetched. This implies that if you want to access the value LOB content after fetching the next set of SQL rows, then you must read and save the value LOB content on the client side. Oracle recommends that you use Persistent and Temporary LOBs in the following scenarios:

- for applications which require reads and writes to the LOB
- where the LOB is expected to last for a longer duration
- when you can't fully read and cache the LOB content at the client side, but rely on the LOB locator to read LOB data from server

> **Caution:**
>
> Before transforming a reference LOB column to a Value LOB column, ensure that your business use case requires the usage of Value LOBs.

4.3 Creating a Value LOB

A value LOB is a read-only temporary LOB that is generated by a SQL statement and is auto-freed at the next SQL fetch. Use value LOBs only for scenarios where a LOB fetched from SQL is only read before the next fetch is performed.

A value LOB is always a temporary LOB irrespective of its origin. When the next fetch is performed on the cursor,Oracle server automatically frees the value LOBs from the previous fetch. So the value LOBs from the previous fetch won't be accessible. A value LOB can be seen as disposable LOB: once it is read, it is not needed anymore, and it is freed. This prevents accumulation of temporary LOBs, which translates to better performance and scalability of the query.

You can create a value LOB in a SQL query in the following ways:

- Creating Value LOBs Using DDL
- Creating Value LOBs Using SQL Operators
- Creating Value LOB in Views
- Creating Value LOBs Using LOB_VALUE Operator

Most of the examples in the following sections use the `agents` table. Following is the structure of the `agents` table.

Column Name	Column Type
ID	NUMBER
NAME	VARCHAR2(100)
VALUELOB	CLOB VALUE
REFERENCELOB	CLOB
CV	BLOB
PHOTO	BLOB VALUE

- Creating Value LOBs Using DDL
  If your application is written in a way that all LOBs from a particular table follow the Value LOB use case, then use the `query as value` syntax for the LOB column as part of the `CREATE TABLE` or `ALTER TABLE` statements.
- Creating Value LOBs Using SQL Operators
  Any SQL operator that has a LOB input and a LOB output will produce a value LOB if the input is a value LOB, as shown in the following examples:
- Creating Value LOBs Using LOB_VALUE Operator
  You can convert any persistent or temporary LOB to a Value LOB by using the `LOB_VALUE` operator, as shown in the following examples.
- Creating Value LOB in Views
  LOB columns with QUERY AS VALUE property can be part of a view. You can create them in two ways.

4.3.1 Creating Value LOBs Using DDL

If your application is written in a way that all LOBs from a particular table follow the Value LOB use case, then use the `query as value` syntax for the LOB column as part of the `CREATE TABLE` or `ALTER TABLE` statements.

This fetches all LOB locators from the specified column as value without changing the rest of the application. The default for create tables is `query as reference` which fetches a LOB locator as a Reference LOB which can be persistent or temporary.

Example

The following example creates a table with the name agent with columns of various LOB types, including `valuelob` and `photo` which are value LOBs:

```
create table agents (id number, name varchar2(100), valuelob clob,
referencelob clob, cv blob)
lob(valuelob) query as value;
alter table agents add (photo blob) lob(photo) query as value;
```

The following example shows how you can switch the query property of a LOB column between `value` or `reference` using the `ALTER TABLE MODIFY LOB` clause.

```
alter table agents modify lob(cv) query as value;
```

> **Note:**
>
> The `query as value` or `query as reference` property only impacts how a LOB is selected out in a query. It does not change how the LOB is physically stored in the table.

When describing a table, a columns defined with `query as value` will contain the `VALUE` keyword with the data type.

```
SQL> desc agents;
Name                  Null?     Type
--------------------- --------  ----------------------------
ID                               NUMBER
NAME                             VARCHAR2(100)
VALUELOB                         CLOB VALUE
REFERENCELOB                     CLOB
CV                               BLOB
PHOTO                            BLOB VALUE
```

The following example shows when you run a query on a column defined with the `query as value` property, it returns a value LOB.

```
select valuelob from agents;
```

> **Note:**
>
> The `query as value` property is applicable only to LOB locators, hence it does not affect the data interface on LOBs. In other words, there is no difference between how the data interface works with LOBs declared with `query as reference` and `query as value`.

4.3.2 Creating Value LOBs Using SQL Operators

Any SQL operator that has a LOB input and a LOB output will produce a value LOB if the input is a value LOB, as shown in the following examples:

```
-- Produces a value LOB
SELECT SUBSTR(valuelob, 5, 5) from agents;

-- Produces a value LOB
SELECT to_blob(valuelob, 0) from agents;

-- Produces a value LOB
SELECT CONCAT(valuelob, valuelob) from agents;
```

Any SQL operator that has a LOB input and a LOB output will produce an old fashioned temporary LOB if the input is a reference LOB, or if the input is not a LOB type, as shown in the following examples:

```
-- Produces a reference Temporary LOB
SELECT SUBSTR(referencelob, 5, 5) from agents;

-- Produces a reference Temporary LOB
SELECT to_blob(substr(referencelob,5,5), 0) from agents;

-- Produces a reference Temporary LOB as name is varchar type
SELECT to_clob(name) from agents;
```

If a SQL operator takes in 2 LOBs, then it is an error if one of them is a value lob and the other is not, as shown in the following example:

```
-- Raises an error
SELECT CONCAT(valuelob, referencelob) from agents;
```

4.3.3 Creating Value LOBs Using LOB_VALUE Operator

You can convert any persistent or temporary LOB to a Value LOB by using the `LOB_VALUE` operator, as shown in the following examples.

```
-- Produces a value LOB
SELECT lob_value(referencelob) from agents;

-- Produces a value LOB
SELECT lob_value(substr(referencelob, 2, 10)) from agents;

-- Produces a value LOB
SELECT lob_value(to_clob(name)) from agents;
```

The `LOB_VALUE()` operator is useful when a LOB column in a table cannot be set as `QUERY AS VALUE` because different queries on that column may need to select the LOB as Reference or Value. Use this operator to convert Reference LOBs to Value LOBs to take advantage of the performance benefits offered by Value LOBs.

PL/SQL functions do not produce Value LOBs. To get a Value LOB from PL/SQL functions, use the `LOB_VALUE()` operator on the output of a PL/SQL function, as shown in the following example.

```
-- Produces a value LOB
SELECT LOB_VALUE(lob_producing_plsql_function(...)) from table;
```

> **Note:**
>
> The reverse conversion of a Value LOB to a Reference LOB is not permitted.

4.3.4 Creating Value LOB in Views

LOB columns with QUERY AS VALUE property can be part of a view. You can create them in two ways.

If view column refers to a value LOB column, view column will be a value LOB as well. The following example shows how `valuelob_v` column is a value LOB as part of a view.

```
-- valuelob_v column is a value LOB column
create view agents_v as
   select id id_v, valuelob valuelob_v from agents;
```

If view column uses a SQL/LOB operator that returns value LOB, it will be value LOB.

```
-- valuelob_v2 column is a value LOB
create view agents_v2 as
    select id id_v2, substr(valuelob, 5, 5) valuelob_v2 from agents;
-- valuelob_v3 column is a value LOB
create view agents_v3 as
    select id id_v3, lob_value(referencelob) valuelob_v3 from agents;
-- valuelob_v4 column is a value LOB
create view agents_v4 as
    select id id_v4, lob_value(substr(referencelob, 5, 5)) valuelob_v4 from
agents;
```

The describe command of SQLPLUS shows if a LOB column for a view is value LOB:

```
SQL> desc agents_v;
Name                                      Null?    Type
----------------------------------------- --------
----------------------------
ID_V                                               NUMBER
NAME_V                                             VARCHAR2(100)
VALUELOB_V                                         CLOB VALUE

SQL> desc agents_v2;
Name                                      Null?    Type
----------------------------------------- --------
----------------------------
ID_V2                                              NUMBER
```

```
NAME_V2                                  VARCHAR2(100)
VALUELOB_V2                              CLOB VALUE
```

4.4 Value LOBs in Queries

Whether a query fetches a Value or a Reference LOB is a compile-time decision, and can be obtained by using a describe on the query.

Wherever a SQL function for JSON returns a LOB value, it returns a reference LOB by default. However, the returning clause can specify that the LOB be value-based. For example:

```
JSON_SERIALIZE (data returning CLOB VALUE)
```

Example: Explain plan for Value LOBs

The explain plan output shows "/* LOB_BY_VALUE */" hint for value LOB in the plan.

```
SQL> explain plan for select valuelob, substr(valuelob, 5, 5) from agents;
...
-------------------------------------------------------------------------------
| Id  | Operation         | Name   | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |        |     1 |  2002 |     2   (0)| 00:00:01 |
|   1 | TABLE ACCESS FULL| AGENTS |     1 |  2002 |     2   (0)| 00:00:01 |
-------------------------------------------------------------------------------
...
Column Projection Information (identified by operation id):
-----------------------------------------------------------
   1 - "VALUELOB" /*+ LOB_BY_VALUE */ [LOB,4000]
...
```

Example: Explain plan for Reference LOBs without LOB_BY_VALUE

```
SQL> explain plan for select referencelob, substr(referencelob, 5, 5) from
agents;
...
-------------------------------------------------------------------------------
| Id  | Operation         | Name   | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |        |     1 |  2002 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS FULL| AGENTS |     1 |  2002 |     2   (0)| 00:00:01 |
-------------------------------------------------------------------------------
...
Column Projection Information (identified by operation id):
-----------------------------------------------------------
   1 - "REFERENCELOB"[LOB,4000]
...
```

In certain situations, the Oracle server automatically determines that a LOB should be selected as a Value LOB.

Example: Explain plan showing automatic conversion to Value LOB

```
SQL> explain plan for select length(referencelob) from agents;
...
-------------------------------------------------------------------------------
| Id  | Operation         | Name   |  Rows | Bytes | Cost (%CPU)|  Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |        |     1 |  2002 |     2   (0)|  00:00:01 |
|   1 |  TABLE ACCESS FULL| AGENTS |     1 |  2002 |     2   (0)|  00:00:01 |
-------------------------------------------------------------------------------
...
Column Projection Information (identified by operation id):
-----------------------------------------------------------
   1 - "REFERENCELOB" /*+ LOB_BY_VALUE */ [LOB,4000]
...
```

> **See Also:**
>
> Example 4-3.

4.5 Performing DML Operations on LOBs with QUERY AS VALUE

The QUERY AS VALUE property is applicable only for SQL queries, so it does not affect any DML on the table. Therefore any DMLs such as `INSERT` or `UPDATE` work identically for LOB columns declared as QUERY AS VALUE or QUERY AS REFERNCE.

DML RETURNING

If a user performs a DML with a RETURNING clause for the LOB locator, the returned LOB locator will be a reference LOB irrespective of the QUERY AS property of the LOB column. Hence a user can INSERT an `empty_lob()` with RETURNING clause, and use the returning locator to write into the persistent LOB, as shown in the following example:

```
-- insert with returning clause
declare
  c clob;
begin
  -- returns a reference to the persistent LOB
  insert into agents values (1, 'My Name', empty_clob(), NULL, NULL, NULL)
      returning valuelob into c;
  -- updates the persistent LOB
  dbms_lob.writeappend(c, length('I am a value LOB'), 'I am a value LOB');
end;
/
```

SELECT FOR UPDATE

Along the same lines, if the user selects out a locator with the `FOR UPDATE` clause, the intention is to update the persistent LOB stored in the LOB column. So the returned LOB locator will be a reference LOB irrespective of the QUERY AS property of the LOB column, as shown in the following example.

```
-- Select for update
declare
  c clob;
begin
  -- returns a reference to the persistent LOB
  select valuelob into c from agents where id = 1 for update;
  -- This updates the persistent LOB
  dbms_lob.writeappend(c, length('!!!!'), '!!!!');
end;
/
```

4.6 Value LOB APIs in Different Programmatic Interfaces

This section lists the value LOB specific APIs in different programmatic interfaces.

- Prefetching of Value LOBs
  Setting a large LOB prefetch size on the client side avoids round trips to server and leads to a drastically improved performance for value LOBs.
- Value LOB API Support
  Once a LOB variable is initialized with either a persistent or a temporary LOB locator, subsequent read operations on the LOB can be performed using APIs such as the `DBMS_LOB` package subprograms.
- PL/SQL APIs for Value LOBs
  This section describes the PL/SQL APIs used with value LOBs.
- OCI APIs for Value LOBs
  This section describes the OCI APIs used with value LOBs.
- Interoperability with Older Clients
  When you send a value LOB to a client that's at version 21c or earlier, the value LOB is converted to a read-only temporary reference LOB.

4.6.1 Prefetching of Value LOBs

Setting a large LOB prefetch size on the client side avoids round trips to server and leads to a drastically improved performance for value LOBs.

The default LOB prefetch size for value LOBs in JDBC, OCI and ODP.NET is 32k.

> **Note:**
>
> In OCI, setting a LOB prefetch size of 0 will be interpreted by Oracle as a prefetch size of 32k for value LOBs, and 0 for reference LOBs. Any other setting of LOB prefetch size is honored for both value and reference LOBs.

Oracle recommends setting the LOB prefetch size large enough to accommodate at least 80% of your LOB read size for value LOBs.

> **See Also:**
>
> Prefetching LOB Data and Length

4.6.2 Value LOB API Support

Once a LOB variable is initialized with either a persistent or a temporary LOB locator, subsequent read operations on the LOB can be performed using APIs such as the `DBMS_LOB` package subprograms.

> **See Also:**
>
> LOB APIs

The operations supported on value LOBs are divided into the following categories:

Table 4-1 Operations supported by value LOB APIs

Category	Value LOB Behavior
Sanity Checking	Supported
Open/Close	Open is allowed in read-only mode. If read-write is specified, an error is thrown.
Read Operations	Supported
Modify Operations	Not supported, since value LOBs are read-only.
Operations involving multiple locators	`OCILobLocatorAssign()` is not supported.
	`OCILobIsEqual()` will always return `FALSE` because `OCILobLocatorAssign` is not supported.
	`OCILobAppend()`, `OCILobCopy2()`,and `OCILobLoadFromFile2()`: These operations support a Value LOB as the source LOB. The destination LOB has to be selected `FOR UPDATE`, which converts it to a reference LOB, hence permitting the operation to go through. The destination LOB cannot be a value LOB.
Operations specific to SecureFiles	Not supported, since value LOBs are temporary LOBs and not Securefiles.
Operations specific to temporary LOBs	`CreateTemporary` cannot produce a value LOB. `FreeTemporary` and `IsTemporary` are supported but not required, since value LOBs are freed automatically.

4.6.3 PL/SQL APIs for Value LOBs

This section describes the PL/SQL APIs used with value LOBs.

Value LOBs exist in the SQL query, as well as on the client-side programmatic interfaces such as JDBC, OCI and ODP.NET. In PL/SQL, a temporary LOB is freed automatically when a user overwrites the variable containing that LOB. Hence any value LOBs passed from SQL to

PL/SQL are converted to read-only temporary LOBs, and have the duration of the variable that holds the LOB. When the variable goes out of scope, the temporary LOB is freed automatically.

In other words, there are no value LOBs in PL/SQL. All LOBs in PL/SQL are reference LOBs. If a Value LOB is sent from SQL to PL/SQL, it becomes a read-only temporary LOB. This LOB supports the APIs listed in Table 4-1. If this LOB is passed from PL/SQL to a client, such as JDBC or OCI, then it continues to be a read-only temporary LOB. This means it follows the same API support as in Table 4-1, but it will not be freed automatically when the next fetch is performed on the cursor. It will be the user's responsibility to free this LOB when they are done with it.

Example 4-1 PL/SQL API for Value LOBs

The following example shows how you can create a value LOB by using the `LOB_VALUE` operator on the output of a PL/SQL function.

```
SELECT LOB_VALUE(lob_producing_plsql_function(...)) from table;
```

OUT Binds

Since `OUT` binds are not part of a SQL query, they cannot return a value LOB. When you attempt to return a value LOB as an `OUT` bind variable, the value LOB is converted to a read-only temporary LOB. It is the responsibility of the application developer to free the LOB after using it since it is a reference LOB. Note that the `LOB_VALUE` operator does not work on `OUT` binds.

4.6.4 OCI APIs for Value LOBs

This section describes the OCI APIs used with value LOBs.

Example 4-2 Explicit Describe returning OCI_ATTR_LOB_IS_VALUE on the Table Using OCIDescribeAny

```
void explicitDescribe()
{
  /* Assume: create table lobtab (c clob) lob(c) query as value */
  ub1         isValueLob = 0;
  OCIParam    *colhd;
  OCIParam    *paramp = NULL;
  OCIParam    *lstHandle = NULL;
  text        *table = "lobtab";
  OCIDescribe *dschp = NULL;

  OCIHandleAlloc(envhp, (void **)&dschp, OCI_HTYPE_DESCRIBE, 0, NULL);
  checkerr(errhp, OCIDescribeAny(svchp, errhp, (void *)table, strlen(table),
                                 OCI_OTYPE_NAME, 0, OCI_PTYPE_TABLE, dschp));

  checkerr(errhp, OCIAttrGet((dvoid *) dschp, (ub4) OCI_HTYPE_DESCRIBE,
                  (dvoid *)&paramp, (ub4*)0, (ub4)OCI_ATTR_PARAM, errhp));

  /* Get the number of columns */
  checkerr(errhp, OCIAttrGet((void *)paramp, OCI_DTYPE_PARAM, (void
*)&noofcols,
                             (ub4 *)0, OCI_ATTR_NUM_COLS, errhp));

  /* Get the column list. */
```

```
  checkerr(errhp, OCIAttrGet(paramp, OCI_DTYPE_PARAM, &lstHandle, 0,
                             OCI_ATTR_LIST_COLUMNS, errhp));

  /* Go through the column list */
  for (int i = 1; i <= noofcols; i++)
  {
    /* Get parameter for column i */
    checkerr(errhp, OCIParamGet(lstHandle, OCI_DTYPE_PARAM,
                                (OCIError *)errhp, (void**)&colhd, (ub4)i),
                                (text *)"param get");

    isValueLob = 0;
    checkerr(errhp,OCIAttrGet(colhd, OCI_DTYPE_PARAM,
                                     &(isValueLob), 0,
                                      OCI_ATTR_LOB_IS_VALUE,
                                      (OCIError *)errhp),
                                      (text*)"attr get");

    printf("Is value lob = %d\n",isValueLob); /* Expected output: Is value
lob = 1 */
   }
   ...
}
```

Example 4-3 Implicit Describe returning OCI_ATTR_LOB_IS_VALUE on a Query's Column Handle

```
text          *select_sql = (text *)"select valuelob from agents"; /*
generates value LOB */
OCILobLocator *lob1;
Boolean        isValLob = 0;
OCIParam      *colhd;

/* Prepare select statement */
checkerr(errhp, OCIStmtPrepare(stmthp, errhp, select_sql,  /* select valuelob
from agents */
                               (ub4) strlen((char *) select_sql), ...);

/* Execute select statement */
checkerr(errhp, OCIStmtExecute(svchp, stmthp, errhp, ...);

/* Implicit Describe: Get parameter for select item #1 */
checkerr(errhp, OCIParamGet(stmthp, OCI_HTYPE_STMT,
                            (OCIError *)errhp,
                            (void**)&colhd, (ub4)1),
                            (text *)"param valuelob column");

/* Check if colhd (valulob column) returns value LOB */
checkerr(errhp,OCIAttrGet(colhd, OCI_DTYPE_PARAM,
                          &(isValueLob), 0,
                          OCI_ATTR_LOB_IS_VALUE,
                          (OCIError *)errhp),
                          (text *)"attr get");
```

Example 4-4 Checking for OCI_ATTR_LOB_IS_VALUE after fetching a LOB Locator

```
void CheckValue(OCILobLocator *lob1)
{
  boolean isValLob = 0;

  /* Check if lob1 is value LOB */
  OCIAttrGet(lob1, OCI_DTYPE_LOB, &isValLob, sizeof(boolean),
             OCI_ATTR_LOB_IS_VALUE, errhp);
}
```

Example 4-5 Checking for OCI_ATTR_LOB_IS_READONLY after fetching a LOB Locator

Recall that there are no Value LOBs in PL/SQL. If a Value LOB is sent from SQL to PL/SQL, it becomes a Read-Only Temporary LOB. This LOB will follow the same API support as in Table 4-1. If this LOB is passed from PL/SQL to a client like JDBC or OCI, then it will continue to be a Read-Only Temporary LOB. The example below shows how to check the read-only property of a LOB locator by using the `OCI_ATTR_LOB_IS_READONLY` attribute.

```
void CheckReadOnly(OCILobLocator *lob1)
{
  boolean isReadOnly = 0;

  /* Check if lob1 is read-only */
  OCIAttrGet(lob1, OCI_DTYPE_LOB, &isReadOnly, sizeof(boolean),
             OCI_ATTR_LOB_IS_READONLY, errhp);
}
```

4.6.5 Interoperability with Older Clients

When you send a value LOB to a client that's at version 21c or earlier, the value LOB is converted to a read-only temporary reference LOB.

This means it will follow the same API support as in <the category table in 4.4.2>, but it will not be freed automatically when the next fetch is performed on the cursor. It will be the user's responsibility to free this LOB when they are done with it.

4.7 Restrictions on Value LOBs

Keep the following restrictions in mind while working with value LOBs.

- Value LOBs are read-only LOBs, so you are not permitted to perform modify operations on them.
- Locator assignment operations, such as `OCILobLocatorAssign()` are not supported on value LOBs.
- Value LOBs is not supported with the following operation in JDBC:
  - Client-side result cache
- Value LOBs are not supported with the following operations in OCI:
  - Scrollable cursors
  - Client-side result cache

- Value LOBs cannot be combined with reference LOBs in the same operation in a query, for example:

```
select concat(valuelob, referencelob) from agents; -- Error expected
```

5
BFILEs

BFILEs are data objects stored in operating system files, outside the database tablespaces. Data stored in a table column of type BFILE is physically located in an operating system file, not in the database. The BFILE column stores a reference to the operating system file.

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

You create BFILEs to hold the following types of data:

- 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 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 BFILEs provided the operating system supports stream-mode access to the operating system files.

- DIRECTORY Objects
  A BFILE locator is initialized by using the function BFILENAME(DIRECTORY, FILENAME). This section describes how to initialize the DIRECTORY Object.
- BFILE Locators
  For BFILEs, the value is stored in a server-side operating system file, in other words, BFILEs are external to the database. The BFILE locator that refers to the file is stored in the database row.
- BFILE APIs
  This section discusses the different operations supported through BFILEs.
- BFILE APIs in Different Programmatic Interfaces
  This section lists all the APIs from different Programmatic Interfaces supported by Oracle Database.

5.1 DIRECTORY Objects

A BFILE locator is initialized by using the function BFILENAME(DIRECTORY, FILENAME). This section describes how to initialize the DIRECTORY Object.

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 you have the required access privilege on the DIRECTORY object. You can also use Oracle Enterprise Manager Cloud Control to manage the DIRECTORY objects.

The DIRECTORY object 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() function in OCI, for initializing a BFILE locator.

- DIRECTORY Name Specification
  You must have `CREATE ANY DIRECTORY` system privilege to create directories.
- Security on Directory Objects
  This section describes the security on `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

5.1.1 DIRECTORY Name Specification

You must have `CREATE ANY DIRECTORY` system privilege to create directories.

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`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')
```

> **WARNING:**
>
> The database does not verify that the directory and path name you specify actually exist. You must ensure to specify a valid directory name in your operating system. If your operating system uses case-sensitive path names, then be sure that you specify the directory name 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` or `abc/../def` or `abc/def/hij..`

On Windows Platform

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";
```

5.1.2 Security on Directory Objects

This section describes the security on DIRECTORY objects.

The DIRECTORY object model has two distinct levels of security:

- SQL DDL: CREATE or DROP a DIRECTORY object
- SQL DML: READ system and object privileges on DIRECTORY objects

DBA Privileges: CREATE / DROP DIRECTORY

The DIRECTORY object is a *system owned* object. Oracle Database supports the following 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

> **WARNING:**
>
> 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.

> **See Also:**
>
> *Oracle Database SQL Language Reference* for information about system owned objects, CREATE DIRECTORY and DROP DIRECTORY

USER Privileges: READ Permission on the Directory

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 or roles and then 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.

> **See Also:**
>
> - Guidelines for DIRECTORY Usage
> - *Oracle Database SQL Language Reference* for information about the `GRANT`, `REVOKE` and `AUDIT` system and object privileges that provide security for `BFILE`s.

Catalog Views on DIRECTORY Objects

Catalog views are provided for `DIRECTORY` objects to enable users to view object names and corresponding paths and privileges. Following are the supported views:

- `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.

5.2 BFILE Locators

For `BFILE`s, the value is stored in a server-side operating system file, in other words, `BFILE`s are external to the database. The `BFILE` locator that refers to the file is stored in the database row.

To associate an operating system file to a `BFILE`, first create a `DIRECTORY` object that is an alias for the full path name to the operating system file. Then, you can initialize an instance of `BFILE` type, using the `BFILENAME` function in SQL or PL/SQL, or `OCILobFileSetName()` in OCI. You can use this `BFILE` instance in one of the following ways:

- If your need for a particular `BFILE` is temporary and limited within the module on which you are working, then you can assign this `BFILE` instance to a PL/SQL or OCI local variable of type `BFILE`. Subsequently, you can use the `BFILE` related APIs on this variable without having to associate this with a column in the database. The `BFILE` API operations on a temporary instance are executed on the client side, without any round-trips to the server.
- You can insert a persistent reference to a `BFILE` in the `BFILE` column using an `INSERT` or `UPDATE` statement. Before using SQL to insert or update a row with a `BFILE`, you must initialize the `BFILE` variable to either `NULL` or a `DIRECTORY` object name and file name.

> **Note:**
>
> The `OCISetAttr()` function does not allow you to set a `BFILE` locator to `NULL`. To insert a `NULL BFILE` in OCI, you must set the bind value to `NULL`.

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;
```

> **See Also:**
>
> Loading BFILEs with SQL*Loader

BFILEs in Objects

If you are using `BFILE`s in objects, you must first set the `BFILE` value, and then flush the object to the database. So, you must first call the `OCIObjectNew()` function, followed by the `OCILobFileSetName()` function and the `OCIObjectFlush()` function respectively.

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 in shared server sessions, `BFILE` operations are bound to one shared server, they cannot migrate from one server to another, and open `BFILE` instances can persist beyond the end of a call to a shared server.

Examples of Creating Directory Objects and BFILE Locators

Many examples in the following sections use the `print_media` table. Following is the structure of the table:

Figure 5-1 print_media table

PRINT_MEDIA Table	
Column name	**Column Type**
product_id	NUMBER (6)
ad_id	NUMBER (6)
ad_composite	BLOB
ad_sourcetext	CLOB
ad_finaltext	CLOB
ad_fltextn	NCLOB
ad_textdocs_ntab	NESTED TABLE
ad_photo	BLOB
ad_graphic	BFILE
ad_header	USER DEFINED TYPE
press_release	LONG

Example 5-1 Inserting BFILEs in SQL and PL/SQL

```
conn system/manager

 -- The DBA creates DIRECTORY object and grants READ to the user
create or replace directory MYDIR as '/your/directory/path/here';
GRANT read ON DIRECTORY MYDIR TO pm;

conn pm/pm

 -- Use BFILENAME to create a BFILE locator for INSERT
INSERT INTO print_media
(product_id, ad_id, ad_composite, ad_sourcetext, ad_graphic)
VALUES
(1, 1, empty_blob(), empty_clob(), BFILENAME('MYDIR','file1.txt'));

-- After this statement, 2 rows point to the same BFILE
```

```
INSERT INTO print_media
(product_id, ad_id, ad_composite, ad_sourcetext, ad_graphic)
    select 2, ad_id, ad_composite, ad_sourcetext, ad_graphic from
print_media;

-- Update the 2nd row to point to a different file
UPDATE print_media SET ad_graphic = BFILENAME('MYDIR','file2.txt')  WHERE
product_id =2;

-- Insert a 3rd row with invalid file name
INSERT INTO print_media
(product_id, ad_id, ad_composite, ad_sourcetext, ad_graphic)
VALUES
(3, 3, empty_blob(), empty_clob(),
BFILENAME('MYDIR','file_does_not_exist.txt'));

-- Insert a NULL for BFILE
INSERT INTO print_media
(product_id, ad_id, ad_composite, ad_sourcetext, ad_graphic)
VALUES
(4, 4, empty_blob(), empty_clob(), NULL);

-- Inserting in PLSQL using a BFILE variable
DECLARE
    f BFILE;
BEGIN
    f := BFILENAME('MYDIR','file5.txt');
    INSERT INTO print_media (product_id, ad_id, ad_composite, ad_sourcetext,
ad_graphic)
    VALUES (5, 5, NULL, NULL, f);
END;
/
SELECT product_id, ad_id, ad_graphic FROM print_media ORDER BY 1,2;
```

Example 5-2 Inserting BFILEs in OCI

```
STATIC TEXT *insstmt = "INSERT INTO print_media (product_id, ad_id,
ad_graphic) VALUES (:1, :1, :2)";
sword insert_bfile()
{
  OCILobLocator *f = (OCILobLocator *)0;

  OCIStmt       *stmthp;
  OCIBind       *bndp1  = (OCIBind *)   0;
  OCIBind       *bndp2  = (OCIBind *)   0;

  ub4            id;

  CHECK_ERROR (OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &stmthp,
                              OCI_HTYPE_STMT, (size_t) 0, (dvoid **) 0));

  /*************** Allocate descriptor **********************/
  CHECK_ERROR (OCIDescriptorAlloc((dvoid *) envhp, (dvoid **) &f,
                                  (ub4)OCI_DTYPE_FILE, (size_t) 0,
                                  (dvoid **) 0));
```

```
  /********** Execute insstmt to insert f ********************/
  id = 6;
  CHECK_ERROR (OCILobFileSetName(envhp, errhp, &f,
                                 (text*)"MYDIR", sizeof("MYDIR") -1,
                                 (text*)"file6.txt",
                                 sizeof("file6.txt") -1));

  CHECK_ERROR (OCIStmtPrepare(stmthp, errhp, insstmt,
                              (ub4) strlen((char *) insstmt),
                              (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT));

  CHECK_ERROR (OCIBindByPos(stmthp, &bndp1, errhp, (ub4) 1, (dvoid *) &id,
                              (sb4) sizeof(id), SQLT_INT, (dvoid *) 0, (ub2
*) 0,
                              (ub2 *)0, (ub4) 0, (ub4*) 0, (ub4)
OCI_DEFAULT));

  CHECK_ERROR (OCIBindByPos(stmthp, &bndp2, errhp, (ub4) 2, (dvoid *) &f4,
                              (sb4) -1, SQLT_BFILE, (dvoid *) 0, (ub2 *) 0,
                              (ub2 *)0, (ub4) 0, (ub4*) 0, (ub4)
OCI_DEFAULT));

  CHECK_ERROR (OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                              (CONST OCISnapshot *) NULL, (OCISnapshot *)
NULL,
                             OCI_DEFAULT));

  /********** Execute insstmt to insert NULL ********************/
  id = 7;
  CHECK_ERROR (OCIStmtPrepare(stmthp, errhp, insstmt,
                              (ub4) strlen((char *) insstmt),
                              (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT));

  CHECK_ERROR (OCIBindByPos(stmthp, &bndp1, errhp, (ub4) 1, (dvoid *) &id,
                              (sb4) sizeof(id), SQLT_INT, (dvoid *) 0, (ub2
*) 0,
                              (ub2 *)0, (ub4) 0, (ub4*) 0, (ub4)
OCI_DEFAULT));

  CHECK_ERROR (OCIBindByPos(stmthp, &bndp2, errhp, (ub4) 2, (dvoid *) NULL,
                              (sb4) -1, SQLT_BFILE, (dvoid *) 0, (ub2 *) 0,
                              (ub2 *)0, (ub4) 0, (ub4*) 0, (ub4)
OCI_DEFAULT));

  CHECK_ERROR (OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                              (CONST OCISnapshot *) NULL, (OCISnapshot *)
NULL,
                             OCI_DEFAULT));

}
```

5.3 BFILE APIs

This section discusses the different operations supported through BFILEs.

Once you initialize a BFILE variable either by using the BFILENAME function or an equivalent API, or by using a SELECT operation on a BFILE column, you can perform read operations on the BFILE using APIs such as DBMS_LOB. Note that BFILE is a read-only data type. So, you cannot update or delete the operating system files, accessed using BFILEs, through the BFILE APIs.
The operations performed on BFILEs are divided into following categories:

Table 5-1 Operations on BFILEs

Category	Operation	Example function /procedure in DBMS_LOB package
Sanity Checking	Check if the BFILE exists on the server	FILEEXISITS
	Get the DIRECTORY object name and file name	FILEGETNAME
	Set the name of a BFILE in a locator without checking if the directory or file exists	BFILENAME
Open / Close	Open a file	OPEN
	Check if the file was opened using the input BFILE locators	ISOPEN
	Close the file	CLOSE
	Close all previously opened files	FILECLOSEALL
Read Operations	Get the length of the BFILE	GETLENGTH
	Read data from the BFILE starting at the specified offset	READ
	Return part of the BFILE value starting at the specified offset using SUBSTR	SUBSTR
	Return the matching position of a pattern in a BFILE using INSTR	INSTR
Operations involving multiple locators	Assign BFILE locator src to BFILE locator dst	dst := src
	Load BFILE data into a LOB	LOADCLOBFROMFILE, LOADBLOBFROMFILE
	Compare all or part of the value of two BFILEs	COMPARE

- Sanity Checking
  Sanity Checking functions on BFILEs enable you to retrieve information about the BFILEs.
- Opening and Closing a BFILE
  You must OPEN a BFILE before performing any operations on it, and CLOSE it before you terminate your program.
- Reading from a BFILE
  You can perform many different read operations on the BFILE data, including reading its length, reading part of the data, or reading the whole data.

- Working with Multiple BFILE Locators
  Some `BFILE` operations accept two locators, at least one of which is a `BFILE` locator. For the assignment and the comparison operations involving BFILES, both the locators must be of `BFILE` type.

5.3.1 Sanity Checking

Sanity Checking functions on BFILEs enable you to retrieve information about the BFILEs.

Recall that the `BFILENAME()` and `OCILobFileSetName()` functions do not verify that the directory and path name you specify actually exist. You can use the sanity checking functions to verify that a `BFILE` exists and to extract the directory and file names from a `BFILE` locator.

5.3.2 Opening and Closing a BFILE

You must `OPEN` a `BFILE` before performing any operations on it, and `CLOSE` it before you terminate your program.

A `BFILE` locator 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 the file must be done from within the same program block using the locator or local copies of it. The `BFILE` locator variable can be used 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.

You must close all the 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.

You can close all open `BFILE`s together by using a procedure like `DBMS_LOB.FILECLOSEALL` or `OCILobFileCloseAll()`.

5.3.3 Reading from a BFILE

You can perform many different read operations on the `BFILE` data, including reading its length, reading part of the data, or reading the whole data.

When reading from a large `BFILE`, you can use the streaming read mode in JDBC or OCI. In JDBC, you can achieve this by using the `getBinaryStream()` method. In OCI, you can achieve it in the way as described in the following section.

Streaming Read in OCI

The most efficient way to read large amounts of `BFILE` data is by using the `OCILobRead2()` function 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.

Amount Parameter

- When calling the DBMS_LOB.READ API, the size of the amount parameter can be larger than the size of the data. However, this 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 the OCILobRead2() function, you can pass a value of UB8MAXVAL for the byte_amt parameter to read to the end of the BFILE.

5.3.4 Working with Multiple BFILE Locators

Some BFILE operations accept two locators, at least one of which is a BFILE locator. For the assignment and the comparison operations involving BFILES, both the locators must be of BFILE type.

Loading a LOB with BFILE data involves special considerations that we will discuss in the following sections:

Loading a LOB with BFILE Data

In PLSQL, the DBMS_LOB.LOADFROMFILE procedure is deprecated in favor of DBMS_LOB.LOADBLOBFROMFILE and DBMS_LOB.LOADCLOBFROMFILE. Specifically, when you use DBMS_LOB.LOADCLOBFROMFILE procedure to load a CLOB or NCLOB instance, it will perform the character set conversions.

Specifying the Amount of BFILE Data to Load

The value you pass for the amount parameter to functions listed in the table below 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 the following table:

Table 5-2 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()	UB8MAXVAL
OCI	OCILobLoadFromFile() (For LOBs less than 4 gigabytes in size.)	UB4MAXVAL

Loading a BLOB with BFILE Data

The DBMS_LOB.LOADBLOBFROMFILE procedure loads a BLOB with data from a BFILE. It can be used to load data into any persistent or temporary BLOB instance. This procedure returns the new source and the destination offsets of the BLOB, which you can then pass into subsequent calls, when used in a loop.

Loading a CLOB with BFILE Data

The DBMS_LOB.LOADCLOBFROMFILE procedure loads a CLOB or NCLOB with character data from a BFILE. It can be used to load data into a persistent or temporary CLOB or NCLOB instance. You can specify the character set ID of the BFILE when calling this procedure and ensure that the character set is properly converted from the BFILE data character set to the destination CLOB or NCLOB character set. This procedure returns the new source and destination offsets of the CLOB or NCLOB, which you can then passe into subsequent calls, when used in a loop.

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.

```
CREATE OR REPLACE PROCEDURE loadCLOB1_proc (dst_loc IN OUT CLOB) IS
  src_loc     BFILE := BFILENAME('MEDIA_DIR','monitor_3060.txt') ;
  amt         NUMBER := DBMS_LOB.LOBMAXSIZE;
  src_offset  NUMBER := 1 ;
  dst_offset  NUMBER := 1 ;
  lang_ctx    NUMBER := DBMS_LOB.DEFAULT_LANG_CTX;
  warning     NUMBER;
BEGIN
  DBMS_OUTPUT.PUT_LINE('------------ LOB LOADCLOBFORMFILE EXAMPLE
------------');
  DBMS_LOB.FILEOPEN(src_loc, DBMS_LOB.FILE_READONLY);

  /* The default_csid can be used when the BFILE encoding is in the same
charset
   * as the destination CLOB/NCLOB charset
   */
   DBMS_LOB.LOADCLOBFROMFILE(dst_loc,src_loc, amt, dst_offset,
src_offset,
       DBMS_LOB.DEFAULT_CSID, lang_ctx,warning) ;
  DBMS_OUTPUT.PUT_LINE(' Amount specified ' || amt ) ;
  DBMS_OUTPUT.PUT_LINE(' Number of bytes read from source: ' ||
(src_offset-1));
  DBMS_OUTPUT.PUT_LINE(' Number of characters written to destination: ' ||
(dst_offset-1) );
  IF (warning = DBMS_LOB.WARN_INCONVERTIBLE_CHAR)
  THEN
    DBMS_OUTPUT.PUT_LINE('Warning: Inconvertible character');
  END IF;
  DBMS_LOB.FILECLOSEALL() ;
END;
/
```

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`.

```
CREATE OR REPLACE PROCEDURE loadCLOB2_proc (dst_loc1 IN OUT NCLOB,dst_loc2 IN
OUT NCLOB) IS
  src_loc     BFILE := BFILENAME('MEDIA_DIR','monitor_3060.txt');
  amt         NUMBER := 100;
  src_offset  NUMBER := 1;
  dst_offset  NUMBER := 1;
  src_osin    NUMBER;
  cs_id       NUMBER := NLS_CHARSET_ID('JA16TSTSET'); /* 998 */
  lang_ctx    NUMBER := dbms_lob.default_lang_ctx;
  warning     NUMBER;
BEGIN
  DBMS_OUTPUT.PUT_LINE('------------ LOB LOADCLOBFORMFILE EXAMPLE
------------');
  DBMS_LOB.FILEOPEN(src_loc, DBMS_LOB.FILE_READONLY);
  DBMS_OUTPUT.PUT_LINE(' BFILE csid is ' || cs_id) ;

  /* Load the first 1KB of the BFILE into dst_loc1 */

  DBMS_OUTPUT.PUT_LINE(' ----------------------------' ) ;
  DBMS_OUTPUT.PUT_LINE('   First load  ' ) ;
  DBMS_OUTPUT.PUT_LINE(' ----------------------------' ) ;

  DBMS_LOB.LOADCLOBFROMFILE(dst_loc1, src_loc, amt, dst_offset, src_offset,
      cs_id, lang_ctx, warning);

  /* the number bytes read may or may not be 1k */

  DBMS_OUTPUT.PUT_LINE(' Amount specified ' || amt ) ;
  DBMS_OUTPUT.PUT_LINE(' Number of bytes read from source: ' ||
      (src_offset-1));
  DBMS_OUTPUT.PUT_LINE(' Number of characters written to destination: ' ||
      (dst_offset-1) );
  if (warning = dbms_lob.warn_inconvertible_char)
  then
    DBMS_OUTPUT.PUT_LINE('Warning: Inconvertible character');
  end if;

  /* load the next 1KB of the BFILE into the dst_loc2 */

  DBMS_OUTPUT.PUT_LINE(' ----------------------------' ) ;
  DBMS_OUTPUT.PUT_LINE('   Second load  ' ) ;
  DBMS_OUTPUT.PUT_LINE(' ----------------------------' ) ;

  /* Notice we are using the src_offset and lang_ctx returned from the
previous
   * load. We do not use value 1001 as the src_offset here because sometimes
```

```
the
   * actual amount read may not be the same as the amount specified.
   */

  src_osin := src_offset;
  dst_offset := 1;
  DBMS_LOB.LOADCLOBFROMFILE(dst_loc2, src_loc, amt, dst_offset, src_offset,
      cs_id, lang_ctx, warning);
  DBMS_OUTPUT.PUT_LINE(' Number of bytes read from source: ' ||
      (src_offset-src_osin) );
  DBMS_OUTPUT.PUT_LINE(' Number of characters written to destination: ' ||
      (dst_offset-1) );
  if (warning = DBMS_LOB.WARN_INCONVERTIBLE_CHAR)
  then
    DBMS_OUTPUT.PUT_LINE('Warning: Inconvertible character');
  end if;
  DBMS_LOB.FILECLOSEALL() ;

END;
/
```

5.4 BFILE APIs in Different Programmatic Interfaces

This section lists all the APIs from different Programmatic Interfaces supported by Oracle Database.

> **Note:**
>
> The PL/SQL `DBMS_LOB` package provides a rich set of operations on `BFILE`s. If you are using a different Programmatic Interface where some of these operations are not provided, then call the corresponding PL/SQL `DBMS_LOB` procedure or function.

- PL/SQL APIs for BFILEs
  This section describes the PL/SQL APIs that you can use with BFILEs.
- JDBC API for BFILEs
  This section describes the JDBC APIs that you can use to work with BFILEs.
- OCI API for BFILEs
  This section describes the OCI APIs that you can use with BFILEs.
- ODP.NET API for BFILEs
  This section describes the ODP.NET APIs that you can use with BFILEs.
- OCCI API for BFILEs
  This section describes the OCCI APIs that you can use with `BFILE`s.
- Pro*C/C++ and Pro*COBOL API for BFILEs
  This section describes Pro*C/C++ and Pro*COBOL APIs APIs you can use for BFILEs.

> **See Also:**
>
> Comparing the LOB Interfaces

5.4.1 PL/SQL APIs for BFILEs

This section describes the PL/SQL APIs that you can use with BFILEs.

> **See Also:**
>
> DBMS_LOB

Table 5-3 DBMS_LOB functions and procedures for BFILEs

Category	Function/ Procedure	Description
Sanity Checking	FILEEXISTS	Checks if the BFILE exists on the server
	FILEGETNAME	Gets the DIRECTORY object name and file name
	BFILENAME	Sets the name of a BFILE in a locator without checking if the directory or file exists
Open/Close	OPEN, FILEOPEN	Opens a file. Use OPEN instead of FILEOPEN.
	ISOPEN, FILEISOPEN	Checks if the file was opened using the input BFILE locators. Use ISOPEN instead of FILEISOPEN.
	CLOSE, FILECLOSE	Closes the file. Use CLOSE instead of FILECLOSE.
	FILECLOSEALL	Closes all previously opened files.
Read Operations	GETLENGTH	Gets the length of the BFILE
	READ	Reads data from the BFILE starting at the specified offset.
	SUBSTR	Returns part of the BFILE value starting at the specified offset.
	INSTR	Returns the matching position of the nth occurrence of the pattern in the BFILE.
Operations involving multiple locators	:= (operator)	Assigns a BFILE locator to another
	LOADCLOBFROMFILE	Loads character data from a file into a LOB
	LOADBLOBFROMFILE	Loads binary data from a file into a LOB
	LOADFROMFILE	Loads BFILE data into a LOB (deprecated)
	COMPARE	Compares the value of two BFILEs.

Example 5-3 PL/SQL API for BFILEs

```
declare
  f           BFILE;
  f2          BFILE;
  b           BLOB;
  c           CLOB;
  dest_offset NUMBER;
  src_offset  NUMBER;
  lang        NUMBER;
  warn        NUMBER;
  buffer      RAW(128);
  amt         NUMBER;
  len         NUMBER;
  pos         NUMBER;
  filename    VARCHAR2(128);
  dirname     VARCHAR2(128);
BEGIN

   /* Select out a BFILE locator */
  SELECT ad_graphic INTO f FROM print_media WHERE product_id = 1 AND ad_id =
1;

  /*---------------------------------------------------------------------*/
  /*---------------------- Sanity Checking ------------------------------*/
  /*---------------------------------------------------------------------*/

  /*-------------- Determining Whether a BFILE Exists ----------------*/
  if DBMS_LOB.FILEEXISTS(f) = 1 then
    DBMS_OUTPUT.PUT_LINE('F exists!');
  else
    DBMS_OUTPUT.PUT_LINE('F does not exist :(');
    return;
  end if;

  /*------ Getting Directory Object Name and File Name of a BFILE ----*/
  DBMS_LOB.FILEGETNAME(f, dirname, filename);
  DBMS_OUTPUT.PUT_LINE('F: directory: '|| dirname ||' filename: '|| filename);

  /*---------------------------------------------------------------------*/
  /*----------------------- Open/Close ----------------------------------*/
  /*---------------------------------------------------------------------*/

  /*----------------------- Opening a BFILE --------------------------*/
  DBMS_LOB.OPEN(f, DBMS_LOB.LOB_READONLY);

  /*-------------- Determining Whether a BFILE Is Open ---------------*/
  if DBMS_LOB.ISOPEN(f) = 1 then
    DBMS_OUTPUT.PUT_LINE('F is open!');
  else
    DBMS_OUTPUT.PUT_LINE('F is not open :(');
  end if;

  /*----------------------- Closing a BFILE --------------------------*/
  DBMS_LOB.CLOSE(f);
```

```
  /*------------ Closing All Open BFILEs with FILECLOSEALL -----------*/
  DBMS_LOB.FILECLOSEALL;

  /*-----------------------------------------------------------------*/
  /*-------------------- BFILE operations ---------------------------*/
  /*-----------------------------------------------------------------*/

  DBMS_LOB.OPEN(f, dbms_lob.lob_readonly);

  /*----------------- Getting the Length of a BFILE ------------------*/
  len := DBMS_LOB.GETLENGTH(f);
  DBMS_OUTPUT.PUT_LINE('dbms_lob.getlength: '||len);

  /*----------------------- Reading BFILE Data ----------------------*/
  amt := 15;
  DBMS_LOB.READ(f, amt, 1, buffer);
  DBMS_OUTPUT.PUT_LINE('dbms_lob.read: '||UTL_RAW.CAST_TO_VARCHAR2(buffer));

  /*--------- Reading a Portion of BFILE Data Using SUBSTR -----------*/
  buffer := DBMS_LOB.SUBSTR(f, 15, 3);
  DBMS_OUTPUT.PUT_LINE('dbms_lob.substr: '||UTL_RAW.CAST_TO_VARCHAR2(buffer));

  /*------ Checking If a Pattern Exists in a BFILE Using INSTR -------*/
  pos :=  DBMS_LOB.INSTR(f, utl_raw.cast_to_raw('BFILE'), 1, 1);
  if pos != 0 then
    DBMS_OUTPUT.PUT_LINE('dbms_lob.instr: "BFILE" word exists in position '
|| pos);
  else
    DBMS_OUTPUT.PUT_LINE('dbms_lob.instr: "BFILE" word does not exist in
file');
  end if;

  /*-----------------------------------------------------------------*/
  /*------------- Operations involving 2 locators -------------------*/
  /*-----------------------------------------------------------------*/

