15

Manipulating Large Objects

ORACLE

Schedule:	Timing	Topic
	70 minutes	Lecture
	45 minutes	Practice
	115 minutes	Total

Objectives

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

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

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 new large object (`LOB`) data types provided in 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.

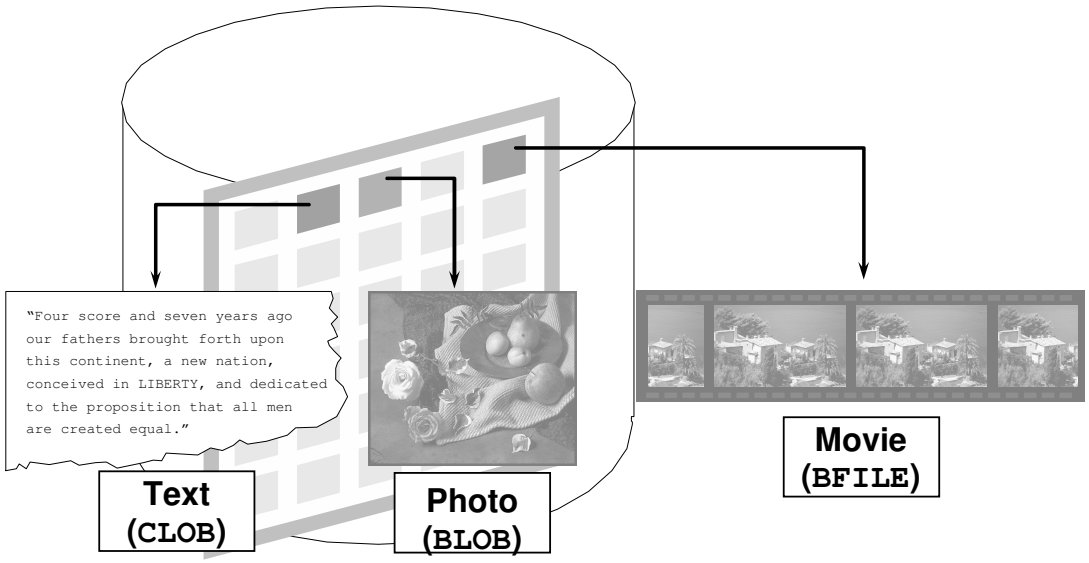
Note: A `LOB` is a data type and should not be confused with an object type.

Instructor Note

`LOB`s are available with Oracle8. They are not part of any option.

What Is a LOB?

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



ORACLE

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 on 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 operating system 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 interpretation 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) are stored in the database.
- External files (BFILE) are stored outside the database.

The Oracle9*i* Server 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

ORACLE

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 9*i* provides a LONG-to-LOB API to migrate from LONG columns to LOB columns.

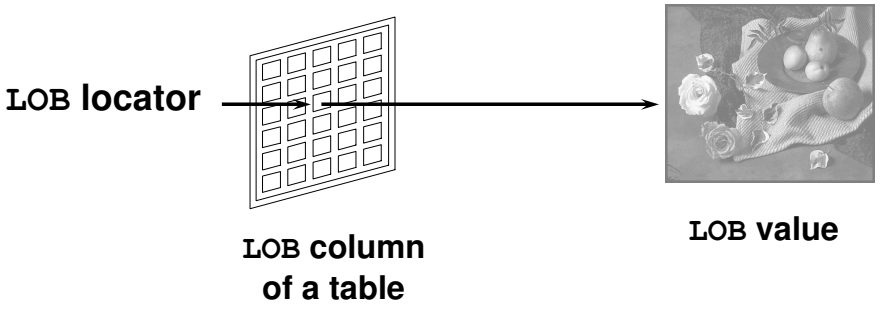
It is beneficial to discuss LOB functionality in comparison to the older types. In the bulleted list below, LONGs refers to LONG and LONG RAW, and LOBs refers to all LOB data types:

- 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 gigabytes; LOBs can be up to 4 gigabytes.
- 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.
- LOBs support random piecewise access to the data through a file-like interface; LONGs are restricted to sequential piecewise access.

The TO_LOB function can be used to covert LONG and LONG RAW values in a column to LOB values. You use this in the SELECT list of a subquery in an INSERT statement.

Anatomy of a LOB

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



Components of a LOB

There are two distinct parts of 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 the 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 DML, you must create a LOB locator. Programmatic interfaces operate on the LOB values, using these locators in a manner similar to operating system file handles.

