8
Data Interface for LOBs

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

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

8.1 Overview of the Data Interface for LOBs

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

These APIs, originally designed for use with legacy data types such as `VARCHAR2, RAW, LONG,` and `LONG RAW`, can also be used with the corresponding LOB data types shown in the following table. The table shows the legacy data types in the *bind or define type* column and the corresponding supported LOB data type in the *LOB column type* column. You can use the data interface for LOBs to store and manipulate character data and binary data in a LOB column just as if it were stored in the corresponding legacy data type. The data interface supports data size up to two gigabytes minus one (2 GB - 1), the maximum size of an `sb4` data type.

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

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

Table 8-1 Corresponding LONG and LOB Data Types in OCI

Bind or Define Type	LOB Column Type	Used For Storing
`SQLT_AFC(n)`	`CLOB`	Character data
`SQLT_CHR`	`CLOB`	Character data

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

Bind or Define Type	LOB Column Type	Used For Storing
SQLT_LNG	CLOB	Character data
SQLT_VCS	CLOB	Character data
SQLT_BIN	BLOB	Binary data
SQLT_LBI	BLOB	Binary data
SQLT_LVB	BLOB	Binary data

8.2 Benefits of Using the Data Interface for LOBs

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

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

- If your application uses `LONG` data types, then you can use the same application with LOB data types with little or no modification of your existing application required. To do so, just convert `LONG` columns in your tables to LOB columns.

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

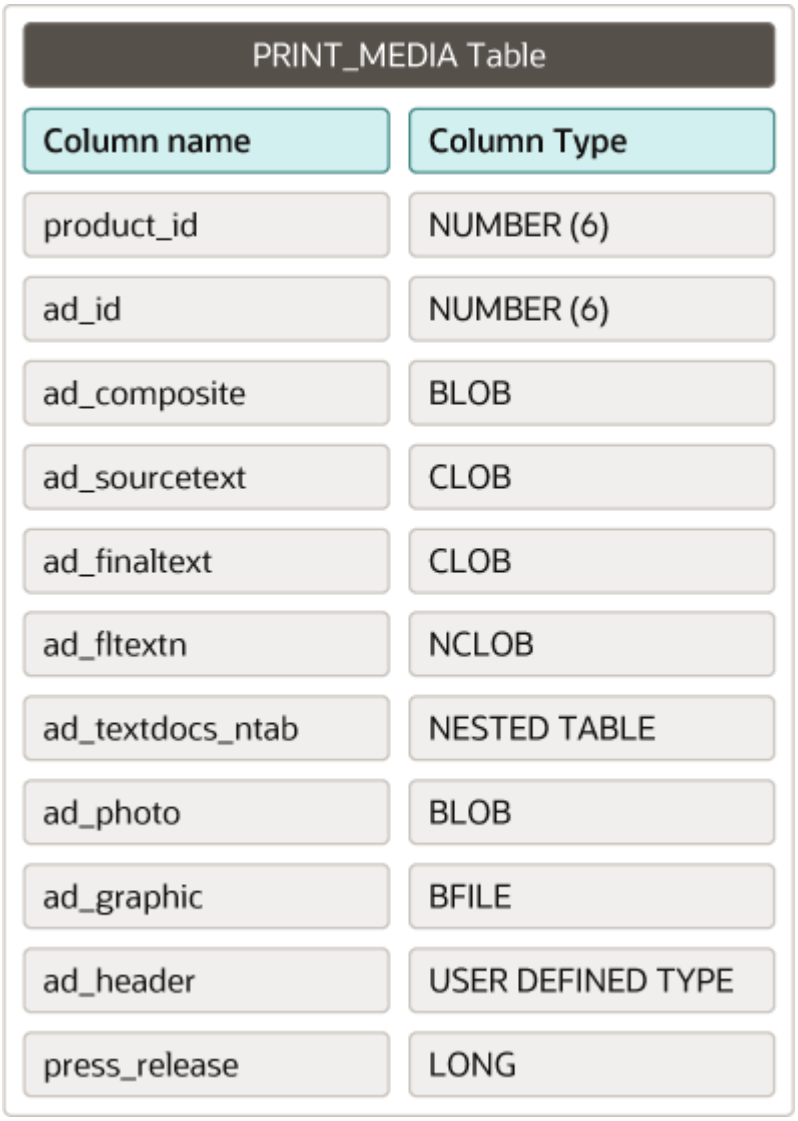
- The Data Interface gives you the best performance if you know the maximum size of your LOB data, and you intend to read or write the entire LOB. A piecewise `INSERT` or fetch using the data interface makes only 1 round-trip the server, as opposed to using LOB API which makes separate round-trips to get the locator and to read/write data.
- You can read LOB data in one `OCIStmtFetch()` call, instead of fetching the LOB locator first and then calling `OCILobRead2()`. This improves performance when you want to read LOB data starting at the beginning.
- You can use array bind and define interfaces to insert and select multiple rows with LOBs in one round trip. Irrespective of whether the LOB data is inserted or fetched using single piece, piecewise or callbacks, it is inserted or fetched in a single round trip for multiple rows when using array binds or defines.

