9 Manipulating Large Objects

Objectives

After completing this lesson, you should be able to do the following:

- Compare and contrast LONG and LOB (large object) data types
- Create and maintain LOB data types
- Differentiate between internal and external LOBs
- Use the DBMS_LOB PL/SQL package
- Describe the use of temporary LOBs

ORACLE

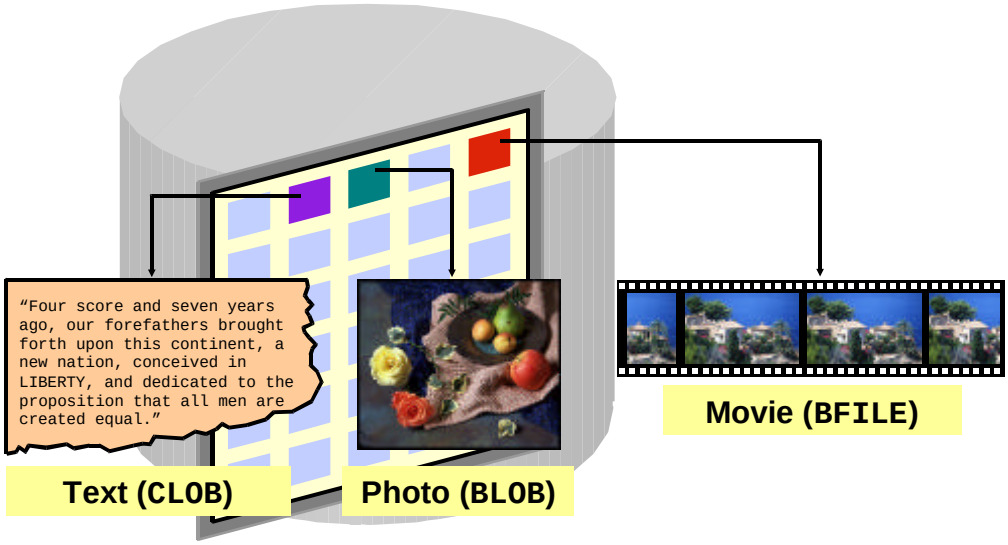
Lesson Aim

Databases have long been used to store large objects. However, the mechanisms built into databases have never been as useful as the large object (LOB) data types that have been provided since Oracle8. This lesson describes the characteristics of the new data types, comparing and contrasting them with earlier data types. Examples, syntax, and issues regarding the LOB types are also presented.

Note: A LOB is a data type and should not be confused with an object type.

What Is a LOB?

LOBs are used to store large unstructured data such as text, graphic images, films, and sound waveforms.



ORACLE

LOB: Overview

A LOB is a data type that is used to store large, unstructured data such as text, graphic images, video clippings, and so on. Structured data, such as a customer record, may be a few hundred bytes, but even small amounts of multimedia data can be thousands of times larger. Also, multimedia data may reside in operating system (OS) files, which may need to be accessed from a database.

There are four large object data types:

- **BLOB** represents a binary large object, such as a video clip.
- **CLOB** represents a character large object.
- **NCLOB** represents a multibyte character large object.
- **BFILE** represents a binary file stored in an OS binary file outside the database. The **BFILE** column or attribute stores a file locator that points to the external file.

LOBs are characterized in two ways, according to their interpretations by the Oracle server (binary or character) and their storage aspects. LOBs can be stored internally (inside the database) or in host files. There are two categories of LOBs:

- Internal LOBs (**CLOB**, **NCLOB**, **BLOB**): Stored in the database
- External files (**BFILE**): Stored outside the database

LOB: Overview (continued)

Oracle Database 10*g* performs implicit conversion between CLOB and VARCHAR2 data types. The other implicit conversions between LOBs are not possible. For example, if the user creates a table T with a CLOB column and a table S with a BLOB column, the data is not directly transferable between these two columns.

BFILEs can be accessed only in read-only mode from an Oracle server.

Contrasting LONG and LOB Data Types

LONG and LONG RAW	LOB
Single LONG column per table	Multiple LOB columns per table
Up to 2 GB	Up to 4 GB
SELECT returns data	SELECT returns locator
Data stored in-line	Data stored in-line or out-of-line
Sequential access to data	Random access to data

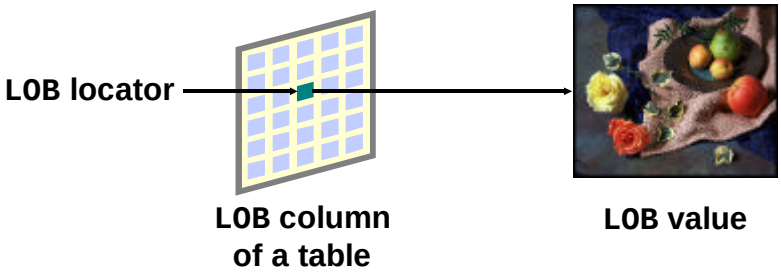
LONG and LOB Data Types

LONG and LONG RAW data types were previously used for unstructured data such as binary images, documents, or geographical information. These data types are superseded by the LOB data types. Oracle Database 10*g* provides a LONG-to-LOB application programming interface (API) to migrate from LONG columns to LOB columns. The following bulleted list compares the LOB functionality with the older types, where LONGs refer to LONG and LONG RAW, and LOBs refer to all LOB data types:

- A table can have multiple LOB columns and object type attributes. A table can have only one LONG column.
- The maximum size of LONGs is 2 GB; LOBs can be up to 4 GB.
- LOBs return the locator; LONGs return the data.
- LOBs store a locator in the table and the data in a different segment, unless the data is less than 4,000 bytes; LONGs store all data in the same data block. In addition, LOBs allow data to be stored in a separate segment and tablespace, or in a host file.
- LOBs can be object type attributes; LONGs cannot be object type attributes.
- LOBs support random piecewise access to the data through a file-like

Anatomy of a LOB

The LOB column stores a locator to the LOB's value.



Components of a LOB

There are two distinct parts to a LOB:

- LOB value: The data that constitutes the real object being stored
- LOB locator: A pointer to the location of the LOB value stored in the database

Regardless of where the value of LOB is stored, a locator is stored in the row. You can think of a LOB locator as a pointer to the actual location of the LOB value.