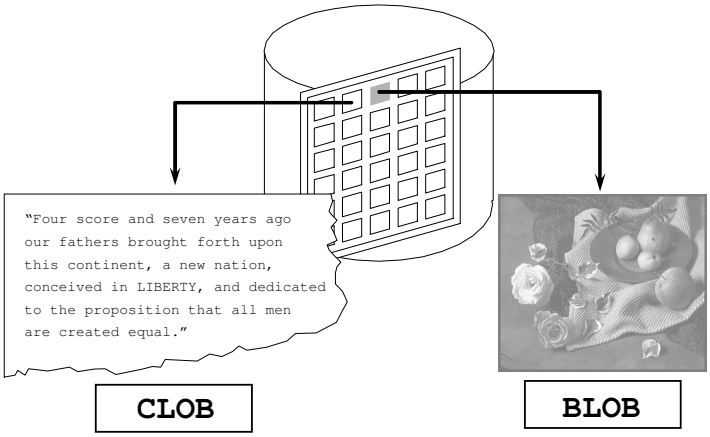
Instructor Note

An internal LOB's value is stored in-line with the other row data if the size of the LOB value is less than 4,000 bytes. When the LOB value is larger than 4,000 bytes, the LOB value is automatically moved out of the row.

When you are creating a table with a LOB column, the default storage is ENABLE STORAGE IN ROW. If you do not want the LOB value stored in the row, even if the size is less than 4,000 bytes, use the storage clause option DISABLE STORAGE IN ROW.

Internal LOBs

The LOB value is stored in the database.



ORACLE

Features of Internal LOBs

The internal LOB is stored inside 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 rollbacks

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
  - JDBC
- The Oracle server provides some support for LOB management through SQL.

ORACLE

How to Manage LOBs

Use the following method to manage an internal LOB:

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.

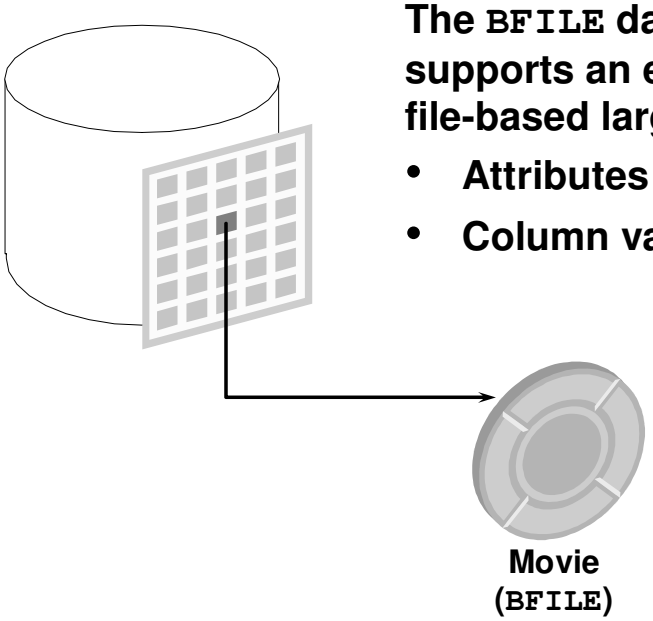
Instructor Note

OCI: Oracle Call Interface

Oracle Objects for OLE (OO4O): Oracle Objects for object linking and embedding

JDBC: Java Database Connectivity

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**

ORACLE

What Are BFILEs?

BFILEs are external large objects (LOBs) stored in operating system files outside of the database tablespaces. The Oracle SQL data type to support these large objects is BFILE. 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, CDROMs, photo CDs, or any such device, but a single LOB cannot extend from one device to another.