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

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

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

Figure 8-1 print_media Table



PRINT_MEDIA Table

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

8.3 Data Interface for LOBs in Java

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

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

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

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

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

The following code snippets work with all JDBC drivers:

Bind:

This is for the non-streaming mode:

```
...
String sql = "insert into print_media (product_id, ad_id, ad_final_text)" +
             " values (:1, :2, :3)";
    PreparedStatement pstmt = conn.prepareStatement(sql);
    pstmt.setInt( 1, 2 );
    pstmt.setInt( 2, 20);
    pstmt.setString( 3, "Java string" );
    int rows = pstmt.executeUpdate();
...
```

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

For the streaming mode, the same code as the preceding works, except that the `setString()` statement is replaced by one of the following:

```
pstmt.setCharacterStream( 3, new LabeledReader(), 1000000 );
pstmt.setAsciiStream( 3, new LabeledAsciiInputStream(), 1000000 );
```

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

Here, `LabeledReader()` and `LabeledAsciiInputStream()` produce character and ASCII streams respectively. If `ad_finaltext` were a `BLOB` column instead of a `CLOB`, then the preceding example works if the bind is of type `RAW`:

```
pstmt.setBytes( 3, <some byte[] array> );
pstmt.setBinaryStream( 3, new LabeledInputStream(), 1000000 );
```

Here, `LabeledInputStream()` produces a binary stream.

Define:

For non-streaming mode:

```
OracleStatement stmt = (OracleStatement)(conn.createStatement());
  stmt.defineColumnType( 1, Types.VARCHAR );
  ResultSet rst = stmt.executeQuery("select ad_finaltext from print_media" );
  while( rst.next() )
     {
       String s = rst.getString( 1 );
       System.out.println( s );
     }
```

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

For streaming mode:

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

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

If `ad_finaltext` were a `BLOB` column instead of a `CLOB`, then the preceding examples work if the define is of type `LONGVARBINARY`:

```
...
OracleStatement stmt = (OracleStatement)conn.createStatement();

stmt.defineColumnType( 1, Types.INTEGER );
```

```
stmt.defineColumnType( 2, Types.LONGVARBINARY );

ResultSet rset = stmt.executeQuery("SELECT ID, LOBCOL FROM LOBTAB");
    while(rset.next())
    {
             /* using getBytes() */
             /*
             byte[] b = rset.getBytes("LOBCOL");
        System.out.println("ID: " + rset.getInt("ID") + "  length: " +
b.length);
             */

            /* using getBinaryStream() */
            InputStream byte_stream = rset.getBinaryStream("LOBCOL");
            byte [] b = new byte [100000];
            int b_len = byte_stream.read(b);
            System.out.println("ID: " + rset.getInt("ID") + "  length: " +
b_len);

            byte_stream.close();
        }
...
```

> **See Also:**
>
> Working with Large Objects and SecureFiles

8.4 Data Interface for LOBs in OCI

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

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

> **Note:**
>
> You can use array bind and define interfaces to insert and select multiple rows with LOBs in one round trip.