  /*----------------- Assigning a BFILE Locator ----------------------*/
  f2 := f; -- where f2 is also a bfile variable

  amt := 15;
  DBMS_LOB.READ(f2, amt, 1, buffer);
  DBMS_OUTPUT.PUT_LINE('assign: dbms_lob.read: '||
UTL_RAW.CAST_TO_VARCHAR2(buffer));

  /*--------------- Loading a LOB with BFILE Data --------------------*/
  /* Select out BLOB and CLOB for update so we can write to them */
  select ad_composite, ad_sourcetext into b, c
  from print_media where product_id = 1 and ad_id = 1 for update;

  /* Load BLOB from BFILE */
  dest_offset := 1;
  src_offset  := 1;

  DBMS_LOB.LOADBLOBFROMFILE(b, f, dbms_lob.lobmaxsize, dest_offset,
src_offset);
```

```
  /* Load CLOB from BFILE, for this operation is necessary to know the charset
   * id of BFILE to read it correctly */
  dest_offset := 1;
  src_offset  := 1;
  lang        := 0;
  /* Specifying the amount as DBMS_LOB.LOBMAXSIZE to copy till end of file */
  DBMS_LOB.LOADCLOBFROMFILE(c, f, DBMS_LOB.LOBMAXSIZE, dest_offset,
src_offset,
                            NLS_CHARSET_ID('utf8'), lang, warn);

  /*-------------- Comparing All or Parts of Two BFILES -------------*/
  SELECT ad_graphic INTO f2 FROM print_media WHERE product_id = 2 AND ad_id =
1;
  DBMS_LOB.OPEN(f2, dbms_lob.lob_readonly);
  if DBMS_LOB.COMPARE(f, f2, 10, 1, 1) = 0 then
    DBMS_OUTPUT.PUT_LINE('dbms_lob.compare: They are equals!!');
  else
    DBMS_OUTPUT.PUT_LINE('dbms_lob.compare: They are not equals :(');
  end if;

  -- Close just f
  DBMS_LOB.CLOSE(f);

  -- Close the rest of bfiles opended
  DBMS_LOB.FILECLOSEALL;

END;
/
```

5.4.2 JDBC API for BFILEs

This section describes the JDBC APIs that you can use to work with BFILEs.

In JDBC, the `oracle.jdbc.OracleBfile` interface provides methods for performing operations on `BFILE` data in the database. It encapsulates the `BFILE` locators, so you do not deal with locators, but instead use methods and properties provided to perform operations and get state information.

To retrieve the locator for the most current row, you must call the `getBFILE()` method on the `OracleResultSet` each time a move operation is made, depending on whether the instance is a `BFILE`.

> **See Also:**
>
> Working with LOBs and BFILEs

Table 5-4 JDBC APIs for BFILEs

Category	Function/ Procedure	Description
Sanity Checking	`boolean fileExists()`	Checks if the `BFILE` exists on the server

Table 5-4 (Cont.) JDBC APIs for BFILEs

Category	Function/ Procedure	Description
	`public java.lang.String getName()`	Gets the file name
	`public java.lang.String getDirAlias()`	Gets the `DIRECTORY` object name
Open/Close	`public void openFile()`	Opens a file.
	`public boolean isFileOpen()`	Checks if the file was opened using the input `BFILE` locators. .
	`public void closeFile()`	Closes the file. Use `CLOSE` instead of `FILECLOSE`.
Read Operations	`long length()`	Gets the length of the `BFILE`
	`public java.io.InputStream getBinaryStream()`	Reads the `BFILE` as a binary stream.
	`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.jdbc.OracleBfile, 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
Operations involving multiple locators	[use equal sign]	Assigns a `BFILE` locator to another

Example 5-4 JDBC API for BFILEs

```
static void run_query() throws Exception {

  try(
      OracleConnection con = getConnection();
      Statement       stmt = con.createStatement();
      ){

    ResultSet   rs       = null;

    OracleBfile f        = null;
    OracleBfile f2       = null;
    OracleBfile f3       = null;

    InputStream in       = null;
    String      output   = null;
    byte        buffer[] = new byte[15];
    long        pos;

    String      filename = null;
    String      dirname  = null;
    long        len      = 0;
```

```
    rs = stmt.executeQuery("select ad_graphic from print_media where
product_id = 1");
    rs.next();
    f = (OracleBfile)((OracleResultSet)rs).getBfile(1);
    rs.close();

    rs = stmt.executeQuery("select ad_graphic from print_media where
product_id = 2");
    rs.next();
    f2 = (OracleBfile)((OracleResultSet)rs).getBfile(1);
    rs.close();

    stmt.close();

    /*-------------------------------------------------------------------*/
    /*---------------------- Sanity Checking ----------------------------*/
    /*-------------------------------------------------------------------*/

    /*-------------- Determining Whether a BFILE Exists ----------------*/
    if (f.fileExists())
      System.out.println("F exists!");
    else
      System.out.println("F does not exist :(");

    /*------ Getting Directory Object Name and File Name of a BFILE ----*/
    dirname  = f.getDirAlias();
    filename = f.getName();

    System.out.println("Directory: " + dirname + " Filename: " + filename);

    /*-------------------------------------------------------------------*/
    /*----------------------- Open/Close --------------------------------*/
    /*-------------------------------------------------------------------*/

    /*----------------------- Opening a BFILE ---------------------------*/
    f.open(LargeObjectAccessMode.MODE_READONLY);

    /*-------------- Determining Whether a BFILE Is Open ---------------*/
    if (f.isOpen())
      System.out.println("F is open!");
    else
      System.out.println("F is not open :(");

    /*----------------------- Closing a BFILE ---------------------------*/
    f.close();

    /*-------------------------------------------------------------------*/
    /*--------------------- BFILE operations ----------------------------*/
    /*-------------------------------------------------------------------*/

    f.open(LargeObjectAccessMode.MODE_READONLY);

    /*----------------- Getting the Length of a BFILE -------------------*/
    len = f.length();
    System.out.println("F Length: "+len);
```

```
    /*------------------------ Reading BFILE Data ----------------------*/
    in = f.getBinaryStream();
    in.read(buffer);
    in.close();

    output = new String(buffer);
    System.out.println("Buffer: " + output);

    /*---- Checking If a Pattern Exists in a BFILE Using POSITION ------*/
    pos = f.position("BFILE".getBytes(), 1);

    if (pos != -1)
      System.out.println("\"BFILE\" word exists in position: " + pos);
    else
      System.out.println("\"BFILE\" word doesn't exist :( " );

    /*-------------------------------------------------------------------*/
    /*------------- Operations involing 2 locators ----------------------*/
    /*-------------------------------------------------------------------*/

    /*----------------- Assigning a BFILE Locator ----------------------*/
    f3 = f;

    in = f3.getBinaryStream();
    in.read(buffer);
    in.close();

    output = new String(buffer);
    System.out.println("assign: Buffer: " + output);

    /*-------------- Comparing All or Parts of Two BFILES -------------*/
    f2.open(LargeObjectAccessMode.MODE_READONLY);
    pos = f.position(f2, 1);

    if (pos != -1)
      System.out.println("f2 exists in position " + pos);
    else
      System.out.println("f2 doesn't exist in position");

    f.close();
    f2.close();
    f3.close();
  }
}
```

5.4.3 OCI API for BFILEs

This section describes the OCI APIs that you can use with BFILEs.

> **See Also:**
>
> LOB and BFILE Operations

Table 5-5 OCI APIs for BFILEs

Category	Function/ Procedure	Description
Sanity Checking	`OCILobFileExists()`	Checks if the `BFILE` exists on the server
	`OCILobFileGetName()`	Gets the `DIRECTORY` object name and the file name
	`OCILobFileSetName()`	Sets the name of a `BFILE` in a locator without checking if the directory or file exists
	`OCILobLocatorIsInit()`	Checks whether a LOB Locator is initialized
Open/Close	`OCILobOpen()` and `OCILobFileOpen()`	Opens a file. Use `OciLobOpen()` instead of `OCILobFileOpen()`.
	`OCILobIsOpen()` and `OCILobFileIsOpen()`	Checks if the file was opened using the input `BFILE` locators. Use `OCILobIsOpen()` instead of `OciLobFileIsOpen()`.
	`OCILobClose()` and `OCILobFileClose()`	Closes the file. Use `OciLobClose()` instead of `OciLobFileClose()`.
	`OCILobFileCloseAll()`	Closes all previously opened files.
Read Operations	`OCILobGetLength2()`	Gets the length of the `BFILE`
	`OCILobRead2()`	Reads data from the `BFILE` starting at the specified offset.
	`OCILobArrayRead()`	Reads data using multiple locators in one round trip.
Operations involving multiple locators	`OCILobLocatorAssign()`	Assigns a `BFILE` locator to another
	`OCILobLoadFromFile2()`	Loads `BFILE` data from a file into a LOB

Example 5-5 OCI API for BFILEs

```
static text *selstmt = (text *) "select ad_graphic, ad_composite,
ad_sourcetext from print_media where product_id = 1 and ad_id = 1 for update"
sword run_query()
{
  OCILobLocator *f = (OCILobLocator *)0;
  OCILobLocator *f2 = (OCILobLocator *)0;

  OCILobLocator *b = (OCILobLocator *)0;
  OCILobLocator *c = (OCILobLocator *)0;

  OCIStmt       *stmthp;
  OCIDefine     *defn1p = (OCIDefine *) 0;
  OCIDefine     *defn2p = (OCIDefine *) 0;
  OCIDefine     *defn3p = (OCIDefine *) 0;

  ub4            bfilelen;
  ub1            lbuf[128];
```

```
  ub8             amt = 15;
  boolean         flag = FALSE;
  ub4             id = 10;

  text            filename[128];
  ub2             filename_len;
  text            dirname[128];
  ub2             dirname_len;

  CHECK_ERROR (OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &stmthp,
                              OCI_HTYPE_STMT, (size_t) 0, (dvoid **) 0));

  /************** Allocate descriptors ***********************/
  CHECK_ERROR (OCIDescriptorAlloc((dvoid *) envhp, (dvoid **) &f,
                                  (ub4)OCI_DTYPE_FILE, (size_t) 0,
                                  (dvoid **) 0));

  CHECK_ERROR (OCIDescriptorAlloc((dvoid *) envhp, (dvoid **) &f2,
                                 (ub4)OCI_DTYPE_FILE, (size_t) 0,
                                 (dvoid **) 0));

  CHECK_ERROR (OCIDescriptorAlloc((dvoid *) envhp, (dvoid **) &b,
                                  (ub4)OCI_DTYPE_LOB, (size_t) 0,
                                  (dvoid **) 0));

  CHECK_ERROR (OCIDescriptorAlloc((dvoid *) envhp, (dvoid **) &c,
                                  (ub4)OCI_DTYPE_LOB, (size_t) 0,
                                  (dvoid **) 0));

  /********** Execute selstmt to get f, b, c **********************/
  CHECK_ERROR (OCIStmtPrepare(stmthp, errhp, selstmt,
                              (ub4) strlen((char *) selstmt),
                              (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT));

  CHECK_ERROR (OCIDefineByPos(stmthp, &defn1p, errhp, (ub4) 1, (dvoid *) &f,
                              (sb4) -1, SQLT_BFILE, (dvoid *) 0, (ub2 *) 0,
                              (ub2 *)0, (ub4) OCI_DEFAULT));
  CHECK_ERROR (OCIDefineByPos(stmthp, &defn2p, errhp, (ub4) 2, (dvoid *) &b,
                              (sb4) -1, SQLT_BLOB, (dvoid *) 0, (ub2 *) 0,
                              (ub2 *)0, (ub4) OCI_DEFAULT));
  CHECK_ERROR (OCIDefineByPos(stmthp, &defn3p, errhp, (ub4) 3, (dvoid *) &c,
                              (sb4) -1, SQLT_CLOB, (dvoid *) 0, (ub2 *) 0,
                              (ub2 *)0, (ub4) OCI_DEFAULT));

  CHECK_ERROR (OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                              (CONST OCISnapshot *) NULL, (OCISnapshot *)
NULL,
                              OCI_DEFAULT));

  /*---------------------------------------------------------------*/
  /*---------------------- Sanity Checking ------------------------*/
  /*---------------------------------------------------------------*/

  /*-------------- Determining Whether a BFILE Exists ----------------*/
  CHECK_ERROR (OCILobFileExists(svchp, errhp, f, &flag));
  printf("OCILobFileExists: %s\n", (flag)?"TRUE":"FALSE");
```

```
  /*------ Getting Directory Object Name and File Name of a BFILE ----*/
  CHECK_ERROR (OCILobFileGetName(envhp, errhp, f, (text*)dirname,
&dirname_len,
                                 (text*)filename, &filename_len));
  printf("OCILobFileGetName: Directory: %.*s Filaname: %.*s \n",
         dirname_len, dirname, filename_len, filename);

  /*-------------------------------------------------------------------*/
  /*----------------------- Open/Close --------------------------------*/
  /*-------------------------------------------------------------------*/

  /*----------------------- Opening a BFILE ---------------------------*/
  CHECK_ERROR (OCILobFileOpen(svchp, errhp, f, OCI_FILE_READONLY));
  printf("OCILobFileOpen: Works\n");

  /*-------------- Determining Whether a BFILE Is Open ---------------*/
  CHECK_ERROR (OCILobFileIsOpen(svchp, errhp, f, &flag));
  printf("OCILobFileIsOpen: %s\n", (flag)?"TRUE":"FALSE");

  /*----------------------- Closing a BFILE ---------------------------*/
  CHECK_ERROR (OCILobFileClose (svchp, errhp, f));

  /*------------ Closing All Open BFILEs with FILECLOSEALL -----------*/
  CHECK_ERROR (OCILobFileCloseAll(svchp, errhp));

  /*-------------------------------------------------------------------*/
  /*-------------------- BFILE operations -----------------------------*/
  /*-------------------------------------------------------------------*/

  CHECK_ERROR (OCILobFileOpen(svchp, errhp, f, OCI_FILE_READONLY));
  printf("OCILobFileOpen: Works\n");

  /*----------------- Getting the Length of a BFILE ------------------*/
  CHECK_ERROR (OCILobGetLength(svchp, errhp, b, &bfilelen));
  printf("OCILobGetLength: loblen: %d \n", bfilelen);

  /*------------------------ Reading BFILE Data -----------------------*/
  CHECK_ERROR (OCILobRead2(svchp, errhp, f, &amt,
                           NULL, (oraub8)1, lbuf,
                           (oraub8)sizeof(lbuf), OCI_ONE_PIECE ,(dvoid*)0,
                           NULL, (ub2)0, (ub1)SQLCS_IMPLICIT));
  printf("OCILobRead2: buf: %.*s amt: %lu\n", amt, lbuf, amt);

  /*-------------------------------------------------------------------*/
  /*------------- Operations involing 2 locators ----------------------*/
  /*-------------------------------------------------------------------*/

  /*----------------- Assigning a BFILE Locator ----------------------*/
  CHECK_ERROR (OCILobLocatorAssign(svchp, errhp, f, &f2));
  printf("OCILobLocatorAssign: Works! \n");

  amt = 15;
  CHECK_ERROR (OCILobRead2(svchp, errhp, f2, &amt,
                          NULL, (oraub8)1, lbuf,
                          (oraub8)sizeof(lbuf), OCI_ONE_PIECE ,(dvoid*)0,
```

```
                          NULL, (ub2)0, (ub1)SQLCS_IMPLICIT));
  printf("OCILobLocatorAssign: OCILobRead2: buf: %.*s amt: %lu\n", amt, lbuf,
amt);

  /*--------------- Loading a LOB with BFILE Data --------------------*/
  /* Load BLOB from BFILE. Specify amount = UB8MAXVAL to copy till end of
bfile */
  CHECK_ERROR (OCILobLoadFromFile2(svchp, errhp, b, f, UB8MAXVAL, 1,1));
  printf("OCILobLoadFromFile2: BLOB case Works\n");

  /* Load CLOB from BFILE. Specify amount = UB8MAXVAL to copy till end of
bfile.
   * Note that there is no character set conversion here. */
  CHECK_ERROR (OCILobLoadFromFile2(svchp, errhp, c, f, UB8MAXVAL, 1,1));
  printf("OCILobLoadFromFile2: CLOB case Works\n");

  /*  Close just f */
  CHECK_ERROR (OCILobFileClose (svchp, errhp, f));

  /* Close the rest of bfiles opened */
  CHECK_ERROR (OCILobFileCloseAll(svchp, errhp));

  OCIDescriptorFree((dvoid *) b, (ub4) SQLT_BLOB);
  OCIDescriptorFree((dvoid *) c, (ub4) SQLT_CLOB);
  OCIDescriptorFree((dvoid *) f, (ub4) SQLT_BFILE);
  OCIDescriptorFree((dvoid *) f2, (ub4) SQLT_BFILE);

  CHECK_ERROR (OCIHandleFree((dvoid *) stmthp, OCI_HTYPE_STMT));
}
```

5.4.4 ODP.NET API for BFILEs

This section describes the ODP.NET APIs that you can use with BFILEs.

> **See Also:**
>
> OracleBFile Class

Table 5-6 ODP.NET methods in OracleBfileClass

Category	Function/Description	Description
Sanity Checking	`FileExists`	Checks if the BFILE exists on the server
	`FileName`	Sets or gets the file name
	`DirectoryName`	Sets or gets the `DIRECTORY` object name
Open/Close	`OpenFile`	Opens a file. Use `OPEN` instead of `FILEOPEN`.
	`IsOpen`	Checks if the file was opened using the input BFILE locators. Use `ISOPEN` instead of `FILEISOPEN`.

Table 5-6 (Cont.) ODP.NET methods in OracleBfileClass

Category	Function/Description	Description
	`CloseFile`	Closes the file.
Read Operations	`Length`	Get the length of the BFILE
	`Value`	Returns the entire LOB data as a string for CLOB and a byte array for BLOB
	`Read`	Reads data from the BFILE starting at the specified offset.
	`Search`	Returns the matching position of the nth occurrence of the pattern in the BFILE.
Operations involving multiple locators	`Compare`	Compares the values of two BFILEs
	`IsEqual`	Check if two LOBs point to the same LOB data

5.4.5 OCCI API for BFILEs

This section describes the OCCI APIs that you can use with `BFILE`s.

In OCCI, 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:**
>
> Bfile Class

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 these 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
- An amount equal to zero (`0`) to read until the end of the `BFILE` in streaming mode

You cannot specify an amount larger than the length of the `BFILE`.

Table 5-7 OCCI Methods for BFILEs

Category	Function/ Procedure	Description
Sanity Checking	`fileExists()`	Checks if the `BFILE` exists on the server
	`getFileName()`	Gets the file name
	`getDirAlias()`	Gets the `DIRECTORY` object name
	`setName()`	Sets the name of a `BFILE` in a locator without checking if the directory or file exists.
	`isInitialized()`	Checks whether a `BFILE` is initialized.
Open/Close	`open()`	Opens a file.
	`isOpen()`	Checks if the file was opened using the input BFILE locators.
	`close()`	Closes the file.
Read Operations	`length()`	Gets the length of the `BFILE`
	`read()`	Reads data from the `BFILE` starting at the specified offset.
Operations involving multiple locators	(operator) =	Assigns a `BFILE` locator to another. Use the assignment operator (=) or the copy constructor.
	`Blob.copy() or Clob.copy()`	Loads `BFILE`data into a LOB

5.4.6 Pro*C/C++ and Pro*COBOL API for BFILEs

This section describes Pro*C/C++ and Pro*COBOL APIs APIs you can use for BFILEs.

> **See Also:**
>
> - Pro*C/C++ Programmer's Guide
> - Pro*COBOL Programmer's Guide

Table 5-8 Pro*C/C++ and Pro*COBOL APIs for BFILEs

Category	Function/ Procedure	Description
Sanity Checking	`DESCRIBE[FILEEXISTS]`	Checks if the `BFILE` exists on the server
	`DESCRIBE[DIRECTORY,FILENAME]`	Gets the directory object name and file name
	`FILE SET`	Sets the name of a `BFILE` in a locator without checking if the directory or file exists
Open/Close	`OPEN`	Opens a file.

Table 5-8 (Cont.) Pro*C/C++ and Pro*COBOL APIs for BFILEs

Category	Function/ Procedure	Description
	DESCRIBE[ISOPEN]	Checks if the file was opened using the input BFILE locators.
	CLOSE	Closes the file.
	FILE CLOSE ALL	Closes all previously opened files.
Read Operations	DESCRIBE[LENGTH]	Gets the length of the BFILE
	READ	Reads data from the BFILE starting at the specified offset.
Operations involving multiple locators	ASSIGN	Assigns a BFILE locator to another
	LOAD FROM FILE	Loads BFILE data into a LOB

6

SQL Semantics for LOBs

You can use various SQL mechanisms to operate on LOBs.

You can access `CLOB` and `NCLOB` data types using SQL `VARCHAR2` semantics, such as SQL string operators and functions. These techniques allow you to use LOBs directly in SQL code and provide an alternative to using LOB-specific APIs for some operations, and are beneficial in the following situations:

- When performing operations on LOBs that are relatively small in size, i.e., up to about 100K bytes
- After migrating your database from `LONG` columns to LOB data types, so that any SQL string functions contained in your existing PL/SQL application continue to work

SQL semantics are not recommended in the following situations, you must use LOB APIs instead:

- When using advanced features such as random access and piece-wise fetch.
- When performing operations on LOBs that are relatively large in size (greater than 1MB), because using SQL semantics can impact performance.

> **Note:**
>
> SQL semantics are used with persistent and temporary LOBs, and do not apply to BFILEs.

- 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 those categories of SQL functions and operators that are supported for use with LOBs.
- Detailed Semantics of SQL Operations on LOBs
  This section explains semantics of SQL operations on LOBs in details.
- Restrictions on SQL Operations on LOBs
  There are many SQL operations that are not supported on LOB columns. This section lists those operations.

6.1 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 those categories of SQL functions and operators that are supported for use with LOBs.

SQL Operations/ Functions	Support
Concatenation	Supported
Comparison	Some comparison functions are not supported for LOBs

SQL Operations/ Functions	Support
Character functions	Supported
Conversion	Some conversion functions are not supported for LOBs
Aggregate functions	Not supported
Unicode functions	Not supported

See Also:

Working with Remote LOBs in SQL and PL/SQL

The following table provides the details on each of the operations that accept `VARCHAR2` types as operands or arguments, or return a `VARCHAR2` value.

- The SQL column identifies the built-in functions and operators that are supported for `CLOB` and `NCLOB` data types. The `LENGTH` function is also supported for the `BLOB` data type.
- The PL/SQL column identifies the PL/SQL built-in functions and operators that are supported on LOBs.
- Functions designated as CNV in the SQL or PL/SQL column in the table 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 6-1 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

Table 6-1 (Cont.) SQL VARCHAR2 Functions and Operators on LOBs

Category	Operator / Function	SQL Example / Comments	SQL	PL/SQL
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
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.	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.	CNV	CNV
Conversion	DECOMPOSE	`DECOMPOSE('str' [CANONICAL | COMPATIBILITY] )` Valid for Unicode character arguments.	CNV	CNV
Conversion	HEXTORAW	`HEXTORAW(CLOB)`	No	CNV

Table 6-1 (Cont.) SQL VARCHAR2 Functions and Operators on LOBs

Category	Operator / Function	SQL Example / Comments	SQL	PL/SQL
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
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

> **See Also:**
>
> - *Oracle Database SQL Language Reference* for syntax details on SQL functions for regular expressions.
> - *Oracle Database Development Guide* for information on using regular expressions with the database.

6.2 Detailed Semantics of SQL Operations on LOBs

This section explains semantics of SQL operations on LOBs in details.

- Return Datatype for SQL Operations on LOBs
  The return data type of SQL functions on LOBs is dependent on the input parameters.
- NULL vs EMPTY LOB: Semantic Difference between LOBs and VARCHAR2
  For the `VARCHAR2` data type, a string of length zero is indistinguishable from a `NULL` value for the column.
- 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.
- CLOBs and NCLOBs Do Not Follow Session Collation Settings
  Learn about various operators on `CLOB`s and `NCLOB`s and compare the operations on `VARCHAR2` and `NVARCHAR2` variables with respect to LOBs in this section.
- Codepoint Semantics
  Codepoint semantics of the `INSTR`, `SUBSTR`, `LENGTH`, and `LIKE` functions differ depending on the data type of the argument passed to the function.

6.2.1 Return Datatype for SQL Operations on LOBs

The return data type of SQL functions on LOBs is dependent on the input parameters.

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.

Example 6-1 `CONCAT` function returning `CLOB`

```
CONCAT(CLOB, VARCHAR2)CLOB
```

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.

6.2.2 NULL vs EMPTY LOB: Semantic Difference between LOBs and VARCHAR2

For the `VARCHAR2` data type, a string of length zero is indistinguishable from a `NULL` value for the column.

For the column of a `LOB` data type, there are three possible states:

1. `NULL`: This means the column has no LOB locator.

2. Zero-length value: This can be achieved by inserting an `EMPTY LOB` into the column, or by using an API such as `DBMS_LOB.TRIM()` to trim the length to zero. In either case, there is a valid LOB locator in the column, but the LOB value length is zero.
3. Non-zero length value.

Due to this difference, the `LENGTH` function differs depending on whether the argument passed is a `LOB` or a character string:

- For a character string of length zero, the `LENGTH` function 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.

Similarly, when used with LOBs, the `IS NULL` and `IS NOT NULL` operators determine whether a LOB locator is stored in the row:

- When you pass an initialized LOB of length zero to the `IS NULL` function, `FALSE` is returned. These semantics are compliant with the SQL 92 standard.
- When you pass a `VARCHAR2` of length zero to the `IS NULL` function, `TRUE` is returned.

6.2.3 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.

The `LENGTH` function, for example, 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;
```

6.2.4 CLOBs and NCLOBs Do Not Follow Session Collation Settings

Learn about various operators on `CLOB`s and `NCLOB`s and compare the operations on `VARCHAR2` and `NVARCHAR2` variables with respect to LOBs in this section.

Standard operators that operate on `CLOB`s and `NCLOB`s without first converting them to `VARCHAR2` or `NVARCHAR2`, are marked as 'Yes' in the SQL or PL/SQL columns of Table 7-1. These operators 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.

These `REGEXP` functions 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`

> **Note:**
>
> CLOBs and NCLOBs support the default USING NLS_COMP option.

> **See Also:**
>
> *Oracle Database Reference* for more information about `NLS_COMP`

6.2.5 Codepoint Semantics

Codepoint semantics of the `INSTR`, `SUBSTR`, `LENGTH`, and `LIKE` functions 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.
- Oracle Database helps to detect half surrogate pair on read or write boundaries in case of SQL functions and in case of read/write through LOB APIs. The behavior is as follows:
  - If the starting offset is in the middle of a surrogate pair, an error is raised for both read and write operations.
  - If the read amount reads only a partial character, increment or decrement the amount by 1 to read complete characters.

    > **Note:**
    >
    > The output amount may vary from the input amount.

  - If the write amount overwrites a partial character, an error is raised to prevent the corruption of existing data caused by overwriting of a partial character in the destination `CLOB` or `NCLOB`.

> **Note:**
>
> This check only applies to the existing data in the `CLOB` or `NCLOB`. You must make sure that the incoming buffer for the write operation starts and ends in complete characters.

6.3 Restrictions on SQL Operations on LOBs

There are many SQL operations that are not supported on LOB columns. This section lists those operations.

Table 6-2 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)...`

Related Topics

- BFILE APIs
  This section discusses the different operations supported through `BFILE`s.

7

PL/SQL Semantics for LOBs

This chapter covers topics related to PL/SQL semantics for LOBs.

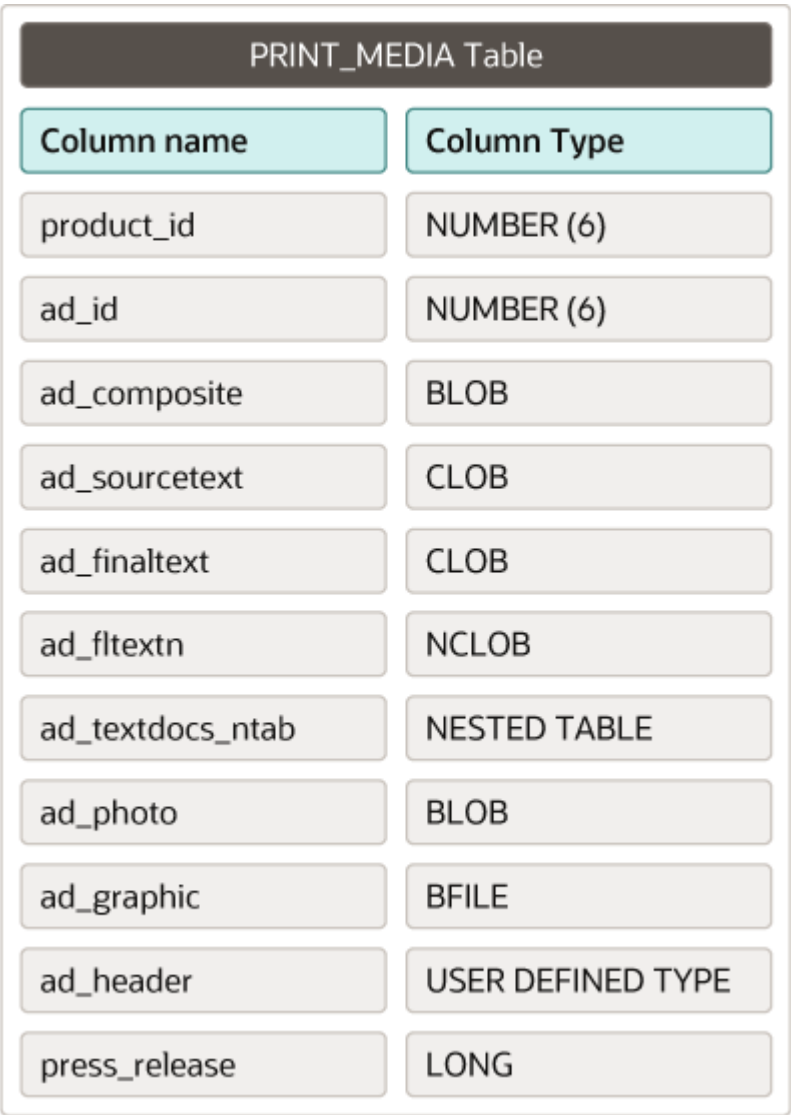
- Implicit Conversion with LOBs
  This section describes the implicit conversion process in PL/SQL from one LOB type to another LOB type or from a LOB type to a non-LOB type.
- Explicit Data Type Conversion Functions
  This section describes the explicit conversion functions in SQL and PL/SQL to convert other data types to and from `CLOB`, `NCLOB`, and `BLOB` data types.
- Temporary LOBs Created by SQL and PL/SQL Built-in Functions
  When a LOB is returned from a SQL or PL/SQL built-in function, then the result returned is a temporary LOB. Similarly, a LOB returned from a user-defined PL/SQL function or procedure, as a value or an OUT parameter, may be a temporary LOB.

7.1 Implicit Conversion with LOBs

This section describes the implicit conversion process in PL/SQL from one LOB type to another LOB type or from a LOB type to a non-LOB type.

Most of the in the following sections use `print_media` table. Following is the structure of `print_media` table:

Figure 7-1 print_media table



PRINT_MEDIA Table	
Column name	**Column Type**
product_id	NUMBER (6)
ad_id	NUMBER (6)
ad_composite	BLOB
ad_sourcetext	CLOB
ad_finaltext	CLOB
ad_fltextn	NCLOB
ad_textdocs_ntab	NESTED TABLE
ad_photo	BLOB
ad_graphic	BFILE
ad_header	USER DEFINED TYPE
press_release	LONG

- Implicit Conversion Between CLOB and NCLOB Data Types in SQL
  This section describes support for implicit conversions between CLOB and NCLOB data types.
- Implicit Conversions Between CLOB and VARCHAR2
  This section describes support for implicit conversions between `CLOB` and `VARCHAR2` data types.
- Implicit Conversions Between BLOB and RAW
  This section describes support for implicit conversions between BLOB and RAW data types.
- Guidelines and Restrictions for Implicit Conversions with LOBs
  This section describes the techniques that you use to access LOB columns or attributes using the Data Interface for LOBs.
- Detailed Examples for Implicit Conversions with LOBs
  The example in this section demonstrates using multiple `VARCHAR` and `RAW` binds in `INSERT` and `UPDATE` operations.

7.1.1 Implicit Conversion Between CLOB and NCLOB Data Types in SQL

This section describes support for implicit conversions between CLOB and NCLOB data types.

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    -- Trim the selected lob to 10 bytes
  9    DBMS_LOB.TRIM(clob1, amt);
 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:

Oracle Database SQL Language Reference for details on implicit conversions supported for all data types.

7.1.2 Implicit Conversions Between CLOB and VARCHAR2

This section describes support for implicit conversions between `CLOB` and `VARCHAR2` data types.

Implicit conversions from `CLOB` to `VARCHAR2` and from `VARCHAR2` to `CLOB` data types are supported in PL/SQL.

See Also:

SQL Semantics for LOBs for details on LOB support in SQL statements.

Note:

While this section uses `VARCHAR2` data type as an example for simplicity, other character types like `CHAR` can also participate in implicit conversions with CLOBs.

Assigning a CLOB to a VARCHAR2 in PL/SQL

When assigning a CLOB to a VARCHAR2, the data stored in the CLOB column is retrieved and stored into the VARCHAR2 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. This is consistent with VARCHAR2 semantics. After successful completion of this assignment operation, the VARCHAR2 variable holds the data as a regular character buffer. This operation can be performed in the following ways:

- SELECT persistent or temporary CLOB data into a character buffer variable such as CHAR or VARCHAR2. In a single SELECT statement, you can have more than one of such defines.
- Assign a CLOB to a VARCHAR2 or CHAR variable.
- Pass CLOB data types to built-in SQL and PL/SQL functions and operators that accept VARCHAR2 arguments, such as the INSTR function and the SUBSTR function.
- Pass CLOB data types to user-defined PL/SQL functions that accept VARCHAR2 data types.

The following example illustrates the way CLOB data is accessed when the CLOBs are treated as VARCHAR2s:

```
DECLARE
  myStoryBuf VARCHAR2(32000);
  myLob  CLOB;
BEGIN
  -- Select a LOB into a VARCHAR2 variable
  SELECT ad_sourcetext INTO myStoryBuf FROM print_media WHERE ad_id = 12001;
  DBMS_OUTPUT.PUT_LINE(myStoryBuf);
  -- Assign a LOB to a VARCHAR2 variable
  SELECT ad_sourcetext INTO myLob FROM print_media WHERE ad_id = 12001;
  myStoryBuf := myLob;
  DBMS_OUTPUT.PUT_LINE(myStoryBuf);
END;
/
```

Assigning a VARCHAR2 to a CLOB in PL/SQL

A VARCHAR2 can be assigned to a CLOB in the following scenarios:

- INSERT or UPDATE character data stored in VARCHAR2 or CHAR variables into a CLOB column. Multiple such binds are allowed in a single INSERT or UPDATE statement.
- Assign a VARCHAR2 or CHAR variable to a CLOB variable.
- Pass VARCHAR2 data types to user-defined PL/SQL functions that accept LOB data types.

```
DECLARE
  myLOB CLOB;
BEGIN
  -- Select a VARCHAR2 into a LOB variable
  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));

  -- Insert a VARCHAR2 into a lob column
  INSERT INTO print_media(product_id, ad_id, AD_SOURCETEXT) VALUES (1000, 1,
```

```
'ABCDE');

  -- Assign a VARCHAR2 to a LOB variable
  myLob := 'XYZ';
 END;
/
```

7.1.3 Implicit Conversions Between BLOB and RAW

This section describes support for implicit conversions between BLOB and RAW data types.

Most discussions related to PL/SQL semantics for implicit conversion between `CLOB` and `VARCHAR2` data types also apply to the implicit conversion process between `BLOB` and `RAW` data types, unless mentioned otherwise. However, to provide concise description, most examples in this chapter do not explicitly mention `BLOB` and `RAW` data types. The following operations involving BLOB data types support implicit conversions:

- `INSERT` or `UPDATE` binary data stored in `RAW` variables into a `BLOB` column. Multiple such binds are allowed in a single `INSERT` or `UPDATE` statement.
- `SELECT` persistent or temporary `BLOB` data into a binary buffer variable such as `RAW`. Multiple such defines are allowed in a single `SELECT` statement.
- Assign a `BLOB` to a `RAW` variable, or assign a `RAW` to a `BLOB` variable.
- Pass `BLOB` data types to built-in or user-defined PL/SQL functions defined to accept the `RAW` data type or pass the `RAW` data type to built-in or user-defined PL/SQL functions defined to accept the `BLOB` data types.

7.1.4 Guidelines and Restrictions for Implicit Conversions with LOBs

This section describes the techniques that you use to access LOB columns or attributes using the Data Interface for 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 in PL/SQL, which is 32767 bytes. For this reason, only LOBs less than 32 kilo bytes in size can be accessed by PL/SQL applications using the data interface for persistent LOBs.

If you must access a LOB with a size more than `32 kilobytes -1` bytes, using the data interface, then you must make JDBC or OCI calls from the PL/SQL code to use the APIs for piecewise insert and fetch.

Use the following guidelines for using the Data Interface to access LOB columns or attributes:

- `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.
- 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 data type, a binary data type, 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 data type, a binary data type, or a LOB locator.

- There are 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.
  - 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. For example, 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 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 data type column, and you cannot bind a buffer of binary data to a character data type 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 BLOB column if the buffer is more than 4000 bytes in size. Similarly, you cannot bind a RAW buffer to a CLOB column if the buffer is more than 4000 bytes in size.

7.1.5 Detailed Examples for Implicit Conversions with LOBs

The example in this section demonstrates using multiple VARCHAR and RAW binds in INSERT and UPDATE operations.

Example 7-1 Using Character and RAW Binds in INSERT and UPDATE Operations

The following example demonstrates using Character and RAW binds for LOB columns in INSERT and UPDATE operations

```
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;
/
```

Example 7-2 Multiple Defines for LOBs in SELECT

The following example demonstrates performing a `SELECT` operation to retrieve multiple persistent or temporary CLOBs from a SQL query into a `VARCHAR2` variable, or a `BLOB` to a `RAW` variable.

```
DECLARE
  ad_src_buffer    VARCHAR2(32000);
  ad_comp_buffer   RAW(32000);
BEGIN
  /* This retrieves the LOB columns if they are up to 32000 bytes,
   * otherwise it raises an exception */
  SELECT ad_sourcetext, ad_composite INTO ad_src_buffer, ad_comp_buffer FROM
print_media
    WHERE product_id=2004 AND ad_id=5;

  /* This retrieves the temporary LOB produced by SUBSTR if it is up to 32000
bytes,
   * otherwise it raises an exception */
  SELECT substr(ad_sourcetext, 2) INTO ad_src_buffer FROM print_media
    WHERE product_id=2004 AND ad_id=5;END;
/
```

Example 7-3 Implicit Conversions between `BLOB` and `RAW`

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.

The 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;
```

Example 7-4 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 a `VARCHAR2` is the formal parameter, or you can pass a `VARCHAR2` as an actual parameter, where a `CLOB` is the formal parameter. The same holds good for `BLOB`s and `RAW`s. 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. This holds for both built-in and user-defined procedures and functions.

The following example demonstrates implicit conversion during procedure calls:

```
CREATE OR REPLACE PROCEDURE foo(vvv IN VARCHAR2, ccc INOUT CLOB) AS
  ...
 BEGIN
  ...
  END;
  /
 DECLARE
  vvv VARCHAR2[32000] := rpad('varchar', 32000, 'varchar')
  ccc CLOB := rpad('clob', 32000, 'clob')
BEGIN
  foo(vvv, ccc); -- No implicit conversion needed here
  foo(ccc, vvv); -- Implicit conversion for both parameters done here
END;
/
```

Example 7-5 Implicit Conversion with PL/SQL built-in functions

The following example illustrates the use of `CLOB`s in PL/SQL built-in functions.

```
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;
/
```

7.2 Explicit Data Type Conversion Functions

This section describes the explicit conversion functions in SQL and PL/SQL to convert other data types to and from `CLOB`, `NCLOB`, and `BLOB` data types.

- `TO_CLOB()`: Converts from `VARCHAR2`, `NVARCHAR2`, or `NCLOB` to a `CLOB`
- `TO_NCLOB()`: Converts from `VARCHAR2`, `NVARCHAR2`, or `CLOB` to an `NCLOB`
- `TO_BLOB(varchar|clob, destcsid,[mime_type])`: Converts the object from its current character set to the given character set in `destcsid`. The resultant object is `BLOB`. Following are various ways in which you can use the conversion function:
  - `TO_BLOB(character, destcsid)`
  - `TO_BLOB(character, destcsid, mime_type)`
  - `TO_BLOB(clob, destcsid)`
  - `TO_BLOB(clob, destcsid, mime_type)`

  If the `destcsid` is 0, then it converts to the database character set ID. The parameter `mime_type` is applicable only to `INSERT` and `UPDATE` statements on Secure File `LOB` columns. If the `mime_type` parameter is used in `SELECT` statements or in temporary or BasicFile `LOB`s, then it is ignored.
- `TO_BLOB(varchar)`: Converts the input to `RAW` before converting to `BLOB`. In other words, `TO_BLOB(HEXTORAW(varchar))` and `TO_BLOB(varchar)` are equivalent.

  > **Note:**
  >
  > `TO_BLOB(CLOB)` is not supported.

- `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.

7.3 Temporary LOBs Created by SQL and PL/SQL Built-in Functions

When a LOB is returned from a SQL or PL/SQL built-in function, then the result returned is a temporary LOB. Similarly, a LOB returned from a user-defined PL/SQL function or procedure, as a value or an OUT parameter, may be a temporary LOB.

In PL/SQL, a temporary LOB has the same lifetime (duration) as the local PL/SQL program variable in which it is stored. 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 a SQL or PL/SQL function returns a temporary LOB, or if a LOB is an OUT parameter for a PL/SQL function or procedure, then you must free it as soon as you are done with it. Failure to do so may cause temporary LOB accumulation and can considerably slow down your system.

The following example illustrates implicit creation of temporary LOBs using SQL built-in functions:

```
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
```

Here is another example of implicit creation of temporary LOBs using PL/SQL built-in functions.

```
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;
/
```

Note that in the preceding example:

- In line number 7, a temporary `CLOB` is implicitly created and is pointed to by the `revisedStory CLOB` locator.
- In line number 13, `myGist` is appended to the end of the temporary LOB, which has the same effect as the following code snippet:

  ```
  DBMS_LOB.WRITEAPPEND(revisedStory, myGist, length(myGist));
  ```

In some scenarios, implicitly created temporary LOBs in PL/SQL statements can change the representation of previously defined LOB locators. The following code snippet explains this scenario:

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;
```

In the preceding example, `myStory` represents a persistent LOB column in the `print_media` table. The `DBMS_LOB.WRITE` procedure writes the data directly to the table without an `UPDATE` statement in the code.

Subsequently in line number 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 now, you must use an `UPDATE` statement. Note that for the previous persistent LOB, the `UPDATE` statement is not required.

> **See Also:**
>
> Working with Remote LOBs in SQL and PL/SQL for PL/SQL functions that support remote `LOBs` and `BFILEs`

8
Data Interface for LOBs

This chapter discusses how to perform DML and Query operations on LOBs. These operations are similar to the ones performed on traditional Character and RAW data types.

- Overview of the Data Interface for LOBs
  The data interface for LOBs includes a set of Java and OCI APIs that are extended to work with the LOB data types.
- Benefits of Using the Data Interface for LOBs
  This section discusses the benefits of the using the Data Interface for LOBs.
- Data Interface for LOBs in Java
  This section discusses the usage of data interface for LOBs in Java.
- Data Interface for LOBs in OCI
  This section discusses OCI functions included in the data interface for LOBs. These OCI functions work for LOB data types exactly the same way as they do for the `VARCHAR` data type.

8.1 Overview of the Data Interface for LOBs

The data interface for LOBs includes a set of Java and OCI APIs that are extended to work with the LOB data types.

These APIs, originally designed for use with legacy data types such as `VARCHAR2, RAW, LONG,` and `LONG RAW`, can also be used with the corresponding LOB data types shown in the following table. The table shows 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. The data interface supports data size up to two gigabytes minus one (2 GB - 1), the maximum size of an `sb4` data type.

> **Note:**
>
> The data interface works for persistent and temporary LOBs and LOBs that are attributes of objects. In this chapter *LOB columns* means LOB columns and LOB attributes.

While most of this discussion focuses on character data types, the same concepts apply to the full set of character and binary data types listed in the following table. `CLOB` also means `NCLOB` in the table.

Table 8-1 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

Table 8-1 (Cont.) Corresponding LONG and LOB Data Types in OCI

Bind or Define Type	LOB Column Type	Used For Storing
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

8.2 Benefits of Using the Data Interface for LOBs

This section discusses the benefits of the using the Data Interface for LOBs.

Following are the benefits of using the Data Interface for LOBs:

- 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` columns in your tables to LOB columns.

  > **See Also:**
  >
  > Migrating Columns to SecureFile LOBs

- The Data Interface gives you the best performance if you know the maximum size of your LOB data, and you intend to read or write the entire LOB. A piecewise `INSERT` or fetch using the data interface makes only 1 round-trip the server, as opposed to using LOB API which makes separate round-trips to get the locator and to read/write data.
- 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. Irrespective of whether the LOB data is inserted or fetched using single piece, piecewise or callbacks, it is inserted or fetched in a single round trip for multiple rows when using array binds or defines.

> **Caution:**
>
> If your application needs to perform random or piecewise read or write calls to LOBs, which means it needs to specify the offset or amount of the operation, then use the LOB APIs instead of the Data Interface.

> **See Also:**
>
> Locator Interface for LOBs

Most of the examples in the following sections use the `print_media` table. Following is the structure of the `print_media` table.

Figure 8-1 print_media Table

PRINT_MEDIA Table

Column name	Column Type
product_id	NUMBER (6)
ad_id	NUMBER (6)
ad_composite	BLOB
ad_sourcetext	CLOB
ad_finaltext	CLOB
ad_fltextn	NCLOB
ad_textdocs_ntab	NESTED TABLE
ad_photo	BLOB
ad_graphic	BFILE
ad_header	USER DEFINED TYPE
press_release	LONG

8.3 Data Interface for LOBs in Java

This section discusses the usage of data interface for LOBs in Java.

You can read and write `CLOB` and `BLOB` data using the same streaming mechanism as for `LONG` and `LONG RAW` data.

For read operations, use the `defineColumnType(nn, Types.LONGVARCHAR)` method or the `defineColumnType(nn, Types.LONGVARBINARY)` method on the persistent or temporary LOBs returned by the `SELECT` statement. This produces a direct stream on the data that is similar to `VARCHAR2` or `RAW` column.

> **Note:**
>
> 1. If you use `VARCHAR` or `RAW` as the `defineColumnType`, then the selected value will be truncated to size 32k.
> 2. Standard JDBC methods such as `getString` or `getBytes` on `ResultSet` and `CallableStatement` are not part of the Data Interface as they use the LOB locator underneath.

To insert character data into a LOB column 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. If the length of the data is known, for better performance, use the versions of `setBinaryStream()` or `setCharacterStream` functions which accept the length parameter. The data interface also supports standard JDBC methods such as `setString` or `setBytes` on `PreparedStatement` to write LOB data. It is easier to code, and in many cases faster, to use these APIs for LOB access. All these techniques reduce database round trips and result in improved performance in many cases.

The following code snippets work with all JDBC drivers:

Bind:

This is for the non-streaming mode:

```
...
String sql = "insert into print_media (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();
...
```

> **Note:**
>
> Oracle supports the non-streaming mode for strings of size up to 2 GB, but your machine's memory may be a limiting factor.

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 );
```

> **Note:**
>
> You can use the streaming interface to insert Gigabyte sized character and binary data into a LOB column.

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" );
  while( rst.next() )
     {
       String s = rst.getString( 1 );
       System.out.println( s );
     }
```

> **Note:**
>
> If the LOB size is greater than 32767 bytes, the data is truncated and no error is thrown.

For streaming mode:

```
OracleStatement stmt = (OracleStatement)(conn.createStatement());
  stmt.defineColumnType( 1, Types.LONGVARCHAR );
  ResultSet rst = stmt.executeQuery("select ad_finaltext from print_media" );
  while(rs.next()) {
    Reader reader = rs.getCharacterStream( 1 );
    int data = 0;
    data = reader.read();
    while( -1 != data ){
      System.out.print( (char)(data) );
      data = reader.read();
      }
    reader.close();
    }
```

> **Note:**
>
> Specifying the datatype as `LONGVARCHAR` lets you select the entire LOB. If the define type is set as `VARCHAR` instead of `LONGVARCHAR`, the data will be truncated at 32k.