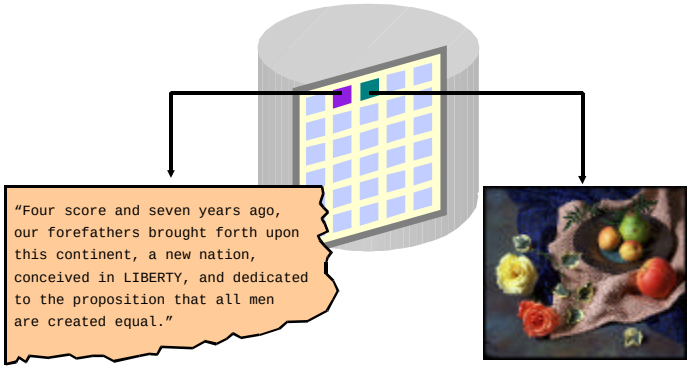
A LOB column does not contain the data; it contains the locator of the LOB value.

When a user creates an internal LOB, the value is stored in the LOB segment and a locator to the out-of-line LOB value is placed in the LOB column of the corresponding row in the table. External LOBs store the data outside the database, so only a locator to the LOB value is stored in the table.

To access and manipulate LOBs without SQL data manipulation language (DML), you must create a LOB locator. The programmatic interfaces operate on the LOB values, using these locators in a manner similar to OS file handles.

Internal LOBs

The LOB value is stored in the database.



ORACLE

Features of Internal LOBs

The internal LOB is stored in the Oracle server. A BLOB, NCLOB, or CLOB can be one of the following:

- An attribute of a user-defined type
- A column in a table
- A bind or host variable
- A PL/SQL variable, parameter, or result

Internal LOBs can take advantage of Oracle features such as:

- Concurrency mechanisms
- Redo logging and recovery mechanisms
- Transactions with COMMIT or ROLLBACK

The BLOB data type is interpreted by the Oracle server as a bitstream, similar to the LONG RAW data type.

The CLOB data type is interpreted as a single-byte character stream.

The NCLOB data type is interpreted as a multiple-byte character stream, based on the byte length of the database national character set.

Managing Internal LOBs

- To interact fully with LOB, file-like interfaces are provided in:
  - PL/SQL package DBMS_LOB
  - Oracle Call Interface (OCI)
  - Oracle Objects for object linking and embedding (OLE)
  - Pro*C/C++ and Pro*COBOL precompilers
  - Java Database Connectivity (JDBC)
- The Oracle server provides some support for LOB management through SQL.

ORACLE

How to Manage LOBs

To manage an internal LOB, perform the following steps:

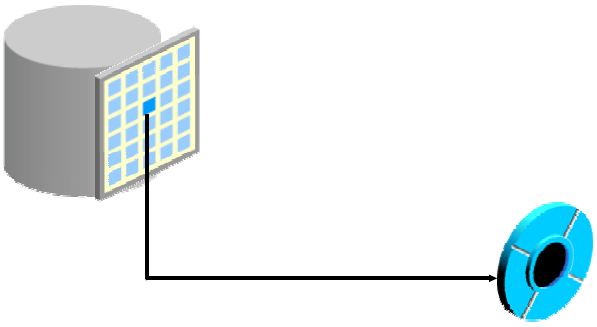
1. Create and populate the table containing the LOB data type.
2. Declare and initialize the LOB locator in the program.
3. Use SELECT FOR UPDATE to lock the row containing the LOB into the LOB locator.
4. Manipulate the LOB with DBMS_LOB package procedures, OCI calls, Oracle Objects for OLE, Oracle precompilers, or JDBC using the LOB locator as a reference to the LOB value.

   You can also manage LOBs through SQL.
5. Use the COMMIT command to make any changes permanent.

What Are BFILEs?

The BFILE data type supports an external or file-based large object as:

- Attributes in an object type
- Column values in a table



Movie (BFILE)

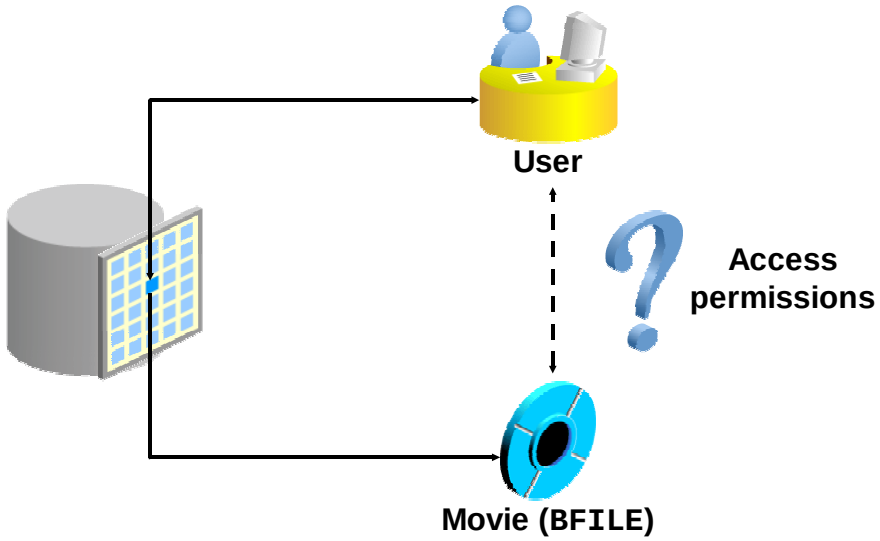
What Are BFILEs?

BFILEs are external large objects (LOBs) stored in OS files that are external to database tables. The BFILE data type stores a locator to the physical file. A BFILE can be in GIF, JPEG, MPEG, MPEG2, text, or other formats. The external LOBs may be located on hard disks, CD-ROMs, photo CDs, or other media, but a single LOB cannot extend from one medium or device to another. The BFILE data type is available so that database users can access the external file system. Oracle Database 10*g* provides:

- Definition of BFILE objects
- Association of BFILE objects to corresponding external files
- Security for BFILEs

The remaining operations that are required for using BFILEs are possible through the DBMS_LOB package and OCI. BFILEs are read-only; they do not participate in transactions. Support for integrity and durability must be provided by the operating system. The file must be created and placed in the appropriate directory, giving the Oracle process privileges to read the file. When the LOB is deleted, the Oracle server does not delete the file. Administration of the files and the OS directory structures can be managed by the database administrator (DBA), system administrator, or user. The maximum size of an external large object depends on the operating system

Securing BFILEs



Securing BFILEs

Unauthenticated access to files on a server presents a security risk. Oracle Database 10*g* can act as a security mechanism to shield the operating system from unsecured access while removing the need to manage additional user accounts on an enterprise computer system.