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

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

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

8.4.1 Binding a LOB in OCI

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

- Regular, piecewise, and callback binds for `INSERT` and `UPDATE` operations
- Array binds for `INSERT` and `UPDATE` operations
- Parameter passing across PL/SQL and OCI boundaries

Piecewise operations can be performed by polling or by providing a callback. To support these operations, the following OCI functions accept the `LONG` and LOB data types listed in Table 8-1.

- `OCIBindByName()` and `OCIBindByPos()`

  These functions create an association between a program variable and a placeholder in the SQL statement or a PL/SQL block for `INSERT` and `UPDATE` operations.

- `OCIBindDynamic()`

  You use this call to register callbacks for dynamic data allocation for `INSERT` and `UPDATE` operations

- `OCIStmtGetPieceInfo()` and `OCIStmtSetPieceInfo()`

  These calls are used to get or set piece information for piecewise operations.

8.4.2 Defining a LOB in OCI

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

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

You can use the following functions

- `OCIDefineByPos()`

This call associates an item in a `SELECT` list with the type and output data buffer.

- `OCIDefineDynamic()`

  This call registers user callbacks for `SELECT` operations if the `OCI_DYNAMIC_FETCH` mode was selected in `OCIDefineByPos()` function call. You can use the `OCIDataServerLengthGet()` function to retrieve LOB length while using dynamic define callback.

When you use these functions with LOB types, the LOB data, and not the locator, is selected into your buffer. Note that in OCI, you cannot specify the amount you want to read using the data interface for LOBs. You can only specify the buffer length of your buffer. The database only reads whatever amount fits into your buffer and the data is truncated.

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

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

When the client character set is in a multibyte format, functions included in the data interface operate the same way with LOB datatypes as they do for `VARCHAR2` data types as follows:

- For a *piecewise* fetch in a multibyte character set, a multibyte character could be cut in the middle, with some bytes at the end of one buffer and remaining bytes in the next buffer.
- For a *regular* fetch, if the buffer cannot hold all bytes of the last character, then Oracle returns as many bytes as fit into the buffer, hence returning partial characters.

8.4.4 Getting LOB Length

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

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

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

8.4.5 Using OCI Functions to Perform INSERT or UPDATE on LOB Columns

This section discusses the various techniques you can use to perform `INSERT` or `UPDATE` operations on LOB columns or attributes using the data interface.

The operations described in this section assume that you have initialized the OCI environment and allocated all necessary handles.

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

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

8.4.5.1 Performing Simple INSERT or UPDATE Operations in One Piece

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

1. Call `OCIStmtPrepare()` to prepare the statement in `OCI_DEFAULT` mode.
2. Call `OCIBindByName()` or `OCIBindbyPos()` in `OCI_DEFAULT` mode to bind a placeholder for LOB as character data or binary data.
3. Call `OCIStmtExecute()` to do the actual `INSERT` or `UPDATE` operation.

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

```
void simple_insert()
{
/* Insert of data into LOB attributes is allowed. */
   ub1 buffer[8000];
   text *insert_sql = (text *)"INSERT INTO Print_media (ad_header) \
               VALUES (adheader_typ(NULL, NULL, NULL,:1))";
   OCIStmtPrepare(stmthp, errhp, insert_sql, strlen((char*)insert_sql),
            (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
   OCIBindByPos(stmthp, &bindhp[0], errhp, 1, (dvoid *)buffer, 2000,
             SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
   OCIStmtExecute(svchp, stmthp, errhp, 1, 0, (const OCISnapshot*) 0,
                  (OCISnapshot*)0, OCI_DEFAULT);
}
```

8.4.5.2 Using Piecewise INSERT and UPDATE Operations with Polling

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

1. Call `OCIStmtPrepare()` to prepare the statement in `OCI_DEFAULT` mode.
2. Call `OCIBindByName()` or `OCIBindbyPos()` in `OCI_DATA_AT_EXEC` mode to bind a LOB as character data or binary data.
3. Call `OCIStmtExecute()` in default mode. Do each of the following in a loop while the value returned from `OCIStmtExecute()` is `OCI_NEED_DATA`. Terminate your loop when the value returned from `OCIStmtExecute()` is `OCI_SUCCESS`.
   - Call `OCIStmtGetPieceInfo()` to retrieve information about the piece to be inserted.
   - Call `OCIStmtSetPieceInfo()` to set information about piece to be inserted.