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");
    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:**
>
> Working with Large Objects and SecureFiles

8.4 Data Interface for LOBs in OCI

This section discusses OCI functions included in the data interface for LOBs. These OCI functions work for LOB data types exactly the same way as they do for the `VARCHAR` data type.

Using these functions, you can perform `INSERT`, `UPDATE` and fetch operations in OCI on LOBs. These techniques are the same as the ones that you use on the other data types for storing 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.

- Binding a LOB in OCI
  This section describes the operations that you can use for binding the LOB data types in OCI.
- Defining a LOB in OCI
  The OCI functions discussed in this section associate a LOB type with a data type and an output buffer.

- Multibyte Character Sets Used in OCI with the Data Interface for LOBs
  This section discusses the functionality of Data Interface for LOBs when the OCI client uses a multibyte character set.
- Getting LOB Length
  This section describes how an OCI application can fetch the LOB length.
- 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.
- Using OCI Data Interface to Fetch LOB Data
  This section discusses techniques you can use to fetch data from persistent or temporary LOBs in OCI using the data interface.
- PL/SQL and C Binds from OCI
  Learn about PL/SQL and C Binds from OCI with respect to LOBs in this section.

> **See Also:**
>
> Runtime Data Allocation and Piecewise Operations in OCI

8.4.1 Binding a LOB in OCI

This section describes the operations that you can use for binding the LOB data types in OCI.

- 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 8-1.

- `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.

8.4.2 Defining a LOB in OCI

The OCI functions discussed in this section associate a LOB type with a data type and an output buffer.

The data interface for LOBs enables the following OCI functions to accept the LONG and LOB data types listed in Table 8-1.

You can use the following functions

- `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. You can use the `OCIDataServerLengthGet()` function to retrieve LOB length while using dynamic define callback.

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.

8.4.3 Multibyte Character Sets Used in OCI with the Data Interface for LOBs

This section discusses the functionality of Data Interface for LOBs when the OCI client uses a multibyte character set.

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 `VARCHAR2` data types 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.

8.4.4 Getting LOB Length

This section describes how an OCI application can fetch the LOB length.

To fetch the LOB data length, use the `OCIServerDataLengthGet()` OCI function. When you access a LOB column using the Data Interface, the server first sends the LOB data length, followed by LOB data. The server first communicates the length of the LOB data, before any conversions are made. The OCI client stores the retrieved LOB length in `define` handle. The OCI application can use the `OCIServerDataLengthGet()` function to access the LOB length.

You can access the LOB length in all fetch modes, that is, single piece, piecewise, and callback. You can also access it inside the callback without incurring a round-trip to the server. However, you should not use it before the fetch operation. In case of piecewise or callback operations, you should use it right after the first piece is fetched.

8.4.5 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.

- Performing Simple INSERT or UPDATE Operations in One Piece
  This section lists the steps to perform simple `INSERT` or `UPDATE` operations in one piece, using the data interface for LOBs.

- Using Piecewise INSERT and UPDATE Operations with Polling
  This section lists the steps to perform piecewise INSERT or UPDATE operations with polling, using the data interface for LOBs.
- Performing Piecewise INSERT and UPDATE Operations with Callback
  This section lists the steps to perform piecewise `INSERT` or `UPDATE` operations with callback, using the data interface for LOBs.
- Performing Array INSERT and UPDATE Operations
  To perform array `INSERT` or `UPDATE` operations using the data interface for LOBs, use any of the techniques discussed in this section.

8.4.5.1 Performing Simple INSERT or UPDATE Operations in One Piece

This section lists the steps to perform simple `INSERT` or `UPDATE` operations in one piece, using the data interface for LOBs.

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.

Following is an example of binding character data for `INSERT` and `UPDATE` operations on a LOB column.

```
void simple_insert()
{
/* 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);
}
```

8.4.5.2 Using Piecewise INSERT and UPDATE Operations with Polling

This section lists the steps to perform piecewise INSERT or UPDATE operations with polling, using the data interface for LOBs.

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.

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) */
}
```

8.4.5.3 Performing Piecewise INSERT and UPDATE Operations with Callback

This section lists the steps to perform piecewise `INSERT` or `UPDATE` operations with callback, using the data interface for LOBs.

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.

You do not need to supply an output callback for pure `IN` binds in OCI to SQL/PLSQL operation. Starting from Oracle Database 21c Release, you do not need to supply an input callback for pure `OUT` binds in OCI to SQL/PLSQL operation.

The following example illustrates binding character 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;
}
```

8.4.5.4 Performing Array INSERT and UPDATE Operations

To perform array `INSERT` or `UPDATE` operations using the data interface for LOBs, use any of the techniques discussed in this section.

Use the `INSERT` or `UPDATE` operations in conjunction with `OCIBindArrayOfStruct()`, or by specifying the number of iterations (*`iter`*), with *`iter`* value greater than 1, in the `OCIStmtExecute()` call. Irrespective of whether the LOB data is inserted using single piece, piecewise or callbacks, it is inserted in a single round trip for multiple rows when using array binds.

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);

}
```

8.4.6 Using OCI Data Interface to Fetch LOB Data

This section discusses techniques you can use to fetch data from persistent or temporary LOBs in OCI using the data interface.

- Performing Simple Fetch Operations in One Piece
  Follow the steps listed in this section for performing a simple fetch operation on LOBs in one piece, using the data interface for LOBs.
- Performing a Piecewise Fetch with Polling
  Follow the steps listed in this section to perform a piecewise fetch operation on a LOB column with polling, using the data interface for LOBs.
- Performing a Piecewise with Callback
  Follow the steps listed in this section to perform a piecewise fetch operation on a LOB column with callback, using the data interface for LOBs.
- Performing an Array Fetch Operation
  Use any of the techniques discussed in this section to perform an array fetch operation in OCI, using the data interface for LOBs.

8.4.6.1 Performing Simple Fetch Operations in One Piece

Follow the steps listed in this section for performing a simple fetch operation on LOBs in one piece, using the data interface for LOBs.

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.

The following example illustrates selecting a persistent LOB or temporary LOB using a simple fetch:

```
void simple_fetch()
{
  word retval;
  text buf[15000];
  /*
    This statement returns a persistent LOB, but can be modified to return a
temporary LOB
    using the query 'SELECT SUBSTR(Ad_sourcetext,5) FROM Print_media WHERE
Product_id = 2004'
  */
  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);
  }
}
```

8.4.6.2 Performing a Piecewise Fetch with Polling

Follow the steps listed in this section to perform a piecewise fetch operation on a LOB column with polling, using the data interface for LOBs.

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. Optionally, you can use `OCIServerDataLengthGet()` to get the LOB length and use it to allocate the buffer to hold the LOB data. 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.

The following example illustrates selecting a LOB column into a character 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 */
}
```

8.4.6.3 Performing a Piecewise with Callback

Follow the steps listed in this section to perform a piecewise fetch operation on a LOB column with callback, using the data interface for LOBs.

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.
6. Inside the callback, you can optionally use `OCIServerDataLengthGet()` to get the LOB length during the first fetch. You can use this value to allocate the buffer to hold LOB data

The following example illustrates selecting a LOB 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;
}
```

This example illustrates selecting a LOB column into a character buffer when using a piecewise fetch with callback, along with fetching the length of LOB data.

```
#define MAX_BUF_SZ 1048576  /* Max allocation size = 1M */
char *buffer = NULL;
ub8   buf_len = 0;

/* Define callback function */
sb4 DefineCbk(void *cbctx, OCIDefine *defnhp, ub4 iter,
              void **bufp, ub4 **alenp, ub1 *piecep,
              void **indp, ub2 **rcodep)
{
  static sword piece = 1;
  boolean isValidLen = FALSE;
  buf_len = 0;

  if (piece == 1)
  {
    OCIServerDataLengthGet(defnhp, &isValidLen, (ub8 *) &buf_len,
                           (OCIError *)cbctx, 0);

    if (buf_len > MAX_BUF_SZ)
      buf_len = MAX_BUF_SZ;

    buffer = (char *)malloc(buf_len);
    *bufp = buffer;
    *alenp = (ub4 *) &buf_len;
  }
  else
  {
    printf("Data = %s\n",buffer);
    buf_len = MAX_BUF_SZ;
  }
  piece++;
  return OCI_CONTINUE;
}

void define_callback()
{
  text      *sqlstmt = (text *)"select lobcol from lob_table";
```

```
  OCIStmtPrepare(stmthp, errhp, sqlstmt, (ub4)strlen( sqlstmt),
                 (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
  OCIDefineByPos(stmthp, &defhp1, errhp, (ub4)1, (dvoid *)0,
                 (sb4) (10 * MAX_BUF_SZ), SQLT_STR, (dvoid *) 0,
                 (ub2 *) 0, (ub2 *) 0, (ub4)OCI_DYNAMIC_FETCH);
  OCIDefineDynamic(defhp1,errhp, errhp,
                   (OCICallbackDefine)DefineCbk);
  OCIStmtExecute(svchp, stmthp, errhp, (ub4) 0, (ub4) 0,
                 (CONST OCISnapshot *) 0, (OCISnapshot *) 0,
                 (ub4) OCI_DEFAULT);
  OCIStmtFetch(stmthp, errhp, 1, OCI_FETCH_NEXT, OCI_DEFAULT);

  buffer[buf_len] = '\0';
  printf(" Data = %s\n",buffer);
  if (buffer)
    free(buffer);
}
```

8.4.6.4 Performing an Array Fetch Operation

Use any of the techniques discussed in this section to perform an array fetch operation in OCI, using the data interface for LOBs.

Use the techniques discussed below, in conjunction with `OCIDefineArrayOfStruct()`, or by specifying the number of iterations (*`iter`*), with the value of *`iter`* greater than 1, in the `OCIStmtExecute()` call. Irrespective of whether the LOB data is fetched using single piece, piecewise or callbacks, it is fetched in a single round trip for multiple rows when using array defines.

The following example illustrates selecting a LOB column into a character 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]);
  }
}
```

8.4.7 PL/SQL and C Binds from OCI

Learn about PL/SQL and C Binds from OCI with respect to LOBs in this section.

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;
```

9

Locator Interface for LOBs

The Locator Interface for LOBs refers to a set of APIs in different programmatic interfaces, which enables you to perform operations on persistent and temporary LOBs using the LOB locator.

These operations typically take an offset, or an amount parameter, or both, as input argument to facilitate efficient random and piecewise operations on the LOB.

- Before You Begin
  Learn about the concepts that you should know before using the programmatic interfaces to work on LOBs, using the LOB locator.
- PL/SQL API for LOBs
  The `DBMS_LOB` package enables you to access and make changes to LOBs in PL/SQL.
- JDBC API for LOBs
  JDBC supports standard Java interfaces `java.sql.Clob` and `java.sql.Blob` for `CLOB`s and `BLOB`s respectively.
- OCI API for LOBs
  Oracle Call Interface (OCI) LOB functions enable you to access and make changes to LOBs in C.
- ODP.NET API for LOBs
  Oracle Data Provider for .NET (ODP.NET) is an ADO.NET provider for the Oracle Database.
- OCCI API for LOBs
  OCCI provides a seamless interface to manipulate objects of user-defined types as C++ class instances.
- Pro*C/C++ and Pro*COBOL API for LOBs
  This section describes the mapping of Pro*C/C++ and Pro*COBOL locators to locator pointers to access a LOB value.

> **See Also:**
>
> BFILE APIs for operations involving the BFILE data type.

9.1 Before You Begin

Learn about the concepts that you should know before using the programmatic interfaces to work on LOBs, using the LOB locator.

- Getting a LOB Locator
  All LOB APIs need a valid LOB locator to be passed as an input. This section discusses various methods to populate LOB variables using a LOB locator.

- LOB Open and Close Operations
  The LOB APIs include operations that enable you to explicitly open and close a LOB instance.
- Read and Write at Chunk Boundaries
  To improve performance, you should perform LOB reads and writes using offsets and amount that are a multiple of the value returned by `GETCHUNKSIZE` function.
- Prefetching LOB Data and Length
  In most clients like JDBC, OCI and ODP.NET, 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. This applies to persistent LOBs, temporary LOBs, and `BFILE`s.
- Determining Character Set ID
  Some LOB APIs such as `DBMS_LOB.LOADCLOBFROMFILE`, `OCILobRead2()` and `OCILobWrite2()` take in a character set ID as an input. To determine the character set ID, you must know the character set name.
- LOB APIs
  Once a LOB variable is initialized with either a persistent or a temporary LOB locator, subsequent read operations on the LOB can be performed using APIs such as the `DBMS_LOB` package subprograms.

9.1.1 Getting a LOB Locator

All LOB APIs need a valid LOB locator to be passed as an input. This section discusses various methods to populate LOB variables using a LOB locator.

All LOB APIs need a valid LOB locator to be passed as an input. Use one of the following methods to populate a LOB variable in your application with a LOB locator:

- Persistent LOBs: First create a table with a LOB column, then insert a value into the LOB column and select out the LOB locator. To modify an existing LOB using a LOB locator, you must lock the row in the table in order to prevent other database users from writing to the LOB during a transaction.

  > **See Also:**
  >
  > – Persistent LOBs for information on how to create a a table with a LOB column and populate it.
  > – Selecting a LOB into a LOB Variable for Read Operations for information on how to select a LOB locator for LOB read operations.
  > – Selecting a LOB into a LOB Variable for Write Operations for information on how to lock the row for LOB modify operations.

- Temporary LOBs: You can create a temporary LOB by using an API like `DBMS_LOB.CREATETEMPORARY` or by invoking a SQL or PL/SQL function that returns a temporary LOB.

  > **See Also:**
  >
  > Temporary LOBs

9.1.2 LOB Open and Close Operations

The LOB APIs include operations that enable you to explicitly open and close a LOB instance.

You can open and close a persistent or temporary LOB instance of any type: `BLOB`, `CLOB` or `NCLOB`. 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

  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 a functional or domain 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 improve the performance of your application if you are doing several write operations on the LOB while it is open. Note that any index on the LOB column is not valid until you explicitly close the LOB.

  If you do not explicitly open the LOB instance, then every modification to the LOB implicitly opens and closes the LOB instance. The database performs index maintenance for any functional and domain indexes on the LOB column on each implicit close of the LOB. This means that the indexes on the LOB are updated as soon as any modification to the LOB instance is made. These indexes are always valid and can be used at any time.

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.

You must close any LOB instance that you explicitly open in the following places:

- 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 in progress in the session).

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, which means that any indexes on the LOB column are not updated. In this situation, you must rebuild your indexes on the LOB column.

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.

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.

9.1.3 Read and Write at Chunk Boundaries

To improve performance, you should perform LOB reads and writes using offsets and amount that are a multiple of the value returned by `GETCHUNKSIZE` function.

If it is appropriate for your application, then you should batch reads and writes until you have enough for an entire chunk instead of issuing several LOB read or write calls that operate on the same LOB chunk.

9.1.4 Prefetching LOB Data and Length

In most clients like JDBC, OCI and ODP.NET, 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. This applies to persistent LOBs, temporary LOBs, and `BFILE`s.

For small to medium sized LOBs, Oracle recommends setting the prefetch length such that about majority of your LOBs are smaller than the prefetch size.

LOB prefetch size can be set at the session level, and can be overwritten at the statement or the column level.

9.1.5 Determining Character Set ID

Some LOB APIs such as `DBMS_LOB.LOADCLOBFROMFILE`, `OCILobRead2()` and `OCILobWrite2()` take in a character set ID as an input. 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.

Although `UTF16` is not allowed as a database or national character set, LOB APIs support it for database conversion purposes. Use character `set ID = 1000` for `UTF16`, or in OCI, you can use `OCI_UTF16ID`.

> **See Also:**
>
> - OCIUnicodeToCharSet() for information on the `OCIUnicodeToCharSet()` function and details on OCI syntax in general.
> - Overview of Globalization Support for detailed information about implementing applications in different languages.

9.1.6 LOB APIs

Once a LOB variable is initialized with either a persistent or a temporary LOB locator, subsequent read operations on the LOB can be performed using APIs such as the `DBMS_LOB` package subprograms.

The operations supported on LOBs are divided into the following categories:

Table 9-1 Operations supported by LOB APIs

Category	Operation	Example function/procedure in DBMS_LOB or OCILob
Sanity Checking	Check if the LOB variable has been initialized	`OCILobLocatorIsInit`
	Find out if the `BLOB` or `CLOB` locator is a SecureFile	`ISSECUREFILE`
Open/Close	Open a LOB	`OPEN`
	Check is a LOB is open	`ISOPEN`
	Close the LOB	`CLOSE`
Read Operations	Get the length of the LOB	`GETLENGTH`
	Get the LOB storage limit for the database configuration	`GET_STORAGE_LIMIT`
	Get the optimum read or write size	`GETCHUNKSIZE`
	Read data from the LOB starting at the specified offset	`READ`
	Return part of the LOB value starting at the specified offset using `SUBSTR`	`SUBSTR`
	Return the matching position of a pattern in a LOB using `INSTR`	`INSTR`
Modify Operations	Write data to the LOB at a specified offset	`WRITE`
	Write data to the end of the LOB	`WRITEAPPEND`
	Erase part of a LOB, starting at a specified offset	`ERASE`
	Trim the LOB value to the specified shorter length	`TRIM`
Operations involving multiple locators	Check whether the two LOB locators are the same	`OCILobIsEqual`
	Compare all or part of the value of two LOBs	`COMPARE`
	Append a LOB value to another LOB	`APPEND`
	Copy all or part of a LOB to another LOB	`COPY`
	Assign LOB locator `src` to LOB locator `dst`	`dst:=src`, `OCILobLocatorAssign`
	Converts a `BLOB` to a `CLOB` or a `CLOB` to a `BLOB`	`CONVERTTOBLOB`, `CONVERTTOCLOB`
	Load `BFILE` data into a LOB	`LOADCLOBFROMFILE`, `LOADBLOBFROMFILE`
Operations Specific to SecureFiles	Returns options (deduplication, compression, encryption) for SecureFiles.	`GETOPTIONS`
	Sets LOB features (deduplication and compression) for SecureFiles	`SETOPTIONS`
	Gets the content string for a SecureFiles.	`GETCONTENTTYPE`

Table 9-1 (Cont.) Operations supported by LOB APIs

Category	Operation	Example function/procedure in DBMS_LOB or OCILob
	Sets the content string for a SecureFiles.	SETCONTENTTYPE
	Delete the data from the LOB at the given offset for the given length	FRAGMENT_DELETE
	Insert the given data (< 32KBytes) into the LOB at the given offset	FRAGMENT_INSERT
	Move the given amount of bytes from the given offset to the new given offset	FRAGMENT_MOVE
	Replace the data at the given offset with the given data (< 32kBytes)	FRAGMENT_REPLACE

See Also:

- Temporary LOBs
- BFILEs
- Comparing the LOB Interfaces

Note:

The DBMS_LOB package provides a rich set of operations on LOBs. If you are using a different programmatic interface, where some of these operations are not provided, then call the corresponding PL/SQL procedure or function in DBMS_LOB package.

Most of the code examples in the following sections use the print_media table with the following structure:

Figure 9-1 print_media table

9.2 PL/SQL API for LOBs

The `DBMS_LOB` package enables you to access and make changes to LOBs in PL/SQL.

> **See Also:**
>
> DBMS_LOB for more information on DBMS_LOB package.

Guidelines for Offset and Amount Parameters in DBMS_LOB Operations

The following guidelines apply to the `offset` and `amount` parameters used in the `DBMS_LOB` PL/SQL package procedures:

- For character data in all formats, either in fixed-width or variable-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 `DBMS_LOB.READ` procedure, the `amount` parameter should be less than or equal to the size of the buffer, which is limited to 32K. However, the `amount` parameter can be larger than the size of the LOB data.

Table 9-2 DBMS_LOB functions and procedures for LOBs

Category	Function/Procedure	Description
Sanity Checking	`ISSECUREFILE`	Find out if the `BLOB` or `CLOB` locator is a SecureFile
Open/Close	`OPEN`	Open a LOB
	`ISOPEN`	Check if a LOB is open
	`CLOSE`	Close the LOB
Read Operations	`GETLENGTH`	
	`GET_STORAGE_LIMIT`	
	`GETCHUNKSIZE`	
	`READ`	
	`SUBSTR`	
	`INSTR`	
Modify Operations	`WRITE`	Write data to the LOB at a specified offset
	`WRITEAPPEND`	Write data to the end of the LOB
	`ERASE`	Erase part of a LOB, starting at a specified offset
	`TRIM`	Trim the LOB value to the specified shorter length
Operations involving multiple locators	`COMPARE`	Compare all or part of the value of two LOBs
	`APPEND`	Append a LOB value to another LOB
	`COPY`	Copy all or part of a LOB to another LOB
	`dst := src`	Assign LOB locator `src` to LOB locator `dst`
	`CONVERTTOBLOB, CONVERTTOCLOB`	Converts a `BLOB` to a `CLOB` or a `CLOB` to a `BLOB`
	`LOADCLOBFROMFILE, LOADBLOBFROMFILE`	Load `BFILE` data into a LOB
Operations specific to SecureFiles	`GETOPTIONS`	Returns options (deduplication, compression, encryption) for SecureFiles.
	`SETOPTIONS`	Sets LOB features (deduplication and compression) for SecureFiles
	`GETCONTENTTYPE`	Gets the content string for a SecureFiles.

Table 9-2 (Cont.) DBMS_LOB functions and procedures for LOBs

Category	Function/Procedure	Description
	SETCONTENTTYPE	Sets the content string for a SecureFiles.
	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)

Example 9-1 PL/SQL API for LOBs

```
DECLARE
   retval    INTEGER;
   clob1     CLOB;
   clob2     CLOB;
   clob3     CLOB;
   blob1     BLOB;
   buf       VARCHAR2(32767);
   buflen    INTEGER := 32760;
   loblen1   INTEGER;

   -- Following are the variables that you need for the convertToBlob and
convertToClob functions
   amt     NUMBER := 0;
   src     NUMBER := 1 ;
   dst     NUMBER := 1 ;
   lang    NUMBER := 0;
   warn    NUMBER;

 BEGIN
   SELECT ad_sourcetext INTO clob1 FROM print_media
     WHERE product_id = 1 AND ad_id = 1;

    -- the select statement is defined with FOR UPDATE so that we can write
to it
   SELECT ad_finaltext INTO clob2 FROM print_media
     WHERE product_id = 1 AND ad_id =1 FOR UPDATE;
   /* Note that all the writes to clob2 will get reflected in the column */

   /*----------------------------------------------------------------*/
   /*---------------------- Sanity Checking -------------------------*/
   /*----------------------------------------------------------------*/
   if DBMS_LOB.ISSECUREFILE(clob1) = TRUE then
     DBMS_OUTPUT.PUT_LINE('CLOB1 is SECUREFILE');
   else
```

```
      DBMS_OUTPUT.PUT_LINE('CLOB1 is BASICFILE');
    end if;

    /*---------------------------------------------------------------------*/
    /*----------------------- Open ----------------------------------------*/
    /*---------------------------------------------------------------------*/
    /* Open clob1 for READs and clob2 for WRITES */
    DBMS_LOB.OPEN(clob1, DBMS_LOB.LOB_READONLY);
    DBMS_LOB.OPEN(clob2, DBMS_LOB.LOB_READWRITE);

    /*---------------------------------------------------------------------*/
    /*-------------------- Reading from a LOB -----------------------------*/
    /*---------------------------------------------------------------------*/
    DBMS_OUTPUT.PUT_LINE('storage limit : ' ||
dbms_lob.get_storage_limit(clob1));
    DBMS_OUTPUT.PUT_LINE('chunk size : ' || dbms_lob.getchunksize(clob1));

    loblen1 := DBMS_LOB.GETLENGTH(clob1);
    DBMS_OUTPUT.PUT_LINE('length : ' || loblen1);

    DBMS_LOB.READ(clob1, buflen, 1, buf);
    DBMS_OUTPUT.PUT_LINE('read : LOB data : ' || buf);
    DBMS_OUTPUT.PUT_LINE('New buflen : ' || buflen);

    DBMS_OUTPUT.PUT_LINE('substr : ' || dbms_lob.substr(clob1, 30, 1));
    DBMS_OUTPUT.PUT_LINE('instr : ' ||
                         DBMS_LOB.INSTR(clob1, 'review of the document', 1,
3));

    /*---------------------------------------------------------------------*/
    /*-------------------- Modifying a LOB --------------------------------*/
    /*---------------------------------------------------------------------*/
    DBMS_LOB.WRITE(clob2, buflen, 10, buf);
    DBMS_LOB.WRITEAPPEND(clob2, buflen, buf);
    buflen := 10;
    DBMS_LOB.ERASE(clob2, buflen, 10);
    DBMS_LOB.TRIM(clob2, 50);

    /* Print the LOB just modified */
    buflen := 32760;
    DBMS_LOB.READ(clob2, buflen, 1, buf);
    DBMS_OUTPUT.PUT_LINE('read : LOB data : ' || buf);
    DBMS_OUTPUT.PUT_LINE('New buflen : ' || buflen);

    /* Error because clob1 is open in READ mode */
    -- DBMS_LOB.WRITE(clob1, buflen, 10, buf);

    /*---------------------------------------------------------------------*/
    /*------------- Operations involving 2 locators ---------------------*/
    /*---------------------------------------------------------------------*/

    retval := DBMS_LOB.COMPARE(clob1, clob2, 100, 1, 1);
    if (retval < 0) then
      DBMS_OUTPUT.PUT_LINE('clob1 is smaller');
    elsif (retval = 0) then
      DBMS_OUTPUT.PUT_LINE('both clobs are equal');
```

```
   else
     DBMS_OUTPUT.PUT_LINE('clob1 is larger');
   end if;

   DBMS_OUTPUT.PUT_LINE('length before append: ' ||
DBMS_LOB.GETLENGTH(clob2));
   DBMS_LOB.APPEND(clob2, clob1);
   DBMS_OUTPUT.PUT_LINE('length after append: ' || DBMS_LOB.GETLENGTH(clob2));

   DBMS_OUTPUT.PUT_LINE('----------- LOB COPY operation --------');
   DBMS_LOB.COPY(clob2, clob1, loblen1, 100, 1);
   DBMS_OUTPUT.PUT_LINE('length after copy: ' || DBMS_LOB.GETLENGTH(clob2));

   /*---------------------------------------------------------------------*/
   /*------------------- Convert CLOB to a BLOB ----------------------*/
   /*---------------------------------------------------------------------*/
   DBMS_LOB.CREATETEMPORARY( blob1, false );
   dst := 1;
   src := 1;
   amt := 5;
   DBMS_LOB.CONVERTTOBLOB(blob1, clob2, amt, dst, src, DBMS_LOB.DEFAULT_CSID,
                          lang, warn);
   DBMS_OUTPUT.PUT_LINE(' Source offset returned      ' || src ) ;
   DBMS_OUTPUT.PUT_LINE(' Destination offset returned ' || dst ) ;
   DBMS_OUTPUT.PUT_LINE(' Length of CLOB      ' ||
dbms_lob.getlength(clob2) ) ;
   DBMS_OUTPUT.PUT_LINE(' Length of BLOB      ' ||
dbms_lob.getlength(blob1) ) ;
   DBMS_OUTPUT.PUT_LINE(' Warning returned    ' || warn);
   DBMS_OUTPUT.PUT_LINE(' OUTPUT BLOB contents = ' || rawtohex(blob1));

   /*---------------------------------------------------------------------*/
   /*-------------------- Convert BLOB to a CLOB ----------------------*/
   /*---------------------------------------------------------------------*/
   DBMS_LOB.CREATETEMPORARY( clob3, false );
   dst := 1;
   src := 1;
   amt := 4;
   DBMS_LOB.CONVERTTOCLOB(clob3, blob1, amt, dst, src, DBMS_LOB.DEFAULT_CSID,
                          lang, warn);
   DBMS_OUTPUT.PUT_LINE(' Source offset returned      ' || src ) ;
   DBMS_OUTPUT.PUT_LINE(' Destination offset returned ' || dst ) ;
   DBMS_OUTPUT.PUT_LINE(' Length of BLOB      ' ||
DBMS_LOB.GETLENGTH(blob1) ) ;
   DBMS_OUTPUT.PUT_LINE(' Length of CLOB      ' ||
DBMS_LOB.GETLENGTH(clob3) ) ;
   DBMS_OUTPUT.PUT_LINE(' Warning returned    ' || warn);
   DBMS_OUTPUT.PUT_LINE(' INPUT BLOB contents =  ' || rawtohex(blob1));
   DBMS_OUTPUT.PUT_LINE(' OUTPUT CLOB contents = ' || clob3);

   /*---------------------------------------------------------------------*/
   /*------------------------ Close ----------------------------------*/
   /*---------------------------------------------------------------------*/
   DBMS_OUTPUT.PUT_LINE('------------- CLOSE ---------------');
   DBMS_LOB.CLOSE(clob2);
```

```
   if (DBMS_LOB.ISOPEN(clob1) = 1) then
     DBMS_LOB.CLOSE(clob1);
   END if;

   COMMIT;
END;
/
```

Example 9-2 PL/SQL APIs for SecureFile specific operations

```
conn pm/pm

-- alter the table to make lob storage as securefile
-- assume tablespace tbs_1 is ASSM
alter table print_media move
lob(ad_composite)  store as securefile (deduplicate compress tablespace tbs_1)
lob(ad_sourcetext) store as securefile (compress tablespace tbs_1)
lob(ad_finaltext)  store as securefile (compress tablespace tbs_1)
lob(ad_photo)      store as securefile (tablespace tbs_1);

SET SERVEROUTPUT ON

DECLARE
 clob1             CLOB;
 blob1             BLOB;
 result            BINARY_INTEGER;

 /* --- variables for setcontenttype, getcontenttype ----*/
 get_media_type    VARCHAR2(128);
 set_media_type    VARCHAR2(128);

 /* --- variables for delta operations --------*/
 amount            INTEGER;
 offset            INTEGER;
 buffer            VARCHAR2(30);
 readbuf           VARCHAR2(50);
 read_amt          INTEGER;
 src_offset        INTEGER;
 dest_offset       INTEGER;
 amount_old        INTEGER;
BEGIN
 -- fetch clob, blob values
 SELECT ad_sourcetext, ad_composite
 INTO   clob1, blob1
 FROM   print_media
 WHERE  product_id = 2056 FOR UPDATE;

 /*-------------------------------------------------------------------*/
 /*---------------------- Get Options --------------------------------*/
 /*-------------------------------------------------------------------*/
 -- check whether compress option is enabled
 result := DBMS_LOB.GETOPTIONS(clob1, DBMS_LOB.OPT_COMPRESS);
 DBMS_OUTPUT.PUT_LINE('Get compress option on ad_sourcetext: '||result);

 -- check whether compress + deduplicate is enabled
```

```
 result := DBMS_LOB.GETOPTIONS(blob1, DBMS_LOB.OPT_DEDUPLICATE +
                                      DBMS_LOB.OPT_COMPRESS);
 DBMS_OUTPUT.PUT_LINE('Get compress + deduplicate option on ad_composite: '||
result);

 /*---------------------------------------------------------------------*/
 /*---------------------- Set Options ----------------------------------*/
 /*---------------------------------------------------------------------*/
 -- turn off compression
 DBMS_LOB.SETOPTIONS(clob1, DBMS_LOB.OPT_COMPRESS, DBMS_LOB.COMPRESS_OFF);
 -- getoptions should be 0 now
 result := DBMS_LOB.GETOPTIONS(clob1, DBMS_LOB.OPT_COMPRESS);
 DBMS_OUTPUT.PUT_LINE('Compress option on clob1: '||result);

 -- turn off deduplication
 DBMS_LOB.SETOPTIONS(blob1, DBMS_LOB.OPT_DEDUPLICATE,
DBMS_LOB.DEDUPLICATE_OFF);
 -- getoptions should be 0 now
 result := DBMS_LOB.GETOPTIONS(blob1, DBMS_LOB.OPT_DEDUPLICATE);
 DBMS_OUTPUT.PUT_LINE('Deduplicate option on blob1: '||result);

 /*---------------------------------------------------------------------*/
 /*----------- Getcontenttype, Setcontenttype ------------------------*/
 /*---------------------------------------------------------------------*/
 -- get contenttype -- should be null as content type is not set yet
 DBMS_OUTPUT.PUT_LINE(CHR(10)||'clob1 contenttype: ' ||
dbms_lob.getcontenttype(clob1));

 set_media_type := 'text/plain';
 DBMS_LOB.SETCONTENTTYPE(clob1, set_media_type);

 DBMS_OUTPUT.PUT_LINE('Clob1 contenttype: ' ||
dbms_lob.getcontenttype(clob1));

 -- setcontenttype for blob
 DBMS_OUTPUT.PUT_LINE('blob1 contenttype: ' ||
dbms_lob.getcontenttype(blob1));
 set_media_type := 'photo/jpeg';
 DBMS_LOB.SETCONTENTTYPE(blob1, set_media_type);

 get_media_type := DBMS_LOB.GETCONTENTTYPE(blob1);
 DBMS_OUTPUT.PUT_LINE('Blob1 contenttype: ' || get_media_type);

 /*---------------------------------------------------------------------*/
 /*---------------------- Fragment Operations ------------------------*/
 /*------------------ Print Before Fragment Operations --------------*/
 read_amt  := 40;
 DBMS_LOB.READ(clob1, read_amt, 1, readbuf);
 DBMS_OUTPUT.PUT_LINE(CHR(10)||'Clob1 before fragment insert: '|| readbuf);
 DBMS_OUTPUT.PUT_LINE(CHR(10)||'Length of clob1 before fragment operations:
'|| dbms_lob.getlength(clob1));

 /*---------------------- Fragment Delete ----------------------------*/
 amount := 100;
 offset := 10;
 DBMS_LOB.FRAGMENT_DELETE(clob1, amount, offset);
```

```
 /*---------------------- Fragment Insert ----------------------------*/
 amount    := 29;
 offset    := 1;
 buffer    := '#Verify lob Delta operations#';
 DBMS_LOB.FRAGMENT_INSERT(clob1, amount, offset, buffer);

 /*---------------------- Fragment Move -----------------------------*/
 amount      := 29;
 src_offset  := 100;
 dest_offset := 1;

 -- fragment move
 DBMS_LOB.FRAGMENT_MOVE(clob1, amount, src_offset, dest_offset);

 /*---------------------- Fragment Replace --------------------------*/
 amount      := 25;
 amount_old  := 29;
 offset      := 100;
 buffer      := '$Verify fragment replace$';

 DBMS_LOB.FRAGMENT_REPLACE(clob1, amount_old, amount, offset,buffer);

 COMMIT;

 /*------------------ Verify After Fragment Operations --------------*/
 read_amt  := 40;
 DBMS_LOB.READ(clob1, read_amt, 1, readbuf);
 DBMS_OUTPUT.PUT_LINE(CHR(10)||'Clob1 after delta insert: '|| readbuf);
 DBMS_OUTPUT.PUT_LINE(CHR(10)||'Length of clob1 after fragment operations:
'|| dbms_lob.getlength(clob1));

EXCEPTION
 WHEN OTHERS THEN
   DBMS_OUTPUT.PUT_LINE(sqlerrm);
END;
/
```

9.3 JDBC API for LOBs

JDBC supports standard Java interfaces `java.sql.Clob` and `java.sql.Blob` for `CLOB`s and `BLOB`s respectively.

In JDBC, you do not deal with locators but instead use methods and properties in the Java APIs to perform operations on LOBs.

When `BLOB` and `CLOB` objects are retrieved as a part of an `ResultSet`, 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. You must call `getBLOB()`, `getCLOB()`, or `getBFILE()` on the `ResultSet` each time a move operation is made depending on whether the instance is a `BLOB`, `CLOB` or `BFILE`.

> **See Also:**
>
> Working with LOBs and BFILEs

Prefetching of LOB Data

When using the JDBC client, 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. This applies to persistent LOBs, temporary LOBs, and `BFILE`s. For small to medium sized LOBs, Oracle recommends setting the prefetch length such that about majority of your LOBs are smaller than the prefetch size.

LOB prefetch size can be set at the session level, and can be overwritten at the statement or the column level. The prefetch size values can be:

- -1 to disable prefetching
- 0 to enable prefetching for metadata only
- 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.

Use `prop.setProperty` to set the prefetch size for the session. The default session prefetch size is 32k for the JDBC Thin Driver.

```
prop.setProperty("oracle.jdbc.defaultLobPrefetchSize","64000");
```

You can overwrite the session level default prefetch size at the statement level as follows:

```
((OracleStatement)stmt).setLobPrefetchSize(100000);
```

You can use the following code snippet to fetch the prefetch size of a statement:

```
int pf = ((OracleStatement)stmt).getLobPrefetchSize() ;
```

You can overwrite the session level default prefetch size at the column level as follows:

```
((OracleStatement)stmt).defineColumnType(1, OracleTypes.CLOB, /
*lobPrefetchSize*/
      32000);
```

> **Note:**
>
> About Prefetching LOB Data

Table 9-3 JDBC methods for LOBs

Category	Function / Procedure	Description
Miscellaneous	`empty_lob()`	Creates an empty LOB
	`isSecureFile()`	Finds out if the `BLOB` or `CLOB` locator is a SecureFile

Table 9-3 (Cont.) JDBC methods for LOBs

Category	Function / Procedure	Description
Open/Close	`open()`	Open a LOB
	`isOpen()`	Check if a LOB is open
	`close()`	Close the LOB
Read Operations	`length()`	Get the length of the LOB
	`getChunkSize()`	Get the optimum read/write size
	`getBytes()`	Read data from the BLOB starting at the specified offset
	`getBinaryStream()`	Streams the BLOB as a binary stream
	`getChars()`	Read data from the CLOB starting at the specified offset
	`getCharacterStream()`	Streams the CLOB as a character stream
	`getAsciiStream()`	Streams the CLOB as an ASCII stream
	`getSubString()`	Return part of the LOB value starting at the specified offset
	`position()`	Return the matching position of a pattern in a LOB
Modify Operations	`setBytes()`	Write data to the BLOB at a specified offset
	`setBinaryStream()`	Sets a binary stream that can be used to write to the `BLOB` value
	`setString()`	Write data to the CLOB at a specified offset
	`setCharacterStream()`	Sets a character stream that can be used to write to the `CLOB` value
	`setAsciiStream()`	Sets an ASCII stream that can be used to write to the `CLOB` value
	`truncate()`	Trim the LOB value to the specified shorter length
Operations involving multiple locators	dst = src	Assign LOB locator src to LOB locator dst

Example 9-3 JDBC API for LOBs

```
static void jdbc_lob_apis() throws Exception {

  System.out.println("Persistent LOBs Test in JDBC "+ TYPE);
  try(
      Connection con = getConnection();
      Statement       stmt = con.createStatement();
      )
  {

    ResultSet    rs      = null;
    Clob  c1      = null;
```

```
    Clob  c2      = null;
    Reader      in      = null;
    long        pos     = 0;
    long        len     = 0;

    rs = stmt.executeQuery("select ad_sourcetext from print_media where
product_id = 1");
    rs.next();
    c1 = rs.getCLOB(1);
    OracleClob c11 = (OracleClob)c1;
    rs.close();

    /*-------------------------------------------------------------------*/
    /*---------------------- Sanity Checking ----------------------------*/
    /*-------------------------------------------------------------------*/
    if (c11.isSecureFile())
      System.out.println("C1 is a Securefile LOB");
    else
      System.out.println("C1 is a Basicfile LOB");


    /*-------------------------------------------------------------------*/
    /*----------------------- Open/Close --------------------------------*/
    /*-------------------------------------------------------------------*/

    /*----------------------- Opening a CLOB ----------------------------*/
    c11.open(LargeObjectAccessMode.MODE_READONLY);

    /*-------------- Determining Whether a CLOB Is Open ----------------*/
    if (c11.isOpen())
      System.out.println("C11 is open!");
    else
      System.out.println("C11 is not open");

    /*----------------------- Closing a CLOB ----------------------------*/
    c11.close();

    /*-------------------------------------------------------------------*/
    /*--------------------- Reading from a LOB --------------------------*/
    /*-------------------------------------------------------------------*/

    /*------------------------ Get CLOB Length --------------------------*/
    len = c1.length();
    System.out.println("CLOB length = " + len);

    /*------------------------ Reading CLOB Data ------------------------*/
    char[] readBuffer = new char[6];
    in = c1.getCharacterStream();
    in.read(readBuffer,0,5);
    in.close();
    String lobContent = new String(readBuffer);
    System.out.println("Buffer with LOB contents: " + lobContent);

    /*------------------------ Substr of a CLOB -------------------------*/
    String subs = c1.getSubString(2, 5);
    System.out.println("LOB substring: " + subs);
```

```
    /*----------------------- Search for a pattern  --------------------*/
    pos = c1.position("aaa", 1);
    System.out.println("Pattern matched at position = " + pos);

    /*-----------------------------------------------------------------*/
    /*------------------- Modifying a LOB -----------------------------*/
    /*-----------------------------------------------------------------*/

    rs = stmt.executeQuery("select ad_sourcetext from print_media where
product_id = 1 for update");
    rs.next();
    c2 = rs.getClob(1);
    OracleClob c22 = (OracleClob)c2;

    /*-------------------- Write to a CLOB ----------------------------*/
    c22.open(LargeObjectAccessMode.MODE_READWRITE);
    c2.setString(3,"modified");
    String msubs = c2.getSubString(1, 15);
    System.out.println("Modified LOB substring: " + msubs);

    /*-------------------- Truncate a CLOB ----------------------------*/
    c2.truncate(20);
    len = c2.length();
    System.out.println("Truncated LOB len = " + len);
    c22.close();

  }
}
```

9.4 OCI API for LOBs

Oracle Call Interface (OCI) LOB functions enable you to access and make changes to LOBs in C.

> **See Also:**
>
> LOB and BFILE Operations

Prefetching LOB Data in OCI

When using the OCI client, 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. This applies to persistent LOBs, temporary LOBs, and `BFILE`s. For small to medium sized LOBs, Oracle recommends setting the prefetch length such that about majority of your LOBs are smaller than the prefetch size.

LOB prefetch size can be set at the session level, and can be overwritten at the statement or the column level.

Use the `OCIAttrSet()` function to set the prefetch size for the session. The default session prefetch size is 0.

