9

Locator Interface for LOBs

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

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

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

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

9.1 Before You Begin

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

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

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

9.1.1 Getting a LOB Locator

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

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

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

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

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

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

9.1.2 LOB Open and Close Operations

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

You can open and close a persistent or temporary LOB instance of any type: `BLOB`, `CLOB` or `NCLOB`. You open a LOB to achieve one or both of the following results:

- Open the LOB in read-only mode

  This ensures that the LOB (both the LOB locator and LOB value) cannot be changed in your session until you explicitly close the LOB. For example, you can open the LOB to ensure that the LOB is not changed by some other part of your program while you are using the LOB in a critical operation. After you perform the operation, you can then close the LOB.

- Open the LOB in read-write mode

  Opening a LOB in read-write mode defers any index maintenance on the LOB column until you close the LOB. Opening a LOB in read-write mode is only useful if there is a functional or domain index on the LOB column, and you do not want the database to perform index maintenance every time you write to the LOB. This technique can improve the performance of your application if you are doing several write operations on the LOB while it is open. Note that any index on the LOB column is not valid until you explicitly close the LOB.

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

The open state of a LOB is associated with the LOB instance, not the LOB locator. The locator does not save any information indicating whether the LOB instance that it points to is open.

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

- Between DML statements that start a transaction, including `SELECT ... FOR UPDATE` and `COMMIT`.
- Within an autonomous transaction block.
- Before the end of a session (when there is no transaction in progress in the session).

If you do not explicitly close the LOB instance, then it is implicitly closed at the end of the session and no index triggers are fired, which means that any indexes on the LOB column are not updated. In this situation, you must rebuild your indexes on the LOB column.

Committing a transaction on the open LOB instance causes an error. When this error occurs, the LOB instance is closed implicitly, any modifications to the LOB instance are saved, and the transaction is committed, but any indexes on the LOB column are not updated. In this situation, you must rebuild your indexes on the LOB column.

If you subsequently rollback the transaction, then the LOB instance is rolled back to its previous state, but the LOB instance is no longer explicitly open.

Keep track of the open or closed state of LOBs that you explicitly open. The following actions cause an error:

- Explicitly opening a LOB instance that has been explicitly open earlier.
- Explicitly closing a LOB instance that is has been explicitly closed earlier.

This occurs whether you access the LOB instance using the same locator or different locators.

9.1.3 Read and Write at Chunk Boundaries

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

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

9.1.4 Prefetching LOB Data and Length

In most clients like JDBC, OCI and ODP.NET, the number of server round trips can be reduced by prefetching part of the data and metadata (length and chunk size) along with the LOB locator during the fetch. This applies to persistent LOBs, temporary LOBs, and `BFILE`s.

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

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

9.1.5 Determining Character Set ID

Some LOB APIs such as `DBMS_LOB.LOADCLOBFROMFILE`, `OCILobRead2()` and `OCILobWrite2()` take in a character set ID as an input. To determine the character set ID, you must know the character set name.

A user can select from the `V$NLS_VALID_VALUES` view, which lists the names of the character sets that are valid as database and national character sets. Then call the function `NLS_CHARSET_ID` with the desired character set name as the one string argument. The character set ID is returned as an integer.

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

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

9.1.6 LOB APIs

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

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

Table 9-1 Operations supported by LOB APIs

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

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

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

See Also:

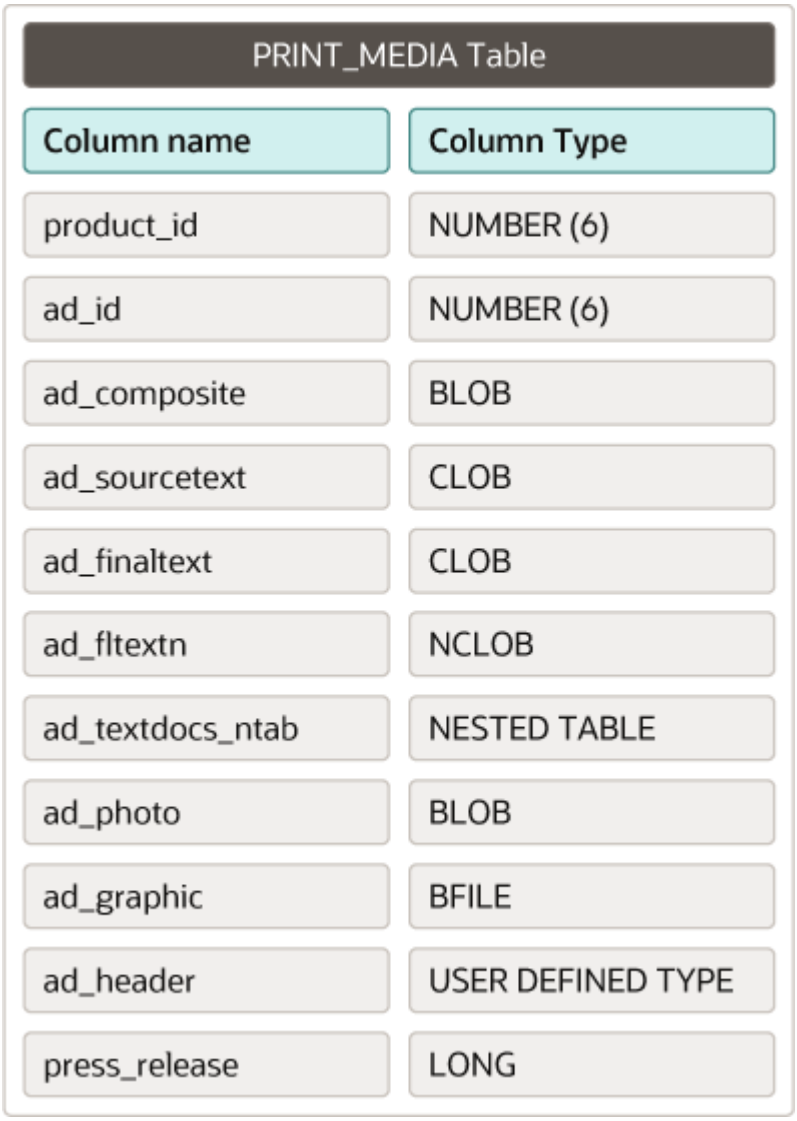
- Temporary LOBs
- BFILEs
- Comparing the LOB Interfaces

Note:

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

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

Figure 9-1 print_media table



9.2 PL/SQL API for LOBs

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

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

Guidelines for Offset and Amount Parameters in DBMS_LOB Operations

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

- For character data in all formats, either in fixed-width or variable-width, the `amount` and `offset` parameters are in characters. This applies to operations on `CLOB` and `NCLOB` data types.
- For binary data, the `offset` and `amount` parameters are in bytes. This applies to operations on `BLOB` data types.
- When using the `DBMS_LOB.READ` procedure, the `amount` parameter should be less than or equal to the size of the buffer, which is limited to 32K. However, the `amount` parameter can be larger than the size of the LOB data.

Table 9-2 DBMS_LOB functions and procedures for LOBs

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

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

Category	Function/Procedure	Description
	SETCONTENTTYPE	Sets the content string for a SecureFiles.
	FRAGMENT_DELETE	Delete the data from the LOB at the given offset for the given length
	FRAGMENT_INSERT	Insert the given data (< 32KBytes) into the LOB at the given offset
	FRAGMENT_MOVE	Move the given amount of bytes from the given offset to the new given offset
	FRAGMENT_REPLACE	Replace the data at the given offset with the given data (< 32kBytes)

Example 9-1 PL/SQL API for LOBs

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

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

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

    -- the select statement is defined with FOR UPDATE so that we can write
to it
   SELECT ad_finaltext INTO clob2 FROM print_media
     WHERE product_id = 1 AND ad_id =1 FOR UPDATE;
   /* Note that all the writes to clob2 will get reflected in the column */

   /*----------------------------------------------------------------*/
   /*---------------------- Sanity Checking -------------------------*/
   /*----------------------------------------------------------------*/
   if DBMS_LOB.ISSECUREFILE(clob1) = TRUE then
     DBMS_OUTPUT.PUT_LINE('CLOB1 is SECUREFILE');
   else
```

```
      DBMS_OUTPUT.PUT_LINE('CLOB1 is BASICFILE');
    end if;

    /*---------------------------------------------------------------------*/
    /*----------------------- Open ----------------------------------------*/
    /*---------------------------------------------------------------------*/
    /* Open clob1 for READs and clob2 for WRITES */
    DBMS_LOB.OPEN(clob1, DBMS_LOB.LOB_READONLY);
    DBMS_LOB.OPEN(clob2, DBMS_LOB.LOB_READWRITE);