The following example illustrates using piecewise `INSERT` with polling using the data interface for LOBs.

```
void piecewise_insert()
{
  text *sqlstmt = (text *)"INSERT INTO Print_media(Product_id, Ad_id,\
                  Ad_sourcetext) VALUES (:1, :2, :3)";
  ub2 rcode;
  ub1 piece, i;
  word product_id = 2004;
  word ad_id = 2;
  ub4 buflen;
  char buf[5000];

  OCIStmtPrepare(stmthp, errhp, sqlstmt, (ub4)strlen((char *)sqlstmt),
                 (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
  OCIBindByPos(stmthp, &bndhp[0], errhp, (ub4) 1,
               (dvoid *) &product_id, (sb4) sizeof(product_id), SQLT_INT,
               (dvoid *) 0, (ub2 *)0, (ub2 *)0,
               (ub4) 0, (ub4 *) 0, (ub4) OCI_DEFAULT);
  OCIBindByPos(stmthp, &bndhp[1], errhp, (ub4) 2,
               (dvoid *) &ad_id, (sb4) sizeof(ad_id), SQLT_INT,
               (dvoid *) 0, (ub2 *)0, (ub2 *)0,
               (ub4) 0, (ub4 *) 0, (ub4) OCI_DEFAULT);
  OCIBindByPos(stmthp, &bndhp[2], errhp, (ub4) 3,
               (dvoid *) 0, (sb4) 15000, SQLT_LNG,
               (dvoid *) 0, (ub2 *)0, (ub2 *)0,
               (ub4) 0, (ub4 *) 0, (ub4) OCI_DATA_AT_EXEC);

  i = 0;
  while (1)
  {
    i++;
    retval = OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                            (CONST OCISnapshot*) 0, (OCISnapshot*) 0,
                            (ub4) OCI_DEFAULT);
    switch(retval)
    {
    case OCI_NEED_DATA:
      memset((void *)buf, (int)'A'+i, (size_t)5000);
      buflen = 5000;
      if (i == 1) piece = OCI_FIRST_PIECE;
      else if (i == 3) piece = OCI_LAST_PIECE;
      else piece = OCI_NEXT_PIECE;

      if (OCIStmtSetPieceInfo((dvoid *)bndhp[2],
                              (ub4)OCI_HTYPE_BIND, errhp, (dvoid *)buf,
                              &buflen, piece, (dvoid *) 0, &rcode))
        {
          printf("ERROR: OCIStmtSetPieceInfo: %d \n", retval);
          break;
        }

      break;
    case OCI_SUCCESS:
      break;
```

```
    default:
      printf( "oci exec returned %d \n", retval);
      report_error(errhp);
      retval = OCI_SUCCESS;
    } /* end switch */
    if (retval == OCI_SUCCESS)
      break;
  } /* end while(1) */
}
```

8.4.5.3 Performing Piecewise INSERT and UPDATE Operations with Callback

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

1. Call `OCIStmtPrepare()` to prepare the statement in `OCI_DEFAULT` mode.
2. Call `OCIBindByName()` or `OCIBindbyPos()` in `OCI_DATA_AT_EXEC` mode to bind a placeholder for the LOB column as character data or binary data.
3. Call `OCIBindDynamic()` to specify the callback.
4. Call `OCIStmtExecute()` in default mode.