```
default_lobprefetch_size = 32000;
OCIAttrSet(authp, OCI_HTYPE_SESSION, &default_lobprefetch_size , 0,
      OCI_ATTR_DEFAULT_LOBPREFETCH_SIZE, errhp));
```

You can overwrite the session level default prefetch size at the column level. For this, you should first set the column level attribute `OCI_ATTR_LOBPREFETCH_LENGTH` to `TRUE` and then set the column level prefetch size attribute `OCI_ATTR_LOBPREFETCH_SIZE` in the define handle to override the session level default lob prefetch size. The following code snippet demonstrates how to set the prefetch size at session level:

```
prefetch_length = TRUE;
status = OCIAttrSet(defhp, OCI_HTYPE_DEFINE, &prefetch_length, 0,
OCI_ATTR_LOBPREFETCH_LENGTH, errhp);

lpf_size = 32000;
OCIAttrSet(defhp, OCI_HTYPE_DEFINE, &lpf_size, sizeof(ub4),
OCI_ATTR_LOBPREFETCH_SIZE, errhp);
```

You can use the following code snippet to get the prefetch size of a define:

```
ub4 get_lpf_size = 0;
OCIAttrGet(defhp, OCI_HTYPE_DEFINE,&get_lpf_size,
0,OCI_ATTR_LOBPREFETCH_SIZE, errhp);
```

> **See Also:**
>
> User Session Handle Attributes

Fixed-width and Varying-width Character Set Rules for OCI

In OCI, for fixed-width client-side character sets, the following rules apply:

- `CLOB`s and `NCLOB`s: `offset` and `amount` parameters are always in characters
- `BLOB`s and `BFILE`s: `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:

  - `CLOB`s and `NCLOB`s: in characters
  - `BLOB`s and `BFILE`s: 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

- **OCILobGetLength2()**:

  Regardless of whether the client-side character set is varying-width, the output length is as follows:

  - `CLOB`s and `NCLOB`s: in characters
  - `BLOB`s and `BFILE`s: in bytes

- **OCILobRead2()**:

  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 buffer `bufp`.

- **OCILobWrite2()**: With client-side character set of varying-width, `CLOB`s and `NCLOB`s:

  - **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`.

- **Amount Operation for OCILob Operations**: For operations such as `OCILobCopy2()`, `OCILobErase2()`, `OCILobLoadFromFile2()`, and `OCILobTrim2()`, the `amount` parameter is in characters for `CLOB`s and `NCLOB`s irrespective of the client-side character set because all these operations refer to the amount of LOB data on the server.

> **See Also:**
>
> Overview of Globalization Support

Amount Parameter

When using the `OCILobRead2()` and `OCILobWrite2()` functions, in order to read or write the entire LOB. you can set the input `amount` parameter as follows:

Table 9-4 Special Amount Parameter Setting to Read/Write the entire LOB

	OCILobRead2	OCILobWrite2
`piece = OCI_ONE_PIECE`	Set amount to `UB8MAXVAL` to read the entire LOB	
Streaming with Polling	Set amount to 0 to read entire data in a loop	Set amount to 0 to continue writing buffer size amount until `OCI_LAST_PIECE`
Streaming with Callback	Set amount 0 to ensure that the callback is called until the entire data is read	Set amount to 0 to ensure that the callback is called until `OCI_LAST_PIECE` is returned by the callback

Table 9-5 OCI Attributes on the OCILobLocator

ATTRIBUTE	OCIAttrSet	OCIAttrGet
OCI_ATTR_LOBEMPTY	Sets the descriptor to be empty LOB	N/A
OCI_ATTR_LOB_REMOTE	N/A	set to TRUE if the lob locator is from a remote database, set to FALSE otherwise
OCI_ATTR_LOB_TYPE	N/A	holds the LOB type (CLOB / BLOB / BFILE)
OCI_ATTR_LOB_IS_VALUE	N/A	set to TRUE if it is from a value LOB, otherwiseFALSE
OCI_ATTR_LOB_IS_READONLY	N/A	set to TRUE if it is a read-only LOB, otherwise FALSE
OCI_ATTR_LOBPREFETCH_LENGTH	When set to TRUE the attribute will enable prefetching and will prefetch the LOB length and the chunk size while performing select operation of LOB locator	set to TRUE if prefetching is turned on for the locator.
OCI_ATTR_LOBPREFETCH_SIZE	Overrides the default prefetch size for LOBs. Has a prerequisite of the OCI_ATTR_LOBPREFETCH_LENGTH attribute to be set to TRUE.	Returns the prefetch size of the locator.

Table 9-6 OCI Functions for LOBs

Category	Function/Procedure	Description
Sanity Checking	OCILobLocatorIsInit()	Checks whether a LOB locator is initialized.
Open/Close	OCILobOpen()	Open a LOB
	OCILobIsOpen()	Check if a LOB is open
	OCILobClose()	Close the LOB
Read Operations	OCILobGetLength2()	Get the length of the LOB
	OCILobGetStorageLimit()	Get the LOB storage limit for the database configuration
	OCILobGetChunkSize()	Get the optimum read / write size
	OCILobRead2()	Read data from the LOB starting at the specified offset
	OCILobArrayRead()	Reads data using multiple locators in one round trip.
	OCILobCharSetId()	Returns the character set ID of a LOB.
	OCILobCharSetForm()	Returns the character set form of a LOB.
Modify Operations	OCILobWrite2()	Write data to the LOB at a specified offset
	OCILobArrayWrite()	Writes data using multiple locators in one round trip.
	OCILobWriteAppend2()	Write data to the end of the LOB

Table 9-6 (Cont.) OCI Functions for LOBs

Category	Function/Procedure	Description
	OCILobErase2()	Erase part of a LOB, starting at a specified offset
	OCILobTrim2()	Trim the LOB value to the specified shorter length
Operations involving multiple locators	OCILobIsEqual()	Checks whether two LOB locators refer to the same LOB.
	OCILobAppend()	Append a LOB value to another LOB
	OCILobCopy2()	Copy all or part of a LOB to another LOB
	OCILobLocatorAssign()	Assign one LOB to another
	OCILobLoadFromFile2()	Load BFILE data into a LOB
Operations specific to SecureFiles	OCILObGetOptions()	Returns options (deduplication, compression, encryption) for SecureFiles.
	OCILObSetOptions()	Sets LOB features (deduplication and compression) for SecureFiles
	OCILobGetContentType()	Gets the content string for a SecureFiles
	OCILobSetContentType()	Sets a content string in a SecureFiles

Example 9-4 OCI API for LOBs

```
/* Define SQL statements to be used in program. */
#define LOB_NUM_QUERIES 2

static text *selstmt[LOB_NUM_QUERIES] = {
    (text *) "select ad_sourcetext from print_media where product_id = 1", /*
0 */
    (text *) "select ad_sourcetext from print_media where product_id = 2 for
update",
};

sword run_query(ub4 index, ub2 dty)
{
  OCILobLocator *c1 = (OCILobLocator *)0;
  OCILobLocator *c2 = (OCILobLocator *)0;

  OCIStmt       *stmthp;
  OCIDefine     *defn1p = (OCIDefine *) 0;
  OCIDefine     *defn2p = (OCIDefine *) 0;
  OCIBind       *bndp1  = (OCIBind *)   0;
  OCIBind       *bndp2  = (OCIBind *)   0;

  ub8            loblen;
  ub1            lbuf[128];
  ub1            inbuf[9] = "modified";
  ub1            inbuf_len = 8;
  ub8            amt = 15;
```

```
  ub8             bamt = 0;
  ub4             csize = 0;
  ub8             slimit = 0;
  boolean         flag = FALSE;
  boolean         boolval = TRUE;
  ub4             id = 10;


  CHECK_ERROR (OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &stmthp,
                              OCI_HTYPE_STMT, (size_t) 0, (dvoid **) 0));

  /************** Allocate descriptors **********************/
  CHECK_ERROR (OCIDescriptorAlloc((dvoid *) envhp, (dvoid **) &c1,
                                  (ub4)OCI_DTYPE_FILE, (size_t) 0,
                                  (dvoid **) 0));

  CHECK_ERROR (OCIDescriptorAlloc((dvoid *) envhp, (dvoid **) &c2,
                                 (ub4)OCI_DTYPE_FILE, (size_t) 0,

  /********** Execute selstmt[0] to get c1  **********************/
  CHECK_ERROR (OCIStmtPrepare(stmthp, errhp, selstmt[0],
                              (ub4) strlen((char *) selstmt[0]),
                              (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT));

  CHECK_ERROR (OCIDefineByPos(stmthp, &defn1p, errhp, (ub4) 1, (dvoid *) &c1,
                              (sb4) -1, SQLT_CLOB, (dvoid *) 0, (ub2 *) 0,
                              (ub2 *)0, (ub4) OCI_DEFAULT));

  CHECK_ERROR (OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                              (CONST OCISnapshot *) NULL, (OCISnapshot *)
NULL,
                              OCI_DEFAULT));

  /********** Execute selstmt[1] to get c2 **********************/
  CHECK_ERROR (OCIStmtPrepare(stmthp, errhp, selstmt[1],
                              (ub4) strlen((char *) selstmt[1]),
                              (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT));

  CHECK_ERROR (OCIDefineByPos(stmthp, &defn1p, errhp, (ub4) 1, (dvoid *) &c2,
                              (sb4) -1, SQLT_CLOB, (dvoid *) 0, (ub2 *) 0,
                              (ub2 *)0, (ub4) OCI_DEFAULT));

  CHECK_ERROR (OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                              (CONST OCISnapshot *) NULL, (OCISnapshot *)
NULL,
                              OCI_DEFAULT));


  /*---------------------------------------------------------------------*/
  /*---------------------- Sanity Checking ----------------------------*/
  /*---------------------------------------------------------------------*/

  CHECK_ERROR (OCILobLocatorIsInit(envhp, errhp, (OCILobLocator *) c1,
                                   &boolval));
  if (boolval)
    printf("LOB locator is initialized! \n");
```

```
else
  printf("LOB locator is NOT initialized \n");


/*---------------------------------------------------------------------*/
/*----------------------- Open/Close ----------------------------------*/
/*---------------------------------------------------------------------*/

/*----------------------- Opening a CLOB ------------------------------*/
CHECK_ERROR (OCILobOpen(svchp, errhp, c1, (ub1)OCI_LOB_READONLY));
printf("OCILobOpen: Works\n");
/*-------------- Determining Whether a CLOB Is Open ----------------*/
CHECK_ERROR (OCILobIsOpen(svchp, errhp, c1,  &boolval));
printf("OCILobIsOpen: %s\n", (boolval)?"TRUE":"FALSE");

/*----------------------- Closing a LOB -------------------------------*/
CHECK_ERROR (OCILobClose(svchp, errhp, c1));
printf("OCILobClose: Works\n");

/*---------------------------------------------------------------------*/
/*--------------------- LOB Read Operations ---------------------------*/
/*---------------------------------------------------------------------*/

printf("OCILobFileOpen: Works\n");

/*----------------- Getting the Length of a LOB --------------------*/
CHECK_ERROR (OCILobGetLength2(svchp, errhp, c1, &loblen));
printf("OCILobGetLength2: loblen: %d \n", loblen);

/*----------------- Getting the Storage Limit of a LOB -------------*/
CHECK_ERROR (OCILobGetStorageLimit(svchp, errhp, c1, &slimit));
printf("OCILobGetStorageLimit: storage limit: %ld \n", slimit);

/*----------------- Getting the Chunk Size of a LOB -----------------*/
CHECK_ERROR (OCILobGetChunkSize(svchp, errhp, c1, &csize));
printf("OCILobGetChunkSize: storage limit: %d \n", csize);

/*------------------------ Reading LOB Data ------------------------*/
CHECK_ERROR (OCILobRead2(svchp, errhp, c1, &amt,
                         NULL, (oraub8)1, lbuf,
                         (oraub8)sizeof(lbuf), OCI_ONE_PIECE ,(dvoid*)0,
                         NULL, (ub2)0, (ub1)SQLCS_IMPLICIT));
printf("OCILobRead2: buf: %.*s amt: %lu\n", amt, lbuf, amt);

/*---------------------------------------------------------------------*/
/*--------------------- Modifying a LOB -------------------------------*/
/*---------------------------------------------------------------------*/

/*----------------------- Writing Data to LOB ------------------------*/
amt = 8;
CHECK_ERROR (OCILobWrite2(svchp, errhp, c2, &bamt, &amt, 1,
              (dvoid *) inbuf, (ub8)inbuf_len, OCI_ONE_PIECE, (dvoid *)0,
              (OCICallbackLobWrite2)0,
              (ub2) 0, (ub1) SQLCS_IMPLICIT));

/*----------------------- Write Append to a LOB ---------------------*/
```

```
  /* Append 8 characters */
  amt = 8;
  CHECK_ERROR (OCILobWriteAppend2(svchp, errhp, c2, &bamt, &amt,
                (dvoid *) inbuf, (ub8)inbuf_len, OCI_ONE_PIECE, (dvoid *)0,
                (OCICallbackLobWrite2)0,
                (ub2) 0, (ub1) SQLCS_IMPLICIT));

  /*---------------------- Erase part of LOB contents ----------------*/
  /* Erase 5 characters */
  amt = 5;
  CHECK_ERROR (OCILobErase2(svchp, errhp, c2, &amt, 2));

  /*------------------------- Trim a LOB -----------------------------*/
  amt = 1000;
  CHECK_ERROR (OCILobTrim2(svchp, errhp, c2, amt));
  printf("OCILobTrim2  Works! \n");

  /*------------------------------------------------------------------*/
  /*------------- Operations involving 2 locators --------------------*/
  /*------------------------------------------------------------------*/

  /*--------------- Check Equality of LOB locators -------------------*/
  CHECK_ERROR ( OCILobIsEqual(envhp, c1, c2, &boolval))
  printf("OCILobIsEqual %s\n", (boolval)?"TRUE":"FALSE");

  /*--------------- Append contents of a LOB to another LOB ----------*/
  CHECK_ERROR(OCILobAppend(svchp, errhp, c2, c1));
  printf("OCILobAppend: Works! \n");

  /*------------------------ LOB Copy --------------------------------*/
  /* Copy 10 characters from offset 1 of source to offset 2 of destination*/
  CHECK_ERROR (OCILobCopy2(svchp, errhp, c2, c1, 10, 2, 1));
  printf("OCILobCopy2: Works! \n");

 /*------------------- LOB Locator Assign ---------------------------*/
  CHECK_ERROR (OCILobLocatorAssign(svchp, errhp, c1, &c2));
  printf("OCILobLocatorAssign: Works! \n");

  /* Free the LOB descriptors which were allocated */
  OCIDescriptorFree((dvoid *) c1, (ub4) SQLT_CLOB);
  OCIDescriptorFree((dvoid *) c2, (ub4) SQLT_CLOB);

  CHECK_ERROR (OCIHandleFree((dvoid *) stmthp, OCI_HTYPE_STMT));
}
```

- Efficiently Reading LOB Data in OCI
  This section describes how to read the contents of a LOB into a buffer.
- Efficiently Writing LOB Data in OCI
  This section describes how to write the contents of a buffer to a LOB.

9.4.1 Efficiently Reading LOB Data in OCI

This section describes how to read the contents of a LOB into a buffer.

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 Also:**
>
> *Oracle Call Interface Programmer's Guide*

LOB Array Read

This section describes how to read LOB data for multiple locators in one round trip, using `OCILobArrayRead()`.

For an OCI application example, assume that the program has a prepared SQL statement such as:

```
SELECT lob1 FROM lob_table;
```

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 */
...
```

LOB Array Read with Streaming

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 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;
}
...
```

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 */
...
}
```

9.4.2 Efficiently Writing LOB Data in OCI

This section describes how to write the contents of a buffer to a LOB.

Streaming Write in OCI

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.

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;
    }
...
```

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);
}

...
```

9.5 ODP.NET API for LOBs

Oracle Data Provider for .NET (ODP.NET) is an ADO.NET provider for the Oracle Database.

ODP.NET offers fast and reliable access to Oracle data and features from any .NET Core or .NET Framework application. ODP.NET also uses and inherits classes and interfaces available in the Microsoft .NET Class Library. The ODP.NET supports the following LOBs as native data types with .NET: `BLOB`, `CLOB`, `NCLOB`, and `BFILE`.

See Also:

- LOB Support
- Obtaining LOB Data

Table 9-7 ODP.NET methods in OracleClob and OracleBlob classes

Category	Function/Procedure	Description
Open/Close	`BeginChunkWrite`	Open a LOB
	`EndChunkWrite`	Close a LOB
	`IsInChunkWriteMode`	Check if a LOB is open
Read Operations	`Length`	Get the length of the LOB
	`OptimumChunkSize`	Get the optimum read/write size
	`Value`	Returns the entire LOB data as a string for CLOB and a byte array for BLOB
	`Read`	Read data from the LOB starting at the specified offset
	`Search`	Return the matching position of a pattern in a LOB using `INSTR`
Modify Operations	`Write`	Write data to the LOB at a specified offset
	`Erase`	Erase part of a LOB, starting at a specified offset
	`SetLength`	Trim the LOB value to the specified shorter length
Operations involving multiple locators	`Compare`	Compare all or part of the value of two LOBs
	`IsEqual`	Check if two LOBs point to the same LOB data
	`Append`	Append a LOB value to another LOB, or append a byte array, string, or character array to an existing LOB
	`CopyTo`	Copy all or part of a LOB to another LOB
	`Clone`	Assign LOB locator `src` to LOB locator `dst`

9.6 OCCI API for LOBs

OCCI provides a seamless interface to manipulate objects of user-defined types as C++ class instances.

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.

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 perform random and piecewise operations on LOBs, which means that you specify the offset or amount of the operation to read or write a part of the LOB value.

OCCI provides these 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 persistent `CLOB`s and `NCLOB`s
- `Blob` class to access and modify data stored in persistent `BLOB`s

> **See Also:**
>
> Syntax information on these classes and details on OCCI in general is available in the*Oracle C++ Call Interface Developer'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.

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.

Fixed-Width Character Set Rules

In OCCI, for *fixed-width* client-side character sets, these rules apply:

- Clob: offset and amount parameters are always in characters
- Blob: 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:
  - Clob(): in characters
  - Blob(): in bytes
- **Amount parameter**: The amount parameter is always as indicated:
  - Clob: in characters, when referring to a server-side LOB
  - Blob: 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
- **Clob.read() and Blob.read()**: 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 OCCI buffer parameter, buffer.
- **Clob.write() and Blob.write()**: With client-side character set of varying-width, CLOBs and NCLOBs:
  - **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.
- **Amount Parameter for Other OCCI Operations:** For the OCCI LOB operations Clob.copy(), Clob.erase(), Clob.trim() irrespective of the client-side character set, the *amount parameter* is in characters for CLOBs and NCLOBs. All these operations refer to the amount of LOB data on the server.

> **See also:**
>
> *Oracle Database Globalization Support Guide*

Table 9-8 OCCI Methods for LOBs

Category	Function/Procedure	Description
Sanity Checking	`Clob/Blob.isInitialized`	Checks whether a LOB locator is initialized.
Open/Close	`Clob/Blob.Open()`	Open a LOB
	`Clob/Blob.isOpen()`	Check if a LOB is open
	`Clob/Blob.Close()`	Close the LOB
Read Operations	`Blob/Clob.length()`	Get the length of the LOB
	`Blob/Clob.getChunkSize()`	Get the optimum read or write size
	`Blob/Clob.read()`	Read data from the LOB starting at the specified offset
	`Clob.getCharSetId()`	Return the character set ID of a LOB
	`Clob.getCharSetForm()`	Return the character set form of a LOB.
Modify Operations	`Blob/Clob.write()`	Write data to the LOB at a specified offset
	`Blob/Clob.trim()`	Trim the LOB value to the specified shorter length
Operations involving multiple locators	`Clob/Blob.operator == and !=`	Checks whether two LOB locators refer to the same LOB.
	`Blob/Clob.append()`	Append a LOB value to another LOB
	`Blob/Clob.copy()`	Copy all or part of a LOB to another LOB, or load from a BFILE into a LOB
	`Clob/Blob.operator =`	Assign one LOB to another
Operations specific to securefiles	`Blob/Clob.getOptions()`	Returns options (deduplication, compression, encryption) for SecureFiles.
	`Blob/Clob.setOptions()`	Sets LOB features (deduplication and compression) for SecureFiles
	`Blob/Clob.getContentType()`	Gets the content string for a SecureFiles
	`Blob/Clob.setContentType()`	Sets a content string in a SecureFiles

9.7 Pro*C/C++ and Pro*COBOL API for LOBs

This section describes the mapping of Pro*C/C++ and Pro*COBOL locators to locator pointers to access a LOB value.

Embedded SQL statements enable you to access data stored in `BLOB`s, `CLOB`s, and `NCLOB`s.

> **See Also:**
>
> *Pro*C/C++ Programmer's Guide* and *Pro*COBOL Programmer's Guide* for detailed documentation, including syntax, host variables, host variable types and example code.

Unlike locators in PL/SQL, locators in Pro*C/C++ and Pro*COBOL are mapped to locator pointers which are then used to refer to the LOB 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.

Table 9-9 Pro*C/C++ and Pro*COBOL Embedded SQL Statements for LOBs

Category	Function/Procedure	Description
Open/Close	`OPEN`	Open a LOB
	`DESCRIBE[ISOPEN]`	Check is a LOB is open
	`CLOSE`	Close the LOB
Read Operations	`DESCRIBE[LENGTH]`	Get the length of the LOB
	`DESCRIBE[CHUNKSIZE]`	Get the optimum read or write size
	`READ`	Read data from the LOB starting at a specified offset
Modify Operations	`WRITE`	Write data to the LOB at a specified offset
	`WRITE APPEND`	Write data to the end of the LOB
	`ERASE`	Erase part of a LOB, starting at a specified offset
	`TRIM`	Trim the LOB value to the specified shorter length
Operations involving multiple locators	`APPEND`	Append a LOB value to another LOB
	`COPY`	Copy all or part of a LOB to another LOB
	`ASSIGN`	Assign one LOB to another
	`LOAD FROM FILE`	Load `BFILE` data into a LOB

10

Distributed LOBs

This section describes the ways in which you can work with LOB data in remote tables.

Distributed LOBs are LOBs that are fetched from one server to another, and may optionally be returned to the client. Distributed LOBs can be persistent or temporary LOBs for both reference and value LOB columns.

In sharding, a table is horizontally partitioned with subsets of rows in a table stored in different sharded databases. The client connects to the coordinator database, which in turn works with shards to provide a consolidated view of a table. Sharded LOBs are an extension of Distributed LOBs. LOB data between different shards is transported as distributed LOBs and the result is provided to the client through the coordinator database.

All Persistent LOBs and Temporary LOBs originating from JSON support Distributed and Sharded LOBs.

- Working with Remote LOBs in SQL and PL/SQL
  This section describes the SQL and PL/SQL functions that are supported on remote LOBs.
- Using the Data Interface on Remote LOBs
  The data interface enables you to bind and define a `CHARACTER` buffer for a `CLOB` column and a `RAW` buffer for a `BLOB` column. This interface is supported for remote LOB columns too.
- Working with Remote Locators
  You can select a persistent LOB locator from a remote table into a local variable and this can be done in any programmatic interface like PL/SQL, JDBC or OCI. The remote columns can be of type `BLOB`, `CLOB` or `NCLOB`.

> **See Also:**
>
> Sharding with LOBs

10.1 Working with Remote LOBs in SQL and PL/SQL

This section describes the SQL and PL/SQL functions that are supported on remote LOBs.

SQL Functions

All the 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 the nested functions is not a LOB type. This includes functions for remote persistent and temporary LOBs and for BFILEs.

Most of the examples in the following sections use `print_media` table. Following is the structure of the table:

PRINT_MEDIA Table

Column name	Column Type
product_id	NUMBER (6)
ad_id	NUMBER (6)
ad_composite	BLOB
ad_sourcetext	CLOB
ad_finaltext	CLOB
ad_fltextn	NCLOB
ad_textdocs_ntab	NESTED TABLE
ad_photo	BLOB
ad_graphic	BFILE
ad_header	USER DEFINED TYPE
press_release	LONG

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
```

PL/SQL functions

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.

```
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 SQL and PL/SQL functions fall under the following non-comprehensive list of categories:

- SQL functions that are not supported on LOBs
  The SQL functions like the `DECODE` function, which are not supported for LOBs, are not supported on remote LOBs as well.
- Functions that accept exactly one LOB argument (where all the other arguments are of non-LOB data types) and does not return a LOB
  The functions, like the LENGTH function, are 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

  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`.

  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 on the same `dblink`. 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;
```

- PL/SQL functions operating on LOBs:
  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;
```

- Multiple LOBs in a query block:
  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
```

- LOB operators and columns are supported if they are in a `SELECT` list and `where` clause in a join query.
- Oracle-provided PL/SQL functions and procedures can return LOB locators.
- Only remote LOBs support SQL operators returning temporary LOBs.
- Only the views supplied by Oracle, support returning LOBs.

10.2 Using the Data Interface on Remote LOBs

The data interface enables you to bind and define a `CHARACTER` buffer for a `CLOB` column and a `RAW` buffer for a `BLOB` column. This interface is supported for remote LOB columns too.

The advantage of using the data interface over using LOB locators is that it makes only one round-trip to the remote server to fetch the LOB data. If used in as part of an array bind or define, it will use only one round-trip for the entire array operation.

The examples discussed in the book use the `print_media` table created in the following two schemas: `dbs1` and `dbs2`. The `CLOB` column of the `print_media` table used in the examples shown is `ad_finaltext`. The examples provided for PL/SQL, OCI, and Java in the following sections use binds and defines for this one column, but multiple columns can also be accessed. Following is the functionality supported:

- You can bind and define a `CLOB` as `VARCHAR2` and a `BLOB` as `RAW`.
- Array binds and defines are supported.

PL/SQL

This section describes how to use the remote data interface with LOBs 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/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.

JDBC

This section demonstrates how to use the remote data interface with LOBs in JDBC.

The following code snippets work with all JDBC drivers:

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();
...
```

Note: Oracle supports the non-streaming mode for strings of size up to 2 GB. However, the memory size of your computer may be a limiting factor.

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 );
```

Note: You can use the streaming interface to insert Gigabyte sized character and binary data into a LOB column.

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 );
     }
```

Note: If the LOB size is greater than 32767 bytes, the data is truncated and no error is thrown.

For streaming mode:

```
OracleStatement stmt = (OracleStatement)(conn.createStatement());
  stmt.defineColumnType( 1, Types.LONGVARCHAR );
  ResultSet rs = stmt.executeQuery("select ad_finaltext from print_media@dbs2" );
  while(rs.next()) {
    Reader reader = rs.getCharacterStream( 1 );
    int data = 0;
    data = reader.read();
    while( -1 != data ){
      System.out.print( (char)(data) );
      data = reader.read();
    }
    reader.close();
  }
```

Note: Specifying the datatype as `LONGVARCHAR` lets you select the entire LOB. If the define type is set as `VARCHAR` instead of `LONGVARCHAR`, the data will be truncated at 32k.

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();
    }
...
```

OCI

This section demonstrates how to use the remote data interface with LOBs in OCI.

The data interface only supports data of size less than 2 gigabytes (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(...);
...
```

For a `BLOB` column, you must use the `SQLT_BIN` type. For example, if you define the `ad_finaltext` column as a `BLOB` column instead of a `CLOB` column, then you must bind and define the column data using the `SQLT_BIN` type. If the LOB is greater than `2GB - 1` bytes in size, then OCI raises a truncation error and the contents of the buffer are undefined.

Remote LOBs

This section discusses the restrictions on the usage of Data Interface on Remote LOBs.

Certain syntax is not supported for remote LOBs.

- 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
  This section describes how to use the remote data interface with LOBs in PL/SQL.
- Remote Data Interface Examples in JDBC
  This section demonstrates how to use the remote data interface with LOBs in JDBC.
- Remote Data Interface Example in OCI
  This section demonstrates how to use the remote data interface with LOBs in OCI.
- Restrictions for Data Interface on Remote LOBs
  This section discusses the restrictions on the usage of Data Interface on Remote LOBs.

See Also:

- Oracle Database JDBC Developer's Guide
- Data Interface for LOBs

10.2.1 Remote Data Interface Example in PL/SQL

This section describes how to use the remote data interface with LOBs 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/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.

10.2.2 Remote Data Interface Examples in JDBC

This section demonstrates how to use the remote data interface with LOBs in JDBC.

The following code snippets work with all JDBC drivers:

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();
...
```

> **Note:**
>
> Oracle supports the non-streaming mode for strings of size up to 2 GB. However, the memory size of your computer may be a limiting factor.

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 );
```

> **Note:**
>
> You can use the streaming interface to insert Gigabyte sized character and binary data into a LOB column.

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 );
     }
```

> **Note:**
>
> If the LOB size is greater than 32767 bytes, the data is truncated and no error is thrown.

For streaming mode:

```
OracleStatement stmt = (OracleStatement)(conn.createStatement());
  stmt.defineColumnType( 1, Types.LONGVARCHAR );
  ResultSet rs = stmt.executeQuery("select ad_finaltext from print_media@dbs2" );
  while(rs.next()) {
    Reader reader = rs.getCharacterStream( 1 );
    int data = 0;
    data = reader.read();
    while( -1 != data ){
      System.out.print( (char)(data) );
      data = reader.read();
    }
    reader.close();
  }
```

> **Note:**
>
> Specifying the datatype as `LONGVARCHAR` lets you select the entire LOB. If the define type is set as `VARCHAR` instead of `LONGVARCHAR`, the data will be truncated at 32k.

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*

10.2.3 Remote Data Interface Example in OCI

This section demonstrates how to use the remote data interface with LOBs in OCI.

The data interface only supports data of size less than 2 gigabytes (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(...);
...
```

For a `BLOB` column, you must use the `SQLT_BIN` type. For example, if you define the `ad_finaltext` column as a `BLOB` column instead of a `CLOB` column, then you must bind and define the column data using the `SQLT_BIN` type. If the LOB is greater than `2GB - 1` bytes in size, then OCI raises a truncation error and the contents of the buffer are undefined.

10.2.4 Restrictions for Data Interface on Remote LOBs

This section discusses the restrictions on the usage of Data Interface on Remote LOBs.

Certain syntax is not supported for remote LOBs.

- 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;
```

- `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`.

10.3 Working with Remote Locators

You can select a persistent LOB locator from a remote table into a local variable and this can be done in any programmatic interface like PL/SQL, JDBC or OCI. The remote columns can be of type `BLOB`, `CLOB` or `NCLOB`.

The following SQL statement is the basis for all the examples with remote LOB locator in this chapter.

```
CREATE TABLE lob_tab (c1 NUMBER, c2 CLOB);
```

In the following example, the table `lob_tab` (with columns `c2` of type `CLOB` and `c1` of type number) defined in the remote database is accessible using database link `db2` and a local `CLOB` variable `lob_var1`.

```
SELECT c2 INTO lob_var1 FROM lob_tab@db2 WHERE c1=1;
SELECT c2 INTO lob_var1 FROM lob_tab@db2 WHERE c1=1 for update;
```

In PL/SQL, the function `dbms_lob.isremote` can be used to check if a particular LOB belongs to a remote table. Similarly, in `OCI`, you can use the `OCI_ATTR_LOB_REMOTE` attribute of `OCILobLocator` to check if a particular LOB belongs to a remote table. For example,

```
IF(dbms_lob.isremote(lob_var1)) THEN
dbms_output.put_line('LOB locator is remote)
ENDIF;
```

- Using Local and Remote Locators as Bind with Queries and DML on Remote Tables
  This section discusses the bind values for queries and DML statements.
- Using Remote Locator
  This section demonstrates the usage of remote locator in PL/SQL and with OCILOB API with examples.

- Using Remote Locators with OCILOB API
  Most `OCILOB` APIs support operations on remote `LOB` locators. The following list of `OCILOB` functions returns an error when a remote `LOB` locator is passed to them:
- Restrictions when using remote LOB locators
  Remote LOB locators have the following restrictions:

> **See Also:**
>
> - ISREMOTE Function
> - OCI_ATTR_LOB_REMOTE Attribute

10.3.1 Using Local and Remote Locators as Bind with Queries and DML on Remote Tables

This section discusses the bind values for queries and DML statements.

For the Queries and DMLs (`INSERT`, `UPDATE`, `DELETE`) with bind values, the following four cases are possible. The first case involves local tables and locators and is the standard LOB functionality, while the other three cases are part of the distributed LOBs functionality and have restrictions listed at the end of this section.

- Local table with local locator as bind value.
- Local table with remote locator as bind value
- Remote table with local locator as bind value
- Remote table with remote locator as bind value

Queries of the following form which use a remote LOB locator as a bind value are supported:

```
SELECT name FROM lob_tab@db2 WHERE length(c1)=length(:lob_v1);
```

In the above query, `c1` is an LOB column and `lob_v1` is a remote locator.

DMLs of the following forms using a remote LOB locator are supported. Here, the bind values can be local, remote persistent, or temporary LOB locators.

```
UPDATE lob_tab@db2 SET c1=:lob_v1;

INSERT into lob_tab@db2 VALUES (:1, :2);
```

You can pass a remote locator to most built-in SQL functions such as `LENGTH, INSTR, SUBSTR`, and `UPPER`. For example:

```
Var lob1 CLOB;
BEGIN
    SELECT c2 INTO lob1 FROM lob_tab@db2 WHERE c1=1;
END;
/
SELECT LENGTH(:lob1) FROM DUAL;
```

> **Note:**
>
> DMLs with `returning` clause are not supported on remote tables for both scalar and LOB columns.

10.3.2 Using Remote Locator

This section demonstrates the usage of remote locator in PL/SQL and with OCILOB API with examples.

PL/SQL

A remote locator can be passed as a parameter to built in PL/SQL functions like `LENGTH`, `INSTR`, `SUBSTR`, `UPPER` and so on which accepts LOB as input. For example,

```
DECLARE
    substr_data VARCHAR2(4000);
    remote_loc CLOB;
BEGIN
    SELECT c2 into remote_loc
    FROM lob_tab@db2 WHERE c1=1;
    substr_data := substr(remote_loc, position, length)
END;
```

All `DBMS_LOB` APIs other than the APIs targeted for BFILEs support operations on remote LOB locators.

The following example shows how to pass remote locator as input to `dbms_lob` operations.

```
DECLARE
  lob CLOB;
  buf VARCHAR2(120) := 'TST';
  amt NUMBER(2);
  len NUMBER(2);
BEGIN
  amt :=30;
  SELECT c2 INTO lob FROM lob_tab@db2 WHERE c1=3 FOR UPDATE;
  DBMS_LOB.WRITE(lob, amt, 1, buf);
  amt :=30;
  DBMS_LOB.READ(lob, amt, 1, buf);
  len := DBMS_LOB.GETLENGTH(lob);
  DBMS_OUTPUT.PUT_LINE(buf);
  DBMS_OUTPUT.PUT_LINE(amt);
  DBMS_OUTPUT.PUT_LINE('get length output = ' || len);
END;
/
```

OCILOB API

Most OCILOB APIs support operations on remote LOB locators. The following list of OCILOB functions returns an error when a remote LOB locator is passed to them:

- OCILobLocatorAssign
- OCILobArrayRead()
- OCILobArrayWrite()
- OCILobLoadFromFile2()

The following example shows how to pass a remote locator to OCILOB API.

```
void select_read_remote_lob()
{
  text *select_sql = (text *)"SELECT c2 lob_tab@dbs1 where c1=1";
  ub4 amtp = 10;
  ub4 nbytes = 0;
  ub4 loblen=0;
  OCILobLocator * one_lob;
  text strbuf[40];

 /* initialize single locator */
 OCIDescriptorAlloc(envhp, (dvoid **) &one_lob,
                 (ub4) OCI_DTYPE_LOB,
                 (size_t) 0, (dvoid **) 0)

 OCIStmtPrepare(stmthp, errhp, select_sql, (ub4)strlen((char*)select_sql),
                 (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);

 OCIDefineByPos(stmthp, &defp, errhp, (ub4) 1,
                     (dvoid *) &one_lob,
                     (sb4) -1,
                     (ub2) SQLT_CLOB,
                     (dvoid *) 0, (ub2 *) 0,
                     (ub2 *) 0, (ub4) OCI_DEFAULT));

 /* fetch the remote locator into the local variable one_lob */
 OCIStmtExecute(svchp, stmthp, errhp, 1, 0, (OCISnapshot *)0,
                 (OCISnapshot *)0, OCI_DEFAULT);

 /* Get the length of the remote LOB */
 OCILobGetLength(svchp, errhp,
                (OCILobLocator *) one_lob, (ub4 *)&loblen)

 printf("LOB length = %d\n", loblen);

 memset((void*)strbuf, (int)'\0', (size_t)40);

 / * Read the data from the remote LOB */
 OCILobRead(svchp, errhp, one_lob, &amtp,
                (ub4) 1, (dvoid *) strbuf, (ub4)& nbytes, (dvoid *)0,
                (OCICallbackLobRead) 0,
                (ub2) 0, (ub1) SQLCS_IMPLICIT));
 printf("LOB content = %s\n", strbuf);
```

```
}
```

> **See Also:**
>
> *OCI Programmer's Guide*, for the complete list of OCILOB APIs

10.3.3 Using Remote Locators with OCILOB API

Most OCILOB APIs support operations on remote LOB locators. The following list of OCILOB functions returns an error when a remote LOB locator is passed to them:

- OCILobLocatorAssign
- OCILobArrayRead()
- OCILobArrayWrite()
- OCILobLoadFromFile2()

The following example shows how to pass a remote locator to OCILOB API.

```
void select_read_remote_lob()
{
  text *select_sql = (text *)"SELECT c2 lob_tab@dbs1 where c1=1";
  ub4 amtp = 10;
  ub4 nbytes = 0;
  ub4 loblen=0;
  OCILobLocator * one_lob;
  text strbuf[40];

 /* initialize single locator */
 OCIDescriptorAlloc(envhp, (dvoid **) &one_lob,
                 (ub4) OCI_DTYPE_LOB,
                 (size_t) 0, (dvoid **) 0)

 OCIStmtPrepare(stmthp, errhp, select_sql, (ub4)strlen((char*)select_sql),
                 (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);

 OCIDefineByPos(stmthp, &defp, errhp, (ub4) 1,
                     (dvoid *) &one_lob,
                     (sb4) -1,
                     (ub2) SQLT_CLOB,
                     (dvoid *) 0, (ub2 *) 0,
                     (ub2 *) 0, (ub4) OCI_DEFAULT));

 /* fetch the remote locator into the local variable one_lob */
 OCIStmtExecute(svchp, stmthp, errhp, 1, 0, (OCISnapshot *)0,
                 (OCISnapshot *)0, OCI_DEFAULT);

 /* Get the length of the remote LOB */
```

```
OCILobGetLength(svchp, errhp,
                (OCILobLocator *) one_lob, (ub4 *)&loblen)

printf("LOB length = %d\n", loblen);

memset((void*)strbuf, (int)'\0', (size_t)40);

/ * Read the data from the remote LOB */
OCILobRead(svchp, errhp, one_lob, &amtp,
                (ub4) 1, (dvoid *) strbuf, (ub4)& nbytes, (dvoid *)0,
                (OCICallbackLobRead) 0,
                (ub2) 0, (ub1) SQLCS_IMPLICIT));
printf("LOB content = %s\n", strbuf);

}
```

> **See Also:**
>
> *OCI Programmer's Guide*, for the complete list of `OCILOB` APIs

10.3.4 Restrictions when using remote LOB locators

Remote LOB locators have the following restrictions:

- You cannot select a remote temporary LOB locator into a local variable using the `SELECT` statement. For example,

  ```
  select substr(c2, 3, 1) from lob_tab@db2 where c1=1
  ```

  The preceding query returns an error.
- Remote LOB functionality is not supported for Index Organized tables (IOT). An attempt to get a locator from a remote IOT table will result in an error.
- Both the local database and the remote database have to be of Database release 12.2 or higher version.
- With distributed LOBs functionality, the tables that you use in the `from` clause or `where` clause should be collocated on the same database. If you use remote locators as bind variables in the `where` clauses, then they should belong to the same remote database. You cannot have one locator from one database (say, DB1) and another locator from another database (say, DB2) to be used as bind variables.
- Collocated tables or locators use the same database link. It is possible to have two different DB Links pointing to the same database. In the following example, both `dblink1` and `dblink2` point to the same remote database, but with different authentication methods. Oracle Database *does not* support such operations.

  ```
  INSERT into tab1@dblink1 SELECT * from tab2@dblink2;
  ```
- Any `DBMS_LOB` or `OCILob` APIs that accept two locators must obtain both the LOB locators through the same database link. Operations, as specified in the following example, are *not supported*:

  ```
  SELECT ad_sourcetext INTO clob1 FROM print_media@db1 WHERE product_id =
  10011;
  ```

```
SELECT ad_sourcetext INTO clob2 FROM print_media@db2 WHERE product_id =
10011;
DBMS_LOB.COPY(clob1, clob2, length(clob2));
```

- Bind values should be of the same LOB type as the column LOB type. For example, you must bind `NCLOB` locators to `NCLOB` columns and `CLOB` locators to `CLOB` columns. Implicit conversion between `NCLOB` and `CLOB` types is not supported in case of remote LOBs.
- DML statements with Array Binds are not supported when the bind operation involves a remote locator, or if the table involved is a remote table.
- You cannot select a `BFILE` column from a remote table into a local variable.

11

Performance Guidelines

This section discusses performance guidelines for applications that use LOB data types.

> **Note:**
>
> From release 23ai onwards, you can experience improved read and write performance for LOBs due to the following enhancements:
>
> - Multiple LOBs in a single transaction are buffered simultaneously. This improves performance when you use mixed workload in a transaction. Mixed workload refers to switching between LOBs while writing within a single transaction. Let's consider that you write to LOB1, then you write to LOB2, and then you want to write again to LOB1 in a single transaction. LOB1 and LOB2 are buffered simultaneously, which provides better throughput and minimizes space fragmentation.
> - Various enhancements, such as acceleration of compressed LOB append and compression unit caching, improve the performance of reads and writes to compressed LOBs.
> - The input-output buffer is adaptively resized based on size of the input data for large writes to LOBs with the NOCACHE option. This improves the performance for large direct writes, such as writes to file systems on DBFS and OFS.

11.1 LOB Performance Guidelines

This section provides performance guidelines while using LOBs through Data Interface or LOB APIs.

LOBs can be accessed using the Data Interface or through the LOB APIs.

- Temporary LOBs
  In addition to the performance guidelines applicable to all LOBs described earlier, following are some guidelines for using temporary LOBs:
- Value LOBs
  Value LOBs are temporary LOBs. Hence all Temporary LOB storage guidelines apply to Value LOBs as well.

11.1.1 All LOBs

Learn about the guidelines to achieve good performance while using LOBs in this section.

The following guidelines will help you get the the best performance when using LOBs, and minimize the number of round trips to the server:

- To minimize I/O:
  - Read and write data at block boundaries. This optimizes I/O in many ways, e.g., by minimizing UNDO generation. For temporary LOBs and securefile LOBs, usable data area of the tablespace block size is returned by the following APIs: `DBMS_LOB.GETCHUNKSIZE` in PLSQL, and `OCILobGetChunkSize()` in OCI. When writing in a loop, design your code so that one write call writes everything that needs to go in a database block, thus ensuring that consecutive writes don't write to the same block.
  - Read and write large pieces of data at a time.
  - The 2 recommendations above can be combined by reading and writing in large whole number multiples of database block size returned by the `DBMS_LOB.GETCHUNKSIZE/ OCILobGetChunkSize()` API.
- To minimize the number of round trips to the server:
  - If you know the maximum size of your lob data, and you intend to read or write the entire LOB, use the Data Interface as outlined below. You can allocate the entire size of lob as a single buffer, or use piecewise / callback mechanisms.
    * For read operations, define the LOB as character/binary type using the `OCIDefineByPos()` function in OCI and the `DefineColumnType()` function in JDBC.
    * For write operations, bind the LOB as character/binary type using the `OCIBindByPos()` function in OCI and the `setString()` or `setBytes()` methods in JDBC.
  - Otherwise, use the LOB APIs as follows:
    * Use LOB prefetching for reads. Define the LOB prefetch size such that it can accommodate majority of the LOB values in the column.
    * Use piecewise or callback mechanism while using `OCILobRead2` or `OCILobWrite2` operations to minimize the roundtrips to the server.

> **See Also:**
>
> Data Interface for Persistent LOBs

11.1.2 Performance Guidelines While Using Persistent LOBs

In addition to the performance guidelines applicable to all LOBs described earlier, here are some performance guidelines while using persistent LOBs.

- Maximize writing to a single LOB in consecutive calls within a transaction. Interleaving DML statements prevent caching from reaching its maximum efficiency.
- Avoid taking savepoints or committing too frequently. This neutralizes the advantage of caching while writing.

> **Note:**
>
> Oracle recommends Securefile LOBs for storing persistent LOBs, hence this chapter focuses only on Securefile storage. All mentions of "LOBs" in the persistent LOB context is for Securefile LOBs unless otherwise mentioned.

11.1.3 Temporary LOBs

In addition to the performance guidelines applicable to all LOBs described earlier, following are some guidelines for using temporary LOBs:

- Temporary LOBs reside in the PGA memory or the temporary tablespace, depending on the size. Please ensure that you have a large enough PGA memory and temporary tablespace for the temporary LOBs used by your application.
- 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 SQL or PL/SQL semantics for LOBs in your applications, then many temporary LOBs are created silently. Ensure that PGA memory and 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 `CLOB`s/`BLOB`s, respectively.
  - Perform a `LONG`-to-`LOB` migration
- 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, you may observe performance degradation.

- In PL/SQL, use NOCOPY to pass temporary LOB parameters by reference whenever possible.

> **See Also:**
>
> *Oracle Database PL/SQL Language Reference*for more information on passing parameters by reference and parameter aliasing

- Temporary LOBs created with the CACHE parameter set to true move through the buffer cache and avoid the disk access.
- Oracle provides `v$temporary_lobs` view to monitor the use of temporary LOBs across all open sessions. Here is an example:

```
SQL> select * from v$temporary_lobs;

       SID CACHE_LOBS NOCACHE_LOBS ABSTRACT_LOBS    CON_ID
---------- ---------- ------------ ------------- ----------
       141          2            3             4          0
       146          0            0             1          0
       148          0            0             1          0
```

Following is the interpretation of output:

  - The `SID` column is the session ID.
  - The `CACHE_LOBS` column shows that session 141 currently has 2 temporary lobs in the temporary tablespace with CACHE turned on.
  - The `NOCACHE_LOBS` column shows that session 141 currently has 3 temporary lobs in the temporary tablespace with CACHE turned off.
  - The `ABSTRACT_LOBS` column shows that session 141 currently has 4 temporary lobs in the PGA memory.
  - The `CON_ID` column is the pluggable database container ID.

- 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.

  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 pointer assignment so that both locators point to the same data as illustrated in the following code snippet:

```
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.

11.1.4 Value LOBs

Value LOBs are temporary LOBs. Hence all Temporary LOB storage guidelines apply to Value LOBs as well.

On the client side, Oracle recommends that you set the LOB prefetch size large enough to accommodate at least 80% of your LOB read size for Value LOBs.

11.2 Moving Data to LOBs in a Threaded Environment

Learn about the recommended procedure to follow while moving data to LOBs in this section.

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

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 use Data Interface to insert character data or raw 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.

11.3 LOB Access Statistics

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 LOB block boundary. Writes aligned to block boundaries are the most efficient write operations. The usable LOB block size of a LOB is available through the LOB API (for example, using PL/SQL, by `DBMS_LOB.GETCHUNKSIZE()`).

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). Oracle recommends that 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"

This example demonstrates how LOB session statistics are updated as the user performs read or write operations on LOBs.

```
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/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/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/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/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/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 block size.
rem

CONNECT pm/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);
    chunk = chunk * floor(32767/chunk); /* integer multiple of chunk */
    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/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;
```

12

Persistent LOBs: Advanced DDL

This chapter describes advanced LOB DDL features to make your application more scalable.

> **Note:**
>
> Unless otherwise stated, all features in this chapter apply to both SecureFile and Basicfile LOBs. However, Oracle strongly recommends SecureFiles for storing and managing LOBs.

- Creating a New LOB Column
  You can provide the LOB storage characteristics when creating a LOB column using the `CREATE TABLE` statement or the `ALTER TABLE ADD COLUMN` statement.
- Altering an Existing LOB Column
  You can use the `ALTER TABLE` statement to change the storage characteristics of a LOB column.
- Creating an Index on LOB Column
  The contents of a LOB are often specific to the application, so an index on the LOB column will usually deal with application logic. You can create a function-based or a domain index on a LOB column to improve the performance of queries accessing data stored in LOB columns. You cannot build a B-tree or bitmap index on a LOB column.
- LOBs in Partitioned Tables
  Partitioning can simplify the manageability of large database objects. This section discusses various aspects of LOBs in partitioned tables.
- LOBs in Index Organized Tables
  Index Organized Tables (IOTs) support LOB and BFILE columns.

12.1 Creating a New LOB Column

You can provide the LOB storage characteristics when creating a LOB column using the `CREATE TABLE` statement or the `ALTER TABLE ADD COLUMN` statement.

For most users, default values for these storage characteristics are sufficient. However, if you want to fine-tune LOB storage, then consider the guidelines discussed in this section.

When defining LOBs in a table, you can explicitly indicate the tablespace and storage characteristics for each persistent LOB column. It is common to use separate tablespaces for large LOBs. SecureFiles is the default storage for LOBs, so the `SECUREFILE` keyword is optional, but is shown for clarity in the following example. The example assumes that `TABLESPACE lobtbs1` is managed with ASSM, because SecureFile LOBs can only be created in tablespaces managed with Automatic Segment Space Management (ASSM).:

```
CREATE TABLE lobtab1 (n NUMBER, c CLOB)
      lob (c) STORE AS SECUREFILE sfsegname
      ( TABLESPACE lobtbs1
        ENABLE STORAGE IN ROW
```

```
  CACHE LOGGING
  RETENTION AUTO
  COMPRESS
  STORAGE (MAXEXTENTS 5)
);
```

To create a BasicFiles LOB, replace the `SECUREFILE` keyword with the `BASICFILE` keyword in the preceding example, and remove the `COMPRESS` keyword, which is specific to SecureFiles.

The data dictionary views `USER_LOBS`, `ALL_LOBS`, and `DBA_LOBS` provide information specific to a LOB column.

> **Note:**
>
> Oracle recommends Securefile LOBs for storing persistent LOBs, so this chapter focuses only on Securefile storage. All mentions of *LOBs* in the persistent LOB context is for Securefile LOBs, unless mentioned otherwise.

> **Note:**
>
> There are no tablespace or storage characteristics that you can specify for `BFILE`s as they are not stored in the database.

Assigning a LOB Data Segment Name

As shown in the previous example, specifying a name for the LOB data segment (`sfsegname` in the example) makes for a much more intuitive working environment. When querying the LOB data dictionary views `USER_LOBS`, `ALL_LOBS`, and `DBA_LOBS`, you see the LOB data segment that you chose instead of system-generated names.

- CREATE TABLE BNF
  The `CREATE TABLE` statement works with LOB storage using parameters that are specific to SecureFiles, BasicFiles LOB storage, or both.
- ENABLE or DISABLE STORAGE IN ROW
  LOB columns store locators that reference the location of the actual LOB value. This section describes how to enable or disable storage in a table row.
- CACHE, NOCACHE, and CACHE READS
  This section discusses the guidelines to follow while creating tables that contain LOBs.
- LOGGING and FILESYSTEM_LIKE_LOGGING
  You can apply the `LOGGING` parameter to LOBs in the same manner as you apply it for other table operations.
- The RETENTION Parameter
  The `RETENTION` parameter for SecureFile LOBs specifies how the database manages the old versions of the LOB data blocks.
- SecureFiles Compression, Deduplication, and Encryption
  In addition to the features supported by BasicFiles, SecureFiles LOB storage supports the following three features that are not available with the BasicFiles LOB storage option: compression, deduplication, and encryption.

- BasicFile Specific Parameters
  This section discusses the storage parameters specific to BasicFiles.
- Restriction on First Extent of a LOB Segment
  This section discusses the first extent requirements on SecureFiles and BasicFiles.
- Summary of CREATE TABLE LOB Storage Parameters for Securefile LOBs
  The table in this section summarizes the parameters of the `CREATE TABLE` statement that relate to Securefile LOB storage.

12.1.1 CREATE TABLE BNF

The `CREATE TABLE` statement works with LOB storage using parameters that are specific to SecureFiles, BasicFiles LOB storage, or both.

The following is the syntax for `CREATE TABLE` in Backus Naur (BNF) notation, parts of which have been simplified to keep the focus on LOB-specific parameters.

> **See Also:**
>
> - *Oracle Database SQL Language Reference*

Example 12-1 BNF for CREATE TABLE

```
CREATE ... TABLE [schema.]table ...;

<column_definition> ::= column [datatype]...

<datatype> ::= ... | BLOB | CLOB | NCLOB | BFILE | ...

<column_properties> ::= ... | LOB_storage_clause | ... |
LOB_partition_storage |...

<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 STORAGE IN ROW [{4000|8000}]
  | 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 ] } }
  ]
<LOB_retention_clause> ::=
  {RETENTION [ MAX | MIN integer | AUTO | NONE ]}
<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
  }
<LOB_partition_storage> ::=
  {PARTITION partition
  { LOB_storage_clause | varray_col_properties }...
  [ (SUBPARTITION subpartition
  { LOB_partitioning_storage | varray_col_properties }...
  )
  ]
  }
<LOB_partitioning_storage> ::=
  {LOB (LOB_item) STORE AS [BASICFILE | SECUREFILE]
  [ LOB_segname [ ( TABLESPACE tablespace | TABLESPACE SET tablespace_set ) ]
  | ( TABLESPACE tablespace | TABLESPACE SET tablespace_set )
  ]
  }
```

12.1.2 ENABLE or DISABLE STORAGE IN ROW

LOB columns store locators that reference the location of the actual LOB value. This section describes how to enable or disable storage in a table row.

Actual LOB values are stored either in the table row (inline) or outside of the table row (out-of-line), depending on the column properties you specify when you create the table and the size of the LOB. The `ENABLE | DISABLE STORAGE IN ROW` clause is used to indicate whether the LOB should be stored inline or out-of-line. The default is `ENABLE STORAGE IN ROW` because it provides a performance benefit for small LOBs.

ENABLE STORAGE IN ROW

If `ENABLE STORAGE IN ROW` is set, the minimum inline size is 4000 and the maximum is 8000. This includes the control information and the LOB value. The default inline size for LOBs is 4000.

If the LOB is stored `IN ROW`,

- Exadata pushdown is enabled for LOBs, including when using securefile compression and encryption.
- In-Memory is enabled for LOBs without securefile compression and encryption.

