**HAPTER 14  
Large Objects**

In this chapter, you will do the following:

Image Learn about large objects (LOBs)

Image See files whose content will be used to populate example LOBs

Image Examine the differences between the different types of LOBs

Image Create tables containing LOBs

Image Use LOBs in SQL and PL/SQL

Image Examine the LONG and LONG RAW types

Image See the Oracle Database 10*g* and 11*g* enhancements to LOBs

**INTRODUCING LARGE OBJECTS (LOBS)**

Today’s websites demand more than just the storage and retrieval of text and numbers: they also require multimedia. Consequently, databases are now being called upon to store items like music and video. Prior to the release of Oracle Database 8, you had to store large blocks of character data using the LONG database type, and large blocks of binary data had to be stored using either the LONG RAW type or the shorter RAW type.

With the release Oracle Database 8, a new class of database types known as *large object*s, or LOBs for short, was introduced. LOBs may be used to store binary data, character data, and references to files. The binary data can contain images, music, video, documents, executables, and so on. LOBs can store up to 128 terabytes of data, depending on the database configuration.

**THE EXAMPLE FILES**

You’ll see the use of the following two files in this chapter:

Image **textContent.txt** A text file

Image **binaryContent.doc** A Microsoft Word file

Image

**NOTE**  
*These files are contained in the* sample\_files *directory, which is created when you extract the Zip file for this book. If you want to follow along with the examples, you should copy the* sample\_files*directory to the C partition on your database server. If you’re using Linux or Unix, you can copy the directory to one of your partitions*.

The file textContent.txt contains an extract from Shakespeare’s play *Macbeth*. The following text is the speech made by Macbeth shortly before he is killed:

To-morrow, and to-morrow, and to-morrow,  
Creeps in this petty pace from day to day,  
To the last syllable of recorded time;  
And all our yesterdays have lighted fools  
The way to a dusty death. Out, out, brief candle!  
Life’s but a walking shadow; a poor player,  
That struts and frets his hour upon the stage,  
And then is heard no more: it is a tale  
Told by an idiot, full of sound and fury,  
Signifying nothing.

The binaryContent.doc file is a Word document that contains the same text as textContent.txt. (A Word document is a binary file.) Although a Word document is used in the examples, you can use any binary file, for example, MP3, DivX, JPEG, MPEG, PDF, or EXE. I have tested the examples with all these types of files.

**LARGE OBJECT TYPES**

There are four LOB types:

Image **CLOB** The character LOB type, which is used to store character data.

Image **NCLOB** The National Character Set LOB type, which is used to store multiple byte character data (typically used for non-English characters). You can learn all about non-English character sets in the *Oracle Database Globalization Support Guide* published by Oracle Corporation.

Image **BLOB** The binary LOB type, which is used to store binary data.

Image **BFILE** The binary FILE type, which is used to store a pointer to a file. The file can be on a hard disk, CD, DVD, Blu-Ray disk, HD-DVD, or any other device that is accessible through the database server’s file system. The file itself is never stored in the database, only a pointer to the file.

Prior to Oracle Database 8 your only choice for storing large amounts of data was to use the LONG and LONG RAW types (you could also use the RAW type for storing binary data of less than 4 kilobytes in size). The LOB types have three advantages over these older types:

Image A LOB can store up to 128 terabytes of data. This is far more data than you can store in a LONG and LONG RAW column, which may only store up to 2 gigabytes of data.

Image A table can have multiple LOB columns, but a table can only have one LONG or LONG RAW column.

Image LOB data can be accessed in random order; LONG and LONG RAW data can be accessed only in sequential order.

A LOB consists of two parts:

Image **The LOB locator** A pointer that specifies the location of the LOB data

Image **The LOB data** The actual character or byte data stored in the LOB

Depending on the amount of data stored in a CLOB, NCLOB or BLOB column, the data will be stored either inside or outside of the table. If the data is less than 4 kilobytes, the data is stored in the same table; otherwise, the data is stored outside the table. With a BFILE column, only the locator is stored in the table—and the locator points to an external file stored in the file system.

**CREATING TABLES CONTAINING LARGE OBJECTS**

You’ll see the use of the following three tables in this section:

Image The clob\_content table, which contains a CLOB column named clob\_column

Image The blob\_content table, which contains a BLOB column named blob\_column

Image The bfile\_content table, which contains a BFILE column named bfile\_column

I’ve provided an SQL\*Plus script named lob\_schema.sql in the SQL directory. This script may be run using Oracle Database 8 and higher. The script creates a user named lob\_user with a password of lob\_password, and it creates the tables and PL/SQL code used in the first part of this chapter. After the script completes, you will be logged in as lob\_user.

The three tables are created using the following statements in the script:

CREATE TABLE clob\_content (

id INTEGER PRIMARY KEY,

clob\_column CLOB NOT NULL

);

CREATE TABLE blob\_content (

id INTEGER PRIMARY KEY,

blob\_column BLOB NOT NULL

);

CREATE TABLE bfile\_content (

id INTEGER PRIMARY KEY,

bfile\_column BFILE NOT NULL

);

**USING LARGE OBJECTS IN SQL**

In this section, you’ll learn how to use SQL to manipulate large objects. You’ll start by examining CLOB and BLOB objects and then move on to BFILE objects.

**Using CLOBs and BLOBs**

The following sections show how to populate CLOB and BLOB objects with data, retrieve the data, and then modify the data.

**Populating CLOBs and BLOBs with Data**

The following INSERT statements add two rows to the clob\_content table; notice the use of the TO\_CLOB() function to convert the text to a CLOB:

INSERT INTO clob\_content (

id, clob\_column

) VALUES (

1, TO\_CLOB('Creeps in this petty pace')

);

INSERT INTO clob\_content (

id, clob\_column

) VALUES (

2, TO\_CLOB(' from day to day')

);

The following INSERT statements add two rows to the blob\_content table; notice the use of the TO\_BLOB() function to convert the numbers to a BLOB (the first statement contains a binary number, and the second contains a hexadecimal number):

INSERT INTO blob\_content (

id, blob\_column

) VALUES (

1, TO\_BLOB('100111010101011111')

);

INSERT INTO blob\_content (

id, blob\_column

) VALUES (

2, TO\_BLOB('A0FFB71CF90DE')

);

**Retrieving Data from CLOBs**

The following query retrieves the rows from the clob\_content table:

**SELECT \***

**FROM clob\_content;**

ID

----------

CLOB\_COLUMN

-------------------------

1

Creeps in this petty pace

2

from day to day

The next query attempts to retrieve the row from the blob\_content table and fails:

**SELECT \***

**FROM blob\_content;**

SP2-0678: Column or attribute type can not be displayed by SQL\*Plus

This example fails because SQL\*Plus cannot display the binary data in a BLOB. You’ll learn how to retrieve the data from a BLOB later in the section "Using Large Objects in PL/SQL."

You can, however, get the non-BLOB columns from the table:

**SELECT id**

**FROM blob\_content;**

ID

----------

1

2

**Modifying the Data in CLOBs and BLOBs**

You should feel free to run the UPDATE and INSERT statements shown in this section. The following UPDATEstatements show how you modify the contents of a CLOB and a BLOB:

**UPDATE clob\_content**

**SET clob\_column = TO\_CLOB('What light through yonder window breaks')**

**WHERE id = 1;**

**UPDATE blob\_content**

**SET blob\_column = TO\_BLOB('1110011010101011111')**

**WHERE id = 1;**

You can also initialize the LOB locator, but not store actual data in the LOB. You do this using the EMPTY\_CLOB() function to store an empty CLOB, and EMPTY\_BLOB() to store an empty BLOB:

**INSERT INTO clob\_content(**

**id, clob\_column**

**) VALUES (**

**3, EMPTY\_CLOB()**

**);**

**INSERT INTO blob\_content(**

**id, blob\_column**

**) VALUES (**

**3, EMPTY\_BLOB()**

**);**

These statements initialize the LOB locator, but set the LOB data to empty.

You can also use EMPTY\_CLOB() and EMPTY\_BLOB() in UPDATE statements when you want to empty out the LOB data. For example:

**UPDATE clob\_content**

**SET clob\_column = EMPTY\_CLOB()**

**WHERE id = 1;**

**UPDATE blob\_content**

**SET blob\_column = EMPTY\_BLOB()**

**WHERE id = 1;**

If you ran any of the INSERT and UPDATE statements shown in this section, go ahead and roll back the changes so that your output matches mine in the rest of this chapter:

**ROLLBACK;**

**Using BFILEs**

A BFILE stores a pointer to a file that is accessible through the database server’s file system. The important point to remember is that these files are stored outside of the database. A BFILE can point to files located on any media: a hard disk, CD, DVD, Blu-Ray, HD-DVD, and so on.

Image

**NOTE**  
*A* BFILE *contains a pointer to an external file. The actual file itself is* never *stored in the database, only a pointer to that file is stored. The file must be accessible through the database server’s file system*.

**Creating a Directory Object**

Before you can store a pointer to a file in a BFILE, you must first create a directory object in the database. The directory object stores the directory in the file system where the files are located. You create a directory object using the CREATE DIRECTORY statement, and to run this statement you must have the CREATE ANY DIRECTORY database privilege.

The following example (contained in lob\_schema.sql) creates a directory object named SAMPLE\_FILES\_DIR for the file system directory C:\sample\_files:

CREATE DIRECTORY SAMPLE\_FILES\_DIR AS 'C:\sample\_files';

Image

**NOTE**  
*Windows uses the backslash character (\) in directories, while Linux and Unix use the forward slash character (/). Also, if your*sample\_files *directory is not stored in the C partition, then you need to specify the appropriate path in the previous example*.

When you create a directory object you must ensure that

Image The actual directory exists in the file system.

Image The user account in the operating system that was used to install the Oracle software has read permission on the directory and on any files that are to be pointed to by a BFILE in the database.

If you’re using Windows, you shouldn’t need to worry about the second point. The Oracle database software should have been installed using a user account that has administrator privileges, and such a user account has read permission on everything in the file system. If you’re using Linux or Unix, you’ll need to grant read access to the appropriate Oracle user account that owns the database (you do this using the chmod command).

**Populating a BFILE Column with a Pointer to a File**

Because a BFILE is just a pointer to an external file, populating a BFILE column is very simple. All you have to do is to use the Oracle database’s BFILENAME() function to populate a BFILE with a pointer to your external file. The BFILENAME() function accepts two parameters: the directory object’s name and the name of the file.

For example, the following INSERT adds a row to the bfile\_content table; notice that the BFILENAME()function is used to populate bfile\_column with a pointer to the textContent.txt file:

INSERT INTO bfile\_content (

id, bfile\_column

) VALUES (

1, BFILENAME(’SAMPLE\_FILES\_DIR', 'textContent.txt')

);

The next INSERT adds a row to the bfile\_content table; notice that the BFILENAME() function is used to populate bfile\_column with a pointer to the binaryContent.doc file:

INSERT INTO bfile\_content (

id, bfile\_column

) VALUES (

2, BFILENAME(’SAMPLE\_FILES\_DIR', 'binaryContent.doc')

);

The following query attempts to retrieve the rows from bfile\_content and fails because SQL\*Plus cannot display the content in a BFILE:

**SELECT \***

**FROM bfile\_content;**

SP2-0678: Column or attribute type can not be displayed by SQL\*Plus

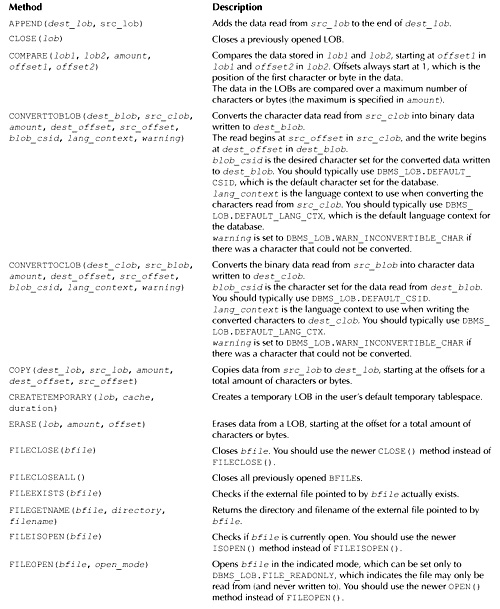
You may use PL/SQL to access the content in a BFILE or a BLOB, and you’ll learn how to do that next.