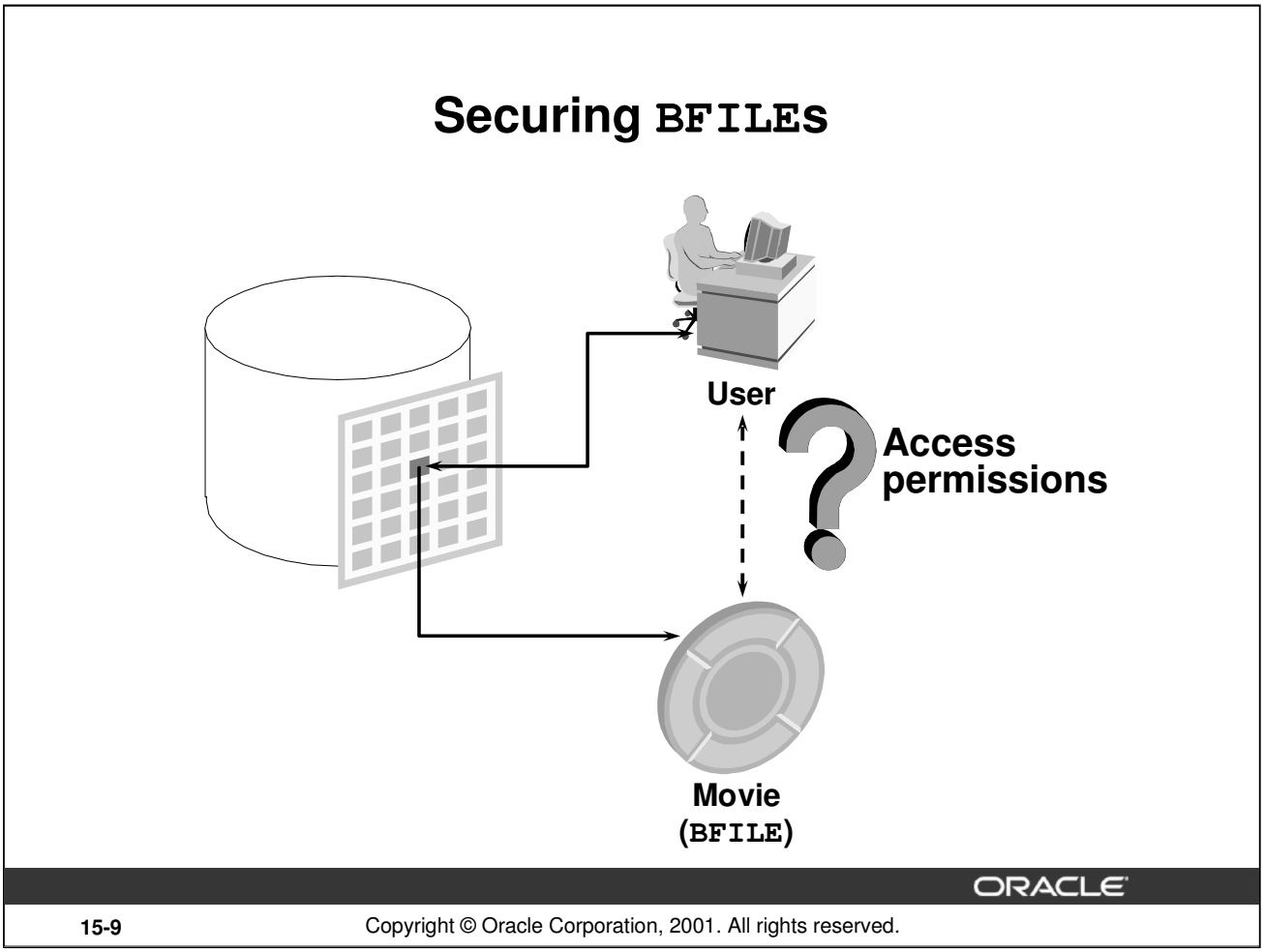
The BFILE data type is available so that database users can access the external file system. The Oracle9*i* server provides for:

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

The rest of the operations required to use BFILEs are possible through the DBMS_LOB package and the Oracle Call Interface.

BFILEs are read-only, so they do not participate in transactions. Any support for integrity and durability must be provided by the operating system. The user must create the file and place it 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. The administration of the actual files and the OS directory structures to house the files is the responsibility of the database administrator (DBA), system administrator, or user. The maximum size of an external large object is operating system dependent but cannot exceed four gigabytes.

Note: BFILEs are available in the Oracle8 database and in later releases.



Securing BFILEs

Unauthenticated access to files on a server presents a security risk. The Oracle9*i* Server 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.