LOBs larger than approximately 32k are stored out-of-line. However, the control information is still stored in the row, thus enabling us to read the out-of-line LOB data faster.

VARCHAR2(32K) and VARRAYs stored as LOBs do not support the increased inlining syntax.

DISABLE STORAGE IN ROW

In some cases, `DISABLE STORAGE IN ROW` is a better choice becase 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.

12.1.3 CACHE, NOCACHE, and CACHE READS

This section discusses the guidelines to follow while creating tables that contain LOBs.

Use the cache options according to the guidelines in the following table:

Table 12-1 Using CACHE, NOCACHE, and CACHE READS Options

Cache Mode	Frequency of Read	Buffer Cache Behavior
`NOCACHE` (default)	Once or occasionally	LOB values are never brought into the buffer cache.
`CACHE READS`	Frequently	LOB values are brought into the buffer cache only during read operations and not during write operations.
`CACHE`	Read the LOB soon after write	LOB pages are placed in the buffer cache during both read and write operations. For storing semi-structured data consider turning on `CACHE` option.

> **Caution:**
>
> If your application frequently writes to LOBs, then using the `CACHE` option can potentially age other non-LOB pages out of the buffer cache prematurely.

12.1.4 LOGGING and FILESYSTEM_LIKE_LOGGING

You can apply the LOGGING parameter to LOBs in the same manner as you apply it for other table operations.

The default value of this parameter is LOGGING. For SecureFiles, the FILESYSTEM_LIKE_LOGGING parameter is equivalent to the NOLOGGING option.

If you set the LOGGING option, then Oracle Database determines the most efficient way to generate the REDO and UNDO logs for the change. Oracle recommends that you keep the LOGGING parameter turned on.

The FILESYSTEM_LIKE_LOGGING or the NOLOGGING option is useful for bulk loads and inserts. When loading data into the LOB, if you do not care about the REDO logs and can restart a failed load, then 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, Oracle recommends that you use the ALTER TABLE statement to modify the LOB storage characteristics for the LOB data segment for normal LOB operations. For example, set the cache option to CACHE or CACHE READS, along with the LOGGING option.

> **See Also:**
>
> Precedence of FORCE LOGGING Settings for more information about overriding the logging behavior at the database level

> **Note:**
>
> For BasicFiles, specifying the CACHE NOLOGGING option results in an error.

12.1.5 The RETENTION Parameter

The RETENTION parameter for SecureFile LOBs specifies how the database manages the old versions of the LOB data blocks.

Unlike other data types, the old versions of the LOB data blocks for SecureFile LOBs are stored in the LOB segment itself and are used to support consistent read operations. Without the corresponding old versions of the LOB data blocks, reading of a LOB at an earlier SCN may fail with ORA-1555. Set the RETENTION parameter as per the following guidelines:

Table 12-2 RETENTION parameter behavior

RETENTION Parameter value	Behavior
MAX	Allows the old versions of the LOB data blocks to fill the entire LOB segment. This minimizes the likelihood of an ORA-1555, if space usage is not a concern. With this setting, the old versions of the LOB data blocks may cause the LOB segment to grow. If you do not set the MAXSIZE attribute, then MAX behaves like AUTO.

Table 12-2 (Cont.) RETENTION parameter behavior

RETENTION Parameter value	Behavior
MIN	Limits the retention of old versions of the LOB data blocks to n seconds. With this setting, you must also specify the retention duration in number of seconds as n. The old versions of the LOB data blocks may also cause the LOB segment to grow.
AUTO	Oracle Database manages the space as efficiently as possible, weighing both time and space needs.
NONE	Set this value if no old version of the LOB data blocks is required for consistent read purposes. This is the most efficient setting in terms of space utilization.
not set (sets to DEFAULT)	Uses the UNDO_RETENTION setting can be set dynamically or manually. If the UNDO_RETENTION parameter is set to a positive value, then it is equivalent to setting the RETENTION parameter to MIN with the same value for retention duration. If the UNDO_RETENTION parameter is set to zero (0), then it is equivalent to setting the RETENTION parameter to NONE.

The SHRINK feature for SecureFile LOBs partially deletes old versions of the LOB data blocks to free extents, regardless of the RETENTION parameter setting. Therefore, it is recommended to have the SHRINK feature only when the RETENTION parameter is set to NONE.

The following SQL code snippet helps you determine the RETENTION parameter for a LOB segment.

```
SELECT RETENTION_TYPE, RETENTION_VALUE FROM USER_LOBs WHERE ...;
```

12.1.6 SecureFiles Compression, Deduplication, and Encryption

In addition to the features supported by BasicFiles, SecureFiles LOB storage supports the following 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.

> **Caution:**
>
> Enabling table or column level compression or encryption does not compress or encrypt the LOB data. To compress or encrypt the LOB data, use SecureFiles compression or encryption by specifying it in the LOB_storage_clause.

> **Note:**
>
> You can enable the compression, deduplication, and encryption features using the `ALTER TABLE` statement. However, if you enable these features using the `ALTER TABLE` statement, then all the data in the SecureFiles LOB storage is read, modified, and written. This can cause the database to lock the table during a potentially lengthy operation. There are online capabilities in the `ALTER TABLE` statement that can help you avoid this issue.

Topics

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

12.1.6.1 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.

Before you enable compression, use the `DBMS_COMPRESSION.GET_COMPRESSION_RATIO` function to estimate the space that you can save by enabling this feature for existing LOBs. This allows you to take an informed decision to enable compression.

Consider the following issues when using the `CREATE TABLE` statement with 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.
- Advanced LOB compression may convert an out-of-line LOB, to an inline LOB, by moving the data from a LOB segment into the table segment (inlined in column).

The following examples demonstrate how to issue `CREATE TABLE` statements for specific compression scenarios:

Example 12-2 Creating a SecureFiles LOB Column with LOW Compression

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

Example 12-3 Creating a SecureFiles LOB Column with MEDIUM (default) Compression

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

Example 12-4 Creating a SecureFiles LOB Column with HIGH Compression

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

Example 12-5 Creating a SecureFiles LOB Column with Disabled Compression

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

Example 12-6 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)
     );
```

12.1.6.2 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.

Consider these issues 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.

Sample Commands

The following examples demonstrate how to issue `CREATE TABLE` statements for specific deduplication scenarios:

Example 12-7 Creating a SecureFiles LOB Column with Deduplication

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

Example 12-8 Creating a SecureFiles LOB Column with Disabled Deduplication

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

Example 12-9 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 12-10 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)
      );
```

12.1.6.3 SecureFiles Encryption

In SecureFiles Encryption, 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.

Consider the following issues when using `CREATE TABLE` statement with SecureFiles Encryption:

- Securefile Encryption encrypts the data stored in the SecureFile LOB column, irrespective of whether the data is stored in-row or out-of-line in the LOB segment. Note that table or column level encryption will not encrypt the data stored out-of-line in the LOB segment.
- SecureFile Encryption relies on a wallet, or Hardware Security Model (HSM), to hold the encryption key. The wallet setup is the same as that described for Transparent Data Encryption (TDE) and Tablespace Encryption, so complete that before using SecureFile encryption.

> **See Also:**
>
> "*Oracle Database Advanced Security Guide* for information about creating and using Oracle wallet with TDE.

- The *`encrypt_algorithm`* indicates the name of the encryption algorithm. Valid algorithms are: `AES192` (default), `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.
- SecureFile Encryption is only supported at the table level on a per-column basis, and not at the per-partition level. Hence all partitions within a LOB column are encrypted.
- `DECRYPT` keeps the LOBs in clear text.
- 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.

The following examples demonstrate how to issue `CREATE TABLE` statements for specific encryption scenarios:

Example 12-11 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 12-12 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 12-13 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_encryption_clause` section of the statement:

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

Example 12-14 Creating a SecureFiles LOB Column with Disabled Encryption

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

12.1.7 BasicFile Specific Parameters

This section discusses the storage parameters specific to BasicFiles.

The following storage parameters are specific to BasicFiles:

> **Caution:**
>
> Oracle strongly recommends that you use SecureFile LOBs for all your LOB needs.

PCTVERSION

When a BasicFiles LOB is modified, a new version of the BasicFiles LOB page is produced in order to support consistent read operations of prior versions of the BasicFiles LOB value. The `PCTVERSION` parameter 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. The `PCTVERSION` parameter has the following preset values:

- Default: 10%
- Minimum: 0
- Maximum: 100

If your application requires several BasicFiles LOB updates that are concurrent with heavy reads of BasicFiles LOB columns, then consider using a higher value for the `PCTVERSION` parameter, such as 20%. 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 the `PCTVERSION` parameter, such as 5% or lower. If existing BasicFiles LOBs are known to be read-only, then you can safely set the `PCTVERSION` parameter to 0% because there will never be any pages needed for old versions of data.

> **Note:**
>
> The `PCTVERSION` parameter and the `RETENTION` parameter are mutually exclusive for BasicFiles LOBs, that is, you can specify either the `PCTVERSION` parameter or the `RETENTION` parameter, but not both.

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 APIs that you use to retrieve the chunk size, return the amount of space used in the LOB chunk to store the LOB value. You can use the following APIs to retrieve the chunk size:

- The `DBMS_LOB.GETCHUNKSIZE` procedure in PL/SQL
- The `OCILobGetChunkSize()` function in OCI

Once you specify the value of the `CHUNK` parameter (when the LOB column is created), you cannot change it without moving the LOB. You can set the `CHUNK` parameter to the data size most frequently accessed or written. It is more efficient to access LOBs in big chunks. If you explicitly specify storage characteristics for the LOB, then make sure that you set the `INITIAL` parameter and the `NEXT` parameter for the LOB data segment storage to a size that is larger than the `CHUNK` size.

For SecureFiles, the `CHUNK` size is an advisory size and is provided for backward compatibility purposes.

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.

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.

12.1.8 Restriction on First Extent of a LOB Segment

This section discusses the first extent requirements on SecureFiles and BasicFiles.

First Extent of a SecureFile LOB Segment

A SecureFile LOB segment can only be created in Locally Managed Tablespace with Automatic Segment Space Management (ASSM). The number of blocks required in the first extent depends on the release. Before 21c, the first extent requires at least 16 blocks. After 21c, the number is 32 if the compatible parameter is greater than or equal to 20.1.0.0.0. Segments created in the previous release will continue to work in the new release. However, they will not be automatically upgraded.

The actual size of the first extent depends on the database block_size. If the tablespace is configured to use uniform extent, the extent must be bigger than the aforementioned number. For example, with `block_size = 8k`, the uniform extent size must be at least 128K pre-21c, or 256K on 21c with compatible parameter set. If the tablespace is configured to use uniform extent that is less than this number, the LOB segment creation will fail.

First Extent of a BasicFile LOB Segment

A BasicFile LOB segment can be created in Dictionary Managed or Locally Managed Tablespaces. The segment requires at least 3 blocks in the first extent. This translates into different extent sizes based on the database block_size. If the tablespace is configured to use uniform extent that contains fewer than 3 blocks, the LOB segment creation will fail.

12.1.9 Summary of CREATE TABLE LOB Storage Parameters for Securefile LOBs

The table in this section summarizes the parameters of the `CREATE TABLE` statement that relate to Securefile LOB storage.

Table 12-3 Parameters of CREATE TABLE Statement Related to LOBs

Parameter	Description
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).
BASICFILE	Specifies BasicFiles LOB storage, the original architecture for LOBs. You must explicitly specify the parameter BASICFILE to use the BasicFiles LOB storage type. For BasicFiles LOBs, specifying any of the SecureFiles LOB options results in an error.
RETENTION	Specifies the retention policy for storing old versions of LOB data to support consistent read. Possible values are: MAX, MIN, AUTO and NONE.
MAXSIZE	Specifies the upper limit of storage space that a LOB may use. The default size is 4000, but this can go up to 8000. 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.
CACHE, NOCACHE, CACHE READS	Specifies when the LOB data in brought into the buffer cache. • NOCACHE: Never brought into buffer cache. • CACHE READS: Only during reads. • CACHE: During reads and writes. The default is NOCACHE.
LOGGING, NOLOGGING, or FILESYSTEM_LIKE_LOGGING	Specifies whether to generate REDO and UNDO for changes to the LOB: • LOGGING: Generate REDO and UNDO for the change • FILESYSTEM_LIKE_LOGGING/NOLOGGING: Log only the metadata. The default is LOGGING.
COMPRESS or NOCOMPRESS	The COMPRESS option turns on Advanced LOB Compression, and NOCOMPRESS turns it off. The default is NOCOMPRESS.
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. The default is KEEP_DUPLICATES.
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. The default is DECRYPT.

12.2 Altering an Existing LOB Column

You can use the `ALTER TABLE` statement to change the storage characteristics of a LOB column.

- ALTER TABLE BNF
  This section has the syntax for `ALTER TABLE` in Backus Naur (BNF) notation, parts of which have been simplified to keep the focus on LOB-specific parameters.
- ALTER TABLE MODIFY vs ALTER TABLE MOVE LOB
  This section compares the storage characteristics while using `ALTER TABLE MODIFY` and `ALTER TABLE MOVE LOB`.
- ALTER TABLE SecureFiles LOB Features
  This section discusses the features of SecureFile LOBs that work with the `ALTER TABLE` statement.

12.2.1 ALTER TABLE BNF

This section has the syntax for `ALTER TABLE` in Backus Naur (BNF) notation, parts of which have been simplified to keep the focus on LOB-specific parameters.

> **See Also:**
>
> ALTER TABLE for more information on usage of `ALTER TABLE` statement.

```
ALTER TABLE [ schema.]table ... [ ... | column_clauses | ... |
move_table_clause] ...;

<column_clauses> ::= ... | [ modify_LOB_storage_clause |
rename_lob_storage_clause ] ...

<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
}. . .

<rename_lob_storage_clause> ::= RENAME LOB(LOB_item) <LOB_RENAME_PARAMETERS>
<LOB_RENAME_PARAMETERS> ::= [ PARTITION | SUBPARTITION | ] <OLD_SEGMENT_NAME>
```

```
TO <NEW_SEGMENT_NAME>

<move_table_clause> ::= MOVE ...[ ... | LOB_storage_clause | ...] ...

<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 STORAGE IN ROW [{4000|8000}]
  | 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 ] } }
  ]

<LOB_retention_clause> ::=
  {RETENTION [ MAX | MIN integer | AUTO | NONE ]}

<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
}
```

12.2.2 ALTER TABLE MODIFY vs ALTER TABLE MOVE LOB

This section compares the storage characteristics while using `ALTER TABLE MODIFY` and `ALTER TABLE MOVE LOB`.

There are two kinds of changes to existing storage characteristics:

1. Some changes to storage characteristics merely apply to the way the data is accessed and do not require moving the entire existing LOB data. For such changes, use the `ALTER TABLE MODIFY LOB` syntax, which uses the `modify_LOB_storage_clause` from the `ALTER TABLE` BNF. Examples of changes that do not require moving the entire existing LOB data are: `RETENTION`, `PCTVERSION`, `CACHE`, `NOCACHELOGGING`, `NOLOGGING`, or `STORAGE` settings, shrinking the space used by the LOB data, and deallocating unused segments.

   > **See Also:**
   >
   > ALTER TABLE

2. Some changes to storage characteristics require changes to the way the data is stored, hence requiring movement of the entire existing LOB data. For such changes use the `ALTER TABLE MOVE LOB` syntax instead of the `ALTER TABLE MODIFY LOB` syntax because the former performs parallel operations on SecureFiles LOBs columns, making it a resource-efficient approach. The `ALTER TABLE MOVE LOB` syntax can process any arbitrary LOB storage clause represented by the `LOB_storage_clause` in the `ALTER TABLE` BNF, and will move the LOB data to a new location.
   Examples of changes that require moving the entire existing LOB data are: `TABLESPACE`, `ENABLE/DISABLE STORAGE IN ROW`, `CHUNK`, `COMPRESSION`, `DEDUPLICATION` and `ENCRYPTION` settings.

   As an alternative to `ALTER TABLE MOVE LOB`, 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 Also:**
>
> - ALTER TABLE for more information about `ALTER TABLE` statement.
> - DBMS_REDEFINITION for more information about `DBMS_REDEFINITION` package.

12.2.3 ALTER TABLE SecureFiles LOB Features

This section discusses the features of SecureFile LOBs that work with the `ALTER TABLE` statement.

- ALTER TABLE with Advanced LOB Compression
  When used with the `ALTER TABLE` statement, advanced LOB compression syntax alters the compression mode of the LOB column. The examples in this section demonstrate how to issue `ALTER TABLE` statements for specific compression scenarios.

- ALTER TABLE with Advanced LOB Deduplication
  When used with the `ALTER TABLE` statement, advanced LOB deduplication syntax alters the deduplication mode of the LOB column.
- ALTER TABLE with SecureFiles Encryption
  The examples in this section demonstrate how to issue ALTER TABLE statements for to enable SecureFiles encryption.

12.2.3.1 ALTER TABLE with Advanced LOB Compression

When used with the `ALTER TABLE` statement, advanced LOB compression syntax alters the compression mode of the LOB column. The examples in this section demonstrate how to issue `ALTER TABLE` statements for specific compression scenarios.

Example: Altering a SecureFiles LOB Column to Enable LOW Compression

```
ALTER TABLE t1 MOVE LOB(a) STORE AS SECUREFILE(COMPRESS LOW)
```

Example: Altering a SecureFiles LOB Column to Disable Compression

```
ALTER TABLE t1 MOVE LOB(a) STORE AS SECUREFILE(NOCOMPRESS)
```

Example: Altering a SecureFiles LOB Column to Enable HIGH Compression

```
ALTER TABLE t1 MOVE LOB(a) STORE AS SECUREFILE(COMPRESS HIGH);
```

Example: Altering a SecureFiles LOB Column to Enable Compression on One partition

```
ALTER TABLE t1 MOVE PARTITION p1 LOB(a) STORE AS SECUREFILE(COMPRESS HIGH);
```

12.2.3.2 ALTER TABLE with Advanced LOB Deduplication

When used with the `ALTER TABLE` statement, advanced LOB deduplication syntax alters the deduplication mode of the LOB column.

Before you enable deduplication, you can use the `GET_LOB_DEDUPLICATION_RATIO` function to estimate the space that you can save by enabling this feature for an existing LOB. You can also use this function to estimate the space that you can save by enabling deduplication, before migrating a BasicFiles LOB to SecureFiles LOB. This enables you to take an informed decision to enable deduplication. See GET_LOB_DEDUPLICATION_RATIO Function in *PL/SQL Packages and Types Reference*.

Disclaimer: The deduplication ratio is an approximate value, which is calculated based on the sampled rows in the LOB column. The actual space that you save when you enable deduplication for the complete table may be different.

The examples in this section demonstrate how to issue `ALTER TABLE` statements for specific deduplication scenarios.

Example: Altering a SecureFiles LOB Column to Disable Deduplication

```
ALTER TABLE t1 MOVE LOB(a) STORE AS SECUREFILE(KEEP_DUPLICATES);
```

Example: Altering a SecureFiles LOB Column to Enable Deduplication

```
ALTER TABLE t1 MOVE LOB(a) STORE AS SECUREFILE(DEDUPLICATE);
```

Example: Altering a SecureFiles LOB Column to Enable Deduplication on One Partition

```
ALTER TABLE t1 MOVE PARTITION p1 LOB(a) STORE AS SECUREFILE(DEDUPLICATE);
```

12.2.3.3 ALTER TABLE with SecureFiles Encryption

The examples in this section demonstrate how to issue ALTER TABLE statements for to enable SecureFiles encryption.

Consider the following points when using the `ALTER TABLE` statement with SecureFiles Encryption:

- The `ALTER TABLE` statement enables and disables SecureFiles Encryption. Using the `REKEY` option with the `ALTER TABLE` statement also enables you to encrypt LOB columns with a new key or algorithm.
- The `DECRYPT` option converts encrypted columns to clear text form.

> **See Also:**
>
> 'CREATE TABLE' Usage Notes for SecureFiles Encryption

Following examples demonstrate how to issue `ALTER TABLE` statements for specific encryption scenarios:

Example: Altering a SecureFiles LOB Column by Encrypting Based on `AES256` encryption

```
ALTER TABLE t1 MOVE LOB(a) STORE AS SECUREFILE(ENCRYPT USING 'AES256');
```

Example: Altering a SecureFiles LOB Column by Encrypting Based on a Password Key

```
ALTER TABLE t1 MOVE LOB(a)
    STORE AS SECUREFILE(ENCRYPT USING 'AES256' IDENTIFIED BY foo);
```

Example: Altering a SecureFiles LOB Column by Regenerating the Encryption key

```
ALTER TABLE t1 REKEY USING 'AES256';
```

12.3 Creating an Index on LOB Column

The contents of a LOB are often specific to the application, so an index on the LOB column will usually deal with application logic. You can create a function-based or a domain index on a LOB column to improve the performance of queries accessing data stored in LOB columns. You cannot build a B-tree or bitmap index on a LOB column.

Function-based and domain indexes are automatically updated when a DML operation is performed on the LOB column, or when a LOB is updated using an API like DBMS_LOB.

You can use the LOB Open/Close API to defer index maintenance to after a bunch of write operations. Opening a LOB in read-write mode defers any index maintenance on the LOB

column until you close the LOB. This is useful when you do not want the database to perform index maintenance every time you write to the LOB. This technique can improve the performance of your application if you are doing several write operations on the LOB while it is open. Any index on the LOB column is not valid until you explicitly close the LOB.

- Function-Based Indexing on LOB Columns
  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.
- Domain Indexing on LOB Columns
  Indexes created by using Extensible Indexing interfaces are known as Domain indexes.

> **See Also:**
>
> Before You Begin

12.3.1 Function-Based Indexing on LOB Columns

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.

> **See Also:**
>
> When to Use Function-Based Indexes

The following example demonstrates the creation of a function-based index on a LOB column using a SQL function:

```
-- Function-Based Index using a SQL function
CREATE INDEX ad_sourcetext_idx_sql ON
print_media(to_char(substr(ad_sourcetext,1,10)));
```

The following example demonstrates the creation of a function-based index on a LOB column using a PL/SQL function:

```
-- Function-Based Index using a PL/SQL function
-- LOB can be an input but cannot be the return type of hte function
CREATE OR REPLACE FUNCTION Ret1st2Char(CLobInput CLOB) RETURN CHAR
DETERMINISTIC IS
       First2Char         CHAR(2) ;
       NoOfChar           INTEGER ;
BEGIN
        NoOfChar := 2 ;
        DBMS_LOB.Read(CLobInput, NoOfChar, 1, First2Char) ;
        RETURN First2Char ;
END ;
/
```

```
CREATE INDEX ad_sourcetext_idx_plsql on
print_media(Ret1st2Char(ad_sourcetext));
```

12.3.2 Domain Indexing on LOB Columns

Indexes created by using Extensible Indexing interfaces are known as Domain indexes.

The database provides extensible indexing interfaces, 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.

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, and image 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 index types.

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. For instance, an inverted index for text documents or a quad-tree for spatial features.
- Build, delete, and update a domain index. The cartridge handles building and maintaining the index structures.
- 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 domain indexes, the database significantly reduces the effort needed to develop high-performance solutions that access complex data types such as LOBs.

- Extensible Optimizer
  Extensible Optmizer enables collection of statistics on user-defined functions and domain indexes.
- Text Indexes on LOB Columns
  If the contents of your LOB column correspond to that of a document type, users are allowed to index such a column using Oracle Text indexes.

> **See Also:**
>
> *Oracle Database Data Cartridge Developer's Guide*

12.3.2.1 Extensible Optimizer

Extensible Optmizer enables collection of statistics on user-defined functions and domain indexes.

The SQL optimizer cannot collect statistics over LOB columns nor can it estimate the cost and selectivity of predicates involving LOB columns. Instead, the Extensible Optimizer functionality allows authors of user-defined functions and domain indexes to create statistics collection, 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.

The Extensible Indexing interfaces enable you to define new operators, indextypes, and domain indexes. For such user-defined operators and domain indexes, the Extensible Optimizer interfaces allows users to control the three main components used by the optimizer to select an execution plan: statistics, selectivity, and cost. This allows the cartridge developer to tune the Extensible Optimizer for efficient execution of queries involving predicates or indexes over complex data types such as LOBs.

> **See Also:**
>
> Extensible Optimizer

12.3.2.2 Text Indexes on LOB Columns

If the contents of your LOB column correspond to that of a document type, users are allowed to index such a column using Oracle Text indexes.

For example, consider the following table `DOCUMENT_TABLE` storing text-based documents on a CLOB column:

```
CREATE TABLE document_table (
    docno NUMBER,
    document CLOB);
```

You can index the contents of the DOCUMENT column with one of the Oracle Text indexing options to speed up text-based queries. The following example will create a SEARCH index used for text-search queries over the DOCUMENT column.

```
CREATE INDEX document_index ON document_table (document) INDEXTYPE IS
CTXSYS.CONTEXT;

CREATE SEARCH INDEX document_index ON document_table (document);
```

> **Note:**
>
> You can create an Oracle Text index on other formats as well. Examples of other formats include PDF, JSON, or XML.

> **See Also:**
>
> Creating Oracle Text Indexes

12.4 LOBs in Partitioned Tables

Partitioning can simplify the manageability of large database objects. This section discusses various aspects of LOBs in partitioned tables.

Very large tables and indexes can be decomposed into smaller and more manageable pieces called partitions, which are entirely transparent to an application. You can partition tables that contain LOB columns. All partitioning schemes supported by Oracle are fully supported on LOBs.

> **See Also:**
>
> Partitions_ Views_ and Other Schema Objects
> Partitioning for All Databases

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.

The following section describes some of the ways you can manipulate LOBs in partitioned tables.

- Partitioning a Table Containing LOB Columns
  All partitioning schemes supported by Oracle are fully supported on LOBs. This section discusses the partitioning of tables with LOB columns.
- Default LOB Storage Attributes
  This section discusses the default LOB storage attributes.
- Partition Maintenance Operation
  This section discusses maintenance operations on partitioned tables with LOB columns.
- Creating an Index on a Table Containing Partitioned LOB Columns
  To improve the performance of queries, you can create local or global indexes on partitioned LOB columns.

12.4.1 Partitioning a Table Containing LOB Columns

All partitioning schemes supported by Oracle are fully supported on LOBs. This section discusses the partitioning of tables with LOB columns.

You can partition a table containing LOB columns using any of 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.

The data dictionary views `USER_LOB_PARTITIONS`, `ALL_LOB_PARTITIONS` and `DBA_LOB_PARTITIONS` provide partition specific information for a LOB column.

Different partitions can have different inline sizes. This is useful if you want to `EXCHANGE` a new table into a partition of an existing table. If a partition level inline size is not specified, the column's table level default is used. In the case of composite partitioning, the sub-partition-level inline size takes precedence over the partition-level inline size, which takes precedence over the table-level inline size. During new partition creation, if inline size values are not specified, the table level defaults are used. Inline size values are never `NULL`.

Example 12-15 A partitioned table with LOB columns:

```
CREATE TABLE print_media
    ( product_id        NUMBER(6),
      ad_id             NUMBER(6),
      ad_sourcetext     CLOB)
    LOB (ad_sourcetext)  STORE AS SECUREFILE (TABLESPACE tbs_2)
    PARTITION BY RANGE(product_id)
    (PARTITION P1 VALUES LESS THAN (1000)
         LOB (ad_sourcetext) STORE AS BASICFILE (TABLESPACE tbs_1),
     PARTITION P2 VALUES LESS THAN (2000)
         LOB (ad_sourcetext) STORE AS (TABLESPACE tbs_2 COMPRESS HIGH),
     PARTITION P3 VALUES LESS THAN (3000));
```

> **See Also:**
>
> Summary of CREATE TABLE LOB Storage Parameters for Securefile LOBs

Example 12-16 A partitioned table with different inline sizes for LOB columns

```
CREATE TABLE print_media
    ( product_id        NUMBER(6),
      ad_content        BLOB,
      ad_sourcetext     CLOB)
      LOB (ad_content) STORE AS SECUREFILE (ENABLE STORAGE IN ROW)
      LOB (ad_sourcetext) STORE AS SECUREFILE (ENABLE STORAGE IN ROW 8000)
TABLESPACE tbs_1
    PARTITION BY RANGE(product_id)
    (PARTITION P1 VALUES LESS THAN (1000)
        LOB (ad_sourcetext) STORE AS SECUREFILE (ENABLE STORAGE IN ROW 4000),
    PARTITION P2 VALUES LESS THAN (2000)
        LOB (ad_sourcetext) STORE AS (ENABLE STORAGE IN ROW 8000 COMPRESS
HIGH),
    PARTITION P3 VALUES LESS THAN (3000)
        LOB (ad_content) STORE AS (DISABLE STORAGE IN ROW));
```

12.4.2 Default LOB Storage Attributes

This section discusses the default LOB storage attributes.

In the above example, the default storage attribute for LOB column `ad_sourcetext` is mentioned as "`STORE AS SECUREFILE (TABLESPACE tbs_2)`". This means that if no LOB storage clause is provided for any partition, this default will be used. In this example, partition `P3` uses tablespace `tbs_2` since no LOB storage is specified. Similarly, `SECUREFILE` is the default storage and is used by partitions `P2` and `P3`, but partition `P1` overrides it to specify BasicFile storage.

The dictionary views `USER_PART_LOBS`, `ALL_PART_LOBS` and `DBA_PART_LOBS` provide information on default LOB storage options for a LOB column in a table.

The table level default LOB storage attribute can be changed, as shown in the example below:

```
ALTER TABLE print_media MODIFY DEFAULT ATTRIBUTES  LOB (ad_sourcetext)
    (TABLESPACE  tbs_1);
```

The change in the default attribute will not affect the existing partitions. Any new partitions created without LOB storage clause will inherit the default values for that column.

12.4.3 Partition Maintenance Operation

This section discusses maintenance operations on partitioned tables with LOB columns.

All partitioning maintenance operations are supported with LOB columns. Here are some examples:

Example 12-17 Adding Partition containing LOBs

```
ALTER TABLE print_media ADD PARTITION P4 VALUES LESS THAN (4000)
         LOB (ad_sourcetext) STORE AS SECUREFILE(TABLESPACE tbs_2);
```

Example 12-18 Modifying Partition Containing LOBs

```
ALTER TABLE print_media MODIFY PARTITION P3 LOB(ad_sourcetext)
     (RETENTION AUTO);
```

Example 12-19 Moving Partition Containing LOBs

```
ALTER TABLE print_media MOVE PARTITION P1 LOB(ad_sourcetext)
    STORE AS (TABLESPACE tbs_3 COMPRESS LOW);
```

The example above moves a LOB partition into a different tablespace, which can be useful if the tablespace is no longer large enough to hold the partition. Move partition can also be used to perform other operations that require moving the LOB data, such as performing a `COMPRESS` operation on the LOB, or changing the `ENABLE / DISABLE STORAGE IN ROW` option.

Example 12-20 Splitting Partitions Containing LOBs

You can split a partition containing LOBs into two 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 print_media SPLIT PARTITION  P1 AT(500) into
(PARTITION P1A LOB(ad_sourcetext) STORE AS (TABLESPACE tbs_1),
PARTITION P1B LOB(ad_sourcetext) STORE AS (TABLESPACE tbs_2)) UPDATE INDEXES;
```

Example 12-21 Merging Partitions Containing LOBs

Merging partitions is useful for reclaiming unused partition space. For example:

```
ALTER TABLE print_media MERGE PARTITIONS P1A, P1B INTO PARTITION P1;
```

Example 12-22 Exchange Partition containing LOB column with non-partitioned table

Exchanging partitions with a table that has partitioned LOB columns using the `ALTER TABLE ... EXCHANGE PARTITION` clause. Exchange partition is a powerful tool to change new data / partitions to a newer storage format without the costly operation of migrating old data. You can exchange partition with LOB data having different storage option, e.g. partition p1 of BasicFile data in Example 11-15 can be exchanged with non-partitioned table with LOB column stored in SecureFile Compressed form:

```
CREATE TABLE print_media_nonpart
    ( product_id NUMBER(6),
      ad_id NUMBER(6),
      ad_sourcetext CLOB)
      LOB (ad_sourcetext) STORE AS SECUREFILE (COMPRESS HIGH);

ALTER TABLE print_media  EXCHANGE PARTITION p1 WITH TABLE print_media_nonpart;
```

12.4.4 Creating an Index on a Table Containing Partitioned LOB Columns

To improve the performance of queries, you can create local or global indexes on partitioned LOB columns.

Only function-based and domain indexes are supported on LOB columns. Other types of indexes, such as unique indexes are not supported with LOBs.

For example:

```
CREATE INDEX ad_sourcetext_idx_sql on print_media
(to_char(substr(ad_sourcetext,1,10)))
      GLOBAL;


CREATE INDEX ad_sourcetext_idx_sql on print_media
(to_char(substr(ad_sourcetext,1,10)))
      LOCAL;
```

12.5 LOBs in Index Organized Tables

Index Organized Tables (IOTs) support LOB and BFILE columns.

For the most part, SQL DDL, DML, and piecewise 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.

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.

LOBs in Partitioned Index-Organized Tables

LOB columns and attributes can be stored in partitioned index-organized tables.

Index-organized tables can have 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.

Restrictions on Index Organized Tables with LOB Columns

The ALTER TABLE MOVE operation cannot be performed on an index organized table with a LOB column in parallel. Instead, use the `NOPARALLEL` clause to move the LOB column for such tables. For example:

```
ALTER TABLE t1 MOVE LOB(a) STORE AS (<tablespace users>) NOPARALLEL;
```

13
Advanced Design Considerations

This section discusses the design considerations for more advanced application development issues.

- Read-Consistent Locators
  Oracle Database provides the same read consistency mechanisms for LOBs as for all other database reads and updates of scalar quantities.
- LOB Locators and Transaction Boundaries
  LOB locators can be used in both transactions as well as transaction IDs.
- LOBs in the Object Cache
  When you copy one object to another in the object cache with a LOB locator attribute, only the LOB locator is copied.
- Guidelines for Creating Terabyte sized LOBs
  To create terabyte LOBs in supported environments, use the following guidelines to make use of all available space in the tablespace for LOB storage.

13.1 Read-Consistent Locators

Oracle Database provides the same read consistency mechanisms for LOBs as for all other database reads and updates of scalar quantities.

Read consistency has some special applications to LOB locators that you must understand. The following sections discuss read consistency and include examples which should be looked at in relationship to each other.

- A Selected Locator Becomes a Read-Consistent Locator
  A read-consistent locator contains the snapshot environment as of the point in time of the `SELECT` operation.
- 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 `UPDATE` operation.
- Example of Updating LOBs Through Updated Locators
  Learn about updating LOBs through Locators in this section.
- Example of Updating a LOB Using SQL DML and DBMS_LOB
  Using the `print_media` table in the following example, a `CLOB` locator is created as `clob_selected`.
- Example of Using One Locator to Update the Same LOB Value
  You may avoid many pitfalls if you use only one locator to update a given LOB value. Learn about it in this section.
- Example of Updating a LOB with a PL/SQL (DBMS_LOB) Bind Variable
  Learn about updating a LOB with a PL/SQL bind variable in this section.
- Example of Deleting a LOB Using Locator
  Learn about deleting a LOB with a PL/SQL bind variable in this section.

- Ensuring Read Consistency
  This script in this section 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 without read inconsistencies.

> **See Also:**
>
> - *Oracle Database Concepts* for general information about read consistency

13.1.1 A Selected Locator Becomes a Read-Consistent Locator

A read-consistent locator contains the snapshot environment as of the point in time of the `SELECT` operation.

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.

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.

13.1.2 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 `UPDATE` operation.

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`.

Example 13-1

```
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;
/
```

13.1.3 Example of Updating LOBs Through Updated Locators

Learn about updating LOBs through Locators in this section.

When you update the value of the persistent LOB through the LOB locator (`L1`), `L1` is updated to contain the current snapshot environment.

This snapshot is 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.

> **Note:**
>
> 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*

13.1.4 Example of Updating a LOB Using SQL DML and DBMS_LOB

Using the `print_media` table 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;
/
```

13.1.5 Example of Using One Locator to Update the Same LOB Value

You may avoid many pitfalls if you use only one locator to update a given LOB value. Learn about it in this section.

> **Note:**
>
> Avoid updating the same LOB with different locators.

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;
/
```

13.1.6 Example of Updating a LOB with a PL/SQL (DBMS_LOB) Bind Variable

Learn about updating a LOB with a PL/SQL bind variable in this section.

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, 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;
/
```

13.1.7 Example of Deleting a LOB Using Locator

Learn about deleting a LOB with a PL/SQL bind variable in this section.

The following example illustrates that LOB content through a locator selected at a given point of time is available even though the LOB is deleted in the same transaction.

In the following example, using table `print_media`, two CLOBs are created as potential locators:`clob_selected` and `clob_copied`.

Note these progressions in the example at times `t1` through `t3`:

- At the time of the first `SELECT INTO` (at `t1`), the value in`ad_sourcetext` for `ad_id` value 20020 is associated with the locator `clob_selected`. The value in `ad_sourcetext` for `ad_id` value 20021 is associated with the locator `clob_copied`.
- The second operation (at `t2`) deletes the row with `ad_id` value 20020. However, a `DBMS_LOB.READ()` of the value through `clob_selected` (at `t1`) reveals that it is a read-consistent locator, continuing to refer to the same value as of the time of its SELECT.
- The third operation (at `t3`), copies the LOB data read through `clob_selected` into the LOB `clob_copied`. `DBMS_LOB.READ()` of the value through `clob_selected` and `clob_copied` are now the same and refer to the same value as of the time of `SELECT` of `clob_selected`.

```
INSERT INTO PRINT_MEDIA VALUES (2056, 20020, EMPTY_BLOB(),
    'abcd', EMPTY_CLOB(), EMPTY_CLOB(), NULL, NULL, NULL, NULL);

INSERT INTO PRINT_MEDIA VALUES (2057, 20021, EMPTY_BLOB(),
    'cdef', EMPTY_CLOB(), EMPTY_CLOB(), NULL, NULL, NULL, NULL);

DECLARE
  clob_selected CLOB;
  clob_copied CLOB;
  buffer VARCHAR2(20);
  read_amount INTEGER := 20;
  read_offset INTEGER := 1;

BEGIN
  -- At time t1:
  SELECT ad_sourcetext INTO clob_selected
      FROM PRINT_MEDIA
      WHERE ad_id = 20020
      FOR UPDATE;

  SELECT ad_sourcetext INTO clob_copied
      FROM PRINT_MEDIA
      WHERE ad_id = 20021
      FOR UPDATE;

  dbms_lob.read(clob_selected, read_amount, read_offset,buffer);
  dbms_output.put_line(buffer);
  -- Produces the output 'abcd'

  dbms_lob.read(clob_copied, read_amount, read_offset,buffer);
```

```
  dbms_output.put_line(buffer);
  -- Produces the output 'cdef'

  -- At time t2: Delete the CLOB associated with clob_selected
  DELETE FROM PRINT_MEDIA WHERE ad_id = 20020;

  dbms_lob.read(clob_selected, read_amount, read_offset,buffer);
  dbms_output.put_line(buffer);
  -- Produces the output 'abcd'

  -- At time t3:
  -- Copy using clob_selected
  dbms_lob.copy(clob_copied, clob_selected, 4000, 1, 1);
  dbms_lob.read(clob_copied, read_amount, read_offset,buffer);
  dbms_output.put_line(buffer);
  -- Produces the output 'abcd'

END;
/
```

13.1.8 Ensuring Read Consistency

This script in this section 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 without 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.

13.2 LOB Locators and Transaction Boundaries

LOB locators can be used in both transactions as well as transaction IDs.

- About LOB Locators and Transaction Boundaries
  Learn about LOB locators and transaction boundaries in this section.

- Read and Write Operations on a LOB Using Locators
  You can always read LOB data using the locator irrespective of whether or not the locator contains a transaction ID. Learn about various aspects of it in this section.
- Selecting the Locator Outside of the Transaction Boundary
  This section has two scenarios that describe techniques for using locators in non-serializable transactions when the locator is selected outside of a transaction.
- Selecting the Locator Within a Transaction Boundary
  This section has two scenarios that describe techniques for using locators in non-serializable transactions when the locator is selected within a transaction.
- LOB Locators Cannot Span Transactions
  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.
- Example of Locator Not Spanning a Transaction
  The example of locator not spanning a transaction uses the `print_media` table.

> **See Also:**
>
> Locator Interface for LOBs for more information about LOB locators

13.2.1 About LOB Locators and Transaction Boundaries

Learn about LOB locators and transaction boundaries in this section.

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.

13.2.2 Read and Write Operations on a LOB Using Locators

You can always read LOB data using the locator irrespective of whether or not the locator contains a transaction ID. Learn about various aspects of it in this section.

- 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 examples Selecting the Locator Outside of the Transaction Boundary, Selecting the Locator Within a Transaction Boundary, LOB Locators Cannot Span Transactions, and Example of Locator Not Spanning a Transaction show the relationship between locators and *non-serializable* transactions

13.2.3 Selecting the Locator Outside of the Transaction Boundary

This section has two scenarios that 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.

13.2.4 Selecting the Locator Within a Transaction Boundary

This section has two scenarios that 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" 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.

13.2.5 LOB Locators Cannot Span Transactions

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.

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.

In the following code example, a `CLOB` locator called `clob_updated` is created and following operations are performed:

- 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.

13.2.6 Example of Locator Not Spanning a Transaction

The example of locator not spanning a transaction uses the `print_media` table.

```
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;
/
```

13.3 LOBs in the Object Cache

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 that refers to *one and the same* LOB *value*. Only when you flush the target LOB, a separate physical copy of the LOB value is made, which is distinct from the source LOB value.

> **See Also:**
>
> Example of Updating LOBs and Read-Consistency for a description of what version of the LOB value is seen by each object if a write operation 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.

Consider the following object cache issues for LOB and BFILE attributes:

- Persistent LOB attributes: Creating an object in the 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 zero (0) length. Once you refresh the object in the object cache, using the `OCI_PIN_LATEST` function, the real LOB locator is read into the attribute, and you can then call the OCI LOB APIs to write data to the LOB.

- `BFILE` attributes: Creating an object in the object cache, sets the `BFILE` attribute to `NULL`.

  When creating an object with a `BFILE` attribute, the `BFILE` is set to `NULL`. You must update it with a valid `DIRECTORY` object name and file name before reading from the `BFILE`.

13.4 Guidelines for Creating Terabyte sized LOBs

To create terabyte 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. 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 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 Terabyte LOBs
  The following example illustrates how to create a tablespace and table to store terabyte LOBs.

13.4.1 Creating a Tablespace and Table to Store Terabyte LOBs

The following example illustrates how to create a tablespace and table to store terabyte 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 1000M NEXT 1000M MAXEXTENTS
                                UNLIMITED));
```

14

Managing LOBs: Database Administration

You must perform various administrative tasks to set up, maintain, and use a database that contains LOBs.

> **Note:**
>
> LOBs are not supported when the Container Database root and Pluggable Databases are in different character sets. For more information, refer to Relocating a PDB Using CREATE PLUGGABLE DATABASE.

- Initialization Parameter for SecureFiles LOBs
  As a database administrator, you can configure the conditions that control or allow creation of SecureFiles LOBs or BasicFiles LOBs. Typically, you set up the `DB_SECUREFILE` parameter in the `init.ora` file for this purpose.
- Database Character Set Considerations
  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.
- Database Utilities for Loading Data into LOBs
  Certain utilities are recommended for bulk loading data into LOB columns as part of the database set up or maintenance tasks.
- LOB Migration with Data Pump
- BFILEs Management
  This section describes various administrative tasks to manage databases that contain `BFILE`s.
- Managing LOB Signatures
  This section describes how to configure LOB signatures.

14.4 LOB Migration with Data Pump

See Migrating LOBs with Data Pump.

14.1 Initialization Parameter for SecureFiles LOBs

As a database administrator, you can configure the conditions that control or allow creation of SecureFiles LOBs or BasicFiles LOBs. Typically, you set up the `DB_SECUREFILE` parameter in the `init.ora` file for this purpose.

The `DB_SECUREFILE` initialization parameter is dynamic and can be modified with the `ALTER SYSTEM` statement in the following way:

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

The valid values for this parameter are described in the following table:

Value	Description
NEVER	Prevents SecureFiles LOBs from being created. If NEVER is specified, then any LOBs that are specified as SecureFiles LOBs are created as BasicFiles LOBs. If storage options are not specified, then the BasicFiles LOB defaults are used. All SecureFiles LOB-specific storage options and features such as compress, encrypt, and deduplicate throw an exception.
IGNORE	Always create BasicFile LOBs, and ignore any errors that the SecureFile LOB options might cause. If IGNORE is specified, then 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.
PREFERRED(default)	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).
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.

14.2 Database Character Set Considerations

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.

> **See Also:**
>
> Choosing a Character Set

14.3 Database Utilities for Loading Data into LOBs

Certain utilities are recommended for bulk loading data into LOB columns as part of the database set up or maintenance tasks.

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

- SQL*Loader
- External Tables
- Oracle Data Pump

- Loading LOBs with SQL*Loader
  Learn about conventional and direct-path loads, when Oracle recommends that you use direct-path loads, and what rules and guidelines you should follow to avoid issues.
- Loading BFILEs with SQL*Loader
  This section describes how to load data from files in the file system into a `BFILE` column using SQL*Loader.
- Loading LOBs with External Tables
  External tables are particularly useful for loading large numbers of records from a single file, so that each record appears in its own row in the table.

14.3.1 Loading LOBs with SQL*Loader

Learn about conventional and direct-path loads, when Oracle recommends that you use direct-path loads, and what rules and guidelines you should follow to avoid issues.

There are two options for loading large object (LOB) data:

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, a direct-path load does not compete with other users for database resources, so it can usually load data at near disk speed. Be aware that there are also other restrictions, security, and backup implications for direct path loads, which you should review.

For each of these options of loading large object data (LOBs), you can use the following techniques to load data into LOBs:

- Loading LOB data from primary data files.

  When you load data from a primary data file, the data for the LOB column is part of the record in the file that you are loading.
- Loading LOB data from a secondary data file using LOB files.

  When you load data from a secondary data file, the data for a LOB column is in a different file from the primary data file. Instead of the data itself, the primary data file contains information about the location of the content of the LOB data in other files.

Recommendations for Using SQL*Loader to Load LOBs

Oracle recommends that you keep the following guidelines and rules in mind when loading LOBs using SQL*Loader:

- Tables that you want to load must already exist in the database. SQL*Loader never creates tables. It loads existing tables that either contain data, or are empty.
- When you load data from LOB files, specify the maximum length of the field corresponding to a LOB-type column. If the maximum length is specified, then SQL*Loader uses this length as a hint to help optimize memory usage. You should ensure that the maximum length you specify does not underestimate the true maximum length.

- If you use conventional path loads, then be aware that 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.
- If you use direct-path loads, then be aware that 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. One way to work around this problem is to use the `ROWS` option to read a smaller number of rows in each data save.

  Only use direct path loads to load XML documents that are known to be valid into XMLtype columns that are stored as CLOBS. Direct path load does not validate the format of XML documents as the are loaded as CLOBs.

  With direct-path loads, errors can be critical. In 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. As a result, if the entire LOB with the error is contained in the first piece, then that LOB column is either empty or truncated.

  You can also use the Direct Path API to load LOBs.

Privileges Required for Using SQL*Loader to Load LOBs

The following privileges are required for using SQL*Loader to load LOBs:

- You must have `INSERT` privileges on the table that you want to load.
- You must have `DELETE` privileges on the table that you want to load, if you want to use the `REPLACE` or `TRUNCATE` option to empty out the old data before loading the new data in its place.

Example 14-1 Loading LOB from a primary data file using Delimited Fields

Review this example to see how to load LOB data in delimited fields. Note the callouts "1" and "2" in **bold**:

Control File Contents

```
LOAD DATA
INFILE 'sample.dat' "str '|'"
INTO TABLE person_table
FIELDS TERMINATED BY ','
   (name        CHAR(25),
1  "RESUME"     CHAR(507) ENCLOSED BY '<startlob>' AND '<endlob>')
```

Data File (`sample.dat`)

```
Julia Nayer,<startlob>        Julia Nayer
                          500 Example Parkway
                          jnayer@example.com ...   <endlob>
2  |Bruce Ernst, .......
```

> **Note:**
>
> The callouts, in bold, to the left of the example correspond to the following notes:
>
> 1. `<startlob>` and `<endlob>` are the enclosure strings. With the default byte-length semantics, the maximum length for a LOB that can be read using `CHAR(507)` is 507 bytes. If character-length semantics were used, then the maximum would be 507 characters. For more information, refer to character-length semantics.
> 2. If the record separator `'|'` had been placed right after `<endlob>` and followed with the newline character, then the newline would have been interpreted as part of the next record. An alternative would be to make the newline part of the record separator (for example, `'|\n'` or, in hexadecimal notation, `X'7C0A'`).

Example 14-2 Loading a LOB from secondary data file, using Delimited Fields:

In this example, note the callout "1" in **bold**:

Control File Contents

```
LOAD DATA
INFILE 'sample.dat'
INTO TABLE person_table
FIELDS TERMINATED BY ','
   (name     CHAR(20),
1  "RESUME"    LOBFILE( CONSTANT 'jqresume') CHAR(2000)
               TERMINATED BY "<endlob>\n")
```

Data File (sample.dat)

```
Johny Quest,
Speed Racer,
```

Secondary Data File (jqresume.txt)

```
    Johny Quest
500 Oracle Parkway
   ... <endlob>
    Speed Racer
400 Oracle Parkway
   ... <endlob>
```

> **Note:**
>
> The callout, in bold, to the left of the example corresponds to the following note:
>
> 1. Because a maximum length of 2000 is specified for `CHAR`, SQL*Loader knows what to expect as the maximum length of the field, which can result in memory usage optimization. *If you choose to specify a maximum length, then you should be sure not to underestimate its value.* The `TERMINATED BY` clause specifies the string that terminates the LOBs. Alternatively, you can use the `ENCLOSED BY` clause. The `ENCLOSED BY` clause allows a bit more flexibility with the relative positioning of the LOBs in the `LOBFILE`, because the LOBs in the `LOBFILE` do not need to be sequential.

Related Topics

- Oracle Call Interface Direct Path Load Interface
- Loading Objects, LOBs, and Collections with SQL*Loader

14.3.2 Loading BFILEs with SQL*Loader

This section describes how to load data from files in the file system into a `BFILE` column using SQL*Loader.

> **Note:**
>
> - 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.
> - 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:**
>
> DIRECTORY Objects for more information

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 `BFILE`s.

> **Note:**
>
> You need to set up the following data structures for certain examples to work:
>
> ```
> CONNECT pm/pm
> CREATE OR REPLACE DIRECTORY adgraphic_photo as '/tmp';
> CREATE OR REPLACE DIRECTORY adgraphic_dir as '/tmp';
> ```

In the following example, only the file name is specified dynamically. The directory name, `adgraphic_photo`, is in quotation marks. Therefore, the string is used as is, and is not capitalized.

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 "adgraphic_photo", FileName))
```

Data file:

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

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,
```

14.3.3 Loading LOBs with External Tables

External tables are particularly useful for loading large numbers of records from a single file, so that each record appears in its own row in the table.

> **Note:**
>
> Loading LOBs with External Tables

- Overview of LOBs and External Tables
  Learn the benefits of using external tables with your database to read and write data, and to understand how to create them.

14.3.3.1 Overview of LOBs and External Tables

Learn the benefits of using external tables with your database to read and write data, and to understand how to create them.

External tables enable you to treat the contents of external files as if they are rows in a table in your Oracle Database. After you create an external table, you can then use SQL statements to read rows from the external table, and insert them into another table.

To perform these operations, Oracle Database uses one of the following access drivers:

- The `ORACLE_LOADER` access driver reads text files and other file formats, similar to SQL Loader.
- The `ORACLE_DATAPUMP` access driver creates binary files that store data returned by a query. It also returns rows from files in binary format.

When you create an external table, you specify column and data types for the external table. The access driver has a list of columns in the data file, and maps the contents of the field in the data file to the column with the same name in the external table. The access driver takes care of finding the fields in the data source, and converting these fields to the appropriate data type for the corresponding column in the external table. After you create an external table, you can load the target table by using an `INSERT AS SELECT` statement.

One of the advantages of using external tables to load data over SQL Loader is that external tables can load data in parallel. The easiest way to do this is to specify the `PARALLEL` clause as part of `CREATE TABLE` for both the external table and the target table.

Example 14-3

This example creates a table, `CANDIDATE`, that can be loaded by an external table. When it is loaded, it then creates an external table, `CANDIDATE_XT`. Next, it executes an `INSERT` statement to load the table. The `INSERT` statement includes the `+APPEND` hint, which uses direct load to insert the rows into the table `CANDIDATES`. The `PARALLEL` parameter tells SQL that the tables can be accessed in parallel.

The `PARALLEL` parameter setting specifies that there can be four (`4`) parallel query processes reading from `CANDIDATE_XT`, and four parallel processes inserting into `CANDIDATE`. Note that LOBs that are stored as `BASICFILE` cannot be loaded in parallel. You can only load `SECUREFILE LOBS` in parallel. The variable *`additional-external-table-info`* indicates where additional external table information can be inserted.