File Location and Access Privileges

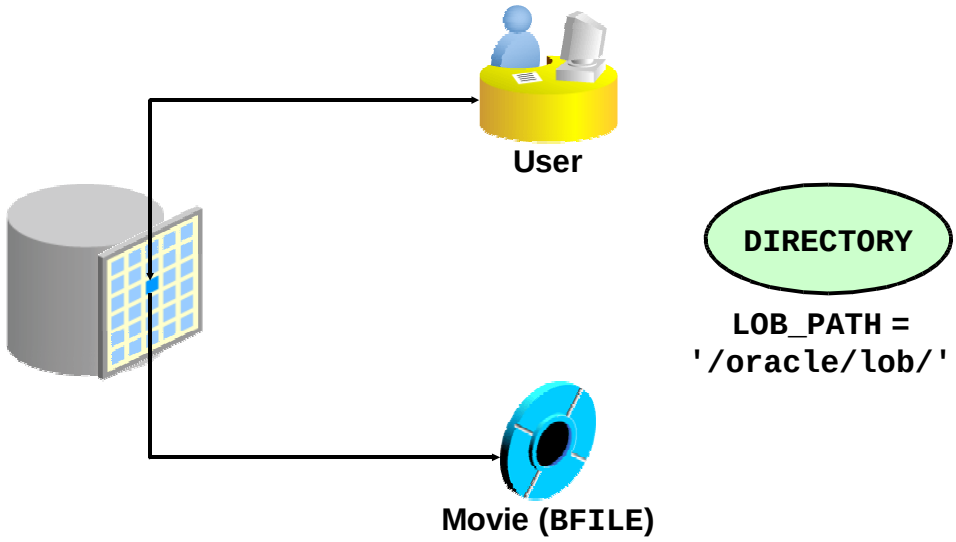
The file must reside on the machine where the database exists. A timeout to read a nonexistent BFILE is based on the OS value.

You can read a BFILE in the same way as you read an internal LOB. However, there could be restrictions related to the file itself, such as:

- Access permissions
- File system space limits
- Non-Oracle manipulations of files
- OS maximum file size

Oracle Database 10*g* does not provide transactional support on BFILEs. Any support for integrity and durability must be provided by the underlying file system and the OS. Oracle backup and recovery methods support only the LOB locators, not the physical BFILEs.

A New Database Object: DIRECTORY



A New Database Object: DIRECTORY

A DIRECTORY is a nonschema database object that enables the administration of access and usage of BFILEs in Oracle Database 10*g*.

A DIRECTORY specifies an alias for a directory on the file system of the server under which a BFILE is located. By granting suitable privileges for these items to users, you can provide secure access to files in the corresponding directories on a user-by-user basis (certain directories can be made read-only, inaccessible, and so on).

Furthermore, these directory aliases can be used while referring to files (open, close, read, and so on) in PL/SQL and OCI. This provides application abstraction from hard-coded path names and gives flexibility in portably managing file locations.

The DIRECTORY object is owned by SYS and created by the DBA (or a user with the CREATE ANY DIRECTORY privilege). The directory objects have object privileges, unlike any other nonschema object. Privileges to the DIRECTORY object can be granted and revoked. Logical path names are not supported.

The permissions for the actual directory depend on the operating system. They may differ from those defined for the DIRECTORY object and could change after the creation of the DIRECTORY object.

Guidelines for Creating DIRECTORY Objects

- Do not create DIRECTORY objects on paths with database files.
- Limit the number of people who are given the following system privileges:
  - CREATE ANY DIRECTORY
  - DROP ANY DIRECTORY
- All DIRECTORY objects are owned by SYS.
- Create directory paths and properly set permissions before using the DIRECTORY object so that the Oracle server can read the file.

ORACLE

Guidelines for Creating DIRECTORY Objects

To associate an OS file with a BFILE, you should first create a DIRECTORY object that is an alias for the full path name to the OS file.

Create DIRECTORY objects by using the following guidelines:

- Directories should point to paths that do not contain database files because tampering with these files could corrupt the database. Currently, only the READ privilege can be given for a DIRECTORY object.
- The CREATE ANY DIRECTORY and DROP ANY DIRECTORY system privileges should be used carefully and not granted to users indiscriminately.
- DIRECTORY objects are not schema objects; all are owned by SYS.
- Create the directory paths with appropriate permissions on the OS before creating the DIRECTORY object. Oracle does not create the OS path.

If you migrate the database to a different OS, then you may need to change the path value of the DIRECTORY object.

The DIRECTORY object information that you create by using the CREATE DIRECTORY command is stored in the DBA_DIRECTORIES and ALL_DIRECTORIES data dictionary views.

Managing BFILEs

The DBA or the system administrator:

1. Creates an OS directory and supplies files
2. Creates a DIRECTORY object in the database
3. Grants the READ privilege on the DIRECTORY object to appropriate database users

The developer or the user:

4. Creates an Oracle table with a column defined as a BFILE data type
5. Inserts rows into the table using the BFILENAME function to populate the BFILE column
6. Writes a PL/SQL subprogram that declares and initializes a LOB locator, and reads BFILE

ORACLE

How to Manage BFILEs

Managing BFILEs requires cooperation between the database administrator and the system administrator and then between the developer and the user of the files.

The database or system administrator should perform the following privileged tasks:

1. Create the operating system (OS) directory (as an Oracle user), and set permissions so that the Oracle server can read the contents of the OS directory. Load files into the OS directory.
2. Create a database DIRECTORY object that references the OS directory.
3. Grant the READ privilege on the database DIRECTORY object to database users requiring access to it.