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

The following example illustrates binding character data to LOB columns using a piecewise `INSERT` with callback:

```
void callback_insert()
{
  word buflen = 15000;
  word product_id = 2004;
  word ad_id = 3;
  text *sqlstmt = (text *) "INSERT INTO Print_media(Product_id, Ad_id,\
                  Ad_sourcetext) VALUES (:1, :2, :3)";
  word pos = 3;

  OCIStmtPrepare(stmthp, errhp, sqlstmt, (ub4)strlen((char *)sqlstmt),
                 (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT)

  OCIBindByPos(stmthp, &bndhp[0], errhp, (ub4) 1,
               (dvoid *) &product_id, (sb4) sizeof(product_id), SQLT_INT,
               (dvoid *) 0, (ub2 *)0, (ub2 *)0,
               (ub4) 0, (ub4 *) 0, (ub4) OCI_DEFAULT);
  OCIBindByPos(stmthp, &bndhp[1], errhp, (ub4) 2,
               (dvoid *) &ad_id, (sb4) sizeof(ad_id), SQLT_INT,
               (dvoid *) 0, (ub2 *)0, (ub2 *)0,
               (ub4) 0, (ub4 *) 0, (ub4) OCI_DEFAULT);
  OCIBindByPos(stmthp, &bndhp[2], errhp, (ub4) 3,
               (dvoid *) 0, (sb4) buflen, SQLT_CHR,
               (dvoid *) 0, (ub2 *)0, (ub2 *)0,
               (ub4) 0, (ub4 *) 0, (ub4) OCI_DATA_AT_EXEC);

  OCIBindDynamic(bndhp[2], errhp, (dvoid *) (dvoid *) &pos,
                 insert_cbk, (dvoid *) 0, (OCICallbackOutBind) 0);
```

```
  OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                 (const OCISnapshot*) 0, (OCISnapshot*) 0,
                 (ub4) OCI_DEFAULT);
} /* end insert_data() */

/* Inbind callback to specify input data. */
static sb4 insert_cbk(dvoid *ctxp, OCIBind *bindp, ub4 iter, ub4 index,
                      dvoid **bufpp, ub4 *alenpp, ub1 *piecep, dvoid **indpp)
{
  static int a = 0;
  word   j;
  ub4    inpos = *((ub4 *)ctxp);
  char   buf[5000];

  switch(inpos)
  {
  case 3:
    memset((void *)buf, (int) 'A'+a, (size_t) 5000);
    *bufpp = (dvoid *) buf;
    *alenpp = 5000 ;
    a++;
    break;
  default: printf("ERROR: invalid position number: %d\n", inpos);
  }

  *indpp = (dvoid *) 0;
  *piecep = OCI_ONE_PIECE;
  if (inpos == 3)
  {
    if (a<=1)
    {
      *piecep = OCI_FIRST_PIECE;
      printf("Insert callback: 1st piece\n");
    }
    else if (a<3)
    {
      *piecep = OCI_NEXT_PIECE;
      printf("Insert callback: %d'th piece\n", a);
    }
    else {
      *piecep = OCI_LAST_PIECE;
      printf("Insert callback: %d'th piece\n", a);
      a = 0;
    }
  }
  return OCI_CONTINUE;
}
```

8.4.5.4 Performing Array INSERT and UPDATE Operations

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

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

The following example illustrates binding character data for LOB columns using an array `INSERT` operation:

```
void array_insert()
{
  ub4 i;
  word buflen;
  word arrbuf1[5];
  word arrbuf2[5];
  text arrbuf3[5][5000];
  text *insstmt = (text *)"INSERT INTO Print_media(Product_id, Ad_id,\
                  Ad_sourcetext) VALUES (:PID, :AID, :SRCTXT)";

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

  OCIBindByName(stmthp, &bndhp[0], errhp,
                (text *) ":PID", (sb4) strlen((char *) ":PID"),
                (dvoid *) &arrbuf1[0], (sb4) sizeof(arrbuf1[0]), SQLT_INT,
                (dvoid *) 0, (ub2 *)0, (ub2 *) 0,
                (ub4) 0, (ub4 *) 0, (ub4) OCI_DEFAULT);

  OCIBindByName(stmthp, &bndhp[1], errhp,
                (text *) ":AID", (sb4) strlen((char *) ":AID"),
                (dvoid *) &arrbuf2[0], (sb4) sizeof(arrbuf2[0]), SQLT_INT,
                (dvoid *) 0, (ub2 *)0, (ub2 *) 0,
                (ub4) 0, (ub4 *) 0, (ub4) OCI_DEFAULT);

  OCIBindByName(stmthp, &bndhp[2], errhp,
                (text *) ":SRCTXT", (sb4) strlen((char *) ":SRCTXT"),
                (dvoid *) arrbuf3[0], (sb4) sizeof(arrbuf3[0]), SQLT_CHR,
                (dvoid *) 0, (ub2 *) 0, (ub2 *) 0,
                (ub4) 0, (ub4 *) 0, (ub4) OCI_DEFAULT);

  OCIBindArrayOfStruct(bndhp[0], errhp sizeof(arrbuf1[0]),
                       indsk, rlsk, rcsk);
  OCIBindArrayOfStruct(bndhp[1], errhp, sizeof(arrbuf2[0]),
                       indsk, rlsk, rcsk);
  OCIBindArrayOfStruct(bndhp[2], errhp, sizeof(arrbuf3[0]),
                       indsk, rlsk, rcsk);

  for (i=0; i<5; i++)
  {
```

```
    arrbuf1[i] = 2004;
    arrbuf2[i] = i+4;
    memset((void *)arrbuf3[i], (int)'A'+i, (size_t)5000);
  }
  OCIStmtExecute(svchp, stmthp, errhp, (ub4) 5, (ub4) 0,
                 (const OCISnapshot*) 0, (OCISnapshot*) 0,
                 (ub4) OCI_DEFAULT);

}
```