```
CREATE TABLE CANDIDATES

  (candidate_id       NUMBER,

   first_name         VARCHAR2(15),
```

```
   last_name          VARCHAR2(20),

   resume             CLOB,

   picture            BLOB

  ) PARALLEL 4;


CREATE TABLE CANDIDATE_XT

  (candidate_id       NUMBER,

   first_name         VARCHAR2(15),

   last_name          VARCHAR2(20),

   resume             CLOB,

   picture            BLOB

  ) PARALLEL 4;

ORGANIZATION EXTERNAL additional-external-table-info PARALLEL 4;

INSERT /*+APPEND*/ INTO CANDIDATE SELECT * FROM CANDIDATE_XT;
```

File Locations for External Tables Created By Access Drivers

All files created or read by `ORACLE_LOADER` and `ORACLE_DATAPUMP` reside in directories pointed to by directory objects. Either the DBA or a user with the `CREATE DIRECTORY` privilege can create a directory object that maps a new to a path on the file system. These users can grant `READ`, `WRITE` or `EXECUTE` privileges on the created directory object to other users. A user granted `READ` privilege on a directory object can use external tables to read files from directory for the directory object. Similarly, a user with `WRITE` privilege on a directory object can use external tables to write files to the directory for the directory object.

Example 14-4 Creating Directory Object

The following example shows how to create a directory object and grant `READ` and `WRITE` access to user `HR`:

```
create directory HR_DIR as /usr/hr/files/exttab;

grant read, write on directory HR_DIR to HR;
```

> **Note:**
>
> When using external tables in an Oracle Real Application Clusters (Oracle RAC) environment, you must make sure that the directory pointed to by the directory object maps to a directory that is accessible from all nodes.

14.5 BFILEs Management

This section describes various administrative tasks to manage databases that contain BFILEs.

- Guidelines for DIRECTORY Usage
  Learn about the guidelines for efficient management of DIRECTORY objects.
- Rules for Using Directory Objects and BFILEs
  You can create a directory object or BFILE objects if these conditions are met.
- Setting Maximum Number of Open BFILEs
  Only limited number of BFILEs can be open simultaneously in each session. Learn to define this number in this section.

14.5.1 Guidelines for DIRECTORY Usage

Learn about the guidelines for efficient management of DIRECTORY objects.

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 or DROP 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. 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. The user must either 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.

> **See Also:**
>
> Security on Directory Objects

14.5.2 Rules for Using Directory Objects and BFILEs

You can create a directory object or BFILE objects if these conditions are met.

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.

14.5.3 Setting Maximum Number of Open BFILEs

Only limited number of BFILEs can be open simultaneously in each session. Learn to define this number in this section.

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.

> **See Also:**
>
> DIRECTORY Objects

14.6 Managing LOB Signatures

This section describes how to configure LOB signatures.

You can configure signature-based security for large object (LOB) locators using the LOB_SIGNATURE_ENABLE initialization parameter.

- To enable signature, set the `LOB_SIGNATURE_ENABLE` initialization parameter at `init.ora`, or using the following `ALTER SYSTEM` command. Also ensure that you have set the compatibility to 12.2.0.2 or above.

  ```
  ALTER SYSTEM SET LOB_SIGNATURE_ENABLE = [TRUE|FALSE];
  ```

- The following `ALTER` statement helps to encrypt, re-key, and delete the signature keys.

  ```
  ALTER DATABASE DICTIONARY [ENCRYPT|REKEY|DELETE] CREDENTIALS;
  ```

For more information, refer to the Oracle Database Security Guide.

> **See Also:**
>
> *Oracle Database Security Guide*

15

Migrating Columns to SecureFile LOBs

Oracle recommends that you migrate your existing columns that use the LONG or LONG RAW datatype or BasicFile LOB storage to the SecureFile LOB storage. This chapter covers various techniques to help with this migration.

All forms of `LONG` data types (`LONG`, `LONG RAW`, `LONG VARCHAR`, `LONG VARRAW`) were deprecated in Oracle8i Release 8.1.6. For succeeding releases, the `LONG` data type was provided for backward compatibility with existing applications. In new applications developed with later releases, Oracle strongly recommends that you use `CLOB` and `NCLOB` data types for large amounts of character data.

> **Note:**
>
> All discussions in this chapter are valid for migrating the `LONG` datatype to `CLOB` or `NCLOB`, and the `LONG RAW` datatype to `BLOB`. Most of the text in this chapter talks just about the `LONG` datatype for brevity.

- Migration Considerations
  This section discusses various factors to be considered while migrating LOB data types or storage.
- Migration Methods
  This section describes various methods you can use to migrate LONG or BasicFile LOB data to SecureFile storage.
- Other Considerations While Migrating LONG Columns to LOBs
  This section describes some more considerations when migrating LONG columns to LOBs.

15.1 Migration Considerations

This section discusses various factors to be considered while migrating LOB data types or storage.

Space requirements

Most migration techniques copy the contents of the table into a new space, and free the old space at the end of the operation. This temporarily doubles the space requirements. If space is limited, then you can perform the BasicFile to SecureFile migration one partition at a time.

Preventing Generation of REDO Data When Migrating

Migrating `LONG` datatype or BasicFiles LOB columns to SecureFile generates redo data, which can slow down the performance during the migration.

Redo changes for a column being converted to SecureFiles 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.

You can prevent redo space generation during migration to SecureFiles LOB by following the following steps:

1. Specify the NOLOGGING storage parameter for any new SecureFiles LOB columns.
2. Turn LOGGING on when the migration is complete.
3. Make a backup of the tablespaces containing the table and the LOB column.

15.2 Migration Methods

This section describes various methods you can use to migrate LONG or BasicFile LOB data to SecureFile storage.

Topics

- Migrating LOBs with SecureFiles Migration Utility
  This is the recommended method to migrate BasicFile LOB data to SecureFile storage. This utility encapsulates all the functionality offered by Online Redefinition and saves you the time and effort involved in manually running a series of API calls.
- Migrating LOBs with Online Redefinition
  Use Online redefinition to migrate LONG or BasicFile LOB data to SecureFile storage by running several API calls.
- Migrating LOBs with Data Pump
  Oracle Data Pump can either recreate tables as they are in your source database, or recreate LOB columns as SecureFile LOBs.

15.2.1 Migrating LOBs with SecureFiles Migration Utility

This is the recommended method to migrate BasicFile LOB data to SecureFile storage. This utility encapsulates all the functionality offered by Online Redefinition and saves you the time and effort involved in manually running a series of API calls.

Advantages

- No need to take the table or partition offline.
- Perform the migration at the database, schema, table or LOB segment level.
- After migrating the data, you can also use the SecureFiles Migration Utility to compress the SecureFile LOBs.

Disadvantages

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

To migrate BasicFile LOB data to SecureFile storage using the SecureFiles migration utility:

1. Run the following command as is to create a table.

```
create table migration_config (ctime date, data clob , constraint c1
check(data is json));
```

2. Make a single entry in the table to specify the schema, table, and columns that you want to migrate. Enter `first` as the value for `run_type` as this is the first time you are running the script. For others, provide values based on your environment.

Example Command

The following example shows a example single entry that specifies the objects that you want to migrate.

```
insert into migration_config values
    (systimestamp,
    '{"schema_name" : ["TEST2"],
    "table_name" : ["TEST1.TAB_DEFERRED_SEGCREATION1",
"TEST1.TAB_NON_LOB1",
    "TEST1.BASIC1A", "TEST1.BASIC3A"],
    "column_name" : ["TEST1.TAB_PARTS1.a",
    "TEST1.BASIC123.a", "TEST1.BASIC125.a"],
    "metadata_schema_name" : "TEMP1",
    "run_type" : "first",
    "directory_path" : "<full path to folder for log files>",
    "compress_storage_rec_threshold" : 5000,
    "trace" : 1}');
```

Where,

- **`schema_name`**: Mandatory. Specify a comma-separated list of schema names that you want to migrate and compress. If you do not specify a value, `{"schema_name" : []}`, the entire schema is not migrated. Instead the script checks for finer granularity that may be specified in the `table_name` or `column_name` arrays.
- **`table_name`**: Mandatory. Specify a comma-separated list of tables that you want to migrate and compress. You must enter the name in the following format, `<schema_name>.<table_name>`, where the schema name prefixes the table name. In the following example, `TEST1` is the name of the schema and `BASIC1A` and `BASIC3A` are the names of the tables.

  ```
  "table_name" : ["TEST1.BASIC1A", "TEST1.BASIC3A"]
  ```

  If you do not specify a value, `"table_name" : []`, then the script checks for finer granularity that may be specified in the `column_name` array.
- **`column_name`**: Mandatory. Specify a comma-separated list of columns that you want to migrate and compress. You must enter the name in the following format, `<schema_name>.<table_name>.<column_name>`, where the schema name and table name prefixes the column name.
  If you do not specify a value, `"column_name" : []`, then all LOB columns belonging to the specified table are migrated.
- **`metadata_schema_name`**: Mandatory. Enter a unique schema name. This is a temporary schema which the script uses to store metadata tables or reports that are generated during the migration. The schema must have a default ASSM tablespace to store intermediate data.
- **`run_type`**: Mandatory. The permitted values are `first`, `second`, and `third` corresponding to the three stages in which you execute the script.
- **`directory_path`**: Mandatory. Enter the complete path to the folder where you want to save the generated log files if `trace` is 1.
- **`compress_storage_rec_threshold`**: Optional. The script recommends compressing LOBs that are above the storage threshold that you enter. The default value is 5000 MB.

- **`trace`**: Optional. Set this to `1` to enable tracing. The log files are saved in the folder path that you specify in `directory_path`. If you do not enter a value or enter any other value, then the log files are not generated.

Example entry to migrate and compress all tables in specified schemas

The following example entry in the `migration_config` table would migrate and then compress all tables and columns in the specified schemas, `TEST1` and `TEST2` when you run the script.

```
insert into migration_config values
    (systimestamp,
    '{"schema_name" : ["TEST1","TEST2"],
    "table_name" : [],
    "segment_name" : [],
    "metadata_schema_name" : "TEMP1",
    "run_type" : "first",
    "directory_path" : "<full_path>",
    "trace" : 1}');
```

Example entry to migrate and compress all schemas, tables, and columns

The following example entry in the `migration_config` table would migrate and then compress ***all*** schemas, tables, and columns when you run the script.

```
insert into migration_config values (
    systimestamp,
    '{"schema_name" : [],
    "table_name" : [],
    "segment_name" : [],
    "metadata_schema_name" : "TEMP1",
    "run_type" : "first",
    "directory_path" : "<full_path>",
    "trace" : 1}');
```

3. Run the script as `SYS` user after creating the table and inserting a row with details of the required configuration.

```
SQL> @securefile_migration_script.sql
```

The following three reports are generated and stored as tables in the temporary table space. You had specified the name of the temporary table space in `metadata_schema_name` in a previous step.

- `sf_migration_table_ddl_report` table provides details about the DDL information for all the tables in specified schema for all users.
- `sf_migration_index_ddl_report` table provides details about index DDL information for all the tables in specified schema for all users.
- `sf_migration_basicfile_report` table lists all the BasicFile LOB segments under all users.

4. Look at the reports and identify if there are any BasicFile LOBs that you do not want to migrate, such as when BasicFile LOB segment is on MSSM tablespace. If you do not want to migrate a BasicFile LOB, change the value of the `Allow migrate` column in the

`sf_migration_basicfile_report` table to **N** for the specific LOB that you do not want to migrate. The default value for all LOBs is **Y**.

5. Create an interim table for the LOBs that you want to migrate. Ensure that the interim table that you create is identical to the original table in all respects except for the property that you intent to change.

   Example Original Table

   For example, let's consider that the existing BasicFile table that you want to migrate has the following properties.

   ```
   CREATE TABLE basic1a
   (
       a CLOB,
       b NUMBER
   );
   ```

   Example Interim Table

   As shown in the following example, the interim table that you create must be identical to the original table, except for the `store as securefile` in the interim table and the name of the interim table must be unique.

   ```
   CREATE TABLE basic1a_int1
   (
       a CLOB,
       b NUMBER
   ) lob(a) store as securefile;
   ```

6. Update the row that you have inserted to change the `run_type` to `second` as shown in the following command.

   ```
   SQL> update migration_config set data = JSON_TRANSFORM(data, SET
   '$.run_type' = 'second');
   ```

7. Run the script as `SYS` user.

   ```
   SQL> @securefile_migration_script.sql
   ```

   The BasicFile LOB data is migrated to SecureFile storage.

   After the migration is completed successfully, the script generates the following reports and stores it as tables in the temporary table space. You had specified the name of the temporary table space in `metadata_schema_name` in a previous step.

   - `sf_migration_lob_statistics_report` table provides details about the LOBs, such as storage, compression ratio, compression recommendation, compression type.
   - `sf_migration_lob_compression_report` table contains recommendations about which LOBs should be compressed in the `COMPRESS_RECOMMENDATION` column as `Y` (yes) or `N` (no). The recommendation is based on the information available in `sf_migration_lob_statistics_report`.

8. Look at `sf_migration_lob_compression_report` and identify if you want to compress the LOBs are per the recommendation. If you do not want to compress the LOBs, skip the next steps.

9. Look at the reports and identify if there are any SecureFile LOBs that you do not want to compress. If you do not want to compress a LOB, change the value of the `compress_recommendation` column in `sf_migration_lob_statistics_report` to **N** for the specific LOB that you do not want to compress. The default value is **Y** for all LOBs.
10. Create an interim table for the LOBs that you want to compress. Ensure that the interim table that you create is identical to the original table in all respects except for the property that you intent to change.

    Example Original Table

    For example, let's consider that the existing BasicFile table that you want to migrate has the following properties.

    ```
    CREATE TABLE basic1a
    (
        a CLOB,
        b NUMBER
    );
    ```

    Example Interim Table

    As shown in the following example, the interim table that you create must be identical to the original table, except for the `LOB(a) STORE AS SECUREFILE seg_basic1a (ENABLE STORAGE IN ROW CACHE LOGGING COMPRESS MEDIUM)` in the interim table and the name of the interim table must be unique.

    ```
    CREATE TABLE comp_basic1a_int1
    (
        a CLOB,
        b NUMBER
    ) LOB(a) STORE AS SECUREFILE seg_basic1a (ENABLE STORAGE IN ROW CACHE
    LOGGING COMPRESS MEDIUM);
    ```

    Where, you can specify `LOW`, `MEDIUM`, and `HIGH` as the options to provide varying degrees of compression.
11. Update the row that you have inserted to change the `run_type` to `third`.

    ```
    SQL> update migration_config set data = JSON_TRANSFORM(data, SET
    '$.run_type' = 'third');
    ```

12. Run the script as `SYS` user.

    ```
    SQL> @securefile_migration_script.sql
    ```

    The `sf_migration_lob_compression_report` report provides details about the updated status of compression.

15.2.2 Migrating LOBs with Online Redefinition

Use Online redefinition to migrate LONG or BasicFile LOB data to SecureFile storage by running several API calls.

Consider using the SecureFiles Migration Utility, which automates this task and saves you the time and effort involved in manually running a series of API calls. See Migrating LOBs with SecureFiles Migration Utility.

While online redefinition for LONG to LOB migration must be performed at the table level, BasicFile to SecureFile migration can be performed at the table or partition level.

Online Redefintion Advantages

- No need not take the table or partition offline
- Can be done in parallel.
  To set up parallel execution of online redefinition, run:

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

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

Example 15-1 Online Redefinition for Migrating Tables from BasicFiles LOB storage to SecureFile LOB storage

```
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/pm

-- This forces the online redefinition to execute in parallel
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;
```

Example 15-2 Online Redefinition for Migrating Tables from the LONG datatype to a SecureFile LOB

The steps for `LONG` to LOB migration are:

- 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.

The following example demonstrates online redefinition for `LONG` to LOB migration.

```
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/pm

-- This forces the online redefinition to execute in parallel
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_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;
```

15.2.3 Migrating LOBs with Data Pump

Oracle Data Pump can either recreate tables as they are in your source database, or recreate LOB columns as SecureFile LOBs.

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

Example:

```
impdp system/manager directory=dpump_dir schemas=lobuser dumpfile=lobuser.dmp
      transform=lob_storage:securefile
```

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

> **See Also:**
>
> TRANSFORM for using TRANSFORM parameter to convert to SecureFile LOBs

You can use the keyword `HIDDEN` to distinguish a default inline LOB size from a user-specified one.

Example:

```
CREATE TABLE <tab> (…) LOB (L1) STORE AS … [ENABLE STORAGE IN ROW [4000|8000]
        HIDDEN];
```

Restrictions on Migrating LOBs with Data Pump

You can't use SecureFile LOBs in non-ASSM tablespace. If the source database contains LOB columns in a tablespace that does not support ASSM, then you'll see an error message when you use Oracle Data Dump to recreate the tables using the securefile clause for LOB columns.

To import non-ASSM tables with LOB columns, run another import for these tables without using `TRANSFORM=LOB_STORAGE:SECUREFILE`.

Example:

```
impdp system/manager directory=dpump_dir schemas=lobuser dumpfile=lobuser.dmp
```

15.3 Other Considerations While Migrating LONG Columns to LOBs

This section describes some more considerations when migrating LONG columns to LOBs.

- Migrating Applications from LONGs to LOBs
  Most APIs that work with `LONG` data types in the PL/SQL, JDBC and OCI environments are enhanced to also work with LOB data types.
- Alternate Methods for LOB Migration
  Online Redefinition is the preferred way for migrating `LONG` data types to LOBs. However, if keeping the application online during the migration is not your primary concern, then you can also use one of the following ways to migrate `LONG` data to LOBs.

15.3.1 Migrating Applications from LONGs to LOBs

Most APIs that work with `LONG` data types in the PL/SQL, JDBC and OCI environments are enhanced to also work with LOB data types.

These APIs are collectively referred to as the data interface for LOBs. Among other things, the data interface provides the following benefits:

- Changes needed are minimal in PL/SQL, JDBC 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:

- Data Interface for LOBs for details on JDBC and OCI APIs included in the data interface.
- SQL Semantics and LOBs for details on SQL syntax supported for LOB data types.
- PL/SQL Semantics for LOBs for details on PL/SQL syntax supported for LOB data types.

Note:

You can use various techniques 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.

However, there are differences between `LONG` and LOB data types that may impact your application migration plans or require you to modify your application.

Identify Application Rewrite Using utldtree.sql

When you migrate your table from `LONG` to LOB column types, certain parts of your PL/SQL application may require rewriting. You can use the utility, `rdbms/admin/utldtree.sql`, to determine which parts.

The `utldtree.sql` 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.

SQL Differences

- Indexes: LONG and LOB data types only support domain and functional indexes.
  - Any domain index on a `LONG` column must be dropped before converting the `LONG` column to LOB column. This index may be manually recreated after the migration.
  - 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 the rebuild.
    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 each selected index, use the command:

```
ALTER INDEX <index> REBUILD
```

- Constraints: The only constraint allowed on `LONG` columns are `NULL` and `NOT NULL`. All constraints of the `LONG` columns are maintained for the new LOB columns. 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.
- Default Values: If you do not specify a default value, then the default value for the `LONG` column becomes the default value of the LOB column.
- Triggers: Most of the existing triggers on your table are still usable. However, you cannot have LOB columns in the `UPDATE OF` list of an `AFTER UPDATE OF` trigger. For example, the following create trigger statement is not valid:

  ```
  CREATE TABLE t(lobcol CLOB);
  CREATE TRIGGER trig AFTER UPDATE OF lobcol ON t ...;
  ```

  `LONG` columns are allowed in such triggers. So, you must drop the `AFTER UPDATE OF` triggers on any `LONG` columns before migrating to LOBs.
- Clustered tables: LOB columns are not allowed in clustered tables, whereas `LONG`s 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.

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);

REM     A zero length string inserts a NULL into the LONG column:
INSERT INTO long_tab values(1, '');

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.

15.3.2 Alternate Methods for LOB Migration

Online Redefinition is the preferred way for migrating LONG data types to LOBs. However, if keeping the application online during the migration is not your primary concern, then you can also use one of the following ways to migrate LONG data to LOBs.

> **See Also:**
>
> Migration Considerations

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` option, 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. This clause can be specified in the `MODIFY` clause.

Other `ALTER TABLE` options are not allowed when converting a `LONG` column to a LOB type column.

Copying a LONG to a LOB Column Using the TO_LOB Operator

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` statement 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;
```

- You cannot use the `TO_LOB` operator 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.

16

Automatic SecureFiles Shrink

The Oracle Database SecureFiles Shrink feature provides manual and automatic methods to free the unused space in SecureFiles LOB segments and release the space back to the containing tablespace. This chapter explains how to use the automatic method called Automatic SecureFiles Shrink with Oracle Database.

16.1 About Manual SecureFiles Shrink

Use the `ALTER TABLE ... SHRINK SPACE` statement to manually shrink a SecureFiles LOB segment. You can also use tools, such as Segment Advisor or a PL/SQL procedure, such as `DBMS_SPACE.SPACE_USAGE` to return information about SecureFiles space usage before deciding on the SecureFiles LOB segments to shrink.

The following points are important when opting for the manual shrink method:

- The manual SecureFiles shrink operation is an online DDL with part of the operations being offline, where offline means concurrent DML are blocked until the shrink activity on the critical section ends. The concurrent DML statements do not fail with ORA-54, but are blocked.
- The manual SecureFiles shrink operation disregards any flavor of undo retention and treats it as if the retention is equal to none. The user cannot expect the LOB retention feature to provide the usual guarantees after invoking the shrink operation. The user may see the `ORA-1555 snapshot too old` error message in queries. Run the shrink operation with caution if this is a concern.

Use *shrink_clause* on SecureFiles LOB segments from release 21c and onward. There are two ways to invoke *shrink_clause*:

1. The following command targets the *specified* LOB column and all its partitions.

   ```
   ALTER TABLE <table_name> MODIFY LOB <lob_column> SHRINK SPACE
   ```

2. The following command cascades the shrink operation for *all* LOB columns and its partitions in the specified table.

   ```
   ALTER TABLE <table_name> SHRINK SPACE CASCADE
   ```

> **See Also:**
>
> - SPACE_USAGE Procedures
> - ALTER TABLE

16.2 About Automatic SecureFiles Shrink

SecureFiles LOB segments can potentially become the largest consumer of space in a database. It may not be feasible for administrators to spend their time checking each SecureFiles LOB segment to shrink. Automatic SecureFiles Shrink uses a framework that enables automatic selection of SecureFiles LOB segments to shrink based on a set criteria and it runs Automatic SecureFiles Shrink in the background.

Automatic SecureFiles Shrink is designed to minimize the functional and performance impact on concurrent workloads. While shrink runs automatically on a SecureFiles LOB segment, all DML and DDL statements that involve the segment will succeed. Space is gradually freed in the SecureFiles LOB segment and the performance impact is minimal.

Automatic SecureFiles Shrink does not have any effect on the BasicFiles LOBs and in-lined SecureFiles LOBs.

> **See Also:**
>
> Reclaiming Unused Space

16.3 Automatic SecureFiles Shrink Features

Automatic SecureFiles Shrink has the following features:

Managed by Database

The database provides background execution and resource management infrastructure for Automatic SecureFiles Shrink. The database also executes the shrink task automatically, at a system-determined interval, within a bounded runtime.

> **See Also:**
>
> Managing Automated Database Maintenance Tasks

Integrates with Pre-Allocation

Automatic SecureFiles Shrink integrates with pre-allocation seamlessly without affecting performance. Automatic SecureFiles Shrink avoids the SecureFiles LOB segments that are recently pre-allocated. Segment pre-allocation is performed in the background for segments that have high demand for free space.

Works with DDL and DML

Automatic SecureFiles Shrink targets only idle segments and skips active SecureFiles LOB segments. User driven DDL and DML statements do not fail and face minimal performance impact when Automatic SecureFiles Shrink works in the background. If Automatic SecureFiles Shrink for a SecureFiles LOB segment comes across locked rows, it skips the locked rows because locked rows are indicative of DML activity or waiting on locked rows may cause deadlocks with user transactions. Automatic SecureFiles Shrink always acquires row locks in the `NOWAIT` mode to avoid deadlock with user transactions.

Targets Idle LOB Segments

To avoid unnecessary block accesses, Automatic SecureFiles Shrink filters SecureFiles LOB segments based on information available in System Global Area (SGA). Automatic SecureFiles Shrink selects only idle SecureFiles LOB segments and skips active LOB segments to minimize performance impact on active SecureFiles LOB segments.

Covers All SecureFiles LOB Segments

The Automatic SecureFiles Shrink task covers all SecureFiles LOB segments in a PDB over several intervals and this includes user-created LOB segments and the SecureFiles LOB segments that are created using features, such as JSON and DBFS.

Performs Shrinks in Iterations

Automatic SecureFiles Shrink does not free all the free space in the selected SecureFiles LOB segments at once. Instead, the Automatic SecureFiles shrink task frees a modest amount of space at every shrink call (iteration). The trickle threshold limit defines the amount of space to shrink in every iteration. Over time, the amount of free space in idle SecureFiles LOB segments approaches the minimum that is specified for pre-allocation.

Executes in the Background

All steps involved in Automatic SecureFiles Shrink, including the selection of SecureFiles LOB segments to shrink, run in the background. After Automatic SecureFiles Shrink is enabled, it comes into effect with the start of a database instance. No directive regarding how Automatic SecureFiles Shrink should operate is required.

Honors Undo Retention

Automatic SecureFiles Shrink respects the undo retention period. It does not allow a query to fail within the undo retention period because an affected SecureFiles LOB segment has been freed, relocated, or reused as a part of an Automatic SecureFiles Shrink task. Unexpired blocks are freed only after the undo retention time.

16.4 SecureFiles Shrink and Undo Retention

This section discusses how the Auto Shrink feature treats different flavors of undo retention.

As part of the Automatic SecureFiles Shrink feature, used blocks in an extent are relocated, and the extents are freed to the tablespace. After an extent is freed to the tablespace, it may be reused and overwritten. Freeing and reusing an extent during shrink can change the expected behavior of undo retention.

Automatic SecureFiles Shrink moves data in a SecureFiles LOB segment to create free space. The feature honors the LOB retention setting by not reusing space until the minimum retention requirement has been met. In practice, concurrent queries may run into `ORA-1555 Snapshots too old` more often compared to the same queries on an idle segment. Automatic SecureFiles Shrink for Autonomous Database skips SecureFiles LOB segments configured with the `RETENTION` parameter set to `MAX`.

16.5 Enable Automatic SecureFiles Shrink

Automatic SecureFiles Shrink affects out-of-line SecureFile LOBs. Automatic SecureFiles Shrink does not have any effect on the BasicFiles LOBs and inline SecureFiles LOBs.

Automatic SecureFiles Shrink is not enabled by default.

- In on-premises environments, run the following command to enable the Automatic SecureFiles Shrink feature.

```
exec DBMS_SPACE.SECUREFILE_SHRINK_ENABLE();
```

In Autonomous Cloud environments, contact your system administrator to enable Automatic SecureFiles Shrink.

16.6 Disable Automatic SecureFiles Shrink

By default, the Automatic SecureFiles Shrink feature is disabled.

- In on-premises environments, run the following command to disable the Automatic SecureFiles Shrink feature.

```
exec DBMS_SPACE.SECUREFILE_SHRINK_DISABLE();
```

In Autonomous Cloud environments, contact your system administrator to disable Automatic SecureFiles Shrink.

16.7 Targets and Limits

Automatic SecureFiles Shrink follows a set of targets and limits to operate optimally.

- Automatic SecureFiles Shrink Targets
  An Automatic SecureFiles Shrink target determines the amount of space that needs to be freed in one automatic shrink run.

- LOB Segment Idle Time Limit
  The LOB segment idle time limit is the minimum time limit a LOB segment can be idle before a shrink can start.

16.7.1 Automatic SecureFiles Shrink Targets

An Automatic SecureFiles Shrink target determines the amount of space that needs to be freed in one automatic shrink run.

In manual shrink, the shrink command continues to free space until there is no free space left. Automatic SecureFiles Shrink sets a shrink target for every shrink run. The target is based on:

- Pre-allocation threshold that determines the minimum limit below which an Automatic SecureFiles Shrink task cannot free space for a segment.
- Trickle threshold that specifies the maximum limit that Automatic SecureFiles Shrink can free space for a LOB segment in one iteration.
- Pre-Allocation Threshold
  A pre-allocation threshold value is used to compute the targeted minimum free space for a LOB segment.
- Automatic SecureFiles Shrink Trickle Threshold
  The trickle threshold controls the maximum amount of space per segment to be freed by one automatic shrink iteration.

16.7.1.1 Pre-Allocation Threshold

A pre-allocation threshold value is used to compute the targeted minimum free space for a LOB segment.

Pre-allocation threshold values are represented in percentages of free space out of the LOB segment size. For example, for a 2 TB LOB segment, the minimum amount of free space is 20 GB (2 TB * 1%). If the actual free space is less than the pre-allocation threshold, pre-allocation triggers in and adds extents to the segment. By default, unexpired space is not considered as free space.

> **Note:**
>
> Automatic SecureFiles Shrink does not free space that is less than the pre-allocation minimum threshold for a LOB segment.

Pre-Allocation Threshold shows the default pre-allocation threshold values for different segment sizes.

> **Note:**
>
> You can adjust the thresholds using optimizations in the future Oracle Database releases.

Table 16-1 Pre-Allocation Threshold

Segment Size	Actual Threshold	Minimum Free Space
<= 1 GB	10%	100 MB
<= 10 GB	5%	500 MB
<= 100 GB	2%	2 GB
> 100 GB	1%	1 GB

16.7.1.2 Automatic SecureFiles Shrink Trickle Threshold

The trickle threshold controls the maximum amount of space per segment to be freed by one automatic shrink iteration.

The shrink run on a LOB segment frees up space incrementally. Therefore, the trickle threshold is an incremental shrink amount, with the shrink task freeing limited space in iterations without concurrent DML and DDL statements facing any noticeable performance impact due to higher latency.

> **Note:**
>
> Automatic SecureFiles Shrink does not free space that is less than the pre-allocation minimum threshold for a LOB segment.

16.7.2 LOB Segment Idle Time Limit

The LOB segment idle time limit is the minimum time limit a LOB segment can be idle before a shrink can start.

The LOB segment idle time limit specifies the minimum amount of time that a LOB segment must be idle to qualify for Automatic SecureFiles Shrink. An idle LOB segment is defined as one that has neither user-driven DML statements nor pre-allocation activity in the past N hours. LOB Segment Idle Time Limit defines the N hours. The default value for LOB segment idle time limit is 1440 minutes, which is 24 hours.

16.8 Selection Criteria for SecureFiles LOB Segments to Shrink

Here are the criteria that automatic SecureFiles shrink uses for selecting SecureFiles LOB segments to shrink.

The Automatic SecureFiles Shrink task excludes the following SecureFiles LOB segments when choosing the SecureFiles LOB segments to shrink:

- The SecureFiles LOB segment is not an idle segment as per LOB Segment Idle Time Limit.
- The SecureFiles LOB segment does not contain extra free space greater than the pre-allocation threshold.
- The SecureFiles LOB segment has `RETENTION MAX`, which means the segment keeps as many unexpired blocks as possible.

- The SecureFiles LOB segment is currently being shrunk.
- The SecureFiles LOB segment does not have enough expired free space that is no longer needed for lob retention requirement. Space that is still needed for lob retention is treated as used space.
- The SecureFiles LOB segment has failed a previous shrink task. Previous shrink calls have failed to free space from the SecureFiles LOB segment. Automatic SecureFiles Shrink identifies the LOB segments that it failed to shrink previously and avoids such segments.

16.9 Automatic SecureFiles Shrink Task

Automatic SecureFiles Shrink performs a series of steps to complete the shrink of SecureFiles LOB segments.

When enabled, a shrink task is performed as one instance of the background action performed on AutoTask. The task runs every 30 minutes and performs the following steps:

1. A shrink task has 60 minutes at the start of the task. As the task progresses, it tracks both the time spent so far and the average duration of a shrink call. The latter is used to predict how long the next shrink call would take. If the time left is not enough for another call, the shrink task exits. If a shrink call goes over the 60-minute mark, it is terminated.
2. Automatic SecureFiles Shrink fetches the next batch of SecureFiles LOB segments from internal catalog tables (which is ordered by the object identifier). The last object identifier in the previous shrink task is used as the starting point for the next shrink task.
3. Automatic SecureFiles Shrink applies the criteria filters from the Selection Criteria for SecureFiles LOB segment to remove the segments that do not qualify for the shrink task.

> **See Also:**
>
> Selection Criteria for SecureFiles LOB Segments to Shrink

4. Once the qualified segment is found, the shrink task can start work on the segment.
5. Before starting the shrink, the shrink target is computed. The shrink target is based on the Pre-Allocation Threshold and the Automatic SecureFiles Shrink Trickle Threshold.
6. Automatic SecureFiles Shrink runs the shrink command. The `ALTER TABLE ... SHRINK SPACE` command is executed using the OCI interface.
7. Automatic SecureFiles Shrink updates the timestamp for the next shrink. This timestamp indicates the earliest time when Automatic SecureFiles Shrink can select this SecureFiles LOB segment again. If space was freed successfully, the timestamp uses the current time. Otherwise, the shrink is assigned a time in the future. If shrink is not successful, a penalty time is assessed to avoid Automatic SecureFiles Shrink from selecting the same segment in future shrink tasks.

See Also:

- LOB Segment Idle Time Limit
- Pre-Allocation Threshold
- Automatic SecureFiles Shrink Trickle Threshold

16.10 Checking Progress

The most straightforward way of checking the result of Automatic SecureFiles Shrink is to compute the combined size of the SecureFiles LOB segments in the database. Alternatively, you can observe space saving at a micro level using the `V$SECUREFILE_SHRINK` table, which reports one row for each SecureFiles LOB segment that is shrunk.

See Also:

V$SECUREFILE_SHRINK to see the results of the previous shrink calls.

17

Introducing the Database File System

This chapter describes the Database File System in details.

- 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.
- What Is Database File System (DBFS)?
  Database File System (DBFS) creates a standard file system interface using a server and clients.

17.1 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.

17.2 What Is Database File System (DBFS)?

Database File System (DBFS) creates a standard file system interface using a server and clients.

- About DBFS
  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.
- DBFS Server
  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. In DBFS, the file server is the Oracle Database.
- DBFS Client Access Methods
  Learn about various methods to access DBFS in this section.

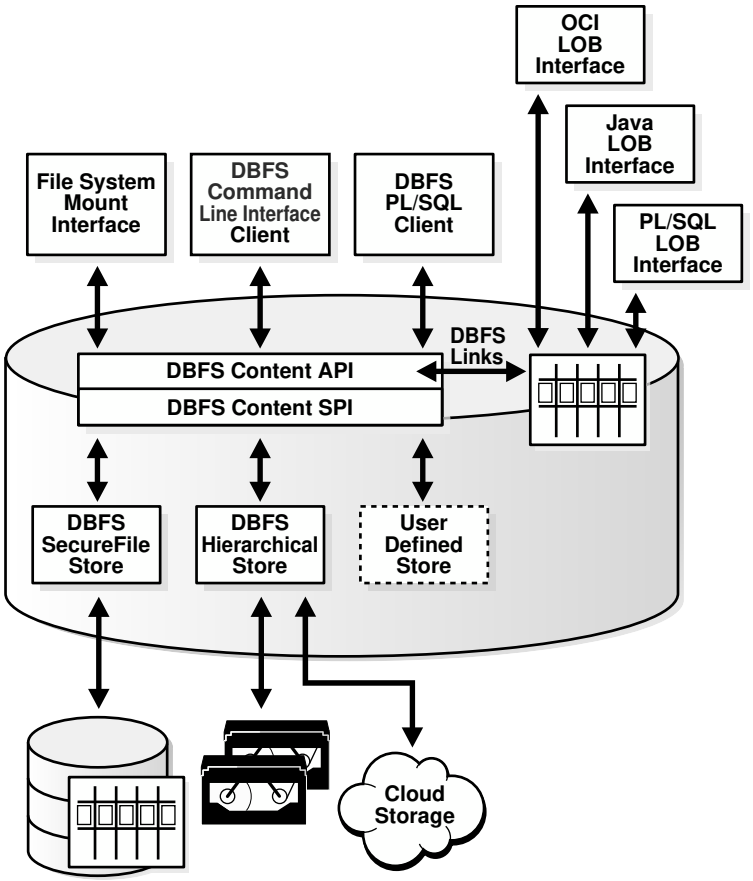
17.2.1 About DBFS

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 `deletefile` function then calls the DBFS Content SPI to perform that function.

Figure 17-1 Database File System (DBFS)



17.2.2 DBFS Server

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. In DBFS, the file server is the Oracle Database.

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

Following are the different types of stores supported by the DBFS Content SPI:

- 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 file system capabilities.
- 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.
- 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 Also:

- Creating Your Own DBFS Store
- DBFS Content API
- DBFS Hierarchical Store

17.2.3 DBFS Client Access Methods

Learn about various methods to access DBFS in this section.

The Database File System offers several access methods.

- PL/SQL Client Interface

  Database applications can access files in the DBFS store directly, through the DBFS Content API 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 `cp`. The command interface creates a direct connection to the database without requiring an OS mount of 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.

- 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 Also:

- Using DBFS
- DBFS Mounting Interface (Linux and Solaris Only)
- Database File System Links for information about using DBFS Links
- PL/SQL Packages for LOBs and DBFS for lists of useful `DBMS_LOB` constants and methods

18
Using DBFS

The DBFS File System implementation includes creating and accessing the file system and managing it.

- Enabling Advanced SecureFiles LOB Features for DBFS
  Using the `@dbfs_create_filesystem.sql` command, you can create a partitioned or non-partitioned file system with the compression and deduplicate options. If you want to specify additional options while creating the file system, use the `DBMS_DBFS_SFS.CREATEFILESYSTEM` procedure.
- Installing DBFS
  DBFS is a part of the Oracle Database installation.
- Creating a DBFS File System
  You can create a partitioned or non-partitioned DBFS File system.
- Accessing DBFS File System
  This section describes the various interfaces through which you can access the DBFS File System.
- Maintaining DBFS
  DBFS administration includes tools that perform diagnostics, manage failover, perform backup, and so on.
- Shrinking and Reorganizing DBFS Filesystems
  DBFS uses Online File system Reorganization to shrink itself, enabling the release of allocated space back to the containing tablespace.
- Dropping a File System
  You can drop a file system by running `DBFS_DROP_FILESYSTEM.SQL`.

18.1 Enabling Advanced SecureFiles LOB Features for DBFS

Using the `@dbfs_create_filesystem.sql` command, you can create a partitioned or non-partitioned file system with the compression and deduplicate options. If you want to specify additional options while creating the file system, use the `DBMS_DBFS_SFS.CREATEFILESYSTEM` procedure.

For information about all the additional options that you can use with the `DBMS_DBFS_SFS.CREATEFILESYSTEM` procedure, see CREATEFILESYSTEM Procedure in *PL/SQL Packages and Types Reference*.

Use the `@dbfs_create_filesystem.sql` command to quickly create, register, and mount a file system. When you use the `DBMS_DBFS_SFS.CREATEFILESYSTEM` procedure to enable additional options while creating a file system, you must additionally run commands to register and mount the file system that you create.

Let's use the `DBMS_DBFS_SFS.CREATEFILESYSTEM` procedure to create a file system with the encryption option.

Before you begin, ensure that you have created a wallet with the encryption key. See Administer Key Management in *SQL Language Reference*.

To create a file system with the encryption option:

1. Run the following command.
   Syntax

   ```
   exec
   dbms_dbfs_sfs.createFilesystem('store_name',tbl_tbs=>'tablespace_name',do_e
   ncrypt=> true | false,encryption=> encryption_type, do_dedup=> true |
   false,do_compress=>true | false);
   ```

   For reference information about the command options, see CREATEFILESYSTEM Procedure in *PL/SQL Packages and Types Reference*.

   Example

   For example, to create a file system in `Test3` store in the `test_fs1` tablespace with the default encryption, compression, and deduplicate options:

   ```
   exec dbms_dbfs_sfs.createFilesystem('test_fs1', tbl_tbs=>'Test3',
   do_encrypt=>true, encryption=>dbms_dbfs_sfs.ENCRYPTION_DEFAULT,
   do_dedup=>true, do_compress=>true);
   ```

   The file system is created with the option you have specified.

2. Run the following command to register the file system that you have created.

   Syntax

   ```
   dbms_dbfs_content.registerStore(store_name => 'filesystem_name',
   provider_name => 'posix',provider_package => 'dbms_dbfs_sfs') ;
   ```

   Example

   For example, run the following command to register the `test_fs1` file system.

   ```
   dbms_dbfs_content.registerStore(store_name => 'test_fs1', provider_name =>
   'posix', provider_package => 'dbms_dbfs_sfs') ;
   ```

3. Run the following command to mount the file system that you have created.

   Syntax

   ```
   dbms_dbfs_content.mountStore(store_name  => 'filesystem_name', store_mount
   => 'filesystem_name');
   ```

   Example

   For example, run the following command to mount the `test_fs1` file system.

   ```
   dbms_dbfs_content.mountStore(store_name  => 'test_fs1', store_mount =>
   'test_fs1');
   ```

18.7 Dropping a File System

You can drop a file system by running `DBFS_DROP_FILESYSTEM.SQL`.

> **Caution:**
>
> When you drop a file system, it deletes all the files and associated metadata. You won't be able to access the files.

1. Log in to the database instance:

   ```
   $ sqlplus dbfs_user/@db_server
   ```

2. Enter the following command:

   ```
   @$ORACLE_HOME/rdbms/admin/dbfs_drop_filesystem.sql file_system_name
   ```

When you drop a file system, it deletes all the files and associated metadata. You won't be able to access the files. If you want to access the file system after dropping a DBFS, you can restore the file system from a database backup or file system backup.

Depending on the backup policy in your organization, you may have a database backup or file system backup. To restore from a database backup, you'll have to restore the entire database and then use the restored file system. To restore the file system from a file system backup, create a new DBFS and restore the file system from the file system backup.

18.2 Installing DBFS

DBFS is a part of the Oracle Database installation.

`$ORACLE_HOME/rdbms/admin` contains these DBFS utility packages:

- 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

18.3 Creating a DBFS File System

You can create a partitioned or non-partitioned DBFS File system.

For both partitioned and non-partitioned DBFS, you can specify one or more of the following storage properties to specify how your files are stored in DBFS: compression and deduplication.

For example, you can configure DBFS as a compressed file system with partitioning. At the time of creating a DBFS file system, you must specify the set of features that you want to enable for the file system.

After creating a DBFS, you can track the usage of the DBFS. If you want to change the storage properties of the DBFS, you can reorganize the DBFS. You can update the metadata of the DBFS by changing the values for parameters, such as `deduplicate`, `compress`, and `partition`. For example, you may have created a DBFS to store all the files in the compressed format. If you want to change this property, you can reorganize the DBFS.

- Privileges Required to Create a DBFS File System
  Database users must certain privileges to create a file system.
- 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.
- Creating a Partitioned File System
  Files in DBFS are hash partitioned. Partitioning creates multiple physical segments in the database, and files are distributed randomly in these partitions.

18.3.1 Privileges Required to Create a DBFS File System

Database users must certain privileges to create a file system.

Following is the minimum set of privileges required for a database user to create a file system:

- `GRANT CONNECT`
- `CREATE SESSION`
- `RESOURCE, CREATE TABLE`
- `CREATE PROCEDURE`
- `DBFS_ROLE`

18.3.2 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.

Before you begin, ensure that you create the file system in an ASSM tablespace to support a SecureFile store.