    /*---------------------------------------------------------------------*/
    /*-------------------- Reading from a LOB -----------------------------*/
    /*---------------------------------------------------------------------*/
    DBMS_OUTPUT.PUT_LINE('storage limit : ' ||
dbms_lob.get_storage_limit(clob1));
    DBMS_OUTPUT.PUT_LINE('chunk size : ' || dbms_lob.getchunksize(clob1));

    loblen1 := DBMS_LOB.GETLENGTH(clob1);
    DBMS_OUTPUT.PUT_LINE('length : ' || loblen1);

    DBMS_LOB.READ(clob1, buflen, 1, buf);
    DBMS_OUTPUT.PUT_LINE('read : LOB data : ' || buf);
    DBMS_OUTPUT.PUT_LINE('New buflen : ' || buflen);

    DBMS_OUTPUT.PUT_LINE('substr : ' || dbms_lob.substr(clob1, 30, 1));
    DBMS_OUTPUT.PUT_LINE('instr : ' ||
                         DBMS_LOB.INSTR(clob1, 'review of the document', 1,
3));

    /*---------------------------------------------------------------------*/
    /*-------------------- Modifying a LOB --------------------------------*/
    /*---------------------------------------------------------------------*/
    DBMS_LOB.WRITE(clob2, buflen, 10, buf);
    DBMS_LOB.WRITEAPPEND(clob2, buflen, buf);
    buflen := 10;
    DBMS_LOB.ERASE(clob2, buflen, 10);
    DBMS_LOB.TRIM(clob2, 50);

    /* Print the LOB just modified */
    buflen := 32760;
    DBMS_LOB.READ(clob2, buflen, 1, buf);
    DBMS_OUTPUT.PUT_LINE('read : LOB data : ' || buf);
    DBMS_OUTPUT.PUT_LINE('New buflen : ' || buflen);

    /* Error because clob1 is open in READ mode */
    -- DBMS_LOB.WRITE(clob1, buflen, 10, buf);

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

    retval := DBMS_LOB.COMPARE(clob1, clob2, 100, 1, 1);
    if (retval < 0) then
      DBMS_OUTPUT.PUT_LINE('clob1 is smaller');
    elsif (retval = 0) then
      DBMS_OUTPUT.PUT_LINE('both clobs are equal');
```

```
   else
     DBMS_OUTPUT.PUT_LINE('clob1 is larger');
   end if;

   DBMS_OUTPUT.PUT_LINE('length before append: ' ||
DBMS_LOB.GETLENGTH(clob2));
   DBMS_LOB.APPEND(clob2, clob1);
   DBMS_OUTPUT.PUT_LINE('length after append: ' || DBMS_LOB.GETLENGTH(clob2));

   DBMS_OUTPUT.PUT_LINE('----------- LOB COPY operation --------');
   DBMS_LOB.COPY(clob2, clob1, loblen1, 100, 1);
   DBMS_OUTPUT.PUT_LINE('length after copy: ' || DBMS_LOB.GETLENGTH(clob2));

   /*---------------------------------------------------------------------*/
   /*------------------- Convert CLOB to a BLOB ----------------------*/
   /*---------------------------------------------------------------------*/
   DBMS_LOB.CREATETEMPORARY( blob1, false );
   dst := 1;
   src := 1;
   amt := 5;
   DBMS_LOB.CONVERTTOBLOB(blob1, clob2, amt, dst, src, DBMS_LOB.DEFAULT_CSID,
                          lang, warn);
   DBMS_OUTPUT.PUT_LINE(' Source offset returned      ' || src ) ;
   DBMS_OUTPUT.PUT_LINE(' Destination offset returned ' || dst ) ;
   DBMS_OUTPUT.PUT_LINE(' Length of CLOB      ' ||
dbms_lob.getlength(clob2) ) ;
   DBMS_OUTPUT.PUT_LINE(' Length of BLOB      ' ||
dbms_lob.getlength(blob1) ) ;
   DBMS_OUTPUT.PUT_LINE(' Warning returned    ' || warn);
   DBMS_OUTPUT.PUT_LINE(' OUTPUT BLOB contents = ' || rawtohex(blob1));