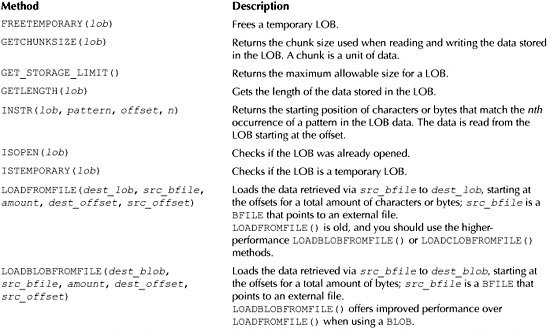
**USING LARGE OBJECTS IN PL/SQL**

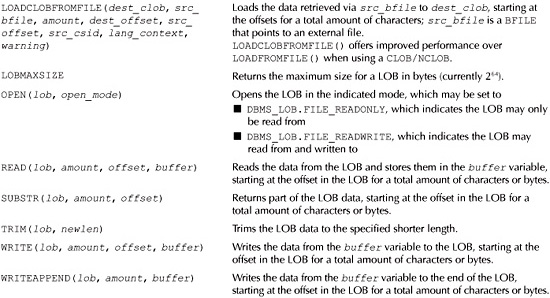
In this section, you’ll learn how to use LOBs in PL/SQL. You’ll start off by examining the methods in the DBMS\_LOB package, which comes with the database. Later, you’ll see plenty of PL/SQL programs that show how to use the DBMS\_LOB methods to read data in a LOB, copy data from one LOB to another, search data in a LOB, copy data from a file to a LOB, copy data from a LOB to a file, and much more.

[Table 14-1](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch14.html#table_14-1) summarizes the most commonly used methods in the DBMS\_LOB package.

In the following sections, you’ll see the details of some of the methods shown in the previous table. You can see all the DBMS\_LOB methods in the *Oracle Database PL/SQL Packages and Types Reference* manual published by Oracle Corporation.







**TABLE 14-1** *DBMS\_LOB Methods*

**APPEND()**

APPEND() adds the data in a source LOB to the end of a destination LOB. There are two versions of APPEND():

DBMS\_LOB.APPEND(

*dest\_lob* IN OUT NOCOPY BLOB,

*src\_lob* IN BLOB

);

DBMS\_LOB.APPEND(

*dest\_lob* IN OUT NOCOPY CLOB/NCLOB CHARACTER SET ANY\_CS,

*src\_lob* IN CLOB/NCLOB CHARACTER SET *dest\_lob*%CHARSET

);

where

Image *dest\_lob* is the destination LOB to which the data is appended.

Image *src\_lob* is the source LOB from which the data is read.

Image CHARACTER SET ANY\_CS means the data in *dest\_lob* can be any character set.

Image CHARACTER SET *dest\_lob*%CHARSET is the character set of dest\_lob.

The following table shows the exception thrown by APPEND().

Image

**CLOSE()**

CLOSE() closes a previously opened LOB. There are three versions of CLOSE():

DBMS\_LOB.CLOSE(

*lob* IN OUT NOCOPY BLOB

);

DBMS\_LOB.CLOSE(

*lob* IN OUT NOCOPY CLOB/NCLOB CHARACTER SET ANY\_CS

);

DBMS\_LOB.CLOSE(

*lob* IN OUT NOCOPY BFILE

);

where lob is the LOB to be closed.

**COMPARE()**

COMPARE() compares the data stored in two LOBs, starting at the offsets over a total amount of characters or bytes. There are three versions of COMPARE():

DBMS\_LOB.COMPARE(

*lob1* IN BLOB,

*lob2* IN BLOB,

*amount* IN INTEGER := 4294967295,

*offset1* IN INTEGER := 1,

*offset2* IN INTEGER := 1

) RETURN INTEGER;

DBMS\_LOB.COMPARE(

*lob1* IN CLOB/NCLOB CHARACTER SET ANY\_CS,

*lob2* IN CLOB/NCLOB CHARACTER SET lob\_1%CHARSET,

*amount* IN INTEGER := 4294967295,

*offset1* IN INTEGER := 1,

*offset2* IN INTEGER := 1

) RETURN INTEGER;

DBMS\_LOB.COMPARE(

*lob1* IN BFILE,

*lob2* IN BFILE,

*amount* IN INTEGER,

*offset1* IN INTEGER := 1,

*offset2* IN INTEGER := 1

) RETURN INTEGER;

where

Image *lob1* and *lob2* are the LOBs to compare.

Image *amount* is the maximum number of characters to read from a CLOB/NCLOB, or the maximum number of bytes to read from a BLOB/BFILE.

Image *offset1* and *offset2* are the offsets in characters or bytes in *lob1* and *lob2* to start the comparison (the offsets start at 1).

COMPARE() returns

Image 0 if the LOBs are identical.

Image 1 if the LOBs aren’t identical.

Image Null if

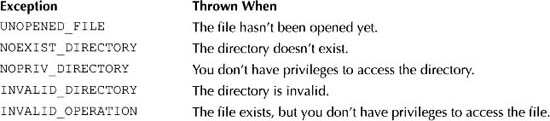
Image *amount* < 1

Image *amount* > LOBMAXSIZE (Note: LOBMAXSIZE is the maximum size of the LOB)

Image *offset1* or *offset2* < 1

Image *offset1* or *offset2* > LOBMAXSIZE

The following table shows the exceptions thrown by COMPARE().



**COPY()**

COPY() copies data from a source LOB to a destination LOB, starting at the offsets for a total amount of characters or bytes. There are two versions of COPY():

DBMS\_LOB.COPY(

*dest\_lob* IN OUT NOCOPY BLOB,

*src\_lob* IN BLOB,

*amount* IN INTEGER,

*dest\_offset* IN INTEGER := 1,

*src\_offset* IN INTEGER := 1

);

DBMS\_LOB.COPY(

*dest\_lob* IN OUT NOCOPY CLOB/NCLOB CHARACTER SET ANY\_CS,

*src\_lob* IN CLOB/NCLOB CHARACTER SET *dest\_lob*%CHARSET,

*amount* IN INTEGER,

*dest\_offset* IN INTEGER := 1,

*src\_offset* IN INTEGER := 1

);

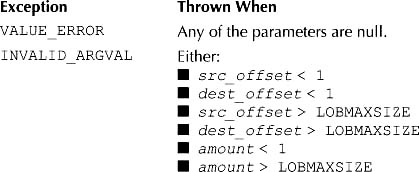
where

Image *dest\_lob* and *src\_lob* are the LOBs to write to and read from, respectively.

Image *amount* is the maximum number of characters to read from a CLOB/NCLOB, or the maximum number of bytes to read from a BLOB/BFILE.

Image *dest\_offset* and *src\_offset* are the offsets in characters or bytes in *dest\_lob* and *src\_lob* to start the copy (the offsets start at 1).

The following table shows the exceptions thrown by COPY().



**CREATETEMPORARY()**

CREATETEMPORARY() creates a temporary LOB in the user’s default temporary tablespace. There are two versions of CREATETEMPORARY():

DBMS\_LOB.CREATETEMPORARY(

*lob* IN OUT NOCOPY BLOB,

*cache* IN BOOLEAN,

*duration* IN PLS\_INTEGER := 10

);

DBMS\_LOB.CREATETEMPORARY (

*lob* IN OUT NOCOPY CLOB/NCLOB CHARACTER SET ANY\_CS,

*cache* IN BOOLEAN,

*duration* IN PLS\_INTEGER := 10

);

where

Image *lob* is the temporary LOB to create.

Image *cache* indicates whether the LOB should be read into the buffer cache (true for yes, false for no).

Image *duration* is a hint (can be set to SESSION, TRANSACTION, or CALL) as to whether the temporary LOB is removed at the end of the session, transaction, or call (the default is SESSION).

The following table shows the exception thrown by CREATETEMPORARY().

Image

**ERASE()**

ERASE() removes data from a LOB, starting at the offset for a total amount of characters or bytes. There are two versions of ERASE():

DBMS\_LOB.ERASE(

*lob* IN OUT NOCOPY BLOB,

*amount* IN OUT NOCOPY INTEGER,

*offset* IN INTEGER := 1

);

DBMS\_LOB.ERASE(

*lob* IN OUT NOCOPY CLOB/NCLOB CHARACTER SET ANY\_CS,

*amount* IN OUT NOCOPY INTEGER,

*offset* IN INTEGER := 1

);

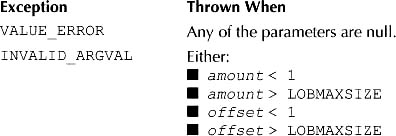
where

Image *lob* is the LOB to erase.

Image *amount* is the maximum number of characters to read from a CLOB/NCLOB, or the number of bytes to read from a BLOB.

Image *offset* is the offset in characters or bytes in *lob* to start the erasure (the offset starts at 1).

The following table shows the exceptions thrown by ERASE().



**FILECLOSE()**

FILECLOSE() closes a BFILE. You should use the newer CLOSE() procedure, as Oracle Corporation does not plan to extend the older FILECLOSE() procedure. I'm only including coverage of FILECLOSE() here so you can understand older programs.

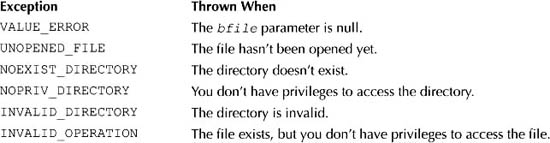
DBMS\_LOB.FILECLOSE(

*bfile* IN OUT NOCOPY BFILE

);

where bfile is the BFILE to close.

The following table shows the exceptions thrown by FILECLOSE().



**FILECLOSEALL()**

FILECLOSEALL() closes all BFILE objects.

DBMS\_LOB.FILECLOSEALL;

The following table shows the exception thrown by FILECLOSEALL().

Image

**FILEEXISTS()**

FILEEXISTS() checks if a file exists.

DBMS\_LOB.FILEEXISTS(

*bfile* IN BFILE

) RETURN INTEGER;

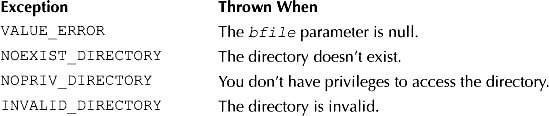
where bfile is a BFILE that points to an external file.

FILEEXISTS() returns

Image 0 if the file doesn’t exist.

Image 1 if the file exists.

The following table shows the exceptions thrown by FILEEXISTS().



**FILEGETNAME()**

FILEGETNAME() returns the directory and filename from a BFILE.

DBMS\_LOB.FILEGETNAME(

*bfile* IN BFILE,

*directory* OUT VARCHAR2,

*filename* OUT VARCHAR2

);

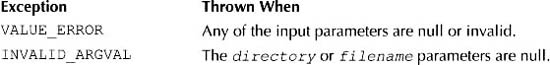
where

Image *bfile* is the pointer to the file.

Image *directory* is the directory where the file is stored.

Image *filename* is the name of the file.