To create a non-partitioned file system:

1. Log in to the database instance as a user with DBFS administrator privileges.

   ```
   $ sqlplus dbfs_user/@db_server
   ```

2. Enter the following command to create the file system.

   Syntax

   ```
   @$ORACLE_HOME/rdbms/admin/dbfs_create_filesystem.sql tablespace_name
        file_system_name
     [compress-high | compress-medium | compress-low | nocompress]
     [deduplicate | nodeduplicate]
     non-partition
   ```

   Example

   For example, to create a file system called `staging_area` in an existing ASSM tablespace `dbfs_tbspc`:

```
$ sqlplus dbfs_user/db_server
   @$ORACLE_HOME/rdbms/admin/dbfs_create_filesystem.sql
   dbfs_tbspc staging_area nocompress nodeduplicate non-partition
```

18.3.3 Creating a Partitioned File System

Files in DBFS are hash partitioned. 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.SQL` while logged in as a user with DBFS administrator privileges.

The tablespace in which you create the file system should be an ASSM tablespace to support Securefile store. Before you begin, ensure that you create the file system in an ASSM tablespace to support SecureFile store.

1. Log in to the database instance:

   ```
   $ sqlplus dbfs_user/@db_server
   ```

2. Enter one of the following commands to create the file system based on your requirement.

   Syntax

   ```
   @$ORACLE_HOME/rdbms/admin/dbfs_create_filesystem_advanced.sql tablespace_name
          file_system_name [compress-high | compress-medium | compress-low |
   nocompress]
     [deduplicate | nodeduplicate]
     [partition | partition-by-itemname | partition-by-guid | partition-by-path]
   ```

   Examples

   - For example, to create a partitioned file system called `staging_area` in an existing ASSM tablespace `dbfs_tbspc`:

     ```
     $ sqlplus dbfs_user/@db_server
            @$ORACLE_HOME/rdbms/admin/dbfs_create_filesystem_advanced.sql dbfs_tbspc
            staging_area nocompress nodeduplicate partition
     ```

   - For example, to create a partitioned file system called `staging_area` in an existing ASSM tablespace `dbfs_tbspc` with the storage properties compress and deduplicate.

     ```
     $ sqlplus dbfs_user/@db_server
            @$ORACLE_HOME/rdbms/admin/dbfs_create_filesystem_advanced.sql dbfs_tbspc
            staging_area compress-medium deduplicate partition
     ```

18.4 Accessing DBFS File System

This section describes the various interfaces through which you can access the DBFS File System.

- DBFS Client Prerequisites
  The DBFS File System client side application, which is named `dbfs_client`, runs on each system that will access to DBFS.
- Multiple Mount Points on DBFS Client
  Starting from Oracle Database Release 21c, a single Database File System (DBFS) client instance can mount multiple DBFS, owned by different database users across different database instances.

- Manager File System
  The Manager File System is the interface between the OS user and the DBFS Client. The OS user can communicate with the Client through limited File System commands.
- DBFS Client Command-Line Interface Operations
  The DBFS client command-line interface allows you to directly access files stored in DBFS.
- DBFS Mounting Interface (Linux and Solaris Only)
  You can mount DBFS using the `dbfs_client` in Linux and Solaris only.
- File System Security Model
  The database manages security in DBFS. It does not use the operating system security model.
- HTTP, WebDAV, and FTP Access to DBFS
  Components that enable HTTP, WebDAV, and FTP access to DBFS over the Internet use various XML DB server protocols.

18.4.1 DBFS Client Prerequisites

The DBFS File System client side application, which is named `dbfs_client`, runs on each system that will access to DBFS.

The prerequisites for the DBFS File System Client, `dbfs_client`, are:

- 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. The `dbfs_client` host must have the `FUSE` Linux package or the Solaris `libfuse` package installed.

> **See Also:**
>
> DBFS Mounting Interface (Linux and Solaris Only) for further details.

The DBFS client command-line interface allows you to perform many pre-defined commands, such as copy files in and out of the DBFS filesystem from any host on the network.

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.

To run DBFS commands, specify `--command` to the DBFS client.

18.4.2 Multiple Mount Points on DBFS Client

Starting from Oracle Database Release 21c, a single Database File System (DBFS) client instance can mount multiple DBFS, owned by different database users across different database instances.

To enable access to multiple database users, the DBFS client has to manage multiple mount points. Each mount point enables one database user to access DBFS.

When the DBFS client provides access to a single database user through a single mount point, it is termed as Single User Mount Version (SUMV) mode and when the DBFS client provides access to multiple database users through multiple mount points, it is termed as Multi User Mount Version (MUMV) mode.

You can start a DBFS client in either of these modes. However, once you start the client in any mode, you cannot switch to the other mode without restarting the client. If a DBFS client is started in the MUMV mode, then the client creates a pseudo file system called Manager File System (MFS), which acts as an interface between the OS user and the DBFS client.

You can start the MUMV mode in two variants, one that can mount DBFS across multiple container databases or one that can mount only DBFS belonging to different pluggable databases of a single container database. The MUMV variant that mounts DBFS from multiple databases is termed as the Cross-Database variant and the one that mounts DBFS for multiple PDBs of a single container database as the CDB variant. Both the variants are started by specifying only the MFS mount points during start up. The DBFS mounts are added by setting extended attributes on the MFS mount point.

- MUMV for CDB Variant
  The CDB variant of the Multi User Mount Version (MUMV) mode manages the mount points of Database File System (DBFS) that belong to different pluggable databases (PDBs) of a single container database (CDB).
- MUMV for Cross-Database Variant
  The Cross-Database variant of the Multi User Mount Version (MUMV) mode manages mount points for Database File System (DBFS) in multiple databases.

18.4.2.1 MUMV for CDB Variant

The CDB variant of the Multi User Mount Version (MUMV) mode manages the mount points of Database File System (DBFS) that belong to different pluggable databases (PDBs) of a single container database (CDB).

Remember the following points while working with the CDB variant of the MUMV mode:

- The DBFS client, managing multiple DBFS mount points of a single container, should be provided with the credentials to connect to a common user of the CDB at `CDB$ROOT`. The DBFS to be mounted, should be created in or exported to this common user in the PDBs.
- A mount point must be specified for the DBFS in every PDB in the given container. The DBFS client connects to the `CDB$ROOT`, using common user credentials, and then switches to the required PDB to access the DBFS through the specified mount point.

18.4.2.2 MUMV for Cross-Database Variant

The Cross-Database variant of the Multi User Mount Version (MUMV) mode manages mount points for Database File System (DBFS) in multiple databases.

Remember the following points while working with the Cross-Database variant of the MUMV mode:

- The DBFS client must have the credentials of a database user on each database to manage the respective DBFS mount points.
- A DBFS mount point must be specified for each database user and a DBFS must be created in their respective schemas.

18.4.3 Manager File System

The Manager File System is the interface between the OS user and the DBFS Client. The OS user can communicate with the Client through limited File System commands.

The Manager File System (MFS) is enabled only in the Multi User Mount Version (MUMV) mode. It treats the various mount points managed by the DBFS Client as files. The MFS provides an easy interface for the OS users to manage multiple mount points.

The MFS does not create or store files on the disk. Only a limited file system operations are allowed on the MFS mount point.

No OS user can create files or directories under the MFS.

- Adding a DBFS Mount Point
  You can add DBFS mount points by specifying extended attributes on the MFS mount points.
- Listing DBFS Mount Points
  Each DBFS mount point has a corresponding file under the MFS directory, `/mnt/mfs`. So, you can use the standard Linux command `ls` to list the DBFS mount points.
- Unmounting a DBFS Mount Point
  The procedure to unmount a DBFS mount point is the same for both the CDB variant and the Cross-Database variant of the MUMV mode.
- Configuration Parameters of DBFS Client
  All configuration parameters of DBFS client in Single User Mount Version (SUMV) mode can also be used with the DBFS client in Multi User Mount Version (MUMV) mode at the time of start up.
- Diagnosability of DBFS Client
  Starting from Oracle Database Release 21c, the DBFS Client writes an alert file in the client trace directory of the configured Automatic Diagnostic Repository (ADR) base.

18.4.3.1 Adding a DBFS Mount Point

You can add DBFS mount points by specifying extended attributes on the MFS mount points.

> **Note:**
>
> The MUMV mode works only in wallet mode, even if you do not specify the `-o wallet` option. As there is no way to provide passwords in the DBFS commend-line interface, you must add all the credentials required by the DBFS client in the wallet.

While using a CDB variant of the MUMV mode, add the mount points for each of the PDB in the CDB by setting the extended attribute on the `/mnt/mfs` directory, where `/mnt/mfs` is the MFS mount point.

Defining the Mount Points in a CDB Variant

Perform the following steps to define the mount points in a CDB variant of the MUMV mode:

1. Start the DBFS client to connect to the common user at the `CDB$ROOT`, specifying the MFS mount point and the wallet alias at the start up:

   ```
   % dbfs_client -o mfs_mount=/mnt/mfs -o cdb=inst_cdb
   ```

   Where, `/mnt/mfs` is the MFS mount point. It can be any empty directory of your choice. `inst_cdb` is the alias insert into the wallet that can connect to the common user in `CDB$ROOT`.

2. Add a DBFS mount point by setting an extended attribute in the following way:

   ```
   % setfattr -n mount_pdb -v " pdb1 /mnt/mp1" /mnt/mfs/
   ```

   Where:

   - `mount_pdb` is the name of the extended attribute to mount a DBFS mount point in CDB variant
   - `pdb1` is the name of the PDB in the particular CDB, which is pointed to by `inst_cdb`
   - `/mnt/mp1` is the mount directory, where the DBFS present in the common user in the PDB `pdb1`, should be mounted
   - `/mnt/mfs` is the MFS mount directory that was used during the start up of the `dbfs_client` command

3. (Optional) Add more DBFS mount points by setting the same extended attribute with different arguments in the following way:

   ```
   % setfattr -n mount_pdb  -v " pdb2 /mnt/mp2" /mnt/mfs
   % setfattr -n mount_pdb  -v " pdb3 /mnt/mp3" /mnt/mfs
   ```

   Where, `pdb2` and `pdb3` are the actual names of the PDBs in the container.

Defining the Mount Points in a Cross-Database Variant

Perform the following steps to define the mount points in a Cross-Database variant of the MUMV mode:

1. Start the DBFS client in MUMV Cross-Database variant by specifying the MFS mount point at the start up in the following way:

   ```
   % dbfs_client -o mfs_mount=/mnt/mfs
   ```

   Where, `/mnt/mfs` is the MFS mount point. It can be any empty directory of your choice

2. Add a DBFS mount point by setting an extended attribute in the following way:

   ```
   % setfattr -n mount -v " inst1 /mnt/mp1" /mnt/mfs/
   ```

   Where,

   - `mount` is the name of the extended attribute to mount a DBFS mount in Cross-Database variant
   - `inst1` is the wallet alias that connects to the DB user, for which DBFS needs to be mounted

- `/mnt/mp1` is the mount directory, where the DBFS should be mounted
- `/mnt/mfs` is the MFS mount directory that was used during the start up of the `dbfs_client` command

3. (Optional) Add more DBFS mount points by setting the same extended attribute with different arguments in the following way:

```
% setfattr -n mount -v "inst2 /mnt/mp2" /mnt/mfs/
% setfattr -n mount -v "inst3 /mnt/mp3" /mnt/mfs/
```

Where, `inst2` and `inst3` are aliases that must exist in the wallet. The DBFS client must have the credentials to connect to the user in the database and they should have at least one DBFS created in their schema.

18.4.3.2 Listing DBFS Mount Points

Each DBFS mount point has a corresponding file under the MFS directory, `/mnt/mfs`. So, you can use the standard Linux command `ls` to list the DBFS mount points.

The following code snippet shows how to list all the DBFS mount points:

```
% ls  -l  /mnt/mfs
```

The content of each file under the `/mnt/mfs` directory, provides details about the parameters used in the corresponding mount point.

The MFS is a *read-only* file system. You cannot create any file or directory within it using any application, apart from the DBFS Client. Anything that appears as a file or a directory under the MFS, is defined by the DBFS Client.

18.4.3.3 Unmounting a DBFS Mount Point

The procedure to unmount a DBFS mount point is the same for both the CDB variant and the Cross-Database variant of the MUMV mode.

You must unmount a mount point using the FUSE executable file, `fusermount`. The following code snippet shows how to drop a DBFS mount point:

```
% fusermount -u /mnt/mp1
```

18.4.3.4 Configuration Parameters of DBFS Client

All configuration parameters of DBFS client in Single User Mount Version (SUMV) mode can also be used with the DBFS client in Multi User Mount Version (MUMV) mode at the time of start up.

All the command-line options passed to the DBFS client in the MUMV mode are inherited by all the DBFS mount points that may be added later. For example, for the following `dbfs_client` command, the DBFS mounted at the `/mnt/mp1` mount point automatically inherits the `spool_max` value as 32 and the `max_threads value` as 16:

```
% dbfs_client -o mfs_mount=/mnt/mfs -o spool_max=32 -o max_threads=16
% setfattr -n mount -v "inst1 /mnt/mp1" /mnt/mfs
```

If you want to configure a mount point differently than the DBFS client, then use the `setfattr` command in the following way:

```
% sefattr -n mount -v "inst2 /mnt/mp2 -o trace_file=/tmp/
clnt.trc,trace_level=1" /mnt/mfs
```

The preceding command enables only the trace for the DBFS client at the `/mnt/mp2` mount point, but does not inherit the `spool_max` and `max_threads` arguments that were specified at the time of start up. The values specified with the `setfattr` command overwrite the values specified during start up.

18.4.3.5 Diagnosability of DBFS Client

Starting from Oracle Database Release 21c, the DBFS Client writes an alert file in the client trace directory of the configured Automatic Diagnostic Repository (ADR) base.

The alert files are generated for every instance of the DBFS client and can be found under the `clients/DBFS/DBFS/trace` directory of the ADR base. The file name is of the format `dbfs_alert_<client_pid>.trc`.

The alert file is different from the trace file. It is always enabled and only important activities of the DBFS clients are written to the alert file.

> **Note:**
>
> The `diagnostic_dest` initialization parameter sets the location of the automatic diagnostic repository. When you use `dbfs_client` or Oracle File Server (OFS) as the file system server, ensure that this parameter does not point to a directory inside `dbfs_client` or OFS as this can produce a dependency cycle and cause the system to hang.

18.4.4 DBFS Client Command-Line Interface Operations

The DBFS client command-line interface allows you to directly access files stored in DBFS.

- About the DBFS Client Command-Line Interface
  The DBFS client command-line interface allows you to perform many pre-defined commands, such as copy files in and out of the DBFS filesystem from any host on the network.
- Listing a Directory
  You can use the `ls` command to list the contents of a directory.
- Copying Files and Directories
  You can use the `cp` command to copy files or directories from the source location to the destination location.
- Removing Files and Directories
  You can use the command `rm` to delete a file or directory.

18.4.4.1 About the DBFS Client Command-Line Interface

The DBFS client command-line interface allows you to perform many pre-defined commands, such as copy files in and out of the DBFS filesystem from any host on the network.

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.

To run DBFS commands, specify `--command` to the DBFS client.

All DBFS content store paths , in command-line interface ,must be preceded by `dbfs:` .This is an example: `dbfs:/staging_area/file1`. All database path names specified must be absolute paths.

```
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.

18.4.4.2 Listing a Directory

You can use the `ls` command to list the contents of a directory.

Use this syntax:

```
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
```

or

```
$ dbfs_client ETLUser@DBConnectString --command  ls -l -a -R dbfs:/staging_area/dir1
```

18.4.4.3 Copying Files and Directories

You can use the `cp` command to copy files or directories from the source location to the destination location.

The `cp` command 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
```

18.4.4.4 Removing Files and Directories

You can use the command rm to delete a file or directory.

The command rm 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
```

or

```
$ dbfs_client ETLUser@DBConnectString --command rm -R  dbfs:/staging_area/dir1
```

18.4.5 DBFS Mounting Interface (Linux and Solaris Only)

You can mount DBFS using the dbfs_client in Linux and Solaris only.

The instructions indicate the different requirements for the Linux and Solaris platforms.

- Installing FUSE on Solaris 11 SRU7 and Later
  You can use dbfs_client as a mount client in Solaris 11 SRU7 and later, if you install FUSE
- Solaris-Specific Privileges
  On Solaris, the user must have the Solaris privilege PRIV_SYS_MOUNT to perform mount and unmount operations on DBFS filesystems.
- About the Mount Command for Solaris and Linux
  The dbfs_client mount command for Solaris and Linux uses specific syntax.
- Mounting a File System with a Wallet
  You can mount a file system with a wallet after configuring various environment variables.
- Mounting a File System with Password at Command Prompt
  You must enter a password at the command prompt to mount a file system using dbfs_client.

- Unmounting a File System
  In Linux, you can run `fusermount` to unmount file systems.
- Mounting DBFS Through fstab Utility for Linux
  In Linux, you can configure `fstab` utility to use `dbfs_client` to mount a DBFS filesystem.
- Mounting DBFS Through the vfstab Utility for Solaris
  On Solaris, file systems are commonly configured using the `vfstab` utility.
- Restrictions on Mounted File Systems
  DBFS supports most file system operations with exceptions.
- Restrictions on Types of Files Stored at DBFS Mount Points
  DBFS should be avoided in scenarios that can cause a file operation on the DBFS files resulting in more data to be written back to the DBFS.

18.4.5.1 Installing FUSE on Solaris 11 SRU7 and Later

You can use `dbfs_client` as a mount client in Solaris 11 SRU7 and later, if you install `FUSE`

Install `FUSE` to use `dbfs_client` as a mount client in Solaris 11 SRU7 and later.

- Run the following package as `root`.

```
pkg install libfuse
```

18.4.5.2 Solaris-Specific Privileges

On Solaris, the user must have the Solaris privilege `PRIV_SYS_MOUNT` to perform mount and unmount operations on DBFS filesystems.

Give the user the Solaris privilege `PRIV_SYS_MOUNT` .

1. Edit `/etc/user_attr`.
2. 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.*
```

18.4.5.3 About the Mount Command for Solaris and Linux

The `dbfs_client` mount command for Solaris and Linux uses specific syntax.

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` or an alias specified in the `tnsnames.ora`.
- *`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.

18.4.5.4 Mounting a File System with a Wallet

You can mount a file system with a wallet after configuring various environment variables.

You must first configure the `LD_LIBRARY_PATH`, `ORACLE_HOME` environment variables and `sqlnet.ora` correctly before mounting a file system with a wallet.

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
   ```

   Using the wallet option runs the `dbfs_client` in the background

> **See Also:**
>
> Using Oracle Wallet with DBFS Client

18.4.5.5 Mounting a File System with Password at Command Prompt

You must enter a password at the command prompt to mount a file system using `dbfs_client`.

- Run the following command at the command prompt and provide the password:

```
$ dbfs_client ETLUser@DBConnectString /mnt/dbfs
  password: xxxxxxx
```

The `dbfs_client` runs in the foreground after the password is provided at the command prompt.

18.4.5.6 Unmounting a File System

In Linux, you can run `fusermount` to unmount file systems.

- Linux
- Solaris

Linux

To run `fusermount` in Linux, do the following:

- Run the following:

```
$ fusermount -u <mount point>
```

Solaris

In Solaris, you can run `umount` to unmount file systems.

- Run the following:

```
$ umount -u <mount point>
```

18.4.5.7 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.

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`.

18.4.5.8 Mounting DBFS Through the vfstab Utility for Solaris

On Solaris, file systems are commonly configured using the `vfstab` utility.

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
   ```

18.4.5.9 Restrictions on Mounted File Systems

DBFS supports most file system operations with exceptions.

The exceptions are:

- `ioctl`
- range locking (file locking is supported)
- asynchronous I/O through `libaio`
- `O_DIRECT` file opens
- hard links
- 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.

> **Note:**
>
> You should not run programs from a DBFS-mounted file system which was mounted with the `direct_io` option.

Oracle does not support exporting DBFS file systems using NFS or Samba.

18.4.5.10 Restrictions on Types of Files Stored at DBFS Mount Points

DBFS should be avoided in scenarios that can cause a file operation on the DBFS files resulting in more data to be written back to the DBFS.

The following scenarios are not exhaustive but provide examples of operations that can make the DBFS and the database interdependent and hence should be avoided:

- **Sample Scenario 1:** DBFS is the destination for the trace files generated by the same database that is hosting the DBFS.
  For example: The act of writing the trace file into the DBFS could generate more trace data to be written back into DBFS.
- **Sample Scenario 2:** The trail file of a database replication is in a DBFS and the DBFS is in the SAME database that is being replicated.
  For example: The act of writing into the trail by the replication process generates redo. This redo could feed back into the replication.
- **Sample Scenario 3:** DBFS is the destination of any database files of the same database.
  For example: The data files, control files, redo log files could make the DBFS and the database inter dependent.

18.4.6 File System Security Model

The database manages security in DBFS. It does not use the operating system security model.

- About the File System Security Model
  DBFS operates under a security model where all file systems created by a user are private to that user, by default.

- Enabling Shared Root Access
  As an operating system user who mounts the file system, you can allow root access to the file system by specifying the `allow_root` option.
- About DBFS Access Among Multiple Database Users
  DBFS allows multiple users to share a subset of the filesystem state.
- Establishing DBFS Access Sharing Across Multiple Database Users
  Learn about sharing access of DBFS to multiple database users in this section.

18.4.6.1 About the File 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, OS `user1` may mount a DBFS filesystem as `READ ONLY`, and OS `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.

18.4.6.2 Enabling Shared Root Access

As an operating system user who mounts the file system, you can allow root access to the file system by specifying the `allow_root` option.

This option requires that the `/etc/fuse.conf` file contain the `user_allow_other` field, as demonstrated in Example 18-1.

Example 18-1 Enabling Root Access for Other Users

```
# Allow users to specify the 'allow_root' mount option.
user_allow_other
```

18.4.6.3 About DBFS Access Among Multiple Database Users

DBFS allows multiple users to share a subset of the filesystem state.

A Single filesystem may be accessed by multiple database users. 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`.

18.4.6.4 Establishing DBFS Access Sharing Across Multiple Database Users

Learn about sharing access of DBFS to multiple database users in this section.

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.

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 `user2` 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> declare
                data blob;
                properties dbms_dbfs_content.properties_t;
            begin
                properties('posix:mode') :=
dbms_dbfs_content.propNumber(33188);
                dbms_dbfs_content.createFile('/user1_mnt/xxx', properties
=> properties, content => data);
            end;
            /
```

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 to `exportFilesystem`.

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`.

18.4.7 HTTP, WebDAV, and FTP Access to DBFS

Components that enable HTTP, WebDAV, and FTP access to DBFS over the Internet use various XML DB server protocols.

- 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`.
- 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.
- 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.
- HTTP Access to DBFS
  Users have read-only access through HTTP/HTTPS protocols.

18.4.7.1 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`.

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.

DBFS Content API for guidelines on DBFS store creation, registration, deregistration, mount, unmount and deletion

18.4.7.2 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.

18.4.7.3 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)

18.4.7.4 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.

18.5 Maintaining DBFS

DBFS administration includes tools that perform diagnostics, manage failover, perform backup, and so on.

- Using Oracle Wallet with DBFS Client
  Learn about using Oracle Wallet in this section.

- DBFS Diagnostics
  The `dbfs_client` program supports multiple levels of tracing to help diagnose problems.
- Preventing Data Loss During Failover Events
  The `dbfs_client` program can failover to one of the other existing database instances if one of the database instances in an Oracle RAC cluster fails.
- Bypassing Client-Side Write Caching
  The sharing and caching semantics for `dbfs_client` are similar to NFS in using the *close-to-open cache consistency* behavior.
- Backing up DBFS
  You have two alternatives for backing up DBFS.
- 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.

18.5.1 Using Oracle Wallet with DBFS Client

Learn about using Oracle Wallet in this section.

An Oracle Wallet allows the DBFS client to mount a DBFS store without requiring the user to enter a password.

> **See Also:**
>
> *Oracle Database Enterprise User Security Administrator's Guide* for more information about creation and management of wallets. Enterprise User Security (EUS) is deprecated with Oracle Database 23ai.

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
   ```

   The `mkstore` wallet management command line tool is deprecated with Oracle Database 23ai, and can be removed in a future release.

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
   ```

18.5.2 DBFS Diagnostics

The `dbfs_client` program supports multiple levels of tracing to help diagnose problems.

The `dbfs_client` can either output traces to a file or to `/var/log/messages` using the `syslog` daemon on Linux.

> **Note:**
>
> The `diagnostic_dest` initialization parameter sets the location of the automatic diagnostic repository. When you use `dbfs_client` or Oracle File Server (OFS) as the file system server, ensure that this parameter does not point to a directory inside `dbfs_client` or OFS as this can produce a dependency cycle and cause the system to hang.

When you trace to a file, the `dbfs_client` program keeps two trace files on disk. `dbfs_client`, rotates the trace files automatically, and limits disk usage to 10 MB.

By default, tracing is turned off except for critical messages which are always logged to `/var/log/messages`.

If `dbfs_client` cannot connect to the Oracle Database, enable tracing using the `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.

18.5.3 Preventing Data Loss During Failover Events

The `dbfs_client` program can failover to one of the other existing database instances if one of the database instances in an Oracle RAC cluster fails.

For `dbfs_client` failover to work correctly, you must modify the Oracle database service and specify failover parameters. Run the `DBMS_SERVICE.MODIFY_SERVICE` procedure to modify the service as shown Example 18-2

Example 18-2 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);
```

Once you have completed the prerequisite, you can prevent data loss during a failover of the DBFS connection after a failure of the back-end Oracle database instance. 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.

- Specify the `-o failover` mount option:

  ```
  $ dbfs_client database_user@database_server -o failover /mnt/dbfs
  ```

18.5.4 Bypassing Client-Side Write Caching

The sharing and caching semantics for `dbfs_client` are similar to NFS in using the *close-to-open cache consistency* behavior.

This allows multiple copies of `dbfs_client` to access the same shared file system. 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.

You can bypass client-side write caching.

- Specify `O_SYNC` when the file is opened.

  To force writes in the cache to disk call `fsync`.

18.5.5 Backing up DBFS

You have 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.

Topics:

- DBFS Backup 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.
- DBFS Backup 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.

18.5.5.1 DBFS Backup 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.

18.5.5.2 DBFS Backup 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.

Specify the `allow_root` mount option if backups are scheduled using the Oracle Secure Backup Administrative Server.

18.5.6 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. Oracle recommends `direct_io` option in `dbfs_client` for optimal performance for reads and writes.

18.6 Shrinking and Reorganizing DBFS Filesystems

DBFS uses Online File system Reorganization to shrink itself, enabling the release of allocated space back to the containing tablespace.

- About Changing DBFS File Systems
  DBFS file systems, like other database segments, grow dynamically with the addition or enlargement of files and directories.
- Advantages of Online Filesystem Reorganization
  DBFS Online Filesystem Reorganization is a powerful data movement facility with these certain advantages.
- 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.
- Required Permissions for Online Filesystem Reorganization
  Database users must have the following set of privileges for Online Filesystem Reorganizaton.
- Invoking Online Filesystem Reorganization
  You can perform an Online Filesystem Reorganization by creating a temporary DBFS filesystem.

18.6.1 About Changing DBFS File Systems

DBFS file systems, 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 file system to the various segments that make up the file system.

However, even if files and directories in the DBFS file system 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 file system 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

18.6.2 Advantages of Online Filesystem Reorganization

DBFS Online Filesystem Reorganization is a powerful data movement facility with these certain advantages.

These are:

- **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.

18.6.3 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:

- Query for a specific function in the DBFS PL/SQL packages.

```
$ sqlplus / as sysdba
SELECT * FROM dba_procedures
WHERE   owner = 'SYS'
    and object_name = 'DBMS_DBFS_SFS'
    and procedure_name = 'REORGANIZEFS';
```

If this query returns a single row similar to the one in this output, 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.

```
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
```

18.6.4 Required Permissions for Online Filesystem Reorganization

Database users must have the following set of privileges for Online Filesystem Reorganizaton.

Users must have these privileges:

- ALTER ANY TABLE
- DROP ANY TABLE
- LOCK ANY TABLE
- CREATE ANY TABLE
- SELECT ANY TABLE
- REDEFINE ANY TABLE
- CREATE ANY TRIGGER
- CREATE ANY INDEX
- CREATE TABLE
- CREATE MATERIALIZED VIEW
- CREATE TRIGGER

18.6.5 Invoking Online Filesystem Reorganization

You can perform an Online Filesystem Reorganization by creating a temporary DBFS filesystem.

> **Note:**
>
> Ensure that you don't create the temporary DBFS filesystem in the SYS schema. DBFS Online Filesystem Reorganization will not work if you create the temporary DBFS filesystem in the SYS schema.

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.

19
DBFS SecureFiles Store

There are certain procedures for setting up and using a DBFS SecureFiles Store.

- Setting Up a SecureFiles Store
  This section shows how to set up a SecureFiles Store.
- Using a DBFS SecureFiles Store File System
  The DBFS Content API provides methods to access and manage a SecureFiles Store file system.
- About DBFS SecureFiles Store Package, DBMS_DBFS_SFS
  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) .
- Database File System (DBFS)— POSIX File Locking
  Starting from Oracle Database 12*c* Release 2(12.2), Oracle supports the Database File system `POSIX` File locking feature.

19.1 Setting Up a SecureFiles Store

This section shows how to set up a SecureFiles Store.

- About Managing Permissions
  You must be a non-SYS database user for all operational access to the Content API and stores.
- Creating or Setting Permissions
  You must grant the `DBFS_ROLE` role to any user that needs to use the DBFS content API.
- Accessing SecureFiles Store
  You should never directly access tables that hold data for a SecureFiles Store file systems.
- Reinitializing SecureFiles Store File Systems
  You can truncate and re-initialize tables associated with an SecureFiles Store.
- Comparison of 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.

19.1.1 About Managing Permissions

You must be a non-SYS 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 to access DBFS Content API.

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.

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

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

19.1.2 Creating or Setting Permissions

You must grant the `DBFS_ROLE` role to any user that needs to use the DBFS content API.

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.

19.1.3 Accessing SecureFiles Store

You should never directly access tables that hold data for a SecureFiles Store file systems.

This is the correct way to access the file systems.

- 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`).

19.1.4 Reinitializing SecureFiles Store File Systems

You can truncate and re-initialize tables associated with an SecureFiles Store.

- Use the procedure `INITFS()`.

  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`, which is associated with the SecureFiles Store `store_name`.

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

19.1.5 Comparison of 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`.

19.2 Using a DBFS SecureFiles Store File System

The DBFS Content API provides methods to access and manage a SecureFiles Store file system.

- DBFS Content API Working Example
  You can create new file and directory elements to populate a SecureFiles Store file system.
- Dropping SecureFiles Store File Systems
  You can use the `unmountStore` method to drop SecureFiles Store file systems.

19.2.1 DBFS Content API Working Example

You can create new file and directory elements to populate a SecureFiles Store file system.

If you have executed the steps in "Setting Up a SecureFiles Store", set the DBFS Content API permissions, created at least one SecureFiles Store reference file system, and mounted it under the mount point `/mnt1`, then you can create a new file and directory elements as demonstrated in Example 19-1.

Example 19-1 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) ||
    '';
   properties      DBMS_DBFS_CONTENT.PROPERTIES_T;

BEGIN
   properties('posix:mode') := DBMS_DBFS_CONTENT.propNumber(16777);
                                          --  drwxr-xr-x  --
   properties('posix:uid')  := DBMS_DBFS_CONTENT.propNumber(0);
   properties('posix:gid')  := DBMS_DBFS_CONTENT.propNumber(0);
   DBMS_DBFS_CONTENT.createDirectory(
            '/mnt1/FS1',
            properties);
```

```
   properties('posix:mode') := DBMS_DBFS_CONTENT.propNumber(33188);
                                                --  -rw-r--r--  --
   DBMS_DBFS_CONTENT.createFile(
            '/mnt1/FS1/hello.c',
            properties,
            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/FS1%'
         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).

> **See Also:**
>
> DBFS Client Access Methods

19.2.2 Dropping SecureFiles Store File Systems

You can 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.

1. Unmount the store.

```
CONNECT sfs_demo/<password>

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

where:

- `store_name` is `FS1`, a case-sensitive unique username.
- `store_mount` is the mount point.

2. Unregister the stores.

```
CONNECT sfs_demo/<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 store.

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

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

19.3 About DBFS SecureFiles Store Package, DBMS_DBFS_SFS

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) .

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) . 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.

See Also:

- See *Oracle Database PL/SQL Packages and Types Reference* for more information about the `DBMS_DBFS_SFS` package.
- 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`.
- Introduction to Large Objects and SecureFiles for advanced features of SecureFiles LOBs.

19.4 Database File System (DBFS)— POSIX File Locking

Starting from Oracle Database 12*c* Release 2(12.2), Oracle supports the Database File system `POSIX` File locking feature.

The DBFS provides file locking support for the following types of applications:

- `POSIX` applications using `DBFS_CLIENT` (in mount mode) as a front-end interface to DBFS.

> **See Also:**
>
> DBFS Client Access Methods

- Applications using `PL/SQL` as an interface to DBFS.

> **Note:**
>
> Oracle supports only Full-file locks in DBFS. Full-file lock implies locking the entire file from byte zero offset to the end of file.

- About Advisory Locking
  Advisory locking is a file locking mechanism that locks the file for a single process.
- About Mandatory Locking
  Mandatory locking is a file locking mechanism that takes support from participating processes.
- File Locking Support
  Enabling the file locking mechanism helps applications to block files for various file system operations.
- Compatibility and Migration Factors of Database Filesystem—File Locking
  The Database Filesystem File Locking feature does not impact the compatibility of `DBFS` and `SFS` store provider with `RDBMS`.
- Examples of Database File System—File Locking
  These examples illustrate the advisory locking and the locking functions available on `UNIX` based systems.
- DBFS Locking Behavior
  This section describes the DBFS locking behavior.
- Scheduling File Locks
  DBFS File Locking feature supports lock scheduling.

19.4.1 About Advisory Locking

Advisory locking is a file locking mechanism that locks the file for a single process.

File locking mechanism cannot independently enforce any form of locking and requires support from the participating processes. For example, if a process `P1` has a `write` lock on file `F1`, the locking API or the operating system does not perform any action to prevent any other process `P2` from issuing a `read` or `write` system call on the file `F1`. This behavior of file locking mechanism is also applicable to other file system operations. The processes that are involved (in file locking mechanism) must follow a lock or unlock protocol provided in a suitable API form by the user-level library. File locking semantics are guaranteed to work as per POSIX standards.

19.4.2 About Mandatory Locking

Mandatory locking is a file locking mechanism that takes support from participating processes.

Mandatory locking is an enforced locking scheme that does not rely on the participating processes to cooperate and/or follow the locking API. For example, if a process `P1` has taken a

write lock on file F1 and if a different process P2 attempts to issue a read/write system call (or any other file system operation) on file F1 , the request is blocked because the concerned file is exclusively locked by process P1.

19.4.3 File Locking Support

Enabling the file locking mechanism helps applications to block files for various file system operations.

The fcntl(), lockf(), and flock() system calls in UNIX and LINUX provide file locking support. These system calls enable applications to use the file locking facility through dbfs_client-FUSE callback interface. File Locks provided by fcntl() are widely known as POSIX file locks and the file locks provided by flock() are known as BSD file locks. The semantics and behavior of POSIX and BSD file locks differ from each other. The locks placed on the same file through fcntl() and flock() are orthogonal to each other. The semantics of file locking functionality designed and implemented in DBFS is similar to POSIX file locks. In DBFS, semantics of file locks placed through flock() system call will be similar to POSIX file locks (such as fcntl()) and not BSD file locks. lockf() is a library call that is implemented as a wrapper over fcntl() system call on most of the UNIX systems, and hence, it provides POSIX file locking semantics. In DBFS, file locks placed through fcntl(), flock(), and lockf() system-calls provide same kind of behavior and semantics of POSIX file locks.

> **Note:**
>
> BSD file locking semantics are not supported.

19.4.4 Compatibility and Migration Factors of Database Filesystem—File Locking

The Database Filesystem File Locking feature does not impact the compatibility of DBFS and SFS store provider with RDBMS.

DBFS_CLIENT is a standalone OCI Client and uses OCI calls and DBMS_FUSE API.

> **Note:**
>
> This feature will be compatible with OraSDK/RSF .

19.4.5 Examples of Database File System—File Locking

These examples illustrate the advisory locking and the locking functions available on UNIX based systems.

The following example uses two running processes — Process A and Process B.

Example 19-2 No locking

```
Process A opens file:

file_desc = open("/path/to/file", O_RDONLY);
/* Reads data into bufffers */
read(fd, buf1, sizeof(buf));
read(fd, buf2, sizeof(buf));
close(file_desc);
```

Subjected to OS scheduling, `process B` can enter any time and issue a `write` system call affecting the integrity of file data.

Example 19-3 Advisory locking used but process B does not follow the protocol

```
Process A opens file:

file_desc = open("/path/to/file", O_RDONLY);
ret = AcquireLock(file_desc, RD_LOCK);
if(ret)
{
    read(fd, buf1, sizeof(buf));
    read(fd, buf2, sizeof(buf));
    ReleaseLock(file_desc);
}
close(file_desc);
```

Subjected to OS scheduling, `process B` can come in any time and still issue a `write` system call ignoring that `process A` already holds a `read` lock.

```
Process B opens file:

file_desc1 = open("/path/to/file", O_WRONLY);
write(file_desc1, buf, sizeof(buf));
close(file_desc1);
```

The above code is executed and leads to inconsistent data in the file.

Example 19-4 Advisory locking used and processes are following the protocol

```
Process A opens file:

file_desc = open("/path/to/file", O_RDONLY);
ret = AcquireLock(file_desc, RD_LOCK);
if(ret)
{
    read(fd, buf1, sizeof(buf));
    read(fd, buf2, sizeof(buf));
    ReleaseLock(file_desc);
}
close(file_desc);
```

```
Process B opens file:

file_desc1 = open("/path/to/file", O_WRONLY);
ret = AcquireLock(file_desc1, WR_LOCK);
/* The above call will take care of checking the existence of a lock */
if(ret)
{
    write(file_desc1, buf, sizeof(buf));
    ReleaseLock(file_desc1);
} close(file_desc1);
```

`Process B` follows the lock API and this API makes sure that the process does not `write` to the file without acquiring a lock.

19.4.6 DBFS Locking Behavior

This section describes the DBFS locking behavior.

The `DBFS` File Locking feature exhibits the following behaviors:

- File locks in `DBFS` are implemented with idempotent functions. If a process issues "N" `read` or `write` lock calls on the same file, only the first call will have an effect, and the subsequent "N-1" calls will be treated as redundant and returns No Operation (NOOP).
- File can be unlocked exactly once. If a process issues "N" `unlock` calls on the same file, only the first call will have an effect, and the subsequent "N-1" calls will be treated as redundant and returns NOOP.
- Lock conversion is supported only from `read` to `write`. If a process `P` holds a `read` lock on file `F` (and `P` is the only process holding the `read` lock), then a `write` lock request by `P` on file `F` will convert the `read` lock to `exclusive`/`write` lock.

19.4.7 Scheduling File Locks

DBFS File Locking feature supports lock scheduling.

This facility is implemented purely on the DBFS client side. Lock request scheduling is required when client application uses blocking call semantics in their `fcntl()`, `lockf()`, and `flock()` calls.

There are two types of scheduling:

- Greedy Scheduling
- Fair Scheduling

Oracle provides the following command line option to switch the scheduling behavior.

```
Mount -o lock_sched_option = lock_sched_option Value;
```

Table 19-1 `lock_sched_option` Value Description

Value	Description
1	Sets the scheduling type to Greedy Scheduling. (Default)
2	Sets the scheduling type to Fair Scheduling.

> **Note:**
>
> Lock Request Scheduling works only on per `DBFS_CLIENT` mount basis. For example, lock requests are not scheduled across multiple mounts of the same file system.

- Greedy Scheduling
  In this scheduling technique, the file lock requests does not follow any guaranteed order.
- Fair Scheduling
  The fair scheduling technique is implemented using a queuing mechanism on per file basis.

19.4.7.1 Greedy Scheduling

In this scheduling technique, the file lock requests does not follow any guaranteed order.

> **Note:**
>
> This is the default scheduling option provided by `DBFS_CLIENT`.

If a file `F` is `read` locked by process `P1`, and if processes `P2` and `P3` submit blocking `write` lock requests on file `F`, the processes `P2` and `P3` will be blocked (using a form of spin lock) and made to wait for its turn to acquire the lock. During the wait, if a process `P4` submits a `read` lock request (blocking call or a non-blocking call) on file `F`, `P4` will be granted the `read` lock even if there are two processes (`P2` and `P3`) waiting to acquire the `write` lock. Once both `P1` and `P4` release their respective `read` locks, one of `P2` and `P3` will succeed in acquiring the lock. But, the order in which processes `P2` and `P3` acquire the lock is not determined. It is possible that process `P2` would have requested first, but the process `P3`'s request might get unblocked and acquire the lock and the process `P2` must wait for `P3` to release the lock.

19.4.7.2 Fair Scheduling

The fair scheduling technique is implemented using a queuing mechanism on per file basis.

For example, if a file `F` is read locked by process `P1`, and processes `P2` and `P3` submit blocking write lock requests on file `F`, these two processes will be blocked (using a form of spin lock) and will wait to acquire the lock. The requests will be queued in the order received by the DBFS client. If a process `P4` submits a read lock request (blocking call or a non-blocking call) on file `F`, this request will be queued even though a read lock can be granted to this process.

DBFS Client ensures that after `P1` releases its read lock, the order in which lock requests are honored is P2->P3 -> P4.

This implies that `P2` will be the first one to get the lock. Once `P2` releases its lock, `P3` will get the lock and so on.

20
DBFS Hierarchical Store

The DBFS Hierarchical Store and related store wallet management work together to store less frequently used data.

20.1 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 external storage devices like tape and 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 provides you the ability to use tape as a storage tier when implementing Information Lifecycle Management (ILM) for database tables or content. 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.

20.2 Setting up the Store

You can create, register, and mount a hierarchical Store.

20.2.1 Creating, Registering, and Mounting the Store

Setting up a hierarchical file system store requires creating, registering, and mounting the store.

Creating, registering, and mounting the store.

1. Call `CREATESTORE`.

> **See Also:**
>
> CREATESTORE Procedure for more information on `CREATESTORE` procedure.

> **Note:**
>
> You create a wallet with the credentials of the Amazon S3 accounts if Amazon S3 is used as the external storage.

2. Set mandatory and optional properties using `DBMS_DBFS_HS.SETSTOREPROPERTY`.

> **See Also:**
>
> SETSTOREPROPERTY Procedure for more information on `SETSTOREPROPERTY` procedure.

3. Register the store using `DBMS_DBFS_CONTENT.REGISTERSTORE`.

> **See Also:**
>
> REGISTERSTORE Procedure for more information on `REGISTERSTORE` procedure.

4. Mount the store using `DBMS_DBFS_CONTENT.MOUNTSTORE`.

> **See Also:**
>
> MOUNTSTORE Procedure for more information on `MOUNTSTORE` procedure.

20.3 Using the Hierarchical Store

You can use the Hierarchical Store as an independent file system or as an archive solution for SecureFile LOBs.

- 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.

- Using Hierarchical Store as an Archive Solution For SecureFiles LOBs
  Use the `DBMS_LOB` package to archive SecureFiles LOBs in a tape or an S3 store.
- Dropping a Hierarchical Store
  You can drop a hierarchical store.
- Compression to Use with the Hierarchical Store
  The DBFS hierarchical store can store its files in compressed forms.
- Program Example Using Tape
  This example program configures and uses a tape store.
- Program Example Using Amazon S3
  This example program configures and uses an Amazon S3 store.

20.3.1 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.

> **See Also:**
>
> DBFS Content API

20.3.2 Using Hierarchical Store as an Archive Solution For SecureFiles LOBs

Use the `DBMS_LOB` package to archive SecureFiles LOBs in a tape or an S3 store.

The `DBMS_LOB` package archives SecureFiles LOBs in a tape or an S3 store. Use the following method to free space in the cache or to force cache resident contents to be written to an external storage device:

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

20.3.3 Dropping a Hierarchical Store

You can drop a hierarchical store.

To drop a hierarchical store, call:

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

20.3.4 Compression to Use with the Hierarchical Store

The DBFS hierarchical store can store its files in compressed forms.

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 performs best and still provides 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.

20.3.5 Program Example Using Tape

This example program configures and uses a tape store.

In the example, you must substitute valid values in some places, as 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 the tape store
  dbms_dbfs_hs.storePush(store_name) ;
  commit ;

end ;
/
```

20.3.6 Program Example Using Amazon S3

This 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 ;
/
```

20.4 The DBMS_DBFS_HS Package

The `DBMS_DBFS_HS` package is a service provider that enables use of tape or Amazon S3 Web service as storage for data.

- Constants for DBMS_DBFS_HS Package
  The `DBMS_DBFS_HS` PL/SQL package constants are very detailed.
- Methods for DBMS_DBFS_HS Package
  There are many methods in the `DBMS_DBFS_HS`package.

20.4.1 Constants for DBMS_DBFS_HS Package

The `DBMS_DBFS_HS` PL/SQL package constants are very detailed.

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

20.4.2 Methods for DBMS_DBFS_HS Package

There are many methods in the `DBMS_DBFS_HS`package.

Table 20-1 summarizes the `DBMS_DBFS_HS` PL/SQL package methods.

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

Table 20-1 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*

Table 20-1 (Cont.) Methods of the DBMS_DBFS_HS PL/SQL Packages

Method	Description
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*

20.5 Views for DBFS Hierarchical Store

The DBFS Hierarchical Stores have several types of views.

- DBA Views
  There are several views available for DBFS Hierarchical Store.
- User Views
  There are several views available for the DBFS Hierarchical Store.

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

20.5.1 DBA Views

There are several views available for DBFS Hierarchical Store.

Following are the views available for DBFS Hierarchical Store:

- 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.

20.5.2 User Views

There are several views available for the DBFS Hierarchical Store.

- `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.

21
Database File System Links

Database File System Links enable storing SecureFiles LOBs in a different location than usual.

- About Database File System Links
  DBFS Links allows storing SecureFiles LOBs transparently in a location separate from the segment where the LOB is normally stored. Instead, you store a link to the LOB in the segment.
- Ways to Create 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`.
- Database File System Links Copy
  The API `DBMS_LOB.COPY_DBFS_LINK(DSTLOB, SRCLOB, FLAGS)` provides the ability to copy a linked SecureFiles LOB.
- The DBMS_LOB Package Used with DBFS
  The `DBMS_LOB` package provides subprograms to operate on, or access and manipulate specific parts of a LOB or complete LOBs.
- DBMS_LOB Constants Used with DBFS
  Certain constants support DBFS link interfaces.
- DBMS_LOB Subprograms Used with DBFS
  You should note that some changes have been made to the `DBMS_LOB` subprograms over time.
- Copying a Linked LOB Between Tables
  You can copy DBFS links from source tables to destination tables.
- 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 can read from a linked SecureFiles LOB even if the data is not cached in the database.

21.1 About Database File System Links

DBFS Links allows storing SecureFiles LOBs transparently in a location separate from the segment where the LOB is normally stored. Instead, you store a link to the LOB in the segment.

The link in the segment must reference a path that uses DBFS Content API to locate the LOB when accessed. This means that the LOB could be stored on another file system, on a tape system, in the cloud, or any other location that can be accessed using DBFS Content API.

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 holds 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:

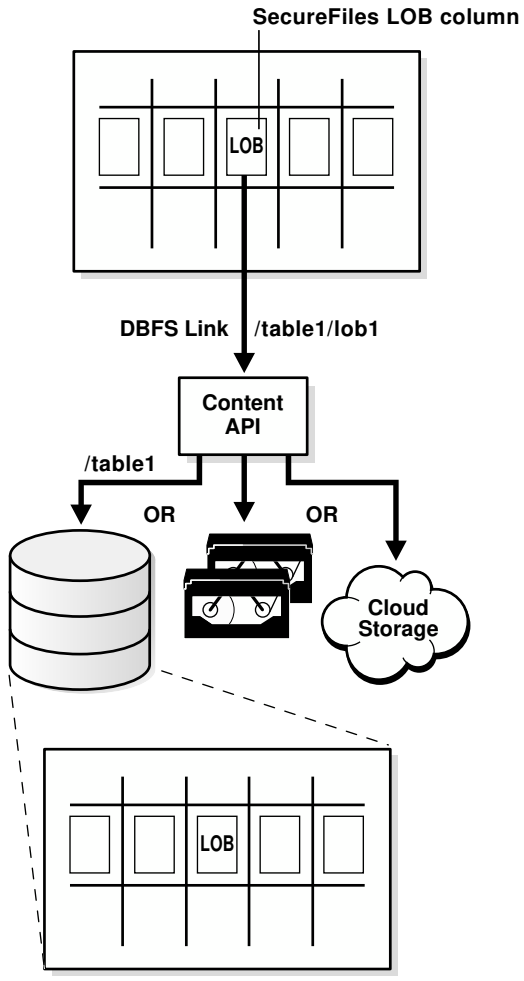
  If the LOB is not already cached in a local area in the database, then it will first be read into the database, modified as needed, and then written back to the DBFS content store defined in the DBFS Link for the LOB in question.