8.4.6 Using OCI Data Interface to Fetch LOB Data

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

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

8.4.6.1 Performing Simple Fetch Operations in One Piece

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

1. Call `OCIStmtPrepare()` to prepare the `SELECT` statement in `OCI_DEFAULT` mode.
2. Call `OCIDefineByPos()` to define a select list position in `OCI_DEFAULT` mode to define a LOB as character data or binary data.
3. Call `OCIStmtExecute()` to run the `SELECT` statement.
4. Call `OCIStmtFetch()` to do the actual fetch.

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

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

```
                Product_id = 2004";

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

  retval = OCIStmtExecute(svchp, stmthp, errhp, (ub4) 0, (ub4) 0,
                          (const OCISnapshot*) 0, (OCISnapshot*) 0,
                          (ub4) OCI_DEFAULT);
  while (retval == OCI_SUCCESS || retval == OCI_SUCCESS_WITH_INFO)
  {
    OCIDefineByPos(stmthp, &defhp, errhp, (ub4) 1, (dvoid *) buf,
                   (sb4) sizeof(buf), (ub2) SQLT_CHR, (dvoid *) 0,
                   (ub2 *) 0, (ub2 *) 0, (ub4) OCI_DEFAULT);
    retval = OCIStmtFetch(stmthp, errhp, (ub4) 1,
                          (ub4) OCI_FETCH_NEXT, (ub4) OCI_DEFAULT);
    if (retval == OCI_SUCCESS || retval == OCI_SUCCESS_WITH_INFO)
      printf("buf = %.*s\n", 15000, buf);
  }
}
```

8.4.6.2 Performing a Piecewise Fetch with Polling

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

1. Call `OCIStmtPrepare()` to prepare the `SELECT` statement in `OCI_DEFAULT` mode.
2. Call `OCIDefinebyPos()` to define a select list position in `OCI_DYNAMIC_FETCH` mode to define the LOB column as character data or binary data.
3. Call `OCIStmtExecute()` to run the `SELECT` statement.
4. Call `OCIStmtFetch()` in default mode. Optionally, you can use `OCIServerDataLengthGet()` to get the LOB length and use it to allocate the buffer to hold the LOB data. Do each of the following in a loop while the value returned from `OCIStmtFetch()` is `OCI_NEED_DATA`. Terminate your loop when the value returned from `OCIStmtFetch()` is `OCI_SUCCESS`.
   - Call `OCIStmtGetPieceInfo()` to retrieve information about the piece to be fetched.
   - Call `OCIStmtSetPieceInfo()` to set information about piece to be fetched.