The following table shows the exceptions thrown by FILEGETNAME().



**FILEISOPEN()**

FILEISOPEN() checks if a file is open. You should use the newer ISOPEN() procedure to check if a file is open in your own programs, as Oracle Corporation does not plan to extend the older FILEISOPEN() method. I'm including coverage of FILEISOPEN() here only so you can understand older programs.

DBMS\_LOB.FILEISOPEN(

*bfile* IN BFILE

) RETURN INTEGER;

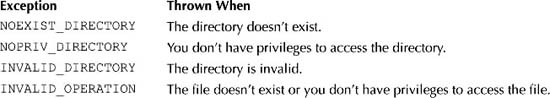
where *bfile* is the pointer to the file.

FILEISOPEN() returns

Image 0 if the file isn’t open.

Image 1 if the file is open.

The following table shows the exceptions thrown by FILEISOPEN().



**FILEOPEN()**

FILEOPEN() opens a file. You should use the newer OPEN() procedure to open a file in your own programs, as Oracle Corporation does not plan to extend the older FILEOPEN() procedure. I'm including coverage of FILEOPEN() here only so you can understand older programs.

DBMS\_LOB.FILEOPEN(

*bfile* IN OUT NOCOPY BFILE,

*open\_mode* IN BINARY\_INTEGER := DBMS\_LOB.FILE\_READONLY

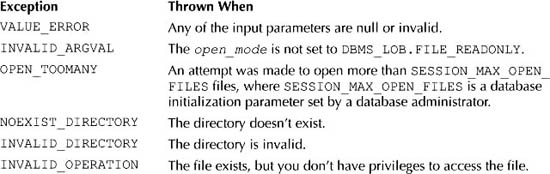
);

where

Image *bfile* is the pointer to the file.

Image *open\_mode* indicates the open mode; the only open mode is DBMS\_LOB.FILE\_READONLY, which indicates the file may be read from.

The following table shows the exceptions thrown by FILEOPEN().



**FREETEMPORARY()**

FREETEMPORARY() frees a temporary LOB from the default temporary tablespace of the user. There are two versions of FREETEMPORARY():

DBMS\_LOB.FREETEMPORARY (

*lob* IN OUT NOCOPY BLOB

);

DBMS\_LOB.FREETEMPORARY (

*lob* IN OUT NOCOPY CLOB/NCLOB CHARACTER SET ANY\_CS

);

where *lob* is the lob to be freed.

The following table shows the exception thrown by FREETEMPORARY().

Image

**GETCHUNKSIZE()**

GETCHUNKSIZE() returns the chunk size when reading and writing LOB data (a chunk is a unit of data). There are two versions of GETCHUNKSIZE():

DBMS\_LOB.GETCHUNKSIZE(

*lob* IN BLOB

) RETURN INTEGER;

DBMS\_LOB.GETCHUNKSIZE(

*lob* IN CLOB/NCLOB CHARACTER SET ANY\_CS

) RETURN INTEGER;

where lob is the LOB to get the chunk size for.

GETCHUNKSIZE() returns

Image The chunk size in bytes for a BLOB

Image The chunk size in characters for a CLOB/NCLOB

The following table shows the exception thrown by GETCHUNKSIZE().

Image

**GET\_STORAGE\_LIMIT()**

GET\_STORAGE\_LIMIT() returns the maximum allowable size for a LOB.

DBMS\_LOB.GET\_STORAGE\_LIMIT()

RETURN INTEGER;

**GETLENGTH()**

GETLENGTH() returns the length of the LOB data. There are three versions of GETLENGTH():

DBMS\_LOB.GETLENGTH(

*lob* IN BLOB

) RETURN INTEGER;

DBMS\_LOB.GETLENGTH(

*lob* IN CLOB/NCLOB CHARACTER SET ANY\_CS

) RETURN INTEGER;

DBMS\_LOB.GETLENGTH(

*bfile* IN BFILE

) RETURN INTEGER;

where

Image *lob* is the BLOB, CLOB, or NCLOB data to get the length of.

Image *bfile* is the BFILE data to get the length of.

GETLENGTH() returns

Image The length in bytes for a BLOB or BFILE

Image The length in characters for a CLOB or NCLOB

The following table shows the exception thrown by GETLENGTH().

Image

**INSTR()**

INSTR() returns the starting position of characters that match the *nth* occurrence of a pattern in the LOB data, starting at an offset. There are three versions of INSTR():

DBMS\_LOB.INSTR(

*lob* IN BLOB,

*pattern* IN RAW,

*offset* IN INTEGER := 1,

*n* IN INTEGER := 1

) RETURN INTEGER;

DBMS\_LOB.INSTR(

*lob* IN CLOB/NCLOB CHARACTER SET ANY\_CS,

*pattern* IN VARCHAR2 CHARACTER SET *lob*%CHARSET,

*offset* IN INTEGER := 1,

*n* IN INTEGER := 1

) RETURN INTEGER;

DBMS\_LOB.INSTR(

*bfile* IN BFILE,

*pattern* IN RAW,

*offset* IN INTEGER := 1,

*n* IN INTEGER := 1

) RETURN INTEGER;

where

Image *lob* is the BLOB, CLOB, or NCLOB to read from.

Image *bfile* is the BFILE to read from.

Image *pattern* is the pattern to search for in the LOB data; the pattern is a group of RAW bytes for a BLOB or BFILE, and a VARCHAR2 character string for a CLOB; the maximum size of the pattern is 16,383 bytes.

Image *offset* is the offset to start reading data from the LOB (the offset starts at 1).

Image *n* is the occurrence of the pattern to search the data for.

INSTR() returns

Image The offset of the start of the pattern (if found)

Image Zero if the pattern isn’t found

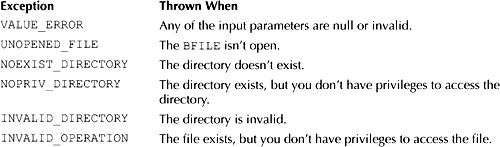
Image Null if

Image Any of the IN parameters are null or invalid

Image *offset* < 1 or *offset* > LOBMAXSIZE

Image *n* < 1 or *n* > LOBMAXSIZE

The following table shows the exceptions thrown by INSTR().



**ISOPEN()**

ISOPEN() checks if the LOB was already opened. There are three versions of ISOPEN():

DBMS\_LOB.ISOPEN(

*lob* IN BLOB

) RETURN INTEGER;

DBMS\_LOB.ISOPEN(

*lob* IN CLOB/NCLOB CHARACTER SET ANY\_CS

) RETURN INTEGER;

DBMS\_LOB.ISOPEN(

*bfile* IN BFILE

) RETURN INTEGER;

where

Image *lob* is the BLOB, CLOB, or NCLOB to check.

Image *bfile* is the BFILE to check.

ISOPEN() returns

Image 0 if the LOB isn’t open.

Image 1 if the LOB is open.

The following table shows the exception thrown by ISOPEN().

Image

**ISTEMPORARY()**

ISTEMPORARY() checks if the LOB is a temporary LOB. There are two versions of ISTEMPORARY():

DBMS\_LOB.ISTEMPORARY(

*lob* IN BLOB

) RETURN INTEGER;

DBMS\_LOB.ISTEMPORARY (

*lob* IN CLOB/NCLOB CHARACTER SET ANY\_CS

) RETURN INTEGER;

where

Image *lob* is the LOB to check.

ISTEMPORARY() returns

Image 0 if the LOB isn’t temporary.

Image 1 if the LOB is temporary.

The following table shows the exception thrown by ISTEMPORARY(). **Exception Thrown When**

Image

**LOADFROMFILE()**

LOADFROMFILE() loads data retrieved via a BFILE into a CLOB, NCLOB, or BLOB, starting at the offsets for a total amount of characters or bytes. You should use the higher-performance LOADCLOBFROMFILE() or LOADBLOBFROMFILE() procedures in your own programs, and I'm including coverage of LOADFROMFILE() here only so you can understand older programs.

There are two versions of LOADFROMFILE():

DBMS\_LOB.LOADFROMFILE(

*dest\_lob* IN OUT NOCOPY BLOB,

*src\_bfile* IN BFILE,

*amount* IN INTEGER,

*dest\_offset* IN INTEGER := 1,

*src\_offset* IN INTEGER := 1

);

DBMS\_LOB.LOADFROMFILE(

*dest\_lob* IN OUT NOCOPY CLOB/NCLOB CHARACTER SET ANY\_CS,

*src\_bfile* IN BFILE,

*amount* IN INTEGER,

*dest\_offset* IN INTEGER := 1,

*src\_offset* IN INTEGER := 1

);

where

Image *dest\_lob* is the LOB into which the data is to be written.

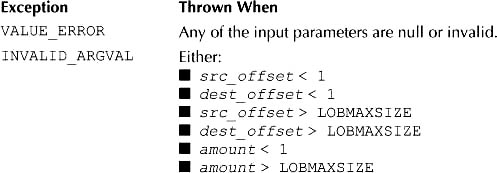
Image *src\_bfile* is the pointer to the file from which the data is to be read.

Image *amount* is the maximum number of bytes or characters to read from src\_bfile.

Image *dest\_offset* is the offset in bytes or characters in *dest\_lob* to start writing data (the offset starts at 1).

Image *src\_offset* is the offset in bytes in *src\_bfile* to start reading data (the offset starts at 1).

The following table shows the exceptions thrown by LOADFROMFILE(). **Exception Thrown When**



**LOADBLOBFROMFILE()**

LOADBLOBFROMFILE() loads data retrieved via a BFILE into a BLOB. LOADBLOBFROMFILE() offers improved performance over the LOADFROMFILE() method when using a BLOB.

DBMS\_LOB.LOADBLOBFROMFILE(

*dest\_blob* IN OUT NOCOPY BLOB,

*src\_bfile* IN BFILE,

*amount* IN INTEGER,

*dest\_offset* IN OUT INTEGER := 1,

*src\_offset* IN OUT INTEGER := 1

);

where

Image *dest\_blob* is the BLOB into which the data is to be written.

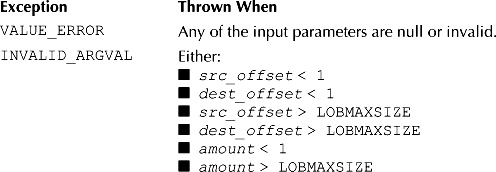
Image *src\_bfile* is the pointer to the file from which the data is to be read.

Image *amount* is the maximum number of bytes to read from src\_bfile.

Image *dest\_offset* is the offset in bytes in *dest\_lob* to start writing data (the offset starts at 1).

Image *src\_offset* is the offset in bytes in *src\_bfile* to start reading data (the offset starts at 1).

The following table shows the exceptions thrown by LOADBLOBFROMFILE().



**LOADCLOBFROMFILE()**

LOADCLOBFROMFILE() loads data retrieved via a BFILE into a CLOB/NCLOB. LOADCLOBFROMFILE()offers improved performance over the LOADFROMFILE() method when using a CLOB/NCLOB. LOADCLOBFROMFILE() also automatically converts binary data to character data.

DBMS\_LOB.LOADCLOBFROMFILE(

*dest\_clob* IN OUT NOCOPY CLOB/NCLOB,

*src\_bfile* IN BFILE,

*amount* IN INTEGER,

*dest\_offset* IN OUT INTEGER,

*src\_offset* IN OUT INTEGER,

*src\_csid* IN NUMBER,

*lang\_context* IN OUT INTEGER,

*warning* OUT INTEGER

);

where

Image *dest\_blob* is the CLOB/NCLOB into which the data is to be written.

Image *src\_bfile* is the pointer to the file from which the data is to be read.

Image *amount* is the maximum number of characters to read from *src\_bfile*.