The designer, application developer, or user should perform the following tasks:

4. Create a database table containing a column defined as the BFILE data type.
5. Insert rows into the table using the BFILENAME function to populate the BFILE column associating the field to an OS file in the named DIRECTORY

Preparing to Use BFILEs

1. Create an OS directory to store the physical data files:

```
mkdir /temp/data_files
```

2. Create a DIRECTORY object by using the CREATE DIRECTORY command:

```
CREATE DIRECTORY data_files
              AS '/temp/data_files';
```

3. Grant the READ privilege on the DIRECTORY object to appropriate users:

```
GRANT READ ON DIRECTORY data_files
   TO SCOTT, MANAGER_ROLE, PUBLIC;
```

ORACLE

Preparing to Use BFILEs

To use a BFILE within an Oracle table, you must have a table with a column of the BFILE type. For the Oracle server to access an external file, the server needs to know the physical location of the file in the OS directory structure.

The database DIRECTORY object provides the means to specify the location of the BFILEs. Use the CREATE DIRECTORY command to specify the pointer to the location where your BFILEs are stored. You need the CREATE ANY DIRECTORY privilege.

Syntax definition: CREATE DIRECTORY *dir_name* AS *os_path*;

In this syntax, *dir_name* is the name of the directory database object, and *os_path* is the location of the BFILEs.
The slide examples show the commands to set up:

- The physical directory (for example /temp/data_files) in the OS
- A named DIRECTORY object, called data_files, that points to the physical directory in the OS
- The READ access right on the directory to be granted to users in the database, providing the privilege to read the BFILEs from the directory

Note: The value of the SESSION_MAX_OPEN_FILES database initialization parameter, which is set to 10 by default, limits the number of BFILEs that can be opened in a session.

Populating BFILE Columns with SQL

- Use the BFILENAME function to initialize a BFILE column. The function syntax is:

```
FUNCTION BFILENAME(directory_alias IN VARCHAR2,
                   filename IN VARCHAR2)
RETURN BFILE;
```

- Example:
  - Add a BFILE column to a table:

```
ALTER TABLE employees ADD video BFILE;
```

  - Update the column using the BFILENAME function:

```
UPDATE employees
  SET video = BFILENAME('DATA_FILES', 'King.avi')
WHERE employee_id = 100;
```


Populating BFILE Columns with SQL

The BFILENAME function is a built-in function that you use to initialize a BFILE column, using the following two parameters:

- *directory_alias* for the name of the DIRECTORY database object that references the OS directory containing the files
- *filename* for the name of the BFILE to be read

The BFILENAME function creates a pointer (or LOB locator) to the external file stored in a physical directory, which is assigned a directory alias name that is used in the first parameter of the function. Populate the BFILE column using the BFILENAME function in either of the following:

- The VALUES clause of an INSERT statement
- The SET clause of an UPDATE statement

An UPDATE operation can be used to change the pointer reference target of the BFILE. A BFILE column can also be initialized to a NULL value and updated later with the BFILENAME function, as shown in the slide.

After the BFILE columns have been associated with a file, subsequent read operations on the BFILE can be performed by using the PL/SQL DBMS_LOB package and OCI. However, these files are read-only when accessed through BFILEs. Therefore, they cannot be updated or deleted through BFILEs.

Populating a BFILE Column with PL/SQL

```
CREATE PROCEDURE set_video(
  dir_alias VARCHAR2, dept_id NUMBER) IS
  filename VARCHAR2(40);
  file_ptr BFILE;
  CURSOR emp_csr IS
    SELECT first_name FROM employees
    WHERE department_id = dept_id FOR UPDATE;
BEGIN
  FOR rec IN emp_csr LOOP
    filename := rec.first_name || '.gif';
    file_ptr := BFILENAME(dir_alias, filename);
    DBMS_LOB.FILEOPEN(file_ptr);
    UPDATE employees SET video = file_ptr
      WHERE CURRENT OF emp_csr;
    DBMS_OUTPUT.PUT_LINE('FILE: ' || filename ||
     ' SIZE: ' || DBMS_LOB.GETLENGTH(file_ptr));
    DBMS_LOB.FILECLOSE(file_ptr);
  END LOOP;
END set_video;
```

ORACLE

Populating a BFILE Column with PL/SQL

The example shows a PL/SQL procedure called `set_video`, which accepts the name of the directory alias referencing the OS file system as a parameter, and a department ID. The procedure performs the following tasks:

- Uses a cursor `FOR` loop to obtain each employee record
- Sets the `filename` by appending `.gif` to the employee's `first_name`
- Creates an in-memory LOB locator for the `BFILE` in the `file_ptr` variable
- Calls the `DBMS_LOB.FILEOPEN` procedure to verify whether the file exists, and to determine the size of the file using the `DBMS_LOB.GETLENGTH` function
- Executes an `UPDATE` statement to write the `BFILE` locator value to the `video BFILE` column
- Displays the file size returned from the `DBMS_LOB.GETLENGTH` function
- Closes the file using the `DBMS_LOB.FILECLOSE` procedure

Suppose that you execute the following call:

```
EXECUTE set_video('DATA_FILE', 60)
```

Sample results are:

```
FILE: Alexander.gif SIZE: 5213
FILE: Bruce.gif SIZE: 26059
```

Using DBMS_LOB Routines with BFILEs

The DBMS_LOB.FILEEXISTS function can check whether the file exists in the OS. The function:

- Returns 0 if the file does not exist
- Returns 1 if the file does exist

```
CREATE FUNCTION get_filesize(file_ptr IN OUT BFILE)
RETURN NUMBER IS
  file_exists BOOLEAN;
  length NUMBER:= -1;
BEGIN
  file_exists := DBMS_LOB.FILEEXISTS(file_ptr)=1;
  IF file_exists THEN
    DBMS_LOB.FILEOPEN(file_ptr);
    length := DBMS_LOB.GETLENGTH(file_ptr);
    DBMS_LOB.FILECLOSE(file_ptr);
  END IF;
  RETURN length;
END;
/
```

ORACLE

Using DBMS_LOB Routines with BFILEs

The set_video procedure on the previous page will terminate with an exception if a file does not exist. To prevent the loop from prematurely terminating, you could create a function, such as get_filesize, to determine whether a given BFILE locator references a file that actually exists on the server's file system. The DBMS_LOB.FILEEXISTS function expects the BFILE locator as a parameter and returns an INTEGER with:

- A value 0 if the physical file does not exist
- A value 1 if the physical file exists

If the BFILE parameter is invalid, one of the three exceptions may be raised:

- NOEXIST_DIRECTORY if the directory does not exist
- NOPRIV_DIRECTORY if database processes do not have privileges for the directory
- INVALID_DIRECTORY if the directory was invalidated after the file was opened

In the get_filesize function, the output of the DBMS_LOB.FILEEXISTS function is compared with value 1 and the result of the condition sets the BOOLEAN variable file_exists. The DBMS_LOB.FILEOPEN call is performed only if the file does exist, preventing unwanted exceptions from occurring

Migrating from LONG to LOB

Oracle Database 10*g* enables the migration of LONG columns to LOB columns.