The following example illustrates selecting a LOB column into a character buffer using a piecewise fetch with polling:

```
void piecewise_fetch()
{
  text buf[15000];
  ub4 buflen=5000;
  word retval;
  text *selstmt = (text *) "SELECT Ad_sourcetext FROM Print_media
                  WHERE Product_id = 2004 AND Ad_id = 2";

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

  OCIDefineByPos(stmthp, &dfnhp, errhp, (ub4) 1,
                 (dvoid *) NULL, (sb4) 100000, SQLT_LNG,
```

```
                (dvoid *) 0, (ub2 *) 0,
                (ub2 *) 0, (ub4) OCI_DYNAMIC_FETCH);

  retval = OCIStmtExecute(svchp, stmthp, errhp, (ub4) 0, (ub4) 0,
                          (CONST OCISnapshot*) 0, (OCISnapshot*) 0,
                          (ub4) OCI_DEFAULT);

  retval = OCIStmtFetch(stmthp, errhp, (ub4) 1 ,
                        (ub2) OCI_FETCH_NEXT, (ub4) OCI_DEFAULT);

  while (retval != OCI_NO_DATA && retval != OCI_SUCCESS)
  {
    ub1 piece;
    ub4 iter;
    ub4 idx;

    genclr((void *)buf, 5000);
    switch(retval)
    {
    case OCI_NEED_DATA:
      OCIStmtGetPieceInfo(stmthp, errhp, &hdlptr, &hdltype,
                          &in_out, &iter, &idx, &piece);
      buflen = 5000;
      OCIStmtSetPieceInfo(hdlptr, hdltype, errhp,
                          (dvoid *) buf, &buflen, piece,
                          (CONST dvoid *) &indp1, (ub2 *) 0);
      retval = OCI_NEED_DATA;
      break;
    default:
      printf("ERROR: piece-wise fetching, %d\n", retval);
      return;
    } /* end switch */
    retval = OCIStmtFetch(stmthp, errhp, (ub4) 1 ,
                          (ub2) OCI_FETCH_NEXT, (ub4) OCI_DEFAULT);
    printf("Data : %.5000s\n", buf);
  } /* end while */
}
```

8.4.6.3 Performing a Piecewise with Callback

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

1. Call `OCIStmtPrepare()` to prepare the statement in `OCI_DEFAULT` mode.
2. Call `OCIDefinebyPos()` to define a select list position in `OCI_DYNAMIC_FETCH` mode to define the LOB column as character data or binary data.
3. Call `OCIStmtExecute()` to run the `SELECT` statement.
4. Call `OCIDefineDynamic()` to specify the callback.
5. Call `OCIStmtFetch()` in default mode.
6. Inside the callback, you can optionally use `OCIServerDataLengthGet()` to get the LOB length during the first fetch. You can use this value to allocate the buffer to hold LOB data

The following example illustrates selecting a LOB column into a LOB buffer when using a piecewise fetch with callback:

```
char buf[5000];
void callback_fetch()
{
  word outpos = 1;
  text *sqlstmt = (text *) "SELECT Ad_sourcetext FROM Print_media WHERE
                  Product_id = 2004 AND Ad_id = 3";

  OCIStmtPrepare(stmthp, errhp, sqlstmt, (ub4)strlen((char *)sqlstmt),
                 (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
  OCIDefineByPos(stmthp, &dfnhp[0], errhp, (ub4) 1,
                 (dvoid *) 0, (sb4)3 * sizeof(buf), SQLT_CHR,
                 (dvoid *) 0, (ub2 *)0, (ub2 *)0,
                 (ub4) OCI_DYNAMIC_FETCH);

  OCIDefineDynamic(dfnhp[0], errhp, (dvoid *) &outpos,
                   (OCICallbackDefine) fetch_cbk);

  OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                 (const OCISnapshot*) 0, (OCISnapshot*) 0,
                 (ub4) OCI_DEFAULT);
  buf[ 4999 ] = '\0';
  printf("Select callback: Last piece: %s\n", buf);
}

/* ------------------------------------------------------------ */
/* Fetch callback to specify buffers. */
/* ------------------------------------------------------------ */
static sb4 fetch_cbk(dvoid *ctxp, OCIDefine *dfnhp, ub4 iter, dvoid **bufpp,
                     ub4 **alenpp, ub1 *piecep, dvoid **indpp, ub2 **rcpp)
{
  static int a = 0;
  ub4 outpos = *((ub4 *)ctxp);
  ub4 len = 5000;
  switch(outpos)
  {
  case 1:
    a ++;
    *bufpp = (dvoid *) buf;
    *alenpp = &len;
    break;
  default:
    *bufpp = (dvoid *) 0;
    *alenpp = (ub4 *) 0;
    printf("ERROR: invalid position number: %d\n", outpos);
  }
  *indpp = (dvoid *) 0;
  *rcpp = (ub2 *) 0;

  buf[len] = '\0';
  if (a<=1)
  {
    *piecep = OCI_FIRST_PIECE;
    printf("Select callback: 0th piece\n");
```

```
  }
  else if (a<3)
  {
    *piecep = OCI_NEXT_PIECE;
    printf("Select callback: %d'th piece: %s\n", a-1, buf);
  }
  else {
    *piecep = OCI_LAST_PIECE;
    printf("Select callback: %d'th piece: %s\n", a-1, buf);
    a = 0;
  }
  return OCI_CONTINUE;
}
```