Image *dest\_offset* is the offset in characters in *dest\_lob* to start writing data (the offset starts at 1).

Image *src\_offset* is the offset in characters in *src\_bfile* to start reading data (the offset starts at 1).

Image *src\_csid* is the character set of *src\_bfile* (you should typically use DBMS\_LOB.DEFAULT\_CSID, which is the default character set for the database).

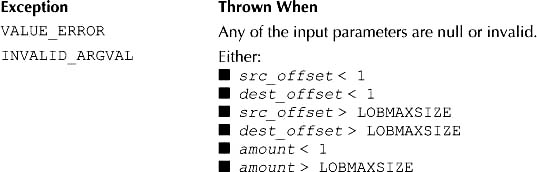
Image *lang\_context* is the language context to use for the load (you should typically use DBMS\_LOB.DEFAULT\_LANG\_CTX, which is the default language context for the database).

Image *warning* is a warning message that contains information if there was a problem with the load; a common problem is that a character in *src\_bfile* cannot be converted to a character in *dest\_lob* (in which case, *warning* is set to DBMS\_LOB.WARN\_INCONVERTIBLE\_CHAR).

Image

**NOTE**  
*You can learn all about character sets, contexts, and how to convert characters from one language to another in the* Oracle Database Globalization Support Guide *published by Oracle Corporation*.

The following table shows the exceptions thrown by LOADCLOBFROMFILE().



**OPEN()**

OPEN() opens a LOB. There are three versions of OPEN():

DBMS\_LOB.OPEN(

*lob* IN OUT NOCOPY BLOB,

*open\_mode* IN BINARY\_INTEGER

);

DBMS\_LOB.OPEN(

*lob* IN OUT NOCOPY CLOB/NCLOB CHARACTER SET ANY\_CS,

*open\_mode* IN BINARY\_INTEGER

);

DBMS\_LOB.OPEN(

*bfile* IN OUT NOCOPY BFILE,

*open\_mode* IN BINARY\_INTEGER := DBMS\_LOB.FILE\_READONLY

);

where

Image *lob* is the LOB to open.

Image *bfile* is the pointer to the file to open.

Image *open\_mode* indicates the open mode; the default is DBMS\_LOB.FILE\_READONLY, which indicates the LOB may only be read from; DBMS\_LOB.FILE\_READWRITE indicates the LOB may be read from and written to.

The following table shows the exception thrown by OPEN().

Image

**READ()**

READ() reads data into a buffer from a LOB. There are three versions of READ():

DBMS\_LOB.READ(

*lob* IN BLOB,

*amount* IN OUT NOCOPY BINARY\_INTEGER,

*offset* IN INTEGER,

*buffer* OUT RAW

);

DBMS\_LOB.READ(

*lob* IN CLOB/NCLOB CHARACTER SET ANY\_CS,

*amount* IN OUT NOCOPY BINARY\_INTEGER,

*offset* IN INTEGER,

*buffer* OUT VARCHAR2 CHARACTER SET *lob*%CHARSET

);

DBMS\_LOB.READ(

*bfile* IN BFILE,

*amount* IN OUT NOCOPY BINARY\_INTEGER,

*offset* IN INTEGER,

*buffer* OUT RAW

);

where

Image *lob* is the CLOB, NCLOB, or BLOB to read from.

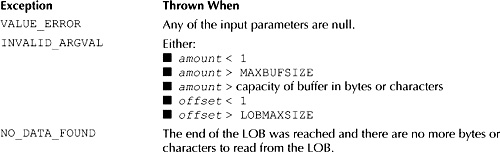
Image *bfile* is the BFILE to read from.

Image *amount* is the maximum number of characters to read from a CLOB/NCLOB, or the maximum number of bytes to read from a BLOB/BFILE.

Image *offset* is the offset to start reading (the offset starts at 1).

Image *buffer* is the variable where the data read from the LOB is to be stored.

The following table shows the exceptions thrown by READ().



**SUBSTR()**

SUBSTR() returns part of the LOB data, starting at the offset for a total amount of characters or bytes. There are three versions of SUBSTR():

DBMS\_LOB.SUBSTR(

*lob* IN BLOB,

*amount* IN INTEGER := 32767,

*offset* IN INTEGER := 1

) RETURN RAW;

DBMS\_LOB.SUBSTR (

*lob* IN CLOB/NCLOB CHARACTER SET ANY\_CS,

*amount* IN INTEGER := 32767,

*offset* IN INTEGER := 1

) RETURN VARCHAR2 CHARACTER SET *lob*%CHARSET;

DBMS\_LOB.SUBSTR (

*bfile* IN BFILE,

*amount* IN INTEGER := 32767,

*offset* IN INTEGER := 1

) RETURN RAW;

where

Image *lob* is the BLOB, CLOB, or NCLOB to read from.

Image *bfile* is the pointer to the file to read from.

Image *amount* is the maximum number of characters read from a CLOB/NCLOB, or the maximum number of bytes to read from a BLOB/BFILE.

Image *offset* is the offset to start reading data from the LOB (the offset starts at 1).

SUBSTR() returns

Image RAW data when reading from a BLOB/BFILE.

Image VARCHAR2 data when reading from a CLOB/NCLOB.

Image Null if

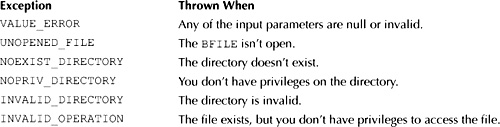
Image *amount* < 1

Image *amount* > 32767

Image *offset* < 1

Image *offset* > LOBMAXSIZE

The following table shows the exceptions thrown by SUBSTR().



**TRIM()**

TRIM() trims the LOB data to the specified shorter length. There are two versions of TRIM():

DBMS\_LOB.TRIM(

*lob* IN OUT NOCOPY BLOB,

*newlen* IN INTEGER

);

DBMS\_LOB.TRIM(

*lob* IN OUT NOCOPY CLOB/NCLOB CHARACTER SET ANY\_CS,

*newlen* IN INTEGER

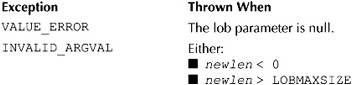
);

where

Image *lob* is the BLOB, CLOB, or NCLOB to trim.

Image *newlen* is the new length (in bytes for a BLOB, or characters for a CLOB/NCLOB).

The following table shows the exceptions thrown by TRIM().



**WRITE()**

WRITE() writes data from a buffer to a LOB. There are two versions of WRITE():

DBMS\_LOB.WRITE(

*lob* IN OUT NOCOPY BLOB,

*amount* IN BINARY\_INTEGER,

*offset* IN INTEGER,

*buffer* IN RAW

);

DBMS\_LOB.WRITE(

*lob* IN OUT NOCOPY CLOB/NCLOB CHARACTER SET ANY\_CS,

*amount* IN BINARY\_INTEGER,

*offset* IN INTEGER,

*buffer* IN VARCHAR2 CHARACTER SET *lob*%CHARSET

);

where

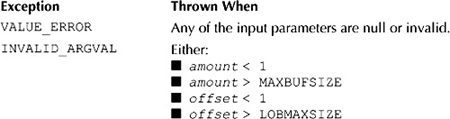
Image *lob* is the LOB to write to.

Image *amount* is the maximum number of characters to write to a CLOB/NCLOB, or the maximum number of bytes to write to a BLOB.

Image *offset* is the offset to start writing data to the LOB (the offset starts at 1).

Image *buffer* is the variable that contains the data to be written to the LOB.

The following table shows the exceptions thrown by WRITE().



**WRITEAPPEND()**

WRITEAPPEND() writes data from the buffer to the end of a LOB, starting at the offset for a total amount of characters or bytes. There are two versions of WRITEAPPEND():

DBMS\_LOB.WRITEAPPEND(

*lob* IN OUT NOCOPY BLOB,

*amount* IN BINARY\_INTEGER,

*buffer* IN RAW

);

DBMS\_LOB.WRITEAPPEND(

*lob* IN OUT NOCOPY CLOB/NCLOB CHARACTER SET ANY\_CS,

*amount* IN BINARY\_INTEGER,

*buffer* IN VARCHAR2 CHARACTER SET *lob*%CHARSET

);

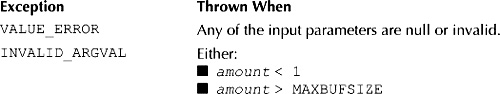
where

Image *lob* is the BLOB, CLOB, or NCLOB to write to.

Image *amount* is the maximum number of characters to write to a CLOB/NCLOB, or the maximum number of bytes to write to a BLOB.

Image *buffer* is the variable that contains the data to be written to the LOB.

The following table shows the exceptions thrown by WRITEAPPEND().



**Example PL/SQL Procedures**

In this section, you’ll see example PL/SQL procedures that use the various methods described in the previous sections. The example procedures are created when you run the lob\_schema.sql script.

**Retrieving a LOB Locator**

The following get\_clob\_locator() procedure gets a LOB locator from the clob\_content table; get\_clob\_locator() performs the following tasks:

Image Accepts an IN OUT parameter named p\_clob of type CLOB; p\_clob is set to a LOB locator inside the procedure. Because p\_clob is IN OUT, the value is passed out of the procedure.

Image Accepts an IN parameter named p\_id of type INTEGER, which specifies the id of a row to retrieve from the clob\_content table.

Image Selects clob\_column from the clob\_content table into p\_clob; this stores the LOB locator of clob\_column in p\_clob.

CREATE PROCEDURE get\_clob\_locator(

p\_clob IN OUT CLOB,

p\_id IN INTEGER

) AS

BEGIN

-- get the LOB locator and store it in p\_clob

SELECT clob\_column

INTO p\_clob

FROM clob\_content

WHERE id = p\_id;

END get\_clob\_locator;

/

The following get\_blob\_locator() procedure does the same thing as the previous procedure, except it gets the locator for a BLOB from the blob\_content table:

CREATE PROCEDURE get\_blob\_locator(

p\_blob IN OUT BLOB,

p\_id IN INTEGER

) AS

BEGIN

-- get the LOB locator and store it in p\_blob

SELECT blob\_column

INTO p\_blob

FROM blob\_content

WHERE id = p\_id;

END get\_blob\_locator;

/

These two procedures are used in the code shown in the following sections.