   /*---------------------------------------------------------------------*/
   /*-------------------- Convert BLOB to a CLOB ----------------------*/
   /*---------------------------------------------------------------------*/
   DBMS_LOB.CREATETEMPORARY( clob3, false );
   dst := 1;
   src := 1;
   amt := 4;
   DBMS_LOB.CONVERTTOCLOB(clob3, blob1, amt, dst, src, DBMS_LOB.DEFAULT_CSID,
                          lang, warn);
   DBMS_OUTPUT.PUT_LINE(' Source offset returned      ' || src ) ;
   DBMS_OUTPUT.PUT_LINE(' Destination offset returned ' || dst ) ;
   DBMS_OUTPUT.PUT_LINE(' Length of BLOB      ' ||
DBMS_LOB.GETLENGTH(blob1) ) ;
   DBMS_OUTPUT.PUT_LINE(' Length of CLOB      ' ||
DBMS_LOB.GETLENGTH(clob3) ) ;
   DBMS_OUTPUT.PUT_LINE(' Warning returned    ' || warn);
   DBMS_OUTPUT.PUT_LINE(' INPUT BLOB contents =  ' || rawtohex(blob1));
   DBMS_OUTPUT.PUT_LINE(' OUTPUT CLOB contents = ' || clob3);

   /*---------------------------------------------------------------------*/
   /*------------------------ Close ----------------------------------*/
   /*---------------------------------------------------------------------*/
   DBMS_OUTPUT.PUT_LINE('------------- CLOSE ---------------');
   DBMS_LOB.CLOSE(clob2);
```

```
   if (DBMS_LOB.ISOPEN(clob1) = 1) then
     DBMS_LOB.CLOSE(clob1);
   END if;

   COMMIT;
END;
/
```

Example 9-2 PL/SQL APIs for SecureFile specific operations

```
conn pm/pm

-- alter the table to make lob storage as securefile
-- assume tablespace tbs_1 is ASSM
alter table print_media move
lob(ad_composite)  store as securefile (deduplicate compress tablespace tbs_1)
lob(ad_sourcetext) store as securefile (compress tablespace tbs_1)
lob(ad_finaltext)  store as securefile (compress tablespace tbs_1)
lob(ad_photo)      store as securefile (tablespace tbs_1);

SET SERVEROUTPUT ON

DECLARE
 clob1             CLOB;
 blob1             BLOB;
 result            BINARY_INTEGER;

 /* --- variables for setcontenttype, getcontenttype ----*/
 get_media_type    VARCHAR2(128);
 set_media_type    VARCHAR2(128);

 /* --- variables for delta operations --------*/
 amount            INTEGER;
 offset            INTEGER;
 buffer            VARCHAR2(30);
 readbuf           VARCHAR2(50);
 read_amt          INTEGER;
 src_offset        INTEGER;
 dest_offset       INTEGER;
 amount_old        INTEGER;
BEGIN
 -- fetch clob, blob values
 SELECT ad_sourcetext, ad_composite
 INTO   clob1, blob1
 FROM   print_media
 WHERE  product_id = 2056 FOR UPDATE;

 /*-------------------------------------------------------------------*/
 /*---------------------- Get Options --------------------------------*/
 /*-------------------------------------------------------------------*/
 -- check whether compress option is enabled
 result := DBMS_LOB.GETOPTIONS(clob1, DBMS_LOB.OPT_COMPRESS);
 DBMS_OUTPUT.PUT_LINE('Get compress option on ad_sourcetext: '||result);

 -- check whether compress + deduplicate is enabled
```

```
 result := DBMS_LOB.GETOPTIONS(blob1, DBMS_LOB.OPT_DEDUPLICATE +
                                      DBMS_LOB.OPT_COMPRESS);
 DBMS_OUTPUT.PUT_LINE('Get compress + deduplicate option on ad_composite: '||
result);

 /*---------------------------------------------------------------------*/
 /*---------------------- Set Options ----------------------------------*/
 /*---------------------------------------------------------------------*/
 -- turn off compression
 DBMS_LOB.SETOPTIONS(clob1, DBMS_LOB.OPT_COMPRESS, DBMS_LOB.COMPRESS_OFF);
 -- getoptions should be 0 now
 result := DBMS_LOB.GETOPTIONS(clob1, DBMS_LOB.OPT_COMPRESS);
 DBMS_OUTPUT.PUT_LINE('Compress option on clob1: '||result);

 -- turn off deduplication
 DBMS_LOB.SETOPTIONS(blob1, DBMS_LOB.OPT_DEDUPLICATE,
DBMS_LOB.DEDUPLICATE_OFF);
 -- getoptions should be 0 now
 result := DBMS_LOB.GETOPTIONS(blob1, DBMS_LOB.OPT_DEDUPLICATE);
 DBMS_OUTPUT.PUT_LINE('Deduplicate option on blob1: '||result);

 /*---------------------------------------------------------------------*/
 /*----------- Getcontenttype, Setcontenttype ------------------------*/
 /*---------------------------------------------------------------------*/
 -- get contenttype -- should be null as content type is not set yet
 DBMS_OUTPUT.PUT_LINE(CHR(10)||'clob1 contenttype: ' ||
dbms_lob.getcontenttype(clob1));

 set_media_type := 'text/plain';
 DBMS_LOB.SETCONTENTTYPE(clob1, set_media_type);