- Data migration consists of the procedure to move existing tables containing LONG columns to use LOBs:

```
ALTER TABLE [<schema>.] <table_name>
    MODIFY (<long_col_name> {CLOB | BLOB | NCLOB})
```

- Application migration consists of changing existing LONG applications for using LOBs.

ORACLE

Migrating from LONG to LOB

Oracle Database 10*g* supports LONG-to-LOB migration using an API. In data migration, existing tables that contain LONG columns need to be moved to use LOB columns. This can be done by using the ALTER TABLE command. In Oracle8*i*, an operator named TO_LOB had to be used to copy a LONG to a LOB. In Oracle Database 10*g*, this operation can be performed using the syntax shown in the slide. You can use the syntax shown to:

- Modify a LONG column to a CLOB or an NCLOB column
- Modify a LONG RAW column to a BLOB column

The constraints of the LONG column (NULL and NOT NULL are the only allowed constraints) are maintained for the new LOB columns. The default value specified for the LONG column is also copied to the new LOB column. For example, suppose you have the following table:

```
CREATE TABLE long_tab (id NUMBER, long_col LONG);
```

To change the long_col column in the long_tab table to the CLOB data type, use:

```
ALTER TABLE long_tab MODIFY ( long_col CLOB );
```


For information about the limitations on LONG-to-LOB migration, refer to the *Oracle Database Application Developer's Guide - Large Objects*. In application

Migrating from LONG to LOB

- Implicit conversion: From LONG (LONG RAW) or a VARCHAR2(RAW) variable to a CLOB (BLOB) variable, and vice versa
- Explicit conversion:
  - TO_CLOB() converts LONG, VARCHAR2, and CHAR to CLOB.
  - TO_BLOB() converts LONG RAW and RAW to BLOB.
- Function and procedure parameter passing:
  - CLOBs and BLOBs are passed as actual parameters
  - VARCHAR2, LONG, RAW, and LONG RAW are formal parameters, and vice versa.
- LOB data is acceptable in most of the SQL and PL/SQL operators and built-in functions.

Migrating from LONG to LOB (continued)

With the new LONG-to-LOB API introduced in Oracle Database 10*g*, data from CLOB and BLOB columns can be referenced by regular SQL and PL/SQL statements.

Implicit assignment and parameter passing: The LONG-to-LOB migration API supports assigning a CLOB (BLOB) variable to a LONG (LONG RAW) or a VARCHAR2(RAW) variable, and vice versa.

Explicit conversion functions: In PL/SQL, the following two new explicit conversion functions have been added in Oracle Database 10*g* to convert other data types to CLOB and BLOB as part of the LONG-to-LOB migration:

- TO_CLOB()converts LONG, VARCHAR2, and CHAR to CLOB
- TO_BLOB()converts LONG RAW and RAW to BLOB

Note: TO_CHAR()is enabled to convert a CLOB to a CHAR type.

Function and procedure parameter passing: This enables the use of CLOBs and BLOBs as actual parameters where VARCHAR2, LONG, RAW, and LONG RAW are formal parameters, and vice versa. In SQL and PL/SQL built-in functions and operators, a CLOB can be passed to SQL and PL/SQL VARCHAR2 built-in functions, behaving exactly like a VARCHAR2. Or, the VARCHAR2 variable can be passed into DBMS_LOB APIs acting like a LOB locator.

DBMS_LOB Package

- Working with LOBs often requires the use of the Oracle-supplied DBMS_LOB package.
- DBMS_LOB provides routines to access and manipulate internal and external LOBs.
- Oracle Database 10*g* enables retrieving LOB data directly using SQL without a special LOB API.
- In PL/SQL, you can define a VARCHAR2 for a CLOB and a RAW for a BLOB.

DBMS_LOB Package

In releases prior to Oracle9*i*, you must use the DBMS_LOB package for retrieving data from LOBs. To create the DBMS_LOB package, the `dbmslob.sql` and `prvtlob.plb` scripts must be executed as SYS. The `catproc.sql` script executes the scripts. Then users can be granted appropriate privileges to use the package.

The package does not support any concurrency control mechanism for BFILE operations. The user is responsible for locking the row containing the destination internal LOB before calling any subprograms that involve writing to the LOB value. These DBMS_LOB routines do not implicitly lock the row containing the LOB.

The two constants, LOBMAXSIZE and FILE_READONLY, defined in the package specification are also used in the procedures and functions of DBMS_LOB—for example, use them to achieve the maximum level of purity to be used in SQL expressions.

The DBMS_LOB functions and procedures can be broadly classified into two types: mutators and observers.