**Reading Data from CLOBs and BLOBs**

The following read\_clob\_example() procedure reads the data from a CLOB and displays the data on the screen; read\_clob\_example() performs the following tasks:

Image Calls get\_clob\_locator() to get a locator and stores it in v\_clob

Image Uses READ() to read the contents of v\_clob into a VARCHAR2 variable named v\_char\_buffer

Image Displays the contents of v\_char\_buffer on the screen

CREATE PROCEDURE read\_clob\_example(

p\_id IN INTEGER

) AS

v\_clob CLOB;

v\_offset INTEGER := 1;

v\_amount INTEGER := 50;

v\_char\_buffer VARCHAR2(50);

BEGIN

-- get the LOB locator and store it in v\_clob

get\_clob\_locator(v\_clob, p\_id);

-- read the contents of v\_clob into v\_char\_buffer, starting at

-- the v\_offset position and read a total of v\_amount characters

DBMS\_LOB.READ(v\_clob, v\_amount, v\_offset, v\_char\_buffer);

-- display the contents of v\_char\_buffer

DBMS\_OUTPUT.PUT\_LINE('v\_char\_buffer = ' || v\_char\_buffer);

DBMS\_OUTPUT.PUT\_LINE('v\_amount = ' || v\_amount);

END read\_clob\_example;

/

The following example turns the server output on and calls read\_clob\_example():

**SET SERVEROUTPUT ON**

**CALL read\_clob\_example(1);**

v\_char\_buffer = Creeps in this petty pace

v\_amount = 25

The following read\_blob\_example() procedure reads the data from a BLOB; read\_blob\_example()performs the following tasks:

Image Calls get\_blob\_locator() to get the locator and stores it in v\_blob

Image Calls READ() to read the contents of v\_blob into a RAW variable named v\_binary\_buffer

Image Displays the contents of v\_binary\_buffer on the screen

CREATE PROCEDURE read\_blob\_example(

p\_id IN INTEGER

) AS

v\_blob BLOB;

v\_offset INTEGER := 1;

v\_amount INTEGER := 25;

v\_binary\_buffer RAW(25);

BEGIN

-- get the LOB locator and store it in v\_blob

get\_blob\_locator(v\_blob, p\_id);

-- read the contents of v\_blob into v\_binary\_buffer, starting at

-- the v\_offset position and read a total of v\_amount bytes

DBMS\_LOB.READ(v\_blob, v\_amount, v\_offset, v\_binary\_buffer);

-- display the contents of v\_binary\_buffer

DBMS\_OUTPUT.PUT\_LINE('v\_binary\_buffer = ' || v\_binary\_buffer);

DBMS\_OUTPUT.PUT\_LINE('v\_amount = ' || v\_amount);

END read\_blob\_example;

/

The following example calls read\_blob\_example():

**CALL read\_blob\_example(1);**

v\_binary\_buffer = 100111010101011111

v\_amount = 9

**Writing to a CLOB**

The following write\_example() procedure writes a string in v\_char\_buffer to v\_clob using WRITE(); notice that the SELECT statement in the procedure uses the FOR UPDATE clause, which is used because the CLOB is written to using WRITE():

CREATE PROCEDURE write\_example(

p\_id IN INTEGER

) AS

v\_clob CLOB;

v\_offset INTEGER := 7;

v\_amount INTEGER := 6;

v\_char\_buffer VARCHAR2(10) := 'pretty';

BEGIN

-- get the LOB locator into v\_clob for update (for update

-- because the LOB is written to using WRITE() later)

SELECT clob\_column

INTO v\_clob

FROM clob\_content

WHERE id = p\_id

FOR UPDATE;

-- read and display the contents of the CLOB

read\_clob\_example(p\_id);

-- write the characters in v\_char\_buffer to v\_clob, starting

-- at the v\_offset position and write a total of v\_amount characters

DBMS\_LOB.WRITE(v\_clob, v\_amount, v\_offset, v\_char\_buffer);

-- read and display the contents of the CLOB

-- and then rollback the write

read\_clob\_example(p\_id);

ROLLBACK;

END write\_example;

/

The following example calls write\_example():

**CALL write\_example(1);**

v\_char\_buffer = Creeps in this petty pace

v\_amount = 25

v\_char\_buffer = Creepsprettyis petty pace

v\_amount = 25

**Appending Data to a CLOB**

The following append\_example() procedure uses APPEND() to copy the data from v\_src\_clob to the end of v\_dest\_clob:

CREATE PROCEDURE append\_example AS

v\_src\_clob CLOB;

v\_dest\_clob CLOB;

BEGIN

-- get the LOB locator for the CLOB in row #2 of

-- the clob\_content table into v\_src\_clob

get\_clob\_locator(v\_src\_clob, 2);

-- get the LOB locator for the CLOB in row #1 of

-- the clob\_content table into v\_dest\_clob for update

-- (for update because the CLOB will be added to using

-- APPEND() later)

SELECT clob\_column

INTO v\_dest\_clob

FROM clob\_content

WHERE id = 1

FOR UPDATE;

-- read and display the contents of CLOB #1

read\_clob\_example(1);

-- use APPEND() to copy the contents of v\_src\_clob to v\_dest\_clob

DBMS\_LOB.APPEND(v\_dest\_clob, v\_src\_clob);

-- read and display the contents of CLOB #1

-- and then rollback the change

read\_clob\_example(1);

ROLLBACK;

END append\_example;

/

The following example calls append\_example():

**CALL append\_example();**

v\_char\_buffer = Creeps in this petty pace

v\_amount = 25

v\_char\_buffer = Creeps in this petty pace from day to day

v\_amount = 41

**Comparing the Data in Two CLOBs**

The following compare\_example() procedure compares the data in v\_clob1 and v\_clob2 using COMPARE():

CREATE PROCEDURE compare\_example AS

v\_clob1 CLOB;

v\_clob2 CLOB;

v\_return INTEGER;

BEGIN

-- get the LOB locators

get\_clob\_locator(v\_clob1, 1);

get\_clob\_locator(v\_clob2, 2);

-- compare v\_clob1 with v\_clob2 (COMPARE() returns 1

-- because the contents of v\_clob1 and v\_clob2 are different)

DBMS\_OUTPUT.PUT\_LINE('Comparing v\_clob1 with v\_clob2');

v\_return := DBMS\_LOB.COMPARE(v\_clob1, v\_clob2);

DBMS\_OUTPUT.PUT\_LINE('v\_return = ' || v\_return);

-- compare v\_clob1 with v\_clob1 (COMPARE() returns 0

-- because the contents are the same)

DBMS\_OUTPUT.PUT\_LINE('Comparing v\_clob1 with v\_clob1');

v\_return := DBMS\_LOB.COMPARE(v\_clob1, v\_clob1);

DBMS\_OUTPUT.PUT\_LINE('v\_return = ' || v\_return);

END compare\_example;

/

The following example calls compare\_example():

**CALL compare\_example();**

Comparing v\_clob1 with v\_clob2

v\_return = 1

Comparing v\_clob1 with v\_clob1

v\_return = 0

Notice that v\_return is 1 when comparing v\_clob1 with v\_clob2, which indicates the LOB data is different; v\_return is 0 when comparing v\_clob1 with v\_clob1, which indicates the LOB data is the same.

**Copying Data from One CLOB to Another**

The following copy\_example() procedure copies some characters from v\_src\_clob to v\_dest\_clobusing COPY():

CREATE PROCEDURE copy\_example AS

v\_src\_clob CLOB;

v\_dest\_clob CLOB;

v\_src\_offset INTEGER := 1;

v\_dest\_offset INTEGER := 7;

v\_amount INTEGER := 5;

BEGIN

-- get the LOB locator for the CLOB in row #2 of

-- the clob\_content table into v\_dest\_clob

get\_clob\_locator(v\_src\_clob, 2);

-- get the LOB locator for the CLOB in row #1 of

-- the clob\_content table into v\_dest\_clob for update

-- (for update because the CLOB will be added to using

-- COPY() later)

SELECT clob\_column

INTO v\_dest\_clob

FROM clob\_content

WHERE id = 1

FOR UPDATE;

-- read and display the contents of CLOB #1

read\_clob\_example(1);

-- copy characters to v\_dest\_clob from v\_src\_clob using COPY(),

-- starting at the offsets specified by v\_dest\_offset and

-- v\_src\_offset for a total of v\_amount characters

DBMS\_LOB.COPY(

v\_dest\_clob, v\_src\_clob,

v\_amount, v\_dest\_offset, v\_src\_offset

);

-- read and display the contents of CLOB #1

-- and then rollback the change

read\_clob\_example(1);

ROLLBACK;

END copy\_example;

/

The following example calls copy\_example():

**CALL copy\_example();**

v\_char\_buffer = Creeps in this petty pace

v\_amount = 25

v\_char\_buffer = Creeps fromhis petty pace

v\_amount = 25

**Using Temporary CLOBs**

The following temporary\_lob\_example() procedure illustrates the use of a temporary CLOB:

CREATE PROCEDURE temporary\_lob\_example AS

v\_clob CLOB;

v\_amount INTEGER;

v\_offset INTEGER := 1;

v\_char\_buffer VARCHAR2(17) := 'Juliet is the sun';

BEGIN

-- use CREATETEMPORARY() to create a temporary CLOB named v\_clob

DBMS\_LOB.CREATETEMPORARY(v\_clob, TRUE);

-- use WRITE() to write the contents of v\_char\_buffer to v\_clob

v\_amount := LENGTH(v\_char\_buffer);

DBMS\_LOB.WRITE(v\_clob, v\_amount, v\_offset, v\_char\_buffer);

-- use ISTEMPORARY() to check if v\_clob is temporary

IF (DBMS\_LOB.ISTEMPORARY(v\_clob) = 1) THEN

DBMS\_OUTPUT.PUT\_LINE('v\_clob is temporary');

END IF;

-- use READ() to read the contents of v\_clob into v\_char\_buffer

DBMS\_LOB.READ(

v\_clob, v\_amount, v\_offset, v\_char\_buffer

);

DBMS\_OUTPUT.PUT\_LINE('v\_char\_buffer = ' || v\_char\_buffer);

-- use FREETEMPORARY() to free v\_clob

DBMS\_LOB.FREETEMPORARY(v\_clob);

END temporary\_lob\_example;

/

The following example calls temporary\_lob\_example():

**CALL temporary\_lob\_example();**

v\_clob is temporary

v\_char\_buffer = Juliet is the sun

**Erasing Data from a CLOB**

The following erase\_example() procedure erases part of a CLOB using ERASE():

CREATE PROCEDURE erase\_example IS

v\_clob CLOB;

v\_offset INTEGER := 2;

v\_amount INTEGER := 5;

BEGIN

-- get the LOB locator for the CLOB in row #1 of

-- the clob\_content table into v\_dest\_clob for update

-- (for update because the CLOB will be erased using

-- ERASE() later)

SELECT clob\_column

INTO v\_clob

FROM clob\_content

WHERE id = 1

FOR UPDATE;

-- read and display the contents of CLOB #1

read\_clob\_example(1);

-- use ERASE() to erase a total of v\_amount characters

-- from v\_clob, starting at v\_offset

DBMS\_LOB.ERASE(v\_clob, v\_amount, v\_offset);

-- read and display the contents of CLOB #1

-- and then rollback the change

read\_clob\_example(1);

ROLLBACK;

END erase\_example;

/

The following example calls erase\_example():

**CALL erase\_example();**

v\_char\_buffer = Creeps in this petty pace

v\_amount = 25

v\_char\_buffer = C in this petty pace

v\_amount = 25

**Searching the Data in a CLOB**

The following instr\_example() procedure uses INSTR() to search the character data stored in a CLOB:

CREATE PROCEDURE instr\_example AS

v\_clob CLOB;

v\_char\_buffer VARCHAR2(50) := 'It is the east and Juliet is the sun';

v\_pattern VARCHAR2(5);

v\_offset INTEGER := 1;

v\_amount INTEGER;

v\_occurrence INTEGER;

v\_return INTEGER;

BEGIN

-- use CREATETEMPORARY() to create a temporary CLOB named v\_clob

DBMS\_LOB.CREATETEMPORARY(v\_clob, TRUE);

-- use WRITE() to write the contents of v\_char\_buffer to v\_clob

v\_amount := LENGTH(v\_char\_buffer);

DBMS\_LOB.WRITE(v\_clob, v\_amount, v\_offset, v\_char\_buffer);

-- use READ() to read the contents of v\_clob into v\_char\_buffer

DBMS\_LOB.READ(v\_clob, v\_amount, v\_offset, v\_char\_buffer);

DBMS\_OUTPUT.PUT\_LINE('v\_char\_buffer = ' || v\_char\_buffer);

-- use INSTR() to search v\_clob for the second occurrence of is,

-- and INSTR() returns 27

DBMS\_OUTPUT.PUT\_LINE(’Searching for second "is"');

v\_pattern := "is";

v\_occurrence := 2;

v\_return := DBMS\_LOB.INSTR(v\_clob, v\_pattern, v\_offset, v\_occurrence);

DBMS\_OUTPUT.PUT\_LINE('v\_return = ' || v\_return);