 DBMS_OUTPUT.PUT_LINE('Clob1 contenttype: ' ||
dbms_lob.getcontenttype(clob1));

 -- setcontenttype for blob
 DBMS_OUTPUT.PUT_LINE('blob1 contenttype: ' ||
dbms_lob.getcontenttype(blob1));
 set_media_type := 'photo/jpeg';
 DBMS_LOB.SETCONTENTTYPE(blob1, set_media_type);

 get_media_type := DBMS_LOB.GETCONTENTTYPE(blob1);
 DBMS_OUTPUT.PUT_LINE('Blob1 contenttype: ' || get_media_type);

 /*---------------------------------------------------------------------*/
 /*---------------------- Fragment Operations ------------------------*/
 /*------------------ Print Before Fragment Operations --------------*/
 read_amt  := 40;
 DBMS_LOB.READ(clob1, read_amt, 1, readbuf);
 DBMS_OUTPUT.PUT_LINE(CHR(10)||'Clob1 before fragment insert: '|| readbuf);
 DBMS_OUTPUT.PUT_LINE(CHR(10)||'Length of clob1 before fragment operations:
'|| dbms_lob.getlength(clob1));

 /*---------------------- Fragment Delete ----------------------------*/
 amount := 100;
 offset := 10;
 DBMS_LOB.FRAGMENT_DELETE(clob1, amount, offset);
```

```
 /*---------------------- Fragment Insert ----------------------------*/
 amount    := 29;
 offset    := 1;
 buffer    := '#Verify lob Delta operations#';
 DBMS_LOB.FRAGMENT_INSERT(clob1, amount, offset, buffer);

 /*---------------------- Fragment Move -----------------------------*/
 amount      := 29;
 src_offset  := 100;
 dest_offset := 1;

 -- fragment move
 DBMS_LOB.FRAGMENT_MOVE(clob1, amount, src_offset, dest_offset);

 /*---------------------- Fragment Replace --------------------------*/
 amount      := 25;
 amount_old  := 29;
 offset      := 100;
 buffer      := '$Verify fragment replace$';

 DBMS_LOB.FRAGMENT_REPLACE(clob1, amount_old, amount, offset,buffer);

 COMMIT;

 /*------------------ Verify After Fragment Operations --------------*/
 read_amt  := 40;
 DBMS_LOB.READ(clob1, read_amt, 1, readbuf);
 DBMS_OUTPUT.PUT_LINE(CHR(10)||'Clob1 after delta insert: '|| readbuf);
 DBMS_OUTPUT.PUT_LINE(CHR(10)||'Length of clob1 after fragment operations:
'|| dbms_lob.getlength(clob1));

EXCEPTION
 WHEN OTHERS THEN
   DBMS_OUTPUT.PUT_LINE(sqlerrm);
END;
/
```

9.3 JDBC API for LOBs

JDBC supports standard Java interfaces `java.sql.Clob` and `java.sql.Blob` for `CLOB`s and `BLOB`s respectively.

In JDBC, you do not deal with locators but instead use methods and properties in the Java APIs to perform operations on LOBs.

When `BLOB` and `CLOB` objects are retrieved as a part of an `ResultSet`, these objects represent LOB locators of the currently selected row. If the current row changes due to a move operation, for example, `rset.next()`, then the retrieved locator still refers to the original LOB row. You must call `getBLOB()`, `getCLOB()`, or `getBFILE()` on the `ResultSet` each time a move operation is made depending on whether the instance is a `BLOB`, `CLOB` or `BFILE`.

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

Prefetching of LOB Data

When using the JDBC client, the number of server round trips can be reduced by prefetching part of the data and metadata (length and chunk size) along with the LOB locator during the fetch. This applies to persistent LOBs, temporary LOBs, and `BFILE`s. For small to medium sized LOBs, Oracle recommends setting the prefetch length such that about majority of your LOBs are smaller than the prefetch size.

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

- -1 to disable prefetching
- 0 to enable prefetching for metadata only
- any value greater than 0 which represents the number of bytes for `BLOB`s and characters for `CLOB`s, to be prefetched along with the locator during fetch operations.

Use `prop.setProperty` to set the prefetch size for the session. The default session prefetch size is 32k for the JDBC Thin Driver.