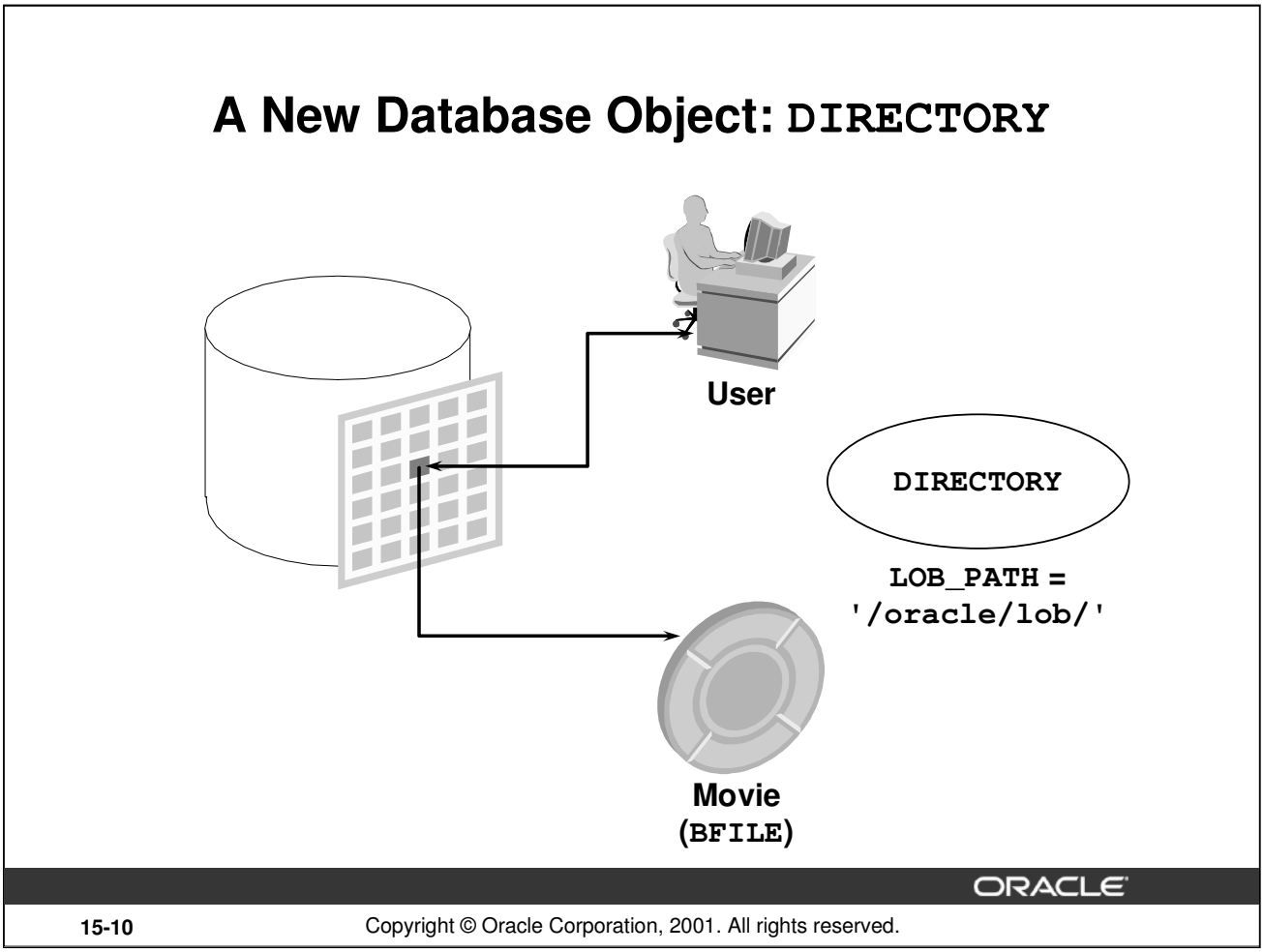
File Location and Access Privileges

The file must reside on the machine where the database exists. A time-out to read a nonexistent `BFILE` is based on the operating system 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

The Oracle9*i* RDBMS does not provide transactional support on `BFILE`s. 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 `BFILE`s.



A New Database Object: DIRECTORY

A DIRECTORY is a nonschema database object that provides for administration of access and usage of BFILEs in an Oracle9*i* Server.

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).

Further, 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 CREATE ANY DIRECTORY privilege). 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 are operating system dependent. 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 operating system file to a BFILE, you should first create a DIRECTORY object that is an alias for the full pathname to the operating system 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 system privileges CREATE ANY DIRECTORY and DROP ANY DIRECTORY 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 prior to creating the DIRECTORY object. Oracle does not create the OS path.

If you migrate the database to a different operating system, 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 data dictionary views DBA_DIRECTORIES and ALL_DIRECTORIES.

Managing BFILEs

- Create an OS directory and supply files.
- Create an Oracle table with a column that holds the BFILE data type.
- Create a DIRECTORY object.
- Grant privileges to read the DIRECTORY object to users.
- Insert rows into the table by using the BFILENAME function.
- Declare and initialize a LOB locator in a program.
- Read the BFILE.

ORACLE

How to Manage BFILEs

Use the following method to manage the BFILE and DIRECTORY objects:

1. Create the 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 the OS directory.
2. Create a table containing the BFILE data type in the Oracle server.
3. Create the DIRECTORY object.
4. Grant the READ privilege to it.
5. Insert rows into the table using the BFILENAME function and associate the OS files with the corresponding row and column intersection.
6. Declare and initialize the LOB locator in a program.
7. Select the row and column containing the BFILE into the LOB locator.
8. Read the BFILE with an OCI or a DBMS_LOB function, using the locator as a reference to the file.

Instructor Note

There is a GRANT READ privilege so users can read from the directory created. This privilege is checked by the DBMS_LOB package.

Preparing to Use BFILEs

- **Create or modify an Oracle table with a column that holds the BFILE data type.**

```
ALTER TABLE employees
  ADD emp_video BFILE;
```

- **Create a DIRECTORY object by using the CREATE DIRECTORY command.**

```
CREATE DIRECTORY dir_name
  AS os_path;
```

- **Grant privileges to read the DIRECTORY object to users.**

```
GRANT READ ON DIRECTORY dir_name TO
user|role|PUBLIC;
```

ORACLE

Preparing to Use BFILEs

In order to use a BFILE within an Oracle table, you need to have a table with a column of BFILE type. For the Oracle server to access an external file, the server needs to know the location of the file on the operating system. The 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*;

Where:		
	dir_name	is the name of the directory database object
	os_path	is the location of the BFILEs

The following commands set up a pointer to BFILEs in the system directory /$HOME/LOG_FILES and give users the privilege to read the BFILEs from the directory.

```
CREATE OR REPLACE DIRECTORY log_files AS '/$HOME/LOG_FILES';
GRANT READ ON DIRECTORY log_files TO PUBLIC;
```

```
Directory created.
Grant succeeded.
```

In a session, the number of BFILEs that can be opened in one session is limited by the parameter SESSION_MAX_OPEN_FILES. This parameter is set in the init.ora file. Its default value is 10.