- The mutators can modify LOB values: APPEND, COPY, ERASE, TRIM, WRITE, FILECLOSE, FILECLOSEALL, and FILEOPEN
- The observers can read LOB values: COMPARE, FILEGETNAME, INSTR,

DBMS_LOB Package

- Modify LOB values:
  APPEND, COPY, ERASE, TRIM, WRITE,
  LOADFROMFILE
- Read or examine LOB values:
  GETLENGTH, INSTR, READ, SUBSTR
- Specific to BFILEs:
  FILECLOSE, FILECLOSEALL, FILEEXISTS,
  FILEGETNAME, FILEISOPEN, FILEOPEN

DBMS_LOB Package (continued)

APPEND	Appends the contents of the source LOB to the destination LOB
COPY	Copies all or part of the source LOB to the destination LOB
ERASE	Erases all or part of a LOB
LOADFROMFILE	Loads BFILE data into an internal LOB
TRIM	Trims the LOB value to a specified shorter length
WRITE	Writes data to the LOB from a specified offset
GETLENGTH	Gets the length of the LOB value
INSTR	Returns the matching position of the *n*th occurrence of the pattern in the LOB
READ	Reads data from the LOB starting at the specified offset
SUBSTR	Returns part of the LOB value starting at the specified offset
FILECLOSE	Closes the file
FILECLOSEALL	Closes all previously opened files
FILEEXISTS	Checks whether the file exists on the server
FILEGETNAME	Gets the directory alias and file name
FILEISOPEN	Checks whether the file was opened using the input BFILE locators
FILEOPEN	Opens a file

DBMS_LOB Package

- NULL parameters get NULL returns.
- Offsets:
  - BLOB, BFILE: Measured in bytes
  - CLOB, NCLOB: Measured in characters
- There are no negative values for parameters.

Using the DBMS_LOB Routines

All functions in the DBMS_LOB package return NULL if any input parameters are NULL. All mutator procedures in the DBMS_LOB package raise an exception if the destination LOB/BFILE is input as NULL.

Only positive, absolute offsets are allowed. They represent the number of bytes or characters from the beginning of LOB data from which to start the operation. Negative offsets and ranges observed in SQL string functions and operators are not allowed. Corresponding exceptions are raised upon violation. The default value for an offset is 1, which indicates the first byte or character in the LOB value.

Similarly, only natural number values are allowed for the amount (BUFSIZ) parameter. Negative values are not allowed.

DBMS_LOB.READ and DBMS_LOB.WRITE

```
PROCEDURE READ (
  lobsrc IN BFILE|BLOB|CLOB ,
  amount IN OUT BINARY_INTEGER,
  offset IN INTEGER,
  buffer OUT RAW|VARCHAR2 )
```

```
PROCEDURE WRITE (
  lobdst IN OUT BLOB|CLOB,
  amount IN OUT BINARY_INTEGER,
  offset IN INTEGER := 1,
  buffer IN RAW|VARCHAR2 )  -- RAW for BLOB
```

ORACLE

DBMS_LOB.READ

Call the READ procedure to read and return piecewise a specified AMOUNT of data from a given LOB, starting from OFFSET. An exception is raised when no more data remains to be read from the source LOB. The value returned in AMOUNT is less than the one specified if the end of the LOB is reached before the specified number of bytes or characters can be read. In the case of CLOBs, the character set of data in BUFFER is the same as that in the LOB.

PL/SQL allows a maximum length of 32,767 for RAW and VARCHAR2 parameters. Ensure that the allocated system resources are adequate to support buffer sizes for the given number of user sessions. Otherwise, the Oracle server raises the appropriate memory exceptions.

Note: BLOB and BFILE return RAW; the others return VARCHAR2.

DBMS_LOB.WRITE

Call the WRITE procedure to write piecewise a specified AMOUNT of data into a given LOB, from the user-specified BUFFER, starting from an absolute OFFSET from the beginning of the LOB value.

Make sure (especially with multibyte characters) that the amount in bytes corresponds to the amount of buffer data. WRITE has no means of checking whether they match, and it will write AMOUNT bytes of the buffer contents into the LOB.

Initializing LOB Columns Added to a Table

- Create the table with columns using the LOB type, or add the LOB columns using ALTER TABLE.

```
ALTER TABLE employees
     ADD (resume CLOB, picture BLOB);
```

- Initialize the column LOB locator value with the DEFAULT option or DML statements using:
  - EMPTY_CLOB() function for a CLOB column
  - EMPTY_BLOB() function for a BLOB column

```
CREATE TABLE emp_hiredata (
   employee_id  NUMBER(6),
   full_name    VARCHAR2(45),
   resume       CLOB DEFAULT EMPTY_CLOB(),
   picture      BLOB DEFAULT EMPTY_BLOB());
```

ORACLE

Initializing LOB Columns Added to a Table

LOB columns are defined by using SQL data definition language (DDL), as in the ALTER TABLE statement in the slide. The contents of a LOB column are stored in the LOB segment, whereas the column in the table contains only a reference to that specific storage area, called the LOB locator. In PL/SQL, you can define a variable of the LOB type, which contains only the value of the LOB locator. You can initialize the LOB locators using:

- EMPTY_CLOB() function to a LOB locator for a CLOB column
- EMPTY_BLOB() function to a LOB locator for a BLOB column

Note: These functions create the LOB locator value and not the LOB content. In general, you use the DBMS_LOB package subroutines to populate the content. The functions are available in Oracle SQL DML, and are not part of the DBMS_LOB package.

The last example in the slide shows how you can use the EMPTY_CLOB() and EMPTY_BLOB() functions in the DEFAULT option in a CREATE TABLE statement. In this way, the LOB locator values are populated in their respective columns when a row is inserted into the table and the LOB columns have not been specified in the INSERT statement.

The next page shows how to use the functions in INSERT and UPDATE statements to initialize the LOB locator values.

Populating LOB Columns

- Insert a row into a table with LOB columns:

```
INSERT INTO emp_hiredata
  (employee_id, full_name, resume, picture)
 VALUES (405, 'Marvin Ellis', EMPTY_CLOB(), NULL);
```

- Initialize a LOB using the EMPTY_BLOB() function:

```
UPDATE emp_hiredata
 SET resume = 'Date of Birth: 8 February 1951',
    picture = EMPTY_BLOB()
 WHERE employee_id = 405;
```

- Update a CLOB column:

```
UPDATE emp_hiredata
 SET resume = 'Date of Birth: 1 June 1956'
 WHERE employee_id = 170;
```

ORACLE

Populating LOB Columns

You can insert a value directly into a LOB column by using host variables in SQL or in PL/SQL, 3GL-embedded SQL, or OCI. You can use the special EMPTY_BLOB() and EMPTY_CLOB() functions in INSERT or UPDATE statements of SQL DML to initialize a NULL or non-NULL internal LOB to empty. To populate a LOB column, perform the following steps:

1. Initialize the LOB column to a non-NULL value—that is, set a LOB locator pointing to an empty or populated LOB value. This is done by using EMPTY_BLOB() and EMPTY_CLOB() functions.
2. Populate the LOB contents by using the DBMS_LOB package routines.