```
prop.setProperty("oracle.jdbc.defaultLobPrefetchSize","64000");
```

You can overwrite the session level default prefetch size at the statement level as follows:

```
((OracleStatement)stmt).setLobPrefetchSize(100000);
```

You can use the following code snippet to fetch the prefetch size of a statement:

```
int pf = ((OracleStatement)stmt).getLobPrefetchSize() ;
```

You can overwrite the session level default prefetch size at the column level as follows:

```
((OracleStatement)stmt).defineColumnType(1, OracleTypes.CLOB, /
*lobPrefetchSize*/
      32000);
```

> **Note:**
>
> About Prefetching LOB Data

Table 9-3 JDBC methods for LOBs

Category	Function / Procedure	Description
Miscellaneous	`empty_lob()`	Creates an empty LOB
	`isSecureFile()`	Finds out if the `BLOB` or `CLOB` locator is a SecureFile

Table 9-3 (Cont.) JDBC methods for LOBs

Category	Function / Procedure	Description
Open/Close	`open()`	Open a LOB
	`isOpen()`	Check if a LOB is open
	`close()`	Close the LOB
Read Operations	`length()`	Get the length of the LOB
	`getChunkSize()`	Get the optimum read/write size
	`getBytes()`	Read data from the BLOB starting at the specified offset
	`getBinaryStream()`	Streams the BLOB as a binary stream
	`getChars()`	Read data from the CLOB starting at the specified offset
	`getCharacterStream()`	Streams the CLOB as a character stream
	`getAsciiStream()`	Streams the CLOB as an ASCII stream
	`getSubString()`	Return part of the LOB value starting at the specified offset
	`position()`	Return the matching position of a pattern in a LOB
Modify Operations	`setBytes()`	Write data to the BLOB at a specified offset
	`setBinaryStream()`	Sets a binary stream that can be used to write to the `BLOB` value
	`setString()`	Write data to the CLOB at a specified offset
	`setCharacterStream()`	Sets a character stream that can be used to write to the `CLOB` value
	`setAsciiStream()`	Sets an ASCII stream that can be used to write to the `CLOB` value
	`truncate()`	Trim the LOB value to the specified shorter length
Operations involving multiple locators	dst = src	Assign LOB locator src to LOB locator dst

Example 9-3 JDBC API for LOBs

```
static void jdbc_lob_apis() throws Exception {

  System.out.println("Persistent LOBs Test in JDBC "+ TYPE);
  try(
      Connection con = getConnection();
      Statement       stmt = con.createStatement();
      )
  {

    ResultSet    rs      = null;
    Clob  c1      = null;
```

```
    Clob  c2      = null;
    Reader      in      = null;
    long        pos     = 0;
    long        len     = 0;

    rs = stmt.executeQuery("select ad_sourcetext from print_media where
product_id = 1");
    rs.next();
    c1 = rs.getCLOB(1);
    OracleClob c11 = (OracleClob)c1;
    rs.close();

    /*-------------------------------------------------------------------*/
    /*---------------------- Sanity Checking ----------------------------*/
    /*-------------------------------------------------------------------*/
    if (c11.isSecureFile())
      System.out.println("C1 is a Securefile LOB");
    else
      System.out.println("C1 is a Basicfile LOB");


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

    /*----------------------- Opening a CLOB ----------------------------*/
    c11.open(LargeObjectAccessMode.MODE_READONLY);

    /*-------------- Determining Whether a CLOB Is Open ----------------*/
    if (c11.isOpen())
      System.out.println("C11 is open!");
    else
      System.out.println("C11 is not open");

    /*----------------------- Closing a CLOB ----------------------------*/
    c11.close();

    /*-------------------------------------------------------------------*/
    /*--------------------- Reading from a LOB --------------------------*/
    /*-------------------------------------------------------------------*/

    /*------------------------ Get CLOB Length --------------------------*/
    len = c1.length();
    System.out.println("CLOB length = " + len);

    /*------------------------ Reading CLOB Data ------------------------*/
    char[] readBuffer = new char[6];
    in = c1.getCharacterStream();
    in.read(readBuffer,0,5);
    in.close();
    String lobContent = new String(readBuffer);
    System.out.println("Buffer with LOB contents: " + lobContent);

    /*------------------------ Substr of a CLOB -------------------------*/
    String subs = c1.getSubString(2, 5);
    System.out.println("LOB substring: " + subs);
```

```
    /*----------------------- Search for a pattern  --------------------*/
    pos = c1.position("aaa", 1);
    System.out.println("Pattern matched at position = " + pos);