Instructor Note

To delete a created directory, use the DROP DIRECTORY command.

The BFILENAME Function

Use the BFILENAME function to initialize a BFILE column.

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

ORACLE

The BFILENAME Function

BFILENAME is a built-in function that initializes a BFILE column to point to an external file. Use the BFILENAME function as part of an INSERT statement to initialize a BFILE column by associating it with a physical file in the server file system. You can use the UPDATE statement to change the reference target of the BFILE. A BFILE can be initialized to NULL and updated later by using the BFILENAME function.

Syntax Definitions

Where:

directory_alias	is the name of the DIRECTORY database object
filename	is the name of the BFILE to be read

Example

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

Once physical files are associated with records using SQL DML, subsequent read operations on the BFILE can be performed using the PL/SQL DBMS_LOB package and OCI. However, these files are read-only when accessed through BFILEs, and so they cannot be updated or deleted through BFILEs.

Instructor Note

You can demonstrate this code with the code example 15_14n.sql file. This script file contains additional statements to disable and enable triggers on EMPLOYEES table.

Loading BFILEs

```
CREATE OR REPLACE PROCEDURE load_emp_bfile
    (p_file_loc IN VARCHAR2) IS
  v_file      BFILE;
  v_filename  VARCHAR2(16);
  CURSOR emp_cursor IS
    SELECT first_name FROM employees
    WHERE department_id = 60 FOR UPDATE;
BEGIN
  FOR emp_record IN emp_cursor LOOP
    v_filename := emp_record.first_name || '.bmp';
    v_file := BFILENAME(p_file_loc, v_filename);
    DBMS_LOB.FILEOPEN(v_file);
    UPDATE employees SET emp_video = v_file
      WHERE CURRENT OF emp_cursor;
    DBMS_OUTPUT.PUT_LINE('LOADED FILE: '||v_filename
      || ' SIZE: ' || DBMS_LOB.GETLENGTH(v_file));
    DBMS_LOB.FILECLOSE(v_file);
  END LOOP;
END load_emp_bfile;
/
```

ORACLE

Loading BFILEs

Example

Load a `BFILE` pointer to an image of each employee into the `EMPLOYEES` table by using the `DBMS_LOB` package. The images are `.bmp` files stored in the `/home/LOG_FILES` directory.

Executing the procedure yields the following results:

```
EXECUTE load_emp_bfile('LOG_FILES')
```

```
LOADED FILE: Alexander.bmp SIZE: 22358
LOADED FILE: Bruce.bmp SIZE: 108082
LOADED FILE: David.bmp SIZE: 78736
LOADED FILE: Valli.bmp SIZE: 373102
LOADED FILE: Diana.bmp SIZE: 78736
PL/SQL procedure successfully completed.
```

Instructor Note

Although the `DIRECTORY` object, represented by the `DIRECTORY_ALIAS` parameter to `BFILENAME()`, need not already be defined before `BFILENAME()` is called by a SQL or PL/SQL program, the `DIRECTORY` object and operating system file must exist by the time you actually use the `BFILE` locator. For example, when the `BFILE` locator is used as a parameter to one of the following operations: `OCILobFileOpen()`, `DBMS_LOB.FILEOPEN()`, `OCILOBOPEN()`, `DBMS_LOB.OPEN()`

You can demonstrate this code with the `15_15s.sql` and `15_15n.sql` files. Run the `15_15s.sql` script first.

Loading BFILEs

Use the DBMS_LOB.FILEEXISTS function to vefiry that the file exists in the operating system. The function returns 0 if the file does not exist, and returns 1 if the file does exist.

```
CREATE OR REPLACE PROCEDURE load_emp_bfile
(p_file_loc IN VARCHAR2)
IS
  v_file        BFILE;   v_filename    VARCHAR2(16);
  v_file_exists BOOLEAN;
  CURSOR emp_cursor IS ...
BEGIN
  FOR emp_record IN emp_cursor LOOP
   v_filename := emp_record.first_name || '.bmp';
   v_file := BFILENAME  (p_file_loc, v_filename);
   v_file_exists := (DBMS_LOB.FILEEXISTS(v_file) = 1);
    IF v_file_exists THEN
      DBMS_LOB.FILEOPEN (v_file); ...
```

ORACLE

Using DBMS_LOB.FILEEXISTS

This function finds out whether a given BFILE locator points to a file that actually exists on the server's file system. This is the specification for the function:

Syntax Definitions

```
FUNCTION DBMS_LOB.FILEEXISTS
  (file_loc IN BFILE)
  RETURN INTEGER;
```

Where:	file_loc	is name of the BFILE locator
	RETURN INTEGER	returns 0 if the physical file does not exist returns 1 if the physical file exists

If the FILE_LOC parameter contains an invalid value, one of three exceptions may be raised.