However, as shown in the slide examples, the two UPDATE statements initialize the resume LOB locator value and populate its contents by supplying a literal value. This can also be done in an INSERT statement. A LOB column can be updated to:

- Another LOB value
- A NULL value
- A LOB locator with empty contents by using the EMPTY_*LOB() built-in function

You can update the LOB by using a bind variable in embedded SQL. When assigning one LOB to another, a new copy of the LOB value is created. Use a

Updating LOB by Using DBMS_LOB in PL/SQL

```
DECLARE
  lobloc CLOB;      -- serves as the LOB locator
  text   VARCHAR2(50) := 'Resigned = 5 June 2000';
  amount NUMBER ;   -- amount to be written
  offset INTEGER;   -- where to start writing
BEGIN
  SELECT resume INTO lobloc FROM emp_hiredata
  WHERE employee_id = 405 FOR UPDATE;
  offset := DBMS_LOB.GETLENGTH(lobloc) + 2;
  amount := length(text);
  DBMS_LOB.WRITE (lobloc, amount, offset, text);
  text := ' Resigned = 30 September 2000';
  SELECT resume INTO lobloc FROM emp_hiredata
  WHERE employee_id = 170 FOR UPDATE;
  amount := length(text);
  DBMS_LOB.WRITEAPPEND(lobloc, amount, text);
  COMMIT;
END;
```

ORACLE

Updating LOB by Using DBMS_LOB in PL/SQL

In the example in the slide, the LOBLOC variable serves as the LOB locator, and the AMOUNT variable is set to the length of the text you want to add. The SELECT FOR UPDATE statement locks the row and returns the LOB locator for the RESUME LOB column. Finally, the PL/SQL WRITE package procedure is called to write the text into the LOB value at the specified offset. WRITEAPPEND appends to the existing LOB value.

The example shows how to fetch a CLOB column in releases before Oracle9*i*. In those releases, it was not possible to fetch a CLOB column directly into a character column. The column value needed to be bound to a LOB locator, which is accessed by the DBMS_LOB package. An example later in this lesson shows that you can directly fetch a CLOB column by binding it to a character variable.

Note: Versions prior to Oracle9*i* did not allow LOBs in the WHERE clause of UPDATE and SELECT statements. Now SQL functions of LOBs are allowed in predicates of WHERE. An example is shown later in this lesson.

Selecting CLOB Values by Using SQL

```
SELECT employee_id, full_name , resume -- CLOB
FROM emp_hiredata
WHERE employee_id IN (405, 170);
```

EMPLOYEE_ID	FULL_NAME	RESUME
405	Marvin Ellis	Date of Birth: 8 February 1951 Resigned = 5 June 2000
170	Joe Fox	Date of Birth: 1 June 1956 Resigned = 30 September 2000

Selecting CLOB Values by Using SQL

It is possible to see the data in a CLOB column by using a SELECT statement. It is not possible to see the data in a BLOB or BFILE column by using a SELECT statement in *i*SQL*Plus. You have to use a tool that can display binary information for a BLOB, as well as the relevant software for a BFILE—for example, you can use Oracle Forms.

Selecting CLOB Values by Using DBMS_LOB

- DBMS_LOB.SUBSTR (lob, amount, start_pos)
- DBMS_LOB.INSTR (lob, pattern)

```
SELECT DBMS_LOB.SUBSTR (resume, 5, 18),
       DBMS_LOB.INSTR (resume,' = ')
FROM   emp_hiredata
WHERE  employee_id IN (170, 405);
```

DBMS_LOB.SUBSTR(RESUME,5,18)	DBMS_LOB.INSTR(RESUME,'=')
Febru	40
June	36

ORACLE

Selecting CLOB Values by Using DBMS_LOB

DBMS_LOB.SUBSTR

Use DBMS_LOB.SUBSTR to display a part of a LOB. It is similar in functionality to the SUBSTR SQL function.

DBMS_LOB.INSTR

Use DBMS_LOB.INSTR to search for information within the LOB. This function returns the numerical position of the information.

Note: Starting with Oracle9*i*, you can also use the SUBSTR and INSTR SQL functions to perform the operations shown in the slide.

Selecting CLOB Values in PL/SQL

```
SET LINESIZE 50 SERVEROUTPUT ON FORMAT WORD_WRAP
DECLARE
  text VARCHAR2(4001);
BEGIN
 SELECT resume INTO text
 FROM emp_hiredata
 WHERE employee_id = 170;
 DBMS_OUTPUT.PUT_LINE('text is: '|| text);
END;
/
```

```
text is: Date of Birth: 1 June 1956 Resigned = 30
September 2000

PL/SQL procedure successfully completed.
```

ORACLE

Selecting CLOB Values in PL/SQL

The slide shows the code for accessing CLOB values that can be implicitly converted to VARCHAR2 in Oracle10*g*. When selected, the RESUME column value is implicitly converted from a CLOB into a VARCHAR2 to be stored in the TEXT variable. Prior to Oracle9*i*, you first retrieved the CLOB locator value into a CLOB variable, and then read the LOB contents specifying the amount and offset in the DBMS_LOB.READ procedure:

```
DECLARE
  rlob   CLOB;
  text   VARCHAR2(4001);
  amt    NUMBER := 4001;
  offset NUMBER := 1;
BEGIN
```

text is: Date of Birth: 1 June 1956 Resigned = 30 September 2000
PL/SQL procedure successfully completed.

```
DBMS_LOB.READ(rlob, amt, offset, text);
DBMS_OUTPUT.PUT_LINE('text is: '|| text);
```

Removing LOBs

- Delete a row containing LOBs:

```
DELETE
FROM  emp_hiredata
WHERE employee_id = 405;
```

- Disassociate a LOB value from a row:

```
UPDATE emp_hiredata
SET resume = EMPTY_CLOB()
WHERE employee_id = 170;
```

ORACLE

Removing LOBs

A LOB instance can be deleted (destroyed) using appropriate SQL DML statements. The SQL statement DELETE deletes a row and its associated internal LOB value. To preserve the row and destroy only the reference to the LOB, you must update the row by replacing the LOB column value with NULL or an empty string, or by using the EMPTY_B/CLOB( ) function.