-- use INSTR() to search v\_clob for the first occurrence of Moon,

-- and INSTR() returns 0 because Moon doesn’t appear in v\_clob

DBMS\_OUTPUT.PUT\_LINE(’Searching for "Moon"');

v\_pattern := "Moon";

v\_occurrence := 1;

v\_return := DBMS\_LOB.INSTR(v\_clob, v\_pattern, v\_offset, v\_occurrence);

DBMS\_OUTPUT.PUT\_LINE('v\_return = ' || v\_return);

-- use FREETEMPORARY() to free v\_clob

DBMS\_LOB.FREETEMPORARY(v\_clob);

END instr\_example;

/

The following example calls instr\_example():

**CALL instr\_example();**

v\_char\_buffer = It is the east and Juliet is the sun

Searching for second 'is'

v\_return = 27

Searching for 'Moon'

v\_return = 0

**Copying Data from a File into a CLOB and a BLOB**

The following copy\_file\_data\_to\_clob() procedure shows how to read text from a file and store it in a CLOB:

CREATE PROCEDURE copy\_file\_data\_to\_clob(

p\_clob\_id INTEGER,

p\_directory VARCHAR2,

p\_file\_name VARCHAR2

) AS

v\_file UTL\_FILE.FILE\_TYPE;

v\_chars\_read INTEGER;

v\_dest\_clob CLOB;

v\_amount INTEGER := 32767;

v\_char\_buffer VARCHAR2(32767);

BEGIN

-- insert an empty CLOB

INSERT INTO clob\_content(

id, clob\_column

) VALUES (

p\_clob\_id, EMPTY\_CLOB()

);

-- get the LOB locator of the CLOB

SELECT clob\_column

INTO v\_dest\_clob

FROM clob\_content

WHERE id = p\_clob\_id

FOR UPDATE;

-- open the file for reading of text (up to v\_amount characters per line)

v\_file := UTL\_FILE.FOPEN(p\_directory, p\_file\_name, 'r', v\_amount);

-- copy the data from the file into v\_dest\_clob one line at a time

LOOP

BEGIN

-- read a line from the file into v\_char\_buffer;

-- GET\_LINE() does not copy the newline character into

-- v\_char\_buffer

UTL\_FILE.GET\_LINE(v\_file, v\_char\_buffer);

v\_chars\_read := LENGTH(v\_char\_buffer);

-- append the line to v\_dest\_clob

DBMS\_LOB.WRITEAPPEND(v\_dest\_clob, v\_chars\_read, v\_char\_buffer);

-- append a newline to v\_dest\_clob because v\_char\_buffer;

-- the ASCII value for newline is 10, so CHR(10) returns newline

DBMS\_LOB.WRITEAPPEND(v\_dest\_clob, 1, CHR(10));

EXCEPTION

-- when there is no more data in the file then exit

WHEN NO\_DATA\_FOUND THEN

EXIT;

END;

END LOOP;

-- close the file

UTL\_FILE.FCLOSE(v\_file);

DBMS\_OUTPUT.PUT\_LINE('Copy successfully completed.');

END copy\_file\_data\_to\_clob;

/

There are a number of things to note about this procedure:

Image UTL\_FILE is a package included with the database and contains methods and types that enable you to read and write files. For example, UTL\_FILE.FILE\_TYPE is an object type used to represent a file.

Image The v\_amount variable is set to 32767, which is the maximum number of characters that can be read from a file during each read operation.

Image The v\_char\_buffer variable is used to store the results read from the file before they are appended to v\_dest\_clob. The maximum length of v\_char\_buffer is set to 32767; this length is large enough to store the maximum number of characters read from the file during each read operation.

Image UTL\_FILE.FOPEN (*directory, file\_name, open\_mode, amount*) opens a file; *open\_mode*can be set to one of the following modes:

Image r to read text

Image w to write text

Image a to append text

Image rb to read bytes

Image wb to write bytes

Image ab to append bytes

Image UTL\_FILE.GET\_LINE(v\_file, v\_char\_buffer) gets a line of text from v\_file into v\_char\_buffer. GET\_LINE() does not add the newline to v\_char\_buffer; because I want the newline, I add it using DBMS\_LOB.WRITEAPPEND(v\_dest\_clob, 1, CHR(10)).

The following example calls copy\_file\_data\_to\_clob() to copy the contents of the file textContent.txt to a new CLOB with an id of 3:

**CALL copy\_file\_data\_to\_clob(3, ’SAMPLE\_FILES\_DIR', 'textContent.txt');**

Copy successfully completed.

The following copy\_file\_data\_to\_blob() procedure shows how to read binary data from a file and store it in a BLOB; notice that a RAW array is used to store the binary data read from the file:

CREATE PROCEDURE copy\_file\_data\_to\_blob(

p\_blob\_id INTEGER,

p\_directory VARCHAR2,

p\_file\_name VARCHAR2

) AS

v\_file UTL\_FILE.FILE\_TYPE;

v\_bytes\_read INTEGER;

v\_dest\_blob BLOB;

v\_amount INTEGER := 32767;

v\_binary\_buffer RAW(32767);

BEGIN

-- insert an empty BLOB

INSERT INTO blob\_content(

id, blob\_column

) VALUES (

p\_blob\_id, EMPTY\_BLOB()

);

-- get the LOB locator of the BLOB

SELECT blob\_column

INTO v\_dest\_blob

FROM blob\_content

WHERE id = p\_blob\_id

FOR UPDATE;

-- open the file for reading of bytes (up to v\_amount bytes at a time)

v\_file := UTL\_FILE.FOPEN(p\_directory, p\_file\_name, 'rb', v\_amount);

-- copy the data from the file into v\_dest\_blob

LOOP

BEGIN

-- read binary data from the file into v\_binary\_buffer

UTL\_FILE.GET\_RAW(v\_file, v\_binary\_buffer, v\_amount);

v\_bytes\_read := LENGTH(v\_binary\_buffer);

-- append v\_binary\_buffer to v\_dest\_blob

DBMS\_LOB.WRITEAPPEND(v\_dest\_blob, v\_bytes\_read/2,

v\_binary\_buffer);

EXCEPTION

-- when there is no more data in the file then exit

WHEN NO\_DATA\_FOUND THEN

EXIT;

END;

END LOOP;

-- close the file

UTL\_FILE.FCLOSE(v\_file);

DBMS\_OUTPUT.PUT\_LINE('Copy successfully completed.');

END copy\_file\_data\_to\_blob;

/

The following example calls copy\_file\_data\_to\_blob() to copy the contents of the file binaryContent.doc to a new BLOB with an id of 3:

**CALL copy\_file\_data\_to\_blob(3, ’SAMPLE\_FILES\_DIR', 'binaryContent.doc');**

Copy successfully completed.

Of course, copy\_file\_data\_to\_blob() can be used to write any binary data contained in a file to a BLOB. The binary data can contain music, video, images, executables, and so on. Go ahead and try this using your own files.

Image

**TIP**  
*You can also bulk-load data into a LOB using the Oracle SQL\*Loader and Data Pump utilities; see the*Oracle Database Large Objects Developer’s Guide *published by Oracle Corporation for details*.

**Copying Data from a CLOB and a BLOB to a File**

The following copy\_clob\_data\_to\_file() procedure shows how to read text from a CLOB and save it to a file:

CREATE PROCEDURE copy\_clob\_data\_to\_file(

p\_clob\_id INTEGER,

p\_directory VARCHAR2,

p\_file\_name VARCHAR2

) AS

v\_src\_clob CLOB;

v\_file UTL\_FILE.FILE\_TYPE;

v\_offset INTEGER := 1;

v\_amount INTEGER := 32767;

v\_char\_buffer VARCHAR2(32767);

BEGIN

-- get the LOB locator of the CLOB

SELECT clob\_column

INTO v\_src\_clob

FROM clob\_content

WHERE id = p\_clob\_id;

-- open the file for writing of text (up to v\_amount characters at a time)

v\_file := UTL\_FILE.FOPEN(p\_directory, p\_file\_name, 'w', v\_amount);

-- copy the data from v\_src\_clob to the file

LOOP

BEGIN

-- read characters from v\_src\_clob into v\_char\_buffer

DBMS\_LOB.READ(v\_src\_clob, v\_amount, v\_offset, v\_char\_buffer);

-- copy the characters from v\_char\_buffer to the file

UTL\_FILE.PUT(v\_file, v\_char\_buffer);

-- add v\_amount to v\_offset

v\_offset := v\_offset + v\_amount;

EXCEPTION

-- when there is no more data in the file then exit

WHEN NO\_DATA\_FOUND THEN

EXIT;

END;

END LOOP;

-- flush any remaining data to the file

UTL\_FILE.FFLUSH(v\_file);

-- close the file

UTL\_FILE.FCLOSE(v\_file);

DBMS\_OUTPUT.PUT\_LINE('Copy successfully completed.');

END copy\_clob\_data\_to\_file;

/

The following example calls copy\_clob\_data\_to\_file() to copy the contents of CLOB #3 to a new file named textContent2.txt:

**CALL copy\_clob\_data\_to\_file(3, ’SAMPLE\_FILES\_DIR', 'textContent2.txt');**

Copy successfully completed.

If you look in the C:\sample\_files directory, you will find the new textContent2.txt file. This file contains identical text to textContent.txt.

The following copy\_blob\_data\_to\_file() procedure shows how to read binary data from a BLOB and save it to a file:

CREATE PROCEDURE copy\_blob\_data\_to\_file(

p\_blob\_id INTEGER,

p\_directory VARCHAR2,

p\_file\_name VARCHAR2

) AS

v\_src\_blob BLOB;

v\_file UTL\_FILE.FILE\_TYPE;

v\_offset INTEGER := 1;

v\_amount INTEGER := 32767;

v\_binary\_buffer RAW(32767);

BEGIN

-- get the LOB locator of the BLOB

SELECT blob\_column

INTO v\_src\_blob

FROM blob\_content

WHERE id = p\_blob\_id;

-- open the file for writing of bytes (up to v\_amount bytes at a time)

v\_file := UTL\_FILE.FOPEN(p\_directory, p\_file\_name, 'wb', v\_amount);

-- copy the data from v\_src\_blob to the file

LOOP

BEGIN

-- read characters from v\_src\_blob into v\_binary\_buffer

DBMS\_LOB.READ(v\_src\_blob, v\_amount, v\_offset, v\_binary\_buffer);

-- copy the binary data from v\_binary\_buffer to the file

UTL\_FILE.PUT\_RAW(v\_file, v\_binary\_buffer);

-- add v\_amount to v\_offset

v\_offset := v\_offset + v\_amount;

EXCEPTION

-- when there is no more data in the file then exit

WHEN NO\_DATA\_FOUND THEN

EXIT;

END;

END LOOP;

-- flush any remaining data to the file

UTL\_FILE.FFLUSH(v\_file);

-- close the file

UTL\_FILE.FCLOSE(v\_file);

DBMS\_OUTPUT.PUT\_LINE('Copy successfully completed.');

END copy\_blob\_data\_to\_file;

/

The following example calls copy\_blob\_data\_to\_file() to copy the contents of BLOB #3 to a new file named binaryContent2.doc:

**CALL copy\_blob\_data\_to\_file(3, ’SAMPLE\_FILES\_DIR', 'binaryContent2.doc');**

Copy successfully completed.

If you look in the C:\sample\_files directory, you will find the new binaryContent2.doc file. This file contains identical text to binaryContent.doc.

Of course, copy\_blob\_data\_to\_file() can be used to write any binary data contained in a BLOB to a file. The binary data can contain music, video, images, executables, and so on.