In the example in the slide, the output of the DBMS_LOB.FILEEXISTS function is compared with value 1 and the result is returned to the BOOLEAN variable V_FILE_EXISTS.

Exception Name	Description
NOEXIST_DIRECTORY	The directory does not exist.
NOPRIV_DIRECTORY	You do not have privileges for the directory.
INVALID_DIRECTORY	The directory was invalidated after the file was opened.

Migrating from LONG to LOB

The Oracle9*i* server allows 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

Oracle9*i* Server supports the LONG-to-LOB migration using API.

Data migration: Where existing tables that contain LONG columns need to be moved to use LOB columns. This can be done 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 Oracle9*i*, 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, if you had a table with the following definition:

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

you can change the LONG_COL column in table LONG_TAB to the CLOB data type as follows:

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

For limitations on the LONG-to-LOB migration, refer to *Oracle9i Application Developer's Guide - Large Objects.*

Application Migration: Where the existing LONG applications change for using LOBs. You can use SQL and PL/SQL to access LONGs and LOBs. This API is provided for both OCI and PL/SQL.

Migrating From LONG to LOB

- **Implicit conversion: 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 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**

ORACLE

Migrating from LONG to LOB (continued)

With the new LONG-to-LOB API introduced in Oracle9*i*, 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 Oracle9*i* to convert other data types to CLOB and BLOB as part of LONG-to-LOB migration:

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

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

Function and procedure parameter passing: This allows all the user-defined procedures and functions to use CLOBs and BLOBs as actual parameters where VARCHAR2, LONG, RAW, and LONG RAW are formal parameters, and vice versa.

Accessing 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.

These details are discussed in detail later in this lesson.

For more information, refer to "Migrating from LONGs to LOBs" in *Oracle9i Application Developer's Guide - Large Objects (LOBs)*.

The DBMS_LOB Package

- Working with LOB often requires the use of the Oracle-supplied package DBMS_LOB.
- DBMS_LOB provides routines to access and manipulate internal and external LOBs.
- Oracle9*i* enables retrieving LOB data directly using SQL, without using any special LOB API.
- In PL/SQL you can define a VARCHAR2 for a CLOB and a RAW for BLOB.

ORACLE

The DBMS_LOB Package

In releases prior to Oracle9*i*, you need to 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.

Two constants are used in the specification of procedures in this package: LOBMAXSIZE and FILE_READONLY. These constants are used in the procedures and functions of DBMS_LOB; for example, you can use them to achieve the maximum possible level of purity so that they can be used in SQL expressions.

Using the DBMS_LOB Routines

Functions and procedures in this package can be broadly classified into two types: mutators or observers. Mutators can modify LOB values, whereas observers can only read LOB values.

- Mutators: APPEND, COPY, ERASE, TRIM, WRITE, FILECLOSE, FILECLOSEALL, and FILEOPEN
- Observers: COMPARE, FILEGETNAME, INSTR, GETLENGTH, READ, SUBSTR, FILEEXISTS, and FILEISOPEN

The 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

ORACLE

The DBMS_LOB Package (continued)

APPEND	Append the contents of the source LOB to the destination LOB
COPY	Copy all or part of the source LOB to the destination LOB
ERASE	Erase all or part of a LOB
LOADFROMFILE	Load BFILE data into an internal LOB
TRIM	Trim the LOB value to a specified shorter length
WRITE	Write data to the LOB from a specified offset
GETLENGTH	Get the length of the LOB value
INSTR	Return the matching position of the *n*th occurrence of the pattern in the LOB
READ	Read data from the LOB starting at the specified offset
SUBSTR	Return part of the LOB value starting at the specified offset
FILECLOSE	Close the file
FILECLOSEALL	Close all previously opened files
FILEEXISTS	Check if the file exists on the server
FILEGETNAME	Get the directory alias and file name
FILEISOPEN	Check if the file was opened using the input BFILE locators
FILEOPEN	Open a file

The 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.

ORACLE

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.

Instructor Note

The offset indicates the starting point when reading from or writing to a large object. If the LOB is a CLOB or NCLOB, this is the *n*th character; if it is a BLOB or BFILE, this is the *n*th byte.

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 will be less than the one specified, if the end of the LOB is reached before the specified number of bytes or characters could 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 32767 for RAW and VARCHAR2 parameters. Make sure the allocated system resources are adequate to support these 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 will write AMOUNT bytes of the buffer contents into the LOB.