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

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

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

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

    if (buf_len > MAX_BUF_SZ)
      buf_len = MAX_BUF_SZ;

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

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

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

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

8.4.6.4 Performing an Array Fetch Operation

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

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

The following example illustrates selecting a LOB column into a character buffer using an array fetch:

```
void array_fetch()
{
  word i;
  text arrbuf[5][5000];
  text *selstmt = (text *) "SELECT Ad_sourcetext FROM Print_media WHERE
                  Product_id = 2004 AND Ad_id >=4";

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

  OCIStmtExecute(svchp, stmthp, errhp, (ub4) 0, (ub4) 0,
                 (const OCISnapshot*) 0, (OCISnapshot*) 0, (ub4) OCI_DEFAULT);

  OCIDefineByPos(stmthp, &defhp1, errhp, (ub4) 1,
                   (dvoid *) arrbuf[0], (sb4) sizeof(arrbuf[0]),
                   (ub2) SQLT_CHR, (dvoid *) 0,
                   (ub2 *) 0, (ub2 *) 0, (ub4) OCI_DEFAULT);

  OCIDefineArrayOfStruct(dfnhp1, errhp, sizeof(arrbuf[0]), indsk,
                         rlsk, rcsk);

  retval = OCIStmtFetch(stmthp, errhp, (ub4) 5,
                        (ub4) OCI_FETCH_NEXT, (ub4) OCI_DEFAULT);
```

```
  if (retval == OCI_SUCCESS || retval == OCI_SUCCESS_WITH_INFO)
  {
     printf("%.5000s\n", arrbuf[0]);
     printf("%.5000s\n", arrbuf[1]);
     printf("%.5000s\n", arrbuf[2]);
     printf("%.5000s\n", arrbuf[3]);
     printf("%.5000s\n", arrbuf[4]);
  }
}
```

8.4.7 PL/SQL and C Binds from OCI

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

When you call a PL/SQL procedure from OCI, and have an `IN` or `OUT` or `IN OUT` bind, you should be able to:

- Bind a variable as `SQLT_CHR` or `SQLT_LNG` where the formal parameter of the PL/SQL procedure is `SQLT_CLOB`, or
- Bind a variable as `SQLT_BIN` or `SQLT_LBI` where the formal parameter is `SQLT_BLOB`

The following two cases work:

Calling PL/SQL Out-binds in the "begin foo(:1); end;" Manner

Here is an example of calling PL/SQL out-binds in the "begin foo(:1); end;" Manner:

```
text *sqlstmt = (text *)"BEGIN get_lob(:c); END; " ;
```

Calling PL/SQL Out-binds in the "call foo(:1);" Manner

Here is an example of calling PL/SQL out-binds in the "call foo(:1);" manner:

```
text *sqlstmt = (text *)"CALL get_lob(:c);" ;
```

In both these cases, the rest of the program has these statements:

```
OCIStmtPrepare(stmthp, errhp, sqlstmt, (ub4)strlen((char *)sqlstmt),
               (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
   curlen = 0;
OCIBindByName(stmthp, &bndhp[3], errhp,
        (text *) ":c", (sb4) strlen((char *) ":c"),
        (dvoid *) buf5, (sb4) LONGLEN, SQLT_CHR,
        (dvoid *) 0, (ub2 *) 0, (ub2 *) 0,
        (ub4) 1, (ub4 *) &curlen, (ub4) OCI_DATA_AT_EXEC);
```

The PL/SQL procedure, `get_lob()`, is as follows:

```
procedure get_lob(c INOUT CLOB) is  -- This might have been column%type
  BEGIN
  ... /* The procedure body could be in PL/SQL or C*/
  END;
```