Note: Replacing a column value with NULL and using EMPTY_B/CLOB are not the same. Using NULL sets the value to null; using EMPTY_B/CLOB ensures that there is nothing in the column.

A LOB is destroyed when the row containing the LOB column is deleted, when the table is dropped or truncated, or when all the LOB data is updated.

You must explicitly remove the file associated with a BFILE using OS commands.

To erase part of an internal LOB, you can use DBMS_LOB.ERASE.

Temporary LOBs

- Temporary LOBs:
  - Provide an interface to support creation of LOBs that act like local variables
  - Can be BLOBs, CLOBs, or NCLOBs
  - Are not associated with a specific table
  - Are created using the DBMS_LOB.CREATETEMPORARY procedure
  - Use DBMS_LOB routines
- The lifetime of a temporary LOB is a session.
- Temporary LOBs are useful for transforming data in permanent internal LOBs.

ORACLE

Temporary LOBs

Temporary LOBs provide an interface to support the creation and deletion of LOBs that act like local variables. Temporary LOBs can be BLOBs, CLOBs, or NCLOBs.

The following are the features of temporary LOBs:

- Data is stored in your temporary tablespace, not in tables.
- Temporary LOBs are faster than persistent LOBs because they do not generate any redo or rollback information.
- Temporary LOBs lookup is localized to each user's own session. Only the user who creates a temporary LOB can access it, and all temporary LOBs are deleted at the end of the session in which they were created.
- You can create a temporary LOB using DBMS_LOB.CREATETEMPORARY.

Temporary LOBs are useful when you want to perform some transformational operation on a LOB (for example, changing an image type from GIF to JPEG). A temporary LOB is empty when created and does not support the EMPTY_B/CLOB functions.

Use the DBMS_LOB package to use and manipulate temporary LOBs.

Creating a Temporary LOB

PL/SQL procedure to create and test a temporary LOB:

```
CREATE OR REPLACE PROCEDURE is_templob_open(
  lob IN OUT BLOB, retval OUT INTEGER) IS
BEGIN
  -- create a temporary LOB
  DBMS_LOB.CREATETEMPORARY (lob, TRUE);
  -- see if the LOB is open: returns 1 if open
  retval := DBMS_LOB.ISOPEN (lob);
  DBMS_OUTPUT.PUT_LINE (
    'The file returned a value...' || retval);
  -- free the temporary LOB
  DBMS_LOB.FREETEMPORARY (lob);
END;
/
```

ORACLE

Creating a Temporary LOB

The example in the slide shows a user-defined PL/SQL procedure, `is_templob_open`, which creates a temporary LOB. This procedure accepts a LOB locator as input, creates a temporary LOB, opens it, and tests whether the LOB is open.

The `is_templob_open` procedure uses the procedures and functions from the DBMS_LOB package as follows:

- The CREATETEMPORARY procedure is used to create the temporary LOB.
- The ISOPEN function is used to test whether a LOB is open: this function returns the value 1 if the LOB is open.
- The FREETEMPORARY procedure is used to free the temporary LOB. Memory increases incrementally as the number of temporary LOBs grows, and you can reuse temporary LOB space in your session by explicitly freeing temporary LOBs.

Summary

In this lesson, you should have learned how to:

- Identify four built-in types for large objects: BLOB, CLOB, NCLOB, and BFILE
- Describe how LOBs replace LONG and LONG RAW
- Describe two storage options for LOBs:
  - Oracle server (internal LOBs)
  - External host files (external LOBs)
- Use the DBMS_LOB PL/SQL package to provide routines for LOB management
- Use temporary LOBs in a session

Summary

There are four LOB data types:

- A BLOB is a binary large object.
- A CLOB is a character large object.
- An NCLOB stores multibyte national character set data.
- A BFILE is a large object stored in a binary file outside the database.

LOBs can be stored internally (in the database) or externally (in an OS file). You can manage LOBs by using the DBMS_LOB package and its procedure.

Temporary LOBs provide an interface to support the creation and deletion of LOBs that act like local variables.

Practice 9: Overview

This practice covers the following topics:

- Creating object types using the CLOB and BLOB data types
- Creating a table with LOB data types as columns
- Using the DBMS_LOB package to populate and interact with the LOB data

ORACLE

Practice 9: Overview

In this practice, you create a table with both BLOB and CLOB columns. Then you use the DBMS_LOB package to populate the table and manipulate the data.

Practice 9

1. **Create a table called `PERSONNEL` by executing the script file `E:\labs\PLPU\labs\lab_09_01.sql`. The table contains the following attributes and data types:**

Column Name	Data Type	Length
ID	NUMBER	6
last_name	VARCHAR2	35
review	CLOB	N/A
picture	BLOB	N/A

2. **Insert two rows into the `PERSONNEL` table, one each for employee `2034` (whose last name is `Allen`) and for employee `2035` (whose last name is Bond). Use the empty function for the `CLOB`, and provide `NULL` as the value for the `BLOB`.**
3. **Examine and execute the `E:\labs\PLPU\labs\lab_09_03.sql` script. The script creates a table named `REVIEW_TABLE`. This table contains annual review information for each employee. The script also contains two statements to insert review details for two employees.**
4. **Update the `PERSONNEL` table.**
   a. **Populate the `CLOB` for the first row, using this subquery in an `UPDATE` statement:**
      ```
      SELECT ann_review
      FROM   review_table
      WHERE  employee_id = 2034;
      ```
   b. **Populate the `CLOB` for the second row, using PL/SQL and the `DBMS_LOB` package. Use the following `SELECT` statement to provide a value for the `LOB` locator.**
      ```
      SELECT ann_review
      FROM   review_table
      WHERE  employee_id = 2035;
      ```

If you have time, complete the following exercise:

5. **Create a procedure that adds a locator to a binary file into the `PICTURE` column of the `COUNTRIES` table. The binary file is a picture of the country flag. The image files are named after the country IDs. You need to load an image file locator into all rows in the Europe region (`REGION_ID` = 1) in the `COUNTRIES` table. A `DIRECTORY` object called `COUNTRY_PIC` referencing the location of the binary files has to be created for you.**
   a. **Add the image column to the `COUNTRIES` table using:**