Instructor Note

These two procedures are given as examples of the routines used from DBMS_LOB. Depending on the audience, you can go through more of the routines using ALL_SOURCE:

```
SELECT text FROM ALL_SOURCE WHERE name = 'DBMS_LOB';
```

Adding LOB Columns to a Table

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

```
Table altered.
```

ORACLE

Adding LOB Columns to a Table

LOB columns are defined by way of SQL data definition language (DDL), as in the ALTER TABLE statement in the slide. The contents of a LOB column is 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 type LOB, which contains only the value of the LOB locator.

Populating LOB Columns

Insert a row into a table with LOB columns:

```
INSERT INTO employees (employee_id, first_name,
    last_name, email, hire_date, job_id,
    salary, resume, picture)
VALUES (405, 'Marvin', 'Ellis', 'MELLIS', SYSDATE,
     'AD_ASST', 4000, EMPTY_CLOB(),NULL);
```

1 row created.

Initialize a LOB column using the EMPTY_BLOB() function:

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

1 row updated.

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 functions EMPTY_BLOB and EMPTY_CLOB in INSERT or UPDATE statements of SQL DML to initialize a NULL or non-NULL internal LOB to empty. These are available as special functions in Oracle SQL DML, and are not part of the DBMS_LOB package.

Before you can start writing data to an internal LOB using OCI or the DBMS_LOB package, the LOB column must be made nonnull, that is, it must contain a locator that points to an empty or populated LOB value. You can initialize a BLOB column's value to empty by using the function EMPTY_BLOB in the VALUES clause of an INSERT statement. Similarly, a CLOB or NCLOB column's value can be initialized by using the function EMPTY_CLOB.

The result of using the function EMPTY_CLOB() or EMPTY_BLOB() means that the LOB is initialized, but not populated with data. To populate the LOB column, you can use an update statement.

You can use an INSERT statement to insert a new row and populate the LOB column at the same time.

When you create a LOB instance, the Oracle server creates and places a locator to the out-of-line LOB value in the LOB column of a particular row in the table. SQL, OCI, and other programmatic interfaces operate on LOBs through these locators.

Populating LOB Columns (continued)

The EMPTY_B/CLOB() function can be used as a DEFAULT column constraint, as in the example below. This initializes the LOB columns with locators.

```
CREATE TABLE emp_hiredata
       (employee_id    NUMBER(6),
        first_name     VARCHAR2(20),
        last_name      VARCHAR2(25),
        resume    CLOB  DEFAULT EMPTY_CLOB(),
        picture   BLOB  DEFAULT EMPTY_BLOB());
```

```
Table created.
```

Updating LOB by Using SQL

UPDATE CLOB column

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

1 row updated.

ORACLE

Updating LOB by Using SQL

You can update a LOB column by setting it to another LOB value, to NULL, or by using the empty function appropriate for the LOB data type (EMPTY_CLOB() or EMPTY_BLOB()). You can update the LOB using a bind variable in embedded SQL, the value of which may be NULL, empty, or populated. When you set one LOB equal to another, a new copy of the LOB value is created. These actions do not require a SELECT FOR UPDATE statement. You must lock the row prior to the update only when updating a piece of the LOB.

Updating LOB by Using DBMS_LOB in PL/SQL

```
DECLARE
  lobloc CLOB;        -- serves as the LOB locator
  text   VARCHAR2(32767):='Resigned: 5 August 2000';
  amount NUMBER ;     -- amount to be written
  offset INTEGER;     -- where to start writing
BEGIN
  SELECT resume INTO lobloc
  FROM   employees
  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   employees
  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 package procedure WRITE 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: In versions prior to Oracle9*i*, Oracle did not allow LOBs in the WHERE clause of UPDATE and SELECT. Now SQL functions of LOBs are allowed in predicates of WHERE. An example is shown later in this lesson.

Instructor Note

EMPTY_B/CLOB() is not the same as NULL, and an IS NULL test on a LOB initialized EMPTY returns FALSE.

Selecting CLOB Values by Using SQL

```
SELECT employee_id, last_name , resume -- CLOB
FROM employees
WHERE employee_id IN (405, 170);
```

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

ORACLE

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.

Instructor Note

Use the *i*SQL*Plus command SET LONG *n* to display more data, if the information in the LOB column is truncated.

Selecting CLOB Values by Using DBMS_LOB

- **DBMS_LOB.SUBSTR(lob_column, no_of_chars, starting)**
- **DBMS_LOB.INSTR (lob_column, pattern)**

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

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

ORACLE

Selecting CLOB Values by Using DBMS_LOB

DBMS_LOB.SUBSTR

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

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 SQL functions SUBSTR and INSTR to perform the operations shown in the slide.

Instructor Note

This works in *i*SQL*Plus because the LOB column is a CLOB and not a BLOB or BFILE.

You can use DBMS_LOB.INSTR to do searches for information in the WHERE clause of a SELECT statement. You may not want do this for performance reasons.

Unlike a LONG, where you can only select data and it is not possible to do any form of manipulation or searching, DBMS_LOB.INSTR/SUBSTR highlights a distinct difference between usage of LONGs and that of LOBs.

Selecting CLOB Values in PL/SQL

```
DECLARE
  text VARCHAR2(4001);
BEGIN
 SELECT resume INTO text
 FROM employees
 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 Oracle9*i*. The value of the column RESUME, when selected into a VARCHAR2 variable TEXT, is implicitly converted.