**Copying Data from a BFILE to a CLOB and a BLOB**

The following copy\_bfile\_data\_to\_clob() procedure shows how to read text from a BFILE and save it to a CLOB:

CREATE PROCEDURE copy\_bfile\_data\_to\_clob(

p\_bfile\_id INTEGER,

p\_clob\_id INTEGER

) AS

v\_src\_bfile BFILE;

v\_directory VARCHAR2(200);

v\_filename VARCHAR2(200);

v\_length INTEGER;

v\_dest\_clob CLOB;

v\_amount INTEGER := DBMS\_LOB.LOBMAXSIZE;

v\_dest\_offset INTEGER := 1;

v\_src\_offset INTEGER := 1;

v\_src\_csid INTEGER := DBMS\_LOB.DEFAULT\_CSID;

v\_lang\_context INTEGER := DBMS\_LOB.DEFAULT\_LANG\_CTX;

v\_warning INTEGER;

BEGIN

-- get the locator of the BFILE

SELECT bfile\_column

INTO v\_src\_bfile

FROM bfile\_content

WHERE id = p\_bfile\_id;

-- use FILEEXISTS() to check if the file exists

-- (FILEEXISTS() returns 1 if the file exists)

IF (DBMS\_LOB.FILEEXISTS(v\_src\_bfile) = 1) THEN

-- use OPEN() to open the file

DBMS\_LOB.OPEN(v\_src\_bfile);

-- use FILEGETNAME() to get the name of the file and the directory

DBMS\_LOB.FILEGETNAME(v\_src\_bfile, v\_directory, v\_filename);

DBMS\_OUTPUT.PUT\_LINE(’Directory = ' || v\_directory);

DBMS\_OUTPUT.PUT\_LINE('Filename = ' || v\_filename);

-- insert an empty CLOB

INSERT INTO clob\_content(

id, clob\_column

) VALUES (

p\_clob\_id, EMPTY\_CLOB()

);

-- get the LOB locator of the CLOB (for update)

SELECT clob\_column

INTO v\_dest\_clob

FROM clob\_content

WHERE id = p\_clob\_id

FOR UPDATE;

-- use LOADCLOBFROMFILE() to get up to v\_amount characters

-- from v\_src\_bfile and store them in v\_dest\_clob, starting

-- at offset 1 in v\_src\_bfile and v\_dest\_clob

DBMS\_LOB.LOADCLOBFROMFILE(

v\_dest\_clob, v\_src\_bfile,

v\_amount, v\_dest\_offset, v\_src\_offset,

v\_src\_csid, v\_lang\_context, v\_warning

);

-- check v\_warning for an inconvertible character

IF (v\_warning = DBMS\_LOB.WARN\_INCONVERTIBLE\_CHAR) THEN

DBMS\_OUTPUT.PUT\_LINE('Warning! Inconvertible character.');

END IF;

-- use CLOSE() to close v\_src\_bfile

DBMS\_LOB.CLOSE(v\_src\_bfile);

DBMS\_OUTPUT.PUT\_LINE('Copy successfully completed.');

ELSE

DBMS\_OUTPUT.PUT\_LINE('File does not exist');

END IF;

END copy\_bfile\_data\_to\_clob;

/

The following example calls copy\_bfile\_data\_to\_clob() to copy the contents of BFILE #1 to a new CLOB with an id of 4:

**CALL copy\_bfile\_data\_to\_clob(1, 4);**

Copy successfully completed.

The next example calls copy\_clob\_data\_to\_file() to copy the contents of CLOB #4 to a new file named textContent3.txt:

**CALL copy\_clob\_data\_to\_file(4, ’SAMPLE\_FILES\_DIR', 'textContent3.txt');**

Copy successfully completed.

If you look in the C:\sample\_files directory, you will find the new textContent3.txt file. This file contains identical text to textContent.txt.

The following copy\_bfile\_data\_to\_blob() procedure shows how to read binary data from a BFILE and save it to a BLOB:

CREATE PROCEDURE copy\_bfile\_data\_to\_blob(

p\_bfile\_id INTEGER,

p\_blob\_id INTEGER

) AS

v\_src\_bfile BFILE;

v\_directory VARCHAR2(200);

v\_filename VARCHAR2(200);

v\_length INTEGER;

v\_dest\_blob BLOB;

v\_amount INTEGER := DBMS\_LOB.LOBMAXSIZE;

v\_dest\_offset INTEGER := 1;

v\_src\_offset INTEGER := 1;

BEGIN

-- get the locator of the BFILE

SELECT bfile\_column

INTO v\_src\_bfile

FROM bfile\_content

WHERE id = p\_bfile\_id;

-- use FILEEXISTS() to check if the file exists

-- (FILEEXISTS() returns 1 if the file exists)

IF (DBMS\_LOB.FILEEXISTS(v\_src\_bfile) = 1) THEN

-- use OPEN() to open the file

DBMS\_LOB.OPEN(v\_src\_bfile);

-- use FILEGETNAME() to get the name of the file and

-- the directory

DBMS\_LOB.FILEGETNAME(v\_src\_bfile, v\_directory, v\_filename);

DBMS\_OUTPUT.PUT\_LINE(’Directory = ' || v\_directory);

DBMS\_OUTPUT.PUT\_LINE('Filename = ' || v\_filename);

-- insert an empty BLOB

INSERT INTO blob\_content(

id, blob\_column

) VALUES (

p\_blob\_id, EMPTY\_BLOB()

);

-- get the LOB locator of the BLOB (for update)

SELECT blob\_column

INTO v\_dest\_blob

FROM blob\_content

WHERE id = p\_blob\_id

FOR UPDATE;

-- use LOADBLOBFROMFILE() to get up to v\_amount bytes

-- from v\_src\_bfile and store them in v\_dest\_blob, starting

-- at offset 1 in v\_src\_bfile and v\_dest\_blob

DBMS\_LOB.LOADBLOBFROMFILE(

v\_dest\_blob, v\_src\_bfile,

v\_amount, v\_dest\_offset, v\_src\_offset

);

-- use CLOSE() to close v\_src\_bfile

DBMS\_LOB.CLOSE(v\_src\_bfile);

DBMS\_OUTPUT.PUT\_LINE('Copy successfully completed.');

ELSE

DBMS\_OUTPUT.PUT\_LINE('File does not exist');

END IF;

END copy\_bfile\_data\_to\_blob;

/

The following example calls copy\_bfile\_data\_to\_blob() to copy the contents of BFILE #2 to a new BLOB with an id of 4:

**CALL copy\_bfile\_data\_to\_blob(2, 4);**

Copy successfully completed.

The next example calls copy\_blob\_data\_to\_file() to copy the contents of BLOB #4 to a new file named binaryContent3.doc:

**CALL copy\_blob\_data\_to\_file(4, ’SAMPLE\_FILES\_DIR', 'binaryContent3.doc');**

Copy successfully completed.

If you look in the C:\sample\_files directory, you will find the new binaryContent3.doc file. This file contains identical text to binaryContent.doc.

This is the end of the coverage on large objects. In the next section, you’ll learn about the LONG and LONG RAWtypes.

**LONG AND LONG RAW TYPES**

I mentioned at the start of this chapter that LOBs are the preferred storage type for large blocks of data, but you may encounter databases that still use the following types:

Image **LONG** Used to store up to 2 gigabytes of character data

Image **LONG RAW** Used to store up to 2 gigabytes of binary data

Image **RAW** Used to store up to 4 kilobytes of binary data

In this section, you’ll learn how to use LONG and LONG RAW types. RAW is used in the same way as a LONG RAW, so I’ve omitted coverage of RAW.

**The Example Tables**

In this section, you’ll see the use of the following two tables:

Image **long\_content** Contains a LONG column named long\_column

Image **long\_raw\_content** Contains a LONG RAW column named long\_raw\_column

These two tables are created by the lob\_schema.sql script using the following statements:

CREATE TABLE long\_content (

id INTEGER PRIMARY KEY,

long\_column LONG NOT NULL

);

CREATE TABLE long\_raw\_content (

id INTEGER PRIMARY KEY,

long\_raw\_column LONG RAW NOT NULL

);

**Adding Data to LONG and LONG RAW Columns**

The following INSERT statements add rows to the long\_content table:

INSERT INTO long\_content (

id, long\_column

) VALUES (

1, 'Creeps in this petty pace'

);

INSERT INTO long\_content (

id, long\_column

) VALUES (

2, ' from day to day'

);

The following INSERT statements add rows to the long\_raw\_content table (the first INSERT contains a binary number, the second a hexadecimal number):

INSERT INTO long\_raw\_content (

id, long\_raw\_column

) VALUES (

1, '100111010101011111'

);

INSERT INTO long\_raw\_content (

id, long\_raw\_column

) VALUES (

2, 'A0FFB71CF90DE'

);

In the next section, you’ll see how to convert LONG and LONG RAW columns to LOBs.

**Converting LONG and LONG RAW Columns to LOBs**

You can convert a LONG to a CLOB using the TO\_LOB() function. For example, the following statement converts long\_column to a CLOB using TO\_LOB() and stores the results in the clob\_content table:

**INSERT INTO clob\_content**

**SELECT 10 + id, TO\_LOB(long\_column)**

**FROM long\_content;**

2 rows created.

You can convert a LONG RAW to a BLOB using the TO\_LOB() function. For example, the following statement converts long\_raw\_column to a BLOB using TO\_LOB() and stores the results in the blob\_content table:

**INSERT INTO blob\_content**

**SELECT 10 + id, TO\_LOB(long\_raw\_column)**

**FROM long\_raw\_content;**

2 rows created.

You can also use the ALTER TABLE statement to convert LONG and LONG RAW columns directly. For example, the following statement converts long\_column to a CLOB:

**ALTER TABLE long\_content MODIFY (long\_column CLOB);**

The next example converts long\_raw\_column to a BLOB:

**ALTER TABLE long\_raw\_content MODIFY (long\_raw\_column BLOB);**

Image

**CAUTION**  
*You should not modify tables that are currently used in a production application*.

Once a LONG or LONG RAW column is converted to a LOB, you can use the rich PL/SQL methods described earlier to access the LOB.

**ORACLE DATABASE 10*G* ENHANCEMENTS TO LARGE OBJECTS**

In this section, you’ll learn about the following enhancements made to large objects in Oracle Database 10*g*:

Image Implicit conversion between CLOB and NCLOB objects

Image Use of the :new attribute when using LOBs in a trigger

I’ve provided an SQL\*Plus script named lob\_schema2.sql in the SQL directory. This script can be run using Oracle Database 10*g* and higher. The script creates a user named lob\_user2 with a password of lob\_password and creates the tables and PL/SQL code used in this section. After the script completes, you will be logged in as lob\_user2.

**Implicit Conversion Between CLOB and NCLOB Objects**

In today’s global business environment, you might have to deal with conversions between Unicode and a national language character set. Unicode is a universal character set that enables you to store text that can be converted into any language; it does this by providing a unique code for every character, regardless of the language. A national character set stores text in a specific language.

In versions of the database below Oracle Database 10*g*, you have to explicitly convert between Unicode text and the national character set text using the TO\_CLOB() and TO\_NCLOB() functions. TO\_CLOB() allows you to convert text stored in a VARCHAR2, NVARCHAR2, or NCLOB to a CLOB. Similarly, TO\_NCLOB() allows you to convert text stored in a VARCHAR2, NVARCHAR2, or CLOB to an NCLOB.

Oracle Database 10*g* and higher implicitly converts Unicode text and national character set text in CLOB and NCLOB objects, which saves you from using TO\_CLOB() and TO\_NCLOB(). You can use this implicit conversion for IN and OUT variables in queries and DML statements as well as for PL/SQL method parameters and variable assignments.