- 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 21-1 Database File System Link



21.2 Ways to Create 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 it is 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.

21.3 Database File System Links Copy

The API `DBMS_LOB.COPY_DBFS_LINK(DSTLOB, SRCLOB, FLAGS)` provides the ability to copy a linked SecureFiles LOB.

sBy 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.

21.4 The DBMS_LOB Package Used with DBFS

The `DBMS_LOB` package provides subprograms to operate on, or access and manipulate specific parts of a LOB or complete LOBs.

The `DBMS_LOB` package applies to both SecureFiles LOB and BasicFiles LOB.

DBMS_LOB Constants Used with SecureFiles LOBs and DBFS and DBMS_LOB Subprograms Used with SecureFiles LOBs and DBFS describe modifications made to the `DBMS_LOB` constants and subprograms with the addition of SecureFiles LOB and Database File System (DBFS).

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* for more information about `DBMS_LOB` package
- Introducing the Database File System

21.5 DBMS_LOB Constants Used with DBFS

Certain constants support DBFS link interfaces.

Table 21-1 lists constants that support DBFS Link interfaces.

See Also:

Oracle Database PL/SQL Packages and Types Reference for complete information about constants used in the PL/SQL `DBMS_LOB` package

Table 21-1 DBMS_LOB Constants That Support DBFS Link Interfaces

Constant	Description
`DBFS_LINK_NEVER`	DBFS link state value
`DBFS_LINK_YES`	DBFS link state value

Table 21-1 (Cont.) DBMS_LOB Constants That Support DBFS Link Interfaces

Constant	Description
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 path names; 1024.
CONTENTTYPE_MAX_SIZE	The maximum 1-byte ASCII characters for content type; 128.

21.6 DBMS_LOB Subprograms Used with DBFS

You should note that some changes have been made to the DBMS_LOB subprograms over time.

Table 21-2 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 21-2 DBMS_LOB Subprograms

Subprogram	Description
COPY_DBFS_LINK	Copies an existing DBFS link into a new LOB

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

Table 21-2 (Cont.) DBMS_LOB Subprograms

Subprogram	Description
COPY_FROM_DBFS_LINK	Copies the specified LOB data from DBFS HSM Store into the database **See Also:** *Oracle Database PL/SQL Packages and Types Reference*
DBFS_LINK_GENERATE_PATHNAME	Returns a unique file path name for creating a DBFS Link **See Also:** *Oracle Database PL/SQL Packages and Types Reference*
GET_DBFS_LINK	Returns the DBFS path name for a LOB **See Also:** *Oracle Database PL/SQL Packages and Types Reference*
GET_DBFS_LINK_STATE	Returns the linking state of a LOB **See Also:** *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 Also:** *Oracle Database PL/SQL Packages and Types Reference*

Table 21-2 (Cont.) DBMS_LOB Subprograms

Subprogram	Description
SET_DBFS_LINK	Links a LOB with a DBFS path name

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

21.7 Copying a Linked LOB Between Tables

You can copy DBFS links from source tables to destination tables.

Use the following code to copy any DBFS Links that are stored in any SecureFiles LOBs in the source table to the destination table.

`CREATE TABLE ... AS SELECT (CTAS)` and `INSERT TABLE ... AS SELECT (ITAS)`

21.8 Online Redefinition and DBFS Links

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

21.9 Transparent Read

DBFS Links can read from a linked SecureFiles LOB even if the data is not cached in the database.

You can read data from the content store where the data is currently stored and stream 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*

22

DBFS Content API

You can enable applications to use the Database File System (DBFS) in several different programming environments.

- Overview of DBFS Content API
  You can enable applications to use DBFS using the DBFS Content API (`DBMS_DBFS_CONTENT`), which is a client-side programmatic API package. You can write applications in SQL, PL/SQL, JDBC, OCI, and other programming environments.
- Stores and DBFS Content API
  The DBFS Content API aggregates the path namespace of one or more stores into a single unified namespace.
- 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.
- 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.
- Querying DBFS Content API Space Usage
  You can query file system space usage statistics.
- DBFS Content API Session Defaults
  Normal client access to the DBFS Content API executes with an implicit context that consists of certain objects.
- 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.
- DBFS Content API Creation Operations
  You must implement the provider SPI so 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).
- DBFS Content API Deletion Operations
  You must implement the provider SPI so 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).
- DBFS Content API Path Get and Put Operations
  You can query existing path items or update them using simple `GET`*`XXX`*`()` and `PUT`*`XXX`*`()` methods.
- DBFS Content API Rename and Move Operations
  You can rename or move path names, possibly across directory hierarchies and mount points, but only within the same store.
- Directory Listings
  Directory listings are handled several different ways.

- DBFS Content API Directory Navigation and Search
  Clients of the DBFS Content API can list or search the contents of directory path names, with optional modes.
- DBFS Content API Locking Operations
  DBFS Content API clients can apply user-level locks,depending on certain criteria.
- DBFS Content API Access Checks
  The DBFS Content API checks the access of specific path names by operations.
- DBFS Content API Abstract Operations
  All of the operations in the DBFS Content API are represented as abstract `opcodes`.
- DBFS Content API Path Normalization
  There is a process for performing API path normalization.
- DBFS Content API Statistics Support
  DBFS provides support to reduce the expense of collecting DBFS Content API statistics.
- DBFS Content API Tracing Support
  Any DBFS Content API user (both clients and providers) can use DBFS Content API tracing, a generic tracing facility.
- Resource and Property Views
  You can see descriptions of Content API structure and properties in certain views.

22.1 Overview of DBFS Content API

You can enable applications to use DBFS using the DBFS Content API (`DBMS_DBFS_CONTENT`), which is a client-side programmatic API package. You can write applications 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 Also:**
>
> DBFS SecureFiles Store

Examples of possible providers include:

- Packaged applications that want to expose data through files.
- Custom applications developers use to leverage the file system interface, such as an application that stores medical images.

22.2 Stores and DBFS Content API

The DBFS Content API aggregates the path namespace of one or more stores into a single unified namespace.

The first component of the path name is used to disambiguate the namespace and then present it to client applications. This allows clients to access the underlying documents using either a full absolute path name represented by a single string, as shown in the following code snippet:

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

The DBFS Content API then takes care of correctly dispatching various operations on path names to the appropriate store provider .

Store providers must conform to the store provider interface (SPI) as declared by the package `DBMS_DBFS_CONTENT_SPI`.

- Creating Your Own DBFS Store
- *Oracle Database PL/SQL Packages and Types Reference* for `DBMS_DBFS_CONTENT` package syntax reference

22.3 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.

- DBFS Content API Role
  Access to the content operational and administrative API (packages, types, tables, and so on) is available through `DBFS_ROLE`.
- Path Name Constants and Types
  Path name constants are modeled after their SecureFiles LOBs store counterparts.
- Path Properties
  Every path name in a store is associated with a set of properties.
- Content IDs
  Content IDs are unique identifiers that represent a path in the store.
- Path Name Types
  Stores can store and provide access to eight types of entities.
- 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.
- Lock Types
  Stores that support locking should implement three types of locks.
- 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.

- 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).
- User-Defined Properties
  You can define your own properties for use in your application.
- 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.
- Exceptions
  DBFS Content API operations can raise any one of the top-level exceptions.
- Property Bundles
  Property bundles are discussed as `property_t` record type and `properties_t`.
- Store Descriptors
  Store descriptors are discussed as `store_t` and `mount_t` records.

> **See Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* for more information

22.3.1 DBFS Content API Role

Access to the content operational and administrative API (packages, types, tables, and so on) is available through `DBFS_ROLE`.

The `DBFS_ROLE` can be granted to all users as needed.

22.3.2 Path Name Constants and Types

Path name constants are modeled after their SecureFiles LOBs store counterparts.

> **See Also:**
>
> DBMS_DBFS_CONTENT Constants for path name constants and their types

22.3.3 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 Also:

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

22.3.4 Content IDs

Content IDs are unique identifiers that represent a path in the store.

See Also:

Oracle Database PL/SQL Packages and Types Reference for details of the `DBMS_DBFS_CONTENT` Content ID constants and properties

22.3.5 Path Name Types

Stores can store and provide access to eight types of entities.

The entities are:

- `type_file`
- `type_directory`
- `type_link`
- `type_reference`
- `type_scoket`
- `type_character`
- `type_block`
- `type_fifo`

Not all stores must implement all directories, links, or references.

> **See Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* for details of the `DBMS_DBFS_CONTENT` constants and path name types

22.3.6 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 the supported features and the ones that are not supported.

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

22.3.7 Lock Types

Stores that support locking should implement three types of locks.

The three types of locks are: `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 Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* for details of the lock types and constants.

22.3.8 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 Also:**
>
> See *Oracle Database PL/SQL Packages and Types Reference* for details of the standard properties and constants

22.3.9 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.

> **See Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* for details of the optional properties and constants

22.3.10 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.

22.3.11 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 Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* for details of the property access flags and constants

22.3.12 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.

> **See Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* for details of the Exceptions

22.3.13 Property Bundles

Property bundles are discussed as `property_t` record type and `properties_t`.

- 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 Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* for details of the `PROPERTY_T` record type

22.3.14 Store Descriptors

Store descriptors are discussed as `store_t` and `mount_t` records.

- A `store_t` is a record that describes a store registered with, and managed by the DBFS Content API .
- 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 Also:**
>
> - Administrative and Query APIs
> - *Oracle Database PL/SQL Packages and Types Reference* for details of the `STORE_T` record type

22.4 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 points (and this is under client control).

- Registering a Content Store
  You can register a new store that is backed by a provider that uses the `provider_package` procedure as the store service provider.
- Unregistering a Content Store
  You can unregister a previously registered store, which invalidates all mount points associated with it.
- Mounting a Registered Store
  You can mount a registered store and bind it to the mount point.
- Unmounting a Previously Mounted Store
  You can unmount a previously mounted store, either by name or by mount point.
- Listing all Available Stores and Their Features
  You can list all the available stores.
- Listing all Available Mount Points
  You can list all available mount points, their backing stores, and the store features.

- Looking Up Specific Stores and Their Features
  You can look up the path name, store name, or mount point of a store.

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

22.4.1 Registering a Content Store

You can register a new store that is backed by a provider that uses the `provider_package` procedure as the store service provider.

The method of registration conforms to the `DBMS_DBFS_CONTENT_SPI` package signature.

- Use the `REGISTERSTORE()` procedure.

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

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

22.4.2 Unregistering a Content Store

You can unregister a previously registered store, which invalidates all mount points associated with it.

Once the store is unregistered, access to the store and its mount points is no longer guaranteed, although a consistent read may provide a temporary illusion of continued access.

- Use the `UNREGISTERSTORE()` procedure.

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

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

22.4.3 Mounting a Registered Store

You can mount a registered store and bind it to the mount point.

- Use the `MOUNTSTORE()`procedure.

After you mount the store, access to the path names in 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 you do not specify a mount point and therefore, it 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).

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

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

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

22.4.4 Unmounting a Previously Mounted Store

You can unmount a previously mounted store, either by name or by mount point.

Attempting to unmount a store by name unmounts all mount points associated with the store.

- Use the `UNMOUNTSTORE()` procedure.

Once unmounted, access to the store or mount-point is no longer guaranteed to work although a consistent read may provide a temporary illusion of continued access. If the `ignore_unknown` argument is `true`, attempts to unmount unknown stores does not raise an exception.

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

22.4.5 Listing all Available Stores and Their Features

You can list all the available stores.

The `store_mount` field of the returned records is set to `null` because mount points are separate from stores themselves.

- Use the `LISTSTORES()` function.

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

22.4.6 Listing all Available Mount Points

You can list all available mount points, their backing stores, and the store features.

A single mount returns a single row, with the `store_mount` field set to `null`.

- Use the `LISTMOUNTS()` function.

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

22.4.7 Looking Up Specific Stores and Their Features

You can look up the path name, store name, or mount point of a store.

- Use `GETSTOREBY`*`XXX`*`()` or `GETFEATUREBY`*`XXX`*`()` functions.

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

22.5 Querying DBFS Content API Space Usage

You can query file system space usage statistics.

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.

- Use the `SPACEUSAGE()` method

where:

- `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, so 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 Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* for details of the `SPACEUSAGE()` method

22.6 DBFS Content API Session Defaults

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

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

22.7 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 Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* for details of the `GETVERSION()` method

22.8 DBFS Content API Creation Operations

You must implement the provider SPI so 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 associated 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. 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.

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, `DBMS_DBFS_CONTENT()` Constants - Optional Properties, and `DBMS_DBFS_CONTENT` Constants - Standard Properties

22.9 DBFS Content API Deletion Operations

You must implement the provider SPI so 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

22.10 DBFS Content API Path Get and Put Operations

You can query existing path items or update them using 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 Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* for details of the `DBMS_DBFS_CONTENT` methods

22.11 DBFS Content API Rename and Move Operations

You can rename or move path names, 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 Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* for details of the `DBMS_DBFS_CONTENT.RENAMEPATH()` methods

22.12 Directory Listings

Directory listings are handled several different ways.

- 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 Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* for details of data structures

22.13 DBFS Content API Directory Navigation and Search

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

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

22.14 DBFS Content API Locking Operations

DBFS Content API clients can apply user-level locks,depending on certain criteria.

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.

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

22.15 DBFS Content API Access Checks

The DBFS Content API checks the access of specific path names by operations.

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"

This is a convenience function for the client; a store that supports access control still internally performs these checks to guarantee security.

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

22.16 DBFS Content API Abstract Operations

All of the operations in the DBFS Content API are represented as abstract `opcodes`.

Clients can use`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 `opcode`s 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 Also:**
>
> - DBFS Content API Access Checks
> - *Oracle Database PL/SQL Packages and Types Reference* for more information about `DBMS_DBFS_CONTENT` Constants - Operation Codes.

22.17 DBFS Content API Path Normalization

There is a process for performing 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 Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* for details of the `DBMS_DBFS_CONTENT.RENAMEPATH()` methods

22.18 DBFS Content API Statistics Support

DBFS provides support to reduce the expense of collecting DBFS Content API statistics.

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.

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

22.19 DBFS Content API Tracing Support

Any DBFS Content API user (both clients and providers) can use DBFS Content API tracing, a generic tracing facility.

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.

> **See Also:**
>
> - Example 22-1 for more information about how to enable tracing using the DBFS Content APIs.
> - *Oracle Database PL/SQL Packages and Types Reference* for details of the `DBMS_DBFS_CONTENT` methods

Example 22-1 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 '');
```

22.20 Resource and Property Views

You can see descriptions of Content API structure and properties in certain views.

Certain views describe the structure and properties of Content API.

See Also:

- *Oracle Database Reference* for more information about `DBFS_CONTENT` views
- *Oracle Database Reference* for more information about `DBFS_CONTENT_PROPERTIES` views

23
Creating Your Own DBFS Store

You can create your own DBFS Store using DBFS Content Store Provider Interface (`DBMS_DBFS_CONTENT_SPI`).

- 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`). It is the basis for existing stores such as the DBFS SecureFiles Store and the DBFS Hierarchical Store, as well as any user-defined DBFS stores that you create.
- DBFS Content Store Provider Interface (DBFS Content SPI)
  The DBFS Content SPI (Store Provider Interface) is a specification only and has no package body.
- Creating a Custom Store Provider
  You can use this example store provider for DBFS, TaBleFileSystem Store Provider ("tbfs"), as a skeleton for custom providers or as a learning tool, to become familiar with the DBFS and its SPI.

23.1 Overview of DBFS Store Creation and Use

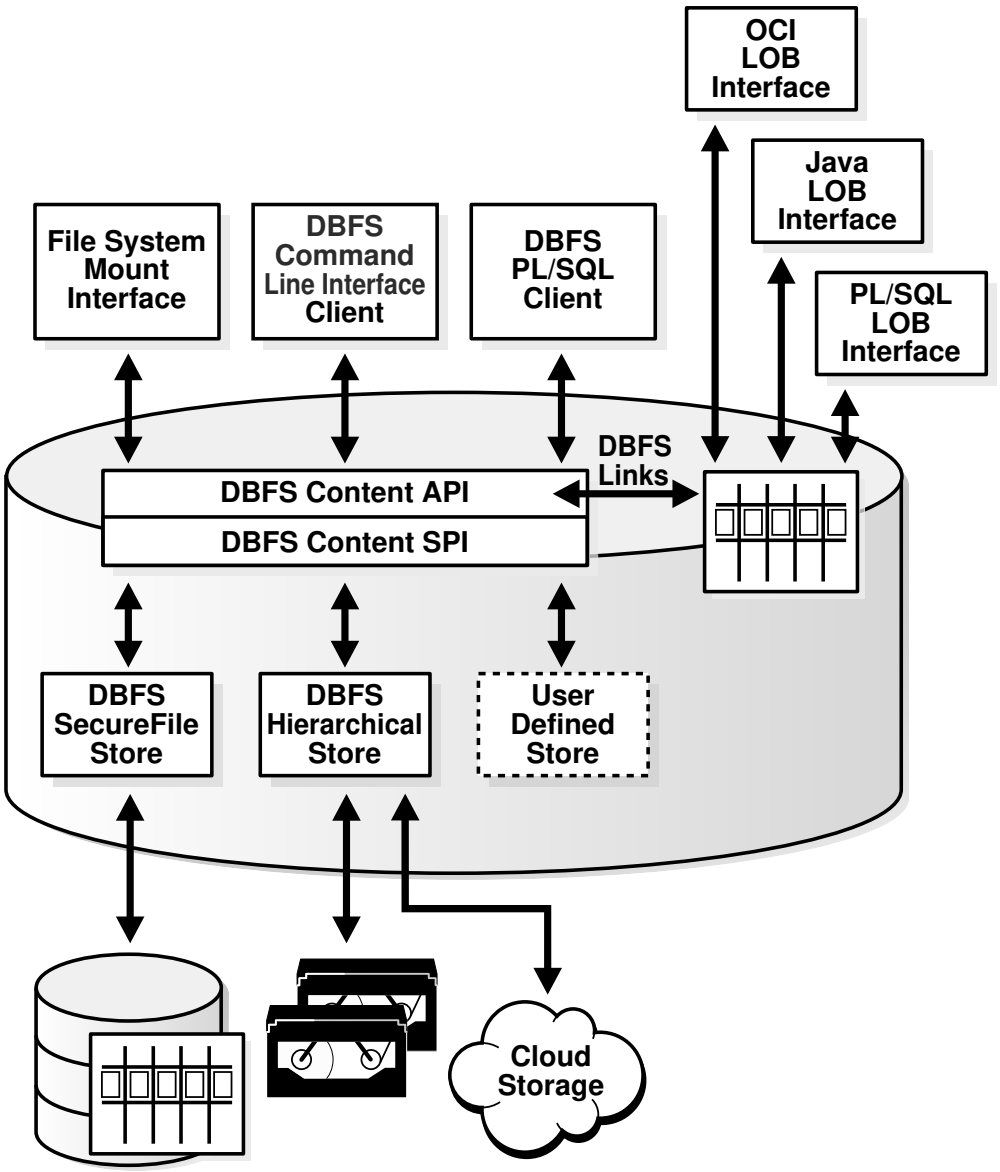
In order to customize a DBFS store, you must implement the DBFS Content SPI (`DBMS_DBFS_CONTENT_SPI`). It is the basis for existing stores such as the DBFS SecureFiles Store and the DBFS Hierarchical Store, as well as any user-defined DBFS stores that you create.

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.

Once you create your DBFS store, you use it much the same way that you would a SecureFiles Store.

> **See Also:**
>
> - DBFS Content API
> - DBFS SecureFiles Store

Figure 23-1 Database File System (DBFS)



23.2 DBFS Content Store Provider Interface (DBFS Content SPI)

The DBFS Content SPI (Store Provider Interface) is a specification only and has no package body.

You must implement the package body 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 the DBFS Content SPI package body as needed. 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`).

Note that all path name references must be store-qualified, that is, the notion of mount points and full absolute path names has 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 SPI implementation is a one-to-many pluggable architecture, the DBFS Content API uses dynamic SQL to invoke methods in the SPI implementation; this may lead to run time errors if your SPI implementation does not follow the specification of SPI implementation given in this document.

There are no explicit initial or final methods to indicate when the DBFS Content API plugs and unplugs a particular SPI implementation. SPI implementations 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

23.3 Creating a Custom Store Provider

You can use this example store provider for DBFS, TaBleFileSystem Store Provider ("tbfs"), as a skeleton for custom providers or as a learning tool, to become familiar with the DBFS and its SPI.

This example store provider for DBFS, exposes a relational table containing a `BLOB` column as a flat, non-hierarchical filesystem, that is, a collection of named files.

To use this example, it is assumed 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.

- Installation and Setup
  You will need certain files for installation and setup of the DBFS TaBleFileSystem Store Provider ("tbfs").
- TBFS Use
  Once the example store provider for DBFS, TaBleFileSystem Store Provider ("tbfs") is installed, files can be added or removed in several different ways and other changes can be made to the TBFS.
- TBFS Internals
  The TBFS is simple because its primary purpose is to serve as a teaching and learning example.
- Example Scripts
  This section describes some example SQL scripts.

23.3.1 Installation and Setup

You will need certain files for installation and setup of the DBFS TaBleFileSystem Store Provider ("tbfs").

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.

23.3.2 TBFS Use

Once the example store provider for DBFS, TaBleFileSystem Store Provider ("tbfs") is installed, files can be added or removed in several different ways and other changes can be made to the TBFS.

A `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.

23.3.3 TBFS Internals

The TBFS is simple because 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.

23.3.4 Example Scripts

This section describes some example SQL scripts.

- Driver Script
  The TBFS.SQL script is the top level driver script.
- Creating a Test User, Tablespace and Table to Backup Filesystem
  The TBL.SQL script creates a test user, a tablespace, the table that backs the filesystem and so on.
- Providing SPI Specification
  The `spec.sql` script provide the SPI specification of the tbfs.
- SPI Implementation of tbfs
  The `body.sql` script provides the SPI implementation of the tbfs.

- Registering and Mounting the DBFS
  The `capi.sql` script registers and mounts the DBFS.

23.3.4.1 Driver Script

The TBFS.SQL script is the top level driver script.

The TBFS.SQL script:

```
set echo on;

@tbl
@spec
@body
@capi

quit;
```

23.3.4.2 Creating a Test User, Tablespace and Table to Backup Filesystem

The TBL.SQL script creates a test user, a tablespace, the table that backs the filesystem and so on.

The TBL.SQL script :

```
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;
```

23.3.4.3 Providing SPI Specification

The `spec.sql` script provide the SPI specification of the tbfs.

The `spec.sql` script:

```
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;
```

23.3.4.4 SPI Implementation of tbfs

The `body.sql` script provides the SPI implementation of the tbfs.

The `body.sql` script:

```
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;
```

23.3.4.5 Registering and Mounting the DBFS

The `capi.sql` script registers and mounts the DBFS.

The `capi.sql` script:

```
connect tbfs/tbfs;

exec dbms_dbfs_content.registerStore('MY_TBFS', 'table', 'TBFS');
exec dbms_dbfs_content.mountStore('MY_TBFS', singleton => true);
commit;
```

24
DBFS Access Using OFS

You can access Database File System (DBFS) using the Oracle File Server (OFS) process.

This chapter provides details about how OFS uses OFSD, a dedicated background process, to manage DBFS. It also provides details about how you can access and manage DBFS.

To access a newly created DBFS across multiple nodes where there are no Oracle Client installations, use OFS to NFS mount the file system. In the absence of an Oracle Client installation, use OFS to mount the newly created DBFS to NFS and use it across multiple nodes. All file system requests are served by threads in the OFS background process.

24.1 About OFS

Oracle File Server (OFS) addresses the need to store PDB specific scripts, logs, trace files and other files produced by running an application in the database.

Additionally, you can use OFS for the following tasks:

- As a staging area where you can host the source data before it is loaded into database tables.
- To store import or export files from Oracle Data Pump process.

Ensure that you do not place core database files such as data, redo, archive log files, and database trace file on OFS as this can produce a dependency cycle and cause the system to hang. Similarly, the `diagnostic_dest` initialization parameter that sets the location of the automatic diagnostic repository should not point to a directory inside OFS.

OFS provides methods and procedures to allow you to create a Database file system using storage that is part of the PDB. You can mount the created file system, unmount it like any other Unix file system using PL/SQL procedures, and destroy the file system when it is no longer in use. When the PDB is destroyed, the file system is also destroyed, which frees up the underlying storage space.

24.2 About Oracle File Server Process

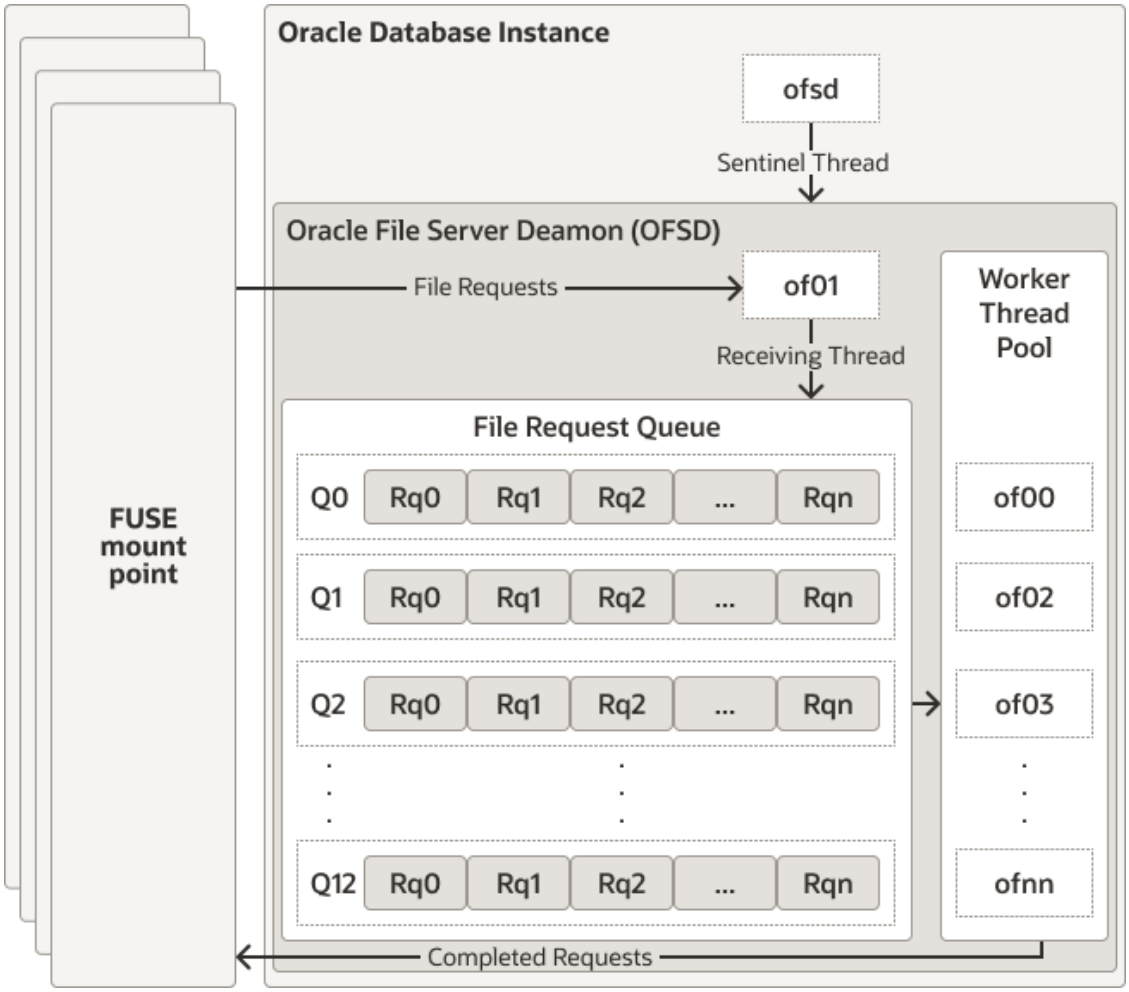
OFS manages the database file system using a non-fatal and dedicated background process called Oracle File Server Deamon (OFSD).

For more information about background process, see Background Processes in the *Database Reference* guide.

When an instance starts, the OFSD process gets spawned on operating system platforms, such as Linux, where OFS is supported. OFSD is multi-threaded and non-fatal. It serves both file system management requests and file requests from each mounted file system.

The centralized server background process model of OFS allows multiple file systems to be mounted and accessed using a limited set of server threads. It allows better resource sharing and a linear scalability with new file server threads created on demand. Both memory and CPU used by these threads are controlled through system wide parameters set in the RDBMS instance.

OFSD process starts two types of threads: receiver thread and worker thread. The receiver thread receives requests from the mounted file system. The name of this thread name is similar to `of01`. The requests received by this thread are placed in a submit queue which is served by different worker threads. The submit queue is hash partitioned to efficiently distribute the incoming requests across all the worker threads. By default, OFSD starts 3 worker threads. You can update the value of the `OFS_THREADS` parameter to increase the number of worker thread.



OFSD supports these 2 types of file systems: DBFS and OFS.

Use the `DBMS_FS` PL/SQL procedures to create, mount, and work with the file systems managed by the OFSD process.

OFSD uses a pool of worker threads to serve requests from multiple file systems that are mounted on the instance. Use `V$OFSMOUNT` to query the mounted file systems. The response that is returned is specific to each PDB. It lists only the file systems that are mounted in the specified PDB.

OFS Configuration Parameters

The following table specifies the parameter that can be tuned to provide file system access for database objects using OFS.

Table 24-1 OFS Configuration Parameters

Parameter Name	Description
OFS_THREADS	Set the number of OFS worker threads to handle OFS requests. Enter an integer value in the range of 3–128. The default value of OFS_THREADS is 4, which includes 1 receiver thread and 3 worker threads. by default, OFSD starts 3 worker threads. Set the value for this parameter based on the total number of mounted file systems and the rate at which file operations are performed in each file system. You can update the number OFS_THREADS to increase the number of DBMS_FS requests that are executed in parallel. This increases the number of worker threads executing the make, mount, unmount, and destroy operations on Oracle file systems in the Oracle database. Set this value after careful consideration as you can only increase this value dynamically and you cannot decrease it. Before changing the value, query the V$OFS_THREADS view to list all the running OFS threads and to retrieve details about the different OFS threads. For information about the columns and data types of this view, see V$OFS_THREADS in *Oracle Database Reference*. Increasing the value of OFS_THREADS, results in a significant reduction of time taken to execute parallel file system requests in environments that contain multiple PDBs. If you set a high value for OFS_THREADS, then the specified number of threads are created. If there is no workload, these threads remain in an idle state and wait for new work.

Note:

The diagnostic_dest initialization parameter sets the location of the automatic diagnostic repository. When you use dbfs_client or Oracle File Server (OFS) as the file system server, ensure that this parameter does not point to a directory inside dbfs_client or OFS as this can produce a dependency cycle and cause the system to hang.

24.4 OFS Client Interface

The OFS interface includes views and procedures that support the OFS operations.

Use the PL/SQL procedures provided by OFS to create, mount, unmount, and destroy file systems. You can query the views to identify the state of the mounted file systems and to collect performance data on individual file I/O operations.

- DBMS_FS Package
  Use the DBMS_FS package to manage the file systems. Use the procedures in this package to create, mount, unmount and destroy a file system in the Oracle Database.
- Views for OFS
  The views that support OFS operations start with V$OFS.

24.4.1 DBMS_FS Package

Use the DBMS_FS package to manage the file systems. Use the procedures in this package to create, mount, unmount and destroy a file system in the Oracle Database.

Multiple PDBs can submit these jobs in parallel. The requests are executed in parallel by different worker threads. You can obtain the status of the operations by querying `V$OFS_THREADS`. OFSD manages the threads. It creates new worker threads, when the number of current worker threads are less than the value of the `OFS_THREADS` parameter.

> **See Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* for more information about Oracle OFS procedures.

The following example illustrates the use of `DBMS_FS` package.

```
BEGIN
 DBMS_FS.MAKE_ORACLE_FS (
  fstype           => 'dbfs',
  fsname           => 'dbfs_fs1',
  mount_options    => 'TABLESPACE=dbfs_fs1_tbspc');
END;
/
BEGIN
 DBMS_FS.MOUNT_ORACLE_FS (
  fstype           => 'dbfs',
  fsname           => 'dbfs_fs1',
  mount_point      => '/oracle/dbfs/testfs',
  mount_options    => 'default_permissions, allow_other');
END;
/
/************** Now you can access the file system. All the FS operations go
here ***************/

BEGIN
 DBMS_FS.UNMOUNT_ORACLE_FS (
  fsname           => 'dbfs_fs1',
  mount_point      => '/oracle/dbfs/testfs',
  unmount_options    => 'force');
END;
/
BEGIN
 DBMS_FS.DESTROY_ORACLE_FS (
  fstype           => 'dbfs',
  fsname           => 'dbfs_fs1');
END;
/
```

24.4.2 Views for OFS

The views that support OFS operations start with `V$OFS`.

Table 24-2 Fixed Views for OFS

View	Description
V$OFS_THREADS	Query this view to list all the running OFS threads and to retrieve details about the different OFS threads, such as the thread type, file system on which the thread is working, time spent by the worker thread on a particular operation. For information about the columns and data types of this view, see V$OFS_THREADS in *Oracle Database Reference*.
V$OFSMOUNT	Query this view to retrieve details about the file systems that are mounted by Oracle File System. For information about the columns and data types of this view, see V$OFSMOUNT in *Oracle Database Reference*.
V$OFS_STATS	Query this view to list the number of times each file operation has been called for a mount point. For information about the columns and data types of this view, see V$OFS_STATS in *Oracle Database Reference*.

24.5 Managing DBFS Locally Using FUSE

Understand how you can manage DBFS using Filesystem in User Space (FUSE).

The FUSE interface in the Linux kernel makes the file systems available to the operating system processes. After mounting the file system, you can export it, and then NFS mount it on other nodes where client applications can access this file system.

- Configuring FUSE
  OFSD exposes the database file system through FUSE. Before using OFSD to mount the database file systems, you must install and configure the FUSE module.
- Accessing OFS in Cloud
  To access files from an OFS mounted on any Cloud environment, you must perform additional steps to configure the environment.

24.5.1 Configuring FUSE

OFSD exposes the database file system through FUSE. Before using OFSD to mount the database file systems, you must install and configure the FUSE module.

If you are running your database instance in a Compute node, configure the FUSE module in that node. The file system gets mounted and is visible through a mounted path on the compute node. In a RAC configuration, configure FUSE in each node, so that the OFS file systems can be mounted independently in each node.

To configure the FUSE module in Cloud or on-premises environment, where the database instance is running:

1. Set read and execute permissions for an Oracle user to use the FUSE executable file, `fusermount`.

   ```
   sudo chmod o+rx /usr/bin/fusermount
   ```

   Use the `fusermount` file to mount and unmount the FUSE user mode file systems.
2. Set the `setuid` bit on the `fusermount` file to permit an Oracle user to mount file systems.

   ```
   sudo chmod u+s /usr/bin/fusermount
   ```

3. Permit other users to access the mounted file system.

```
sudo sh -c ''echo user_allow_other >> /etc/fuse.conf''
```

4. Optional. By default, the maximum number of file systems that you can mount using FUSE is 1000. If you are running a large number of PDBs and need to configure a separate file system for each PDB, then run the following command to increase the number of file systems that can be mounted using FUSE. The following command increases the number of file systems that can be mounted using FUSE to 4000.

```
sudo sh -c ''echo mount_max=4000 >> /etc/fuse.conf''
```

5. Allow all users to read the `fuse.conf` file, so that the Oracle process can read this file at run time.

```
sudo chmod a+r /etc/fuse.conf
```

24.5.2 Accessing OFS in Cloud

To access files from an OFS mounted on any Cloud environment, you must perform additional steps to configure the environment.

To access files in an OFS mount in the Cloud environment, you may need to perform additional configuration. It may not be possible to export the OFS mount point from database node to client node due to security reasons. This may hinder client applications from accessing the OFS files through operating system commands and utilities and the OFS mount path may not be available to access using system calls. In such situations, Oracle recommends that you use the `utl_file` package to access files in the OFS mount. For information about UTL file package, see Summary of UTL_FILE Subprograms in *PL/SQL Packages and Types Reference*.

You can also use the `impdp` and `expdp` command-line clients to access files in the OFS mount. See Oracle Data Pump Import and Oracle Data Pump Export in the *Utilities guide*.

To configure the environment so that client applications in Cloud can access files in an OFS:

1. Create a directory object using the OFS mount path.

   The following sample code displays how you can create a directory object called `pdb1_ofsdir` when `/u03/dbfs/<pdbid>/data` is the OFS mount directory on the db node.

```
CREATE DIRECTORY pdb1_ofsdir AS '/u03/dbfs/<pdbid>/data/';
```

2. Grant access to the user to access the directory object.

   For more information on creating a directory object and setting access permissions on it, see CREATE DIRECTORY in *PL/SQL Packages and Types Reference*.

Do not access the OFS files by directly querying or modifying the DBFS tables. Do not use `dbfs_client` when the DBFS file system is mounted through OFS or else it could lead to metadata and data inconsistency. To access the OFS files, use the `UTL_FILE` package in addition to the procedures listed in the `DBMS_FS` package. See FS_EXISTS and LIST_FILES in *PL/SQL Packages and Types Reference*.

Accessing DBFS and OFS with an NFS Account

NFS is a widely used protocol to access any local file system across network. OFS makes use of this protocol and enables access to any DBFS file system that is mounted on the compute node.

NFS enables the compute node to be accessible across all nodes that are authorized to access the file system.

24.6.1 Accessing OFS with an NFS Account

You can export an OFS mount to a specified list of nodes and NFS mount it on them. This allows users to access the contents of an OFS mount point from a node where the database is not running. The NFS exports may not work in cloud environments due to security reasons, but you can use it in on-premise environments.

NFS v3 is a stateless protocol because of which it encapsulates each `readdir` request between `opendir` and `releasedir` calls. This may lead to poor performance when you want to list directories that have a large number of files. Therefore, OFS maintains a directory cache which persists across the `opendir` and `releasedir` calls. Do not use the `no_rbt_cache` mount option to avoid inconsistent directory cache listing and to utilize the benefits of directory cache.

24.6.2 Prerequisites to Access Storage Through NFS Server

Learn about the prerequisites to access storage through NFS server.

Following are the prerequisites:

- DBFS file system must be created before using OFS.
- You should be able to mount the file systems exported by the database.
- NFS server must be configured with `KERNEL` module.

> **Note:**
>
> The `KERNEL` module is supported through `FUSE` driver for Linux.

24.6.3 NFS Security

OFS uses the OS authentication model to authorize NFS client users. If the user accesses a local node (where the Oracle instance is running), the access to each file in the file system is controlled through **Unix Access Control List** set for each object. For NFS clients, you can configure Kerberos to control access.

On Linux, OFS uses FUSE to receive file system requests from the OS kernel or NFS client. This requires `user_allow_other` parameter to be set in `/etc/fuse.conf` configuration file if an OS user other than the `root` user and oracle user need to access the file system.

> **Note:**
>
> Users can also be configured with an Oracle password to log into Oracle client tools like `SQL* Plus` to execute SQL statements.

If the network is not secure, the customer is advised to setup Kerberos to authenticate the user using OS NFS.

> **Note:**
>
> - The Kerberos authentication is available from NFS version 4 onwards. If the OFS is exported via NFS version 3, then the authentication is performed using `AUTH_SYS`.
> - For local node, the authentication is performed using `AUTH_SYS` irrespective of how the OFS is exported (NFS version 3 or NFS version 4).

- About Kerberos
  Kerberos uses encryption technology, Key Distribution Center (KDC), and an arbitrator to perform secure authentication on open networks.
- Configuring Kerberos Server
  To configure a Kerberos Server in a Linux system:

24.6.3.1 About Kerberos

Kerberos uses encryption technology, Key Distribution Center (KDC), and an arbitrator to perform secure authentication on open networks.

Kerberos is the widely used security mechanism that provides all three flavors of security:

- Authentication
- Integrity check
- Privacy

Kerberos Infrastructure consists of Kerberos software, secured authentication servers, centralized account and password store, and systems configured to authenticate through the Kerberos protocol. The OS NFS server handles the complete authentication and integrity checks by using kerberos principal name as the user name. Once the authentication is performed, the requests passed to the Oracle kernel are handled based on the user name passed through the **VFS I/O** request.

24.6.3.2 Configuring Kerberos Server

To configure a Kerberos Server in a Linux system:

1. Install Kerberos software in the Linux system.
2. Check if the daemons are running using the following commands.

```
# /sbin/chkconfig krb5kdc on
# /sbin/chkconfig kadmin on
```

3. If the daemons are not running use the following commands to start the daemons manually:

```
# /etc/rc.d/init.d/krb5kdc start
# /etc/rc.d/init.d/kadmin start
```

4. Add user principal using the `kadmin.local` command.

   Example:

```
kadmin.local: addprinc <scott>
```

A

Comparing the LOB Interfaces

The tables in this section compare the eight LOB programmatic interfaces by listing their functions and methods used to operate on LOBs. The tables are split in two only to accommodate all eight interfaces.

APIs for BLOBs and CLOBs

Table A-1 APIs for BLOBs and CLOBs (PL/SQL, JDBC, OCI, OCCI)

PL/SQL: DBMS_LOB (dbmslob.sql)	JDBC (Java) interfaces java.sql.Clob and java.sql.Blob	OCI (C/ocip.h)	OCCI (C++/occiData.h) classes: Clob and Blob
NA	NA	OCILobLocatorIsInit()	isInitialized()
ISSECUREFILE	isSecureFile()	NA	NA
OPEN	open()	OCILobOpen()	Open()
ISOPEN	isOpen()	OCILobIsOpen()	isOpen()
CLOSE	close()	OCILobClose()	Close()
CREATETEMPORARY	createTemporary	OCILobCreateTemporary()	NA
FREETEMPORARY	freeTemporary	OCILobFreeTemporary()	NA
ISTEMPORARY	isTemporary	OCILobIsTemporary()	NA
GETLENGTH	length()	OCIGetLobLength2()	length()
GET_STORAGE_LIMIT	NA	OCILobGetStorageLimit()	NA
GETCHUNKSIZE	getChunkSize()	OCILobGetChunkSize()	getChunkSize()
READ	Blob: getBytes() getBinaryStream() Clob: getChars() getCharacterStream() getAsciiStream()	OciLobRead2() OCILobArrayRead()	read()
SUBSTR	getSubString	NA	NA
INSTR	position	NA	NA
NA	NA	OCILobCharSetId()	getCharSetId() (Clob only)
NA	NA	OCILobCharSetForm()	getCharSetForm (Clob only)
WRITE	Blob: setBytes() setBinaryStream() Clob: setString() setCharacterStream()	OCILobWrite2() OCILobArrayWrite()	write

Table A-1 (Cont.) APIs for BLOBs and CLOBs (PL/SQL, JDBC, OCI, OCCI)

PL/SQL: DBMS_LOB (dbmslob.sql)	JDBC (Java) interfaces java.sql.Clob and java.sql.Blob	OCI (C/ocip.h)	OCCI (C++/occiData.h) classes: Clob and Blob
WRITEAPPEND	use length() and then putString() or putBytes()	OCILobWriteAppend2( )	NA
ERASE	NA	OCILobErase2()	NA
TRIM	truncate()	OCILobTrim2()	trim
NA	equal	OCILobIsEqual()	Use operators == / !=
COMPARE	Use DBMS_LOB	NA	NA
APPEND	Use length() and then putString() or putBytes()	OCILobWriteAppend2( )	NA
COPY	Use read and write	OCILobCopy2()	copy()
Use operator :=	Use operator =	OCILobLocatorAssig n()	use operator =
CONVERTTOBLOB	NA	NA	NA
CONVERTTOCLOB	NA	NA	NA
NA	NA	NA	closeStream()
GETOPTIONS	NA	OCILobGetOptions()	getOptions()
SETOPTIONS	NA	OCILobSetOptions()	setOptions()
GETCONTENTTYPE	NA	OciLobGetContentTyp e()	getContentType()
SETCONTENTTYPE	NA	OciLobSetContentTyp e()	setContentType()
FRAGMENT_DELETE FRAGMENT_INSERT FRAGMENT_MOVE FRAGMENT_REPLACE	NA	NA	NA

Table A-2 APIs for BLOB and CLOB (PL/SQL, .NET, Pro*C/C++, Pro COBOL)

PL/SQL: DBMS_LOB (dbmslob.sql)	ODP.NET Classes: OracleClob and OracleBlob	Pro*C/C++ and Pro*COBOL
OPEN	BeginChunkWrite	OPEN
ISOPEN	IsInChunkWriteMode	DESCRIBE [ISOPEN]
CLOSE	EndChunkWrite	CLOSE
CREATETEMPORARY	Add()	CREATE TEMPORARY
FREETEMPORARY	Dispose() and Close()	FREE TEMPORARY
ISTEMPORARY	IsTemporary()	DESCRIBE [ISTEMPORARY]
GETLENGTH	Length()	DESCRIBE [LENGTH]
GETCHUNKSIZE	OptimumChunkSize()	DESCRIBE [CHUNKSIZE]
READ	Value Read	READ
INSTR	Search	NA

Table A-2 (Cont.) APIs for BLOB and CLOB (PL/SQL, .NET, Pro*C/C++, Pro COBOL)

PL/SQL: DBMS_LOB (dbmslob.sql)	ODP.NET Classes: OracleClob and OracleBlob	Pro*C/C++ and Pro*COBOL
WRITE	Write	WRITE
WRITEAPPEND	Append	WRITE APPEND
ERASE	Erase	ERASE
TRIM	SetLength	TRIM
NA	IsEqual	NA
COMPARE	Compare	NA
APPEND	Append	APPEND
COPY	CopyTo	COPY
Use operator :=	Clone	ASSIGN

APIs for BFILEs

Table A-3 APIs for BFILEs (PL/SQL, JDBC, OCI, OCCI)

PL/SQL: DBMS_LOB (dbmslob.sql)	JDBC (Java) interface oracle.jdbc.OracleBfile	OCI (C/ociap.h)	OCCI (C++/occiData.h) class: Bfile
FILEEXISTS	fileExists	OciLobFileExist()	fileExists()
FILEGETNAME	getDirAlias, getName	OCILobFileGetName()	getDirAlias()getFileName()
SQL BFILENAME operator	SQL BFILENAME operator	OCILobFileSetName()	setName()
OPEN	openFile	OCILobOpen()	open()
ISOPEN	isFileOpen()	OCILobIsOpen()	isOpen()
CLOSE	closeFile	OCILobClose()	close()
FILECLOSEALL	Use DBMS_LOB	OCILobFileCloseAll()	NA
GETLENGTH	length	OCILobGetLength2()	length()
READ	getBytes()getBinaryStream()	OCILobRead()OCILobArrayRead()	read
SUBSTR	getBytes	NA	NA
INSTR	position	NA	NA
Use operator :=	Use operator =	OCILobLocatorAssign()	Use operator =
LOADCLOBFROMFILE LOADBLOBFROMFILE	NA	OCILobLoadFromFile2()	Blob.copy() or Clob.copy()
COMPARE	NA	NA	NA
NA	equal	OCILobIsEqual()	Use operators ==/!=

Table A-4 APIs for BFILEs (PL/SQL, ODP.NET, Pro*C/C++ and Pro*COBOL)

PL/SQL: DBMS_LOB (dbmslob.sql)	ODP.NET Class: OracleBfile	Pro*C/C++ and Pro*COBOL
FILEEXISTS	FileExists	DESCRIBE [FILEEXISTS]
FILEGETNAME	DirectoryName, Filename	DESCRIBE [DIRECTORY, FILENAME]
SQL BFILENAME operator	DirectoryName, Filename	FILE SET
OPEN	OpenFile	OPEN
ISOPEN	IsOpen()	DESCRIBE [ISOPEN]
CLOSE	CloseFile	CLOSE
FILECLOSEALL	NA	FILE CLOSE ALL
GETLENGTH	Length	DESCRIBE [LENGTH]
READ	Value,Read	READ
SUBSTR	NA	NA
INSTR	Search	NA
Use operator :=	NA	NA
LOADCLOBFROMFILE LOADBLOBFROMFILE	NA	NA
COMPARE	Compare	NA
NA	IsEqual	NA