In prior releases, to access a CLOB column, first you must to retrieve the CLOB column value into a CLOB variable and specify the amount and offset size. Then you use the DBMS_LOB package to read the selected value. The code using DBMS_LOB is as follows:

```
DECLARE
  rlob clob;
  text VARCHAR2(4001);
  amt number := 4001;
  offset number := 1;
BEGIN
 SELECT resume INTO rlob
 FROM employees
 WHERE employee_id = 170;
 DBMS_LOB.READ(rlob, amt, offset, text);
 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.
```

Removing LOBs

Delete a row containing LOBs:

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

1 row deleted.

Disassociate a LOB value from a row:

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

1 row updated.

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 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 implicitly when all the LOB data is updated.

You must explicitly remove the file associated with a BFILE using operating system 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 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.

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.

Instructor Note

Temporary LOBs are introduced in Oracle8*i*.

The default lifetime of a temporary LOB is a session. It can also be cleaned up at the end of the call; this is specified at the time of creation.

Creating a Temporary LOB

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

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

```
Procedure created.
```

ORACLE

Creating a Temporary LOB

The example in the slide shows a user-defined PL/SQL procedure, `IsTempLOBOpen`, that 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 `IsTempLOBOpen` 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 `LOB`s grows, and you can reuse temporary `LOB` space in your session by explicitly freeing temporary `LOB`s.

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:**
  - **The 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**

ORACLE

Summary

There are four LOB data types:

- A BLOB is a binary large object.
- A CLOB is a character large object.
- A 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 operating system file). You can manage LOBs by using the DBMS_LOB package and its procedures.

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

Practice 15 Overview

This practice covers the following topics:

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

ORACLE

Practice 15 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 15

1. Create a table called PERSONNEL by executing the script file lab15_1.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 employees 2034 and 2035. Use the empty function for the CLOB, and provide NULL as the value for the BLOB.
3. Examine and execute the script lab15_3.sql. 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 the following subquery in a SQL 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.

```
SELECT ann_review
FROM   review_table
WHERE  employee_id = 2035;
```

Practice 15 (continued)

If you have time

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. The image files are named after the country IDs. You need to load an image file locator into all rows in Europe region (REGION_ID = 1) in the COUNTRIES table. The DIRECTORY object name that stores a pointer to the location of the binary files is called COUNTRY_PIC. This object is already created for you.

   a. Use the command below to add the image column to the COUNTRIES table (or use lab15_5_add.sql)

      ```
      ALTER TABLE countries ADD (picture BFILE);
      ```

   b. Create a PL/SQL procedure called load_country_image that reads a locator into your picture column. Have the program test to see if the file exists, using the function DBMS_LOB.FILEEXISTS. If the file is not existing, your procedure should display a message that the file can not be opened. Have your program report information about the load to the screen.

   c. Invoke the procedure by passing the name of the directory object COUNTRY_PIC as parameter. Note that you should pass the directory object in single quotation marks.

   Sample output follows:

```
LOADING LOCATORS TO IMAGES...
LOADED LOCATOR TO FILE: BE.tif SIZE: 7444
LOADED LOCATOR TO FILE: CH.tif SIZE: 7444
LOADED LOCATOR TO FILE: DE.tif SIZE: 7444
LOADED LOCATOR TO FILE: DK.tif SIZE: 7444
LOADED LOCATOR TO FILE: FR.tif SIZE: 7444
LOADED LOCATOR TO FILE: IT.tif SIZE: 7444
LOADED LOCATOR TO FILE: NL.tif SIZE: 7444
LOADED LOCATOR TO FILE: UK.tif SIZE: 7444
TOTAL FILES UPDATED: 8
PL/SQL procedure successfully completed.
```

Instructor Note

The class needs to add a BFILE column to the COUNTRIES table to complete question 5 of Practice 15. The script is stored in lab\lab15_5_add.sql.

This practice question 5 requires the instructor to run the file lab\lab15_5_setup.sql. The script creates the DIRECTORY objects and grants READ permissions to these objects. Edit the file and fill in the database connection for the classroom. Also, fill in the file location on the server to identify where the COUNTRY_PIC and LOG_FILES are located.

The steps in this script must be followed before the students can complete question 5 of Practice 15. After running the above script, you can verify that the directory alias is created by querying the ALL_DIRECTORIES data dictionary view as follows:

```
SELECT directory_name, directory_path FROM all_directories;
```