Let’s take a look at an example. The following statement creates a table named nclob\_content that contains an NCLOB column named nclob\_column:

CREATE TABLE nclob\_content (

id INTEGER PRIMARY KEY,

nclob\_column NCLOB

);

The following nclob\_example() procedure shows the implicit conversion of a CLOB to an NCLOB, and vice versa:

CREATE PROCEDURE nclob\_example

AS

v\_clob CLOB := 'It is the east and Juliet is the sun';

v\_nclob NCLOB;

BEGIN

-- insert v\_clob into nclob\_column; this implicitly

-- converts the CLOB v\_clob to an NCLOB, storing

-- the contents of v\_clob in the nclob\_content table

INSERT INTO nclob\_content (

id, nclob\_column

) VALUES (

1, v\_clob

);

-- select nclob\_column into v\_clob; this implicitly

-- converts the NCLOB stored in nclob\_column to a

-- CLOB, retrieving the contents of nclob\_column

-- into v\_clob

SELECT nclob\_column

INTO v\_clob

FROM nclob\_content

WHERE id = 1;

-- display the contents of v\_clob

DBMS\_OUTPUT.PUT\_LINE('v\_clob = ' || v\_clob);

END nclob\_example;

/

The following example turns the server output on and calls nclob\_example():

**SET SERVEROUTPUT ON**

**CALL nclob\_example();**

v\_clob = It is the east and Juliet is the sun

**Use of the :new Attribute When Using LOBs in a Trigger**

In Oracle Database 10*g* and higher, you can use the :new attribute when referencing LOBs in a BEFORE UPDATE or BEFORE INSERT row level trigger. The following example creates a trigger named before\_clob\_content\_update; the trigger fires when the clob\_content table is updated and displays the length of the new data in clob\_column; notice that :new is used to access the new data in clob\_column:

CREATE TRIGGER before\_clob\_content\_update

BEFORE UPDATE

ON clob\_content

FOR EACH ROW

BEGIN

DBMS\_OUTPUT.PUT\_LINE('clob\_content changed');

DBMS\_OUTPUT.PUT\_LINE(

'Length = ' || DBMS\_LOB.GETLENGTH(:new.clob\_column)

);

END before\_clob\_content\_update;

/

The following example updates the clob\_content table, causing the trigger to be fired:

**UPDATE clob\_content**

**SET clob\_column = 'Creeps in this petty pace'**

**WHERE id = 1;**

clob\_content changed

Length = 25

**ORACLE DATABASE 11*G* ENHANCEMENTS TO LARGE OBJECTS**

In this section, you’ll learn about the following enhancements made to large objects in Oracle Database 11*g*:

Image Encryption of BLOB, CLOB, and NCLOB data, which prevents unauthorized viewing and modification of the data

Image Compression to squeeze BLOB, CLOB, and NCLOB data

Image De-duplication of BLOB, CLOB, and NCLOB data to automatically detect and remove repeated data

**Encrypting LOB Data**

You can disguise your data using encryption so that unauthorized users cannot view or modify it. You should encrypt sensitive data such as credit card numbers, social security numbers, and so on.

Before you can encrypt data, either you or a database administrator needs to set up a "wallet" to store security details. The data in a wallet includes a private key for encrypting and decrypting data. In this section, you’ll see how to create a wallet, encrypt LOB data, and encrypt regular column data.

**Creating a Wallet**

To create a wallet, you must first create a directory called wallet in the directory $ORACLE\_BASE\admin\$ORACLE\_SID, where ORACLE\_BASE is the base directory where the Oracle database software is installed, and ORACLE\_SID is the system identifier for the database in which the wallet is to be created. For example, on my computer running Windows XP and Oracle Database 11*g*, I created my wallet directory in C:\oracle\_11g\admin\orcl.

Once the wallet directory is created, you need to run SQL\*Plus, connect to the database using a privileged user account (for example, system), and run an ALTER SYSTEM command to set the password for the wallet encryption key, as shown here:

SQL> **CONNECT system/manager**

SQL> **ALTER SYSTEM SET ENCRYPTION KEY IDENTIFIED BY "testpassword123";**

System altered.

Once this is done, a file called ewallet.p12 appears in the wallet directory, and the database automatically opens the wallet. The encryption key password is stored in the wallet, and is used to encrypt and decrypt data behind the scenes.

I’ve provided an SQL\*Plus script named lob\_schema3.sql in the SQL directory. This script may be run using Oracle Database 11*g*. The script creates a user named lob\_user3 with a password of lob\_password, and it also creates the tables used later in this section. After the script completes, you will be logged in as lob\_user3.

**Encrypting LOB Data**

You can encrypt the data stored in a BLOB, CLOB, or NCLOB to prevent unauthorized access to that data; you cannot encrypt a BFILE, because the file itself is stored outside the database.

You can use the following algorithms to encrypt data:

Image **3DES168** The Triple-DES (Data Encryption Standard) algorithm with a key length of 168 bits.

Image **AES128** The Advanced Encryption Standard algorithm with a key length of 128 bits. The AES algorithms were developed to replace the older algorithms based on DES.

Image **AES192** The Advanced Encryption Standard algorithm with a key length of 192 bits.

Image **AES256** The Advanced Encryption Standard algorithm with a key length of 256 bits. This is the most secure encryption algorithm supported by the Oracle database.

The following statement creates a table with a CLOB whose contents are to be encrypted using the AES128 algorithm; notice the use of the ENCRYPT and SECUREFILE keywords, which are required when encrypting data:

CREATE TABLE clob\_content (

id INTEGER PRIMARY KEY,

clob\_column CLOB ENCRYPT USING 'AES128'

) LOB(clob\_column) STORE AS SECUREFILE (

CACHE

);

As you can see, the contents of clob\_column will be encrypted using the AES128 algorithm. If you omit the USING keyword and the algorithm, then the default AES192 algorithm is used.

The CACHE keyword in the CREATE TABLE statement indicates that the database places data from the LOB into the buffer cache for faster access. The options you can use for buffer caching are as follows:

Image **CACHE READS** Use when the LOB data will be frequently read, but written only once or occasionally.

Image **CACHE** Use when the LOB data will be frequently read and frequently written.

Image **NOCACHE** Use when the LOB data will be read once or occasionally and written once or occasionally. This is the default option.

The following INSERT statements add two rows to the clob\_content table:

INSERT INTO clob\_content (

id, clob\_column

) VALUES (

1, TO\_CLOB('Creeps in this petty pace')

);

INSERT INTO clob\_content (

id, clob\_column

) VALUES (

2, TO\_CLOB(' from day to day')

);

The data supplied to clob\_column in these statements are automatically encrypted behind the scenes by the database.

The following query retrieves the rows from the clob\_content table:

**SELECT \***

**FROM clob\_content;**

ID

----------

CLOB\_COLUMN

-------------------------

1

Creeps in this petty pace

2

from day to day

When the data is retrieved, it is automatically decrypted by the database and then returned to SQL\*Plus.

As long as the wallet is open, you can store and retrieve encrypted data; when the wallet is closed, you cannot. Let’s see what happens when the wallet is closed; the following statements connect as the system user and close the wallet:

**CONNECT system/manager**

**ALTER SYSTEM SET WALLET CLOSE;**

If you now attempt to connect as lob\_user3 and retrieve clob\_column from the clob\_content table, you get the error ORA-28365: wallet is not open:

**CONNECT lob\_user3/lob\_password**

**SELECT clob\_column**

**FROM clob\_content;**

ORA-28365: wallet is not open

You can still retrieve and modify the contents of unencrypted columns; for example, the following query retrieves the id column from the clob\_content table:

**SELECT id**

**FROM clob\_content;**

ID

----------

1

2

The following statements connect as the system user and re-open the wallet:

**CONNECT system/manager**

**ALTER SYSTEM SET WALLET OPEN IDENTIFIED BY "testpassword123";**

Once this is done, you can retrieve and modify the contents of clob\_column from the clob\_content table.

**Encrypting Column Data**

You can also encrypt regular column data. This feature was introduced in Oracle Database 10*g* Release 2. For example, the following statement creates a table named credit\_cards with an encrypted column named card\_number:

CREATE TABLE credit\_cards (

card\_number NUMBER(16, 0) ENCRYPT,

first\_name VARCHAR2(10),

last\_name VARCHAR2(10),

expiration DATE

);

You can use the same algorithms to encrypt a column as for a LOB: 3DES168, AES128, AES192 (the default), and AES256. Because I didn’t specify an algorithm after the ENCRYPT keyword for the card\_numbercolumn, the default AES192 algorithm is used.

The following INSERT statements add two rows to the credit\_cards table:

INSERT INTO credit\_cards (

card\_number, first\_name, last\_name, expiration

) VALUES (

1234, 'Jason', 'Bond', '03-FEB-2008'

);

INSERT INTO credit\_cards (

card\_number, first\_name, last\_name, expiration

) VALUES (

5768, ’Steve', 'Edwards', '07-MAR-2009'

);

As long as the wallet is open, you can retrieve and modify the contents of the card\_number column. If the wallet is closed, you get the error ORA-28365: wallet is not open. You saw examples that illustrate these concepts in the previous section, so I won’t repeat similar examples here.

Accessing data in an encrypted column introduces additional overhead. The overhead for encrypting or decrypting a column is estimated by Oracle Corporation to be about 5 percent; this means a SELECT or an INSERT takes about 5 percent more time to complete. The total overhead depends on the number of encrypted columns and their frequency of access; therefore, you should only encrypt columns that contain sensitive data.

Image

**NOTE**  
*If you are interested in learning more about wallets and database security generally, you should read the* Advanced Security Administrator’s Guide *published by Oracle Corporation*.

**Compressing LOB Data**

You can compress the data stored in a BLOB, CLOB, or NCLOB to reduce storage space. For example, the following statement creates a table with a CLOB whose contents are to be compressed; notice the use of the COMPRESS keyword:

CREATE TABLE clob\_content3 (

id INTEGER PRIMARY KEY,

clob\_column CLOB

) LOB(clob\_column) STORE AS SECUREFILE (

COMPRESS

CACHE

);

Image

**NOTE**  
*Even though the table does not contain encrypted data, the* SECUREFILE *clause must be used*.

When you add data to the LOB, it will be automatically compressed by the database; similarly, when you read data from a LOB, it will be automatically decompressed. You can use COMPRESS HIGH for maximum data compression; the default is COMPRESS MEDIUM, and the MEDIUM keyword is optional. The higher the compression, the higher the overhead when reading and writing LOB data.

**Removing Duplicate LOB Data**

You can configure a BLOB, CLOB, or NCLOB so that any duplicate data supplied to it is automatically removed; this process is known as de-duplicating data and can save storage space. For example, the following statement creates a table with a CLOB whose contents are to be de-duplicated; notice the use of the DEDUPLICATE LOBkeywords:

CREATE TABLE clob\_content2 (

id INTEGER PRIMARY KEY,

clob\_column CLOB

) LOB(clob\_column) STORE AS SECUREFILE (

DEDUPLICATE LOB

CACHE

);

Any duplicate data added to the LOB will be automatically removed by the database. The database uses the SHA1 secure hash algorithm to detect duplicate data.

You can learn even more about large objects in the *Oracle Database Large Objects Developer’s Guide* published by Oracle Corporation.

**SUMMARY**

In this chapter, you have learned the following:

Image LOBs may be used to store binary data, character data, and references to external files. LOBs can store up to 128 terabytes of data.

Image There are four LOB types: CLOB, NCLOB, BLOB, and BFILE.

Image A CLOB stores character data.

Image An NCLOB stores multiple byte character data.

Image A BLOB stores binary data.

Image A BFILE stores a pointer to a file located in the file system.

Image A LOB consists of two parts: a locator, which specifies the location of the LOB data, and the data itself.

Image The DBMS\_LOB PL/SQL package contains methods for accessing LOBs.