    /*-----------------------------------------------------------------*/
    /*------------------- Modifying a LOB -----------------------------*/
    /*-----------------------------------------------------------------*/

    rs = stmt.executeQuery("select ad_sourcetext from print_media where
product_id = 1 for update");
    rs.next();
    c2 = rs.getClob(1);
    OracleClob c22 = (OracleClob)c2;

    /*-------------------- Write to a CLOB ----------------------------*/
    c22.open(LargeObjectAccessMode.MODE_READWRITE);
    c2.setString(3,"modified");
    String msubs = c2.getSubString(1, 15);
    System.out.println("Modified LOB substring: " + msubs);

    /*-------------------- Truncate a CLOB ----------------------------*/
    c2.truncate(20);
    len = c2.length();
    System.out.println("Truncated LOB len = " + len);
    c22.close();

  }
}
```

9.4 OCI API for LOBs

Oracle Call Interface (OCI) LOB functions enable you to access and make changes to LOBs in C.

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

Prefetching LOB Data in OCI

When using the OCI client, the number of server round trips can be reduced by prefetching part of the data and metadata (length and chunk size) along with the LOB locator during the fetch. This applies to persistent LOBs, temporary LOBs, and `BFILE`s. For small to medium sized LOBs, Oracle recommends setting the prefetch length such that about majority of your LOBs are smaller than the prefetch size.

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

Use the `OCIAttrSet()` function to set the prefetch size for the session. The default session prefetch size is 0.

```
default_lobprefetch_size = 32000;
OCIAttrSet(authp, OCI_HTYPE_SESSION, &default_lobprefetch_size , 0,
      OCI_ATTR_DEFAULT_LOBPREFETCH_SIZE, errhp));
```

You can overwrite the session level default prefetch size at the column level. For this, you should first set the column level attribute `OCI_ATTR_LOBPREFETCH_LENGTH` to `TRUE` and then set the column level prefetch size attribute `OCI_ATTR_LOBPREFETCH_SIZE` in the define handle to override the session level default lob prefetch size. The following code snippet demonstrates how to set the prefetch size at session level:

```
prefetch_length = TRUE;
status = OCIAttrSet(defhp, OCI_HTYPE_DEFINE, &prefetch_length, 0,
OCI_ATTR_LOBPREFETCH_LENGTH, errhp);

lpf_size = 32000;
OCIAttrSet(defhp, OCI_HTYPE_DEFINE, &lpf_size, sizeof(ub4),
OCI_ATTR_LOBPREFETCH_SIZE, errhp);
```

You can use the following code snippet to get the prefetch size of a define:

```
ub4 get_lpf_size = 0;
OCIAttrGet(defhp, OCI_HTYPE_DEFINE,&get_lpf_size,
0,OCI_ATTR_LOBPREFETCH_SIZE, errhp);
```

> **See Also:**
>
> User Session Handle Attributes

Fixed-width and Varying-width Character Set Rules for OCI

In OCI, for fixed-width client-side character sets, the following rules apply:

- `CLOB`s and `NCLOB`s: `offset` and `amount` parameters are always in characters
- `BLOB`s and `BFILE`s: `offset` and `amount` parameters are always in bytes

The following rules apply only to varying-width client-side character sets:

- **Offset parameter**:

  Regardless of whether the client-side character set is varying-width, the offset parameter is always as follows:

  - `CLOB`s and `NCLOB`s: in characters
  - `BLOB`s and `BFILE`s: in bytes
- **Amount parameter**:

  The amount parameter is always as follows:

  - When referring to a server-side LOB: in characters
  - When referring to a client-side buffer: in bytes

- **OCILobGetLength2()**:

  Regardless of whether the client-side character set is varying-width, the output length is as follows:

  - `CLOB`s and `NCLOB`s: in characters
  - `BLOB`s and `BFILE`s: in bytes

- **OCILobRead2()**:

  With client-side character set of varying-width, `CLOB`s and `NCLOB`s:

  - **Input amount** is in characters. Input amount refers to the number of characters to read from the server-side `CLOB` or `NCLOB`.
  - **Output amount** is in bytes. Output amount indicates how many bytes were read into the buffer `bufp`.

- **OCILobWrite2()**: With client-side character set of varying-width, `CLOB`s and `NCLOB`s:

  - **Input amount** is in bytes. The input amount refers to the number of bytes of data in the input buffer `bufp`.
  - **Output amount** is in characters. The output amount refers to the number of characters written into the server-side `CLOB` or `NCLOB`.

- **Amount Operation for OCILob Operations**: For operations such as `OCILobCopy2()`, `OCILobErase2()`, `OCILobLoadFromFile2()`, and `OCILobTrim2()`, the `amount` parameter is in characters for `CLOB`s and `NCLOB`s irrespective of the client-side character set because all these operations refer to the amount of LOB data on the server.

> **See Also:**
>
> Overview of Globalization Support

Amount Parameter

When using the `OCILobRead2()` and `OCILobWrite2()` functions, in order to read or write the entire LOB. you can set the input `amount` parameter as follows:

Table 9-4 Special Amount Parameter Setting to Read/Write the entire LOB

	OCILobRead2	OCILobWrite2
`piece = OCI_ONE_PIECE`	Set amount to `UB8MAXVAL` to read the entire LOB	
Streaming with Polling	Set amount to 0 to read entire data in a loop	Set amount to 0 to continue writing buffer size amount until `OCI_LAST_PIECE`
Streaming with Callback	Set amount 0 to ensure that the callback is called until the entire data is read	Set amount to 0 to ensure that the callback is called until `OCI_LAST_PIECE` is returned by the callback

Table 9-5 OCI Attributes on the OCILobLocator

ATTRIBUTE	OCIAttrSet	OCIAttrGet
OCI_ATTR_LOBEMPTY	Sets the descriptor to be empty LOB	N/A
OCI_ATTR_LOB_REMOTE	N/A	set to TRUE if the lob locator is from a remote database, set to FALSE otherwise
OCI_ATTR_LOB_TYPE	N/A	holds the LOB type (CLOB / BLOB / BFILE)
OCI_ATTR_LOB_IS_VALUE	N/A	set to TRUE if it is from a value LOB, otherwiseFALSE
OCI_ATTR_LOB_IS_READONLY	N/A	set to TRUE if it is a read-only LOB, otherwise FALSE
OCI_ATTR_LOBPREFETCH_LENGTH	When set to TRUE the attribute will enable prefetching and will prefetch the LOB length and the chunk size while performing select operation of LOB locator	set to TRUE if prefetching is turned on for the locator.
OCI_ATTR_LOBPREFETCH_SIZE	Overrides the default prefetch size for LOBs. Has a prerequisite of the OCI_ATTR_LOBPREFETCH_LENGTH attribute to be set to TRUE.	Returns the prefetch size of the locator.

Table 9-6 OCI Functions for LOBs

Category	Function/Procedure	Description
Sanity Checking	OCILobLocatorIsInit()	Checks whether a LOB locator is initialized.
Open/Close	OCILobOpen()	Open a LOB
	OCILobIsOpen()	Check if a LOB is open
	OCILobClose()	Close the LOB
Read Operations	OCILobGetLength2()	Get the length of the LOB
	OCILobGetStorageLimit()	Get the LOB storage limit for the database configuration
	OCILobGetChunkSize()	Get the optimum read / write size
	OCILobRead2()	Read data from the LOB starting at the specified offset
	OCILobArrayRead()	Reads data using multiple locators in one round trip.
	OCILobCharSetId()	Returns the character set ID of a LOB.
	OCILobCharSetForm()	Returns the character set form of a LOB.
Modify Operations	OCILobWrite2()	Write data to the LOB at a specified offset
	OCILobArrayWrite()	Writes data using multiple locators in one round trip.
	OCILobWriteAppend2()	Write data to the end of the LOB

Table 9-6 (Cont.) OCI Functions for LOBs

Category	Function/Procedure	Description
	`OCILobErase2()`	Erase part of a LOB, starting at a specified offset
	`OCILobTrim2()`	Trim the LOB value to the specified shorter length
Operations involving multiple locators	`OCILobIsEqual()`	Checks whether two LOB locators refer to the same LOB.
	`OCILobAppend()`	Append a LOB value to another LOB
	`OCILobCopy2()`	Copy all or part of a LOB to another LOB
	`OCILobLocatorAssign()`	Assign one LOB to another
	`OCILobLoadFromFile2()`	Load `BFILE` data into a LOB
Operations specific to SecureFiles	`OCILObGetOptions()`	Returns options (deduplication, compression, encryption) for SecureFiles.
	`OCILObSetOptions()`	Sets LOB features (deduplication and compression) for SecureFiles
	`OCILobGetContentType()`	Gets the content string for a SecureFiles
	`OCILobSetContentType()`	Sets a content string in a SecureFiles

Example 9-4 OCI API for LOBs

```
/* Define SQL statements to be used in program. */
#define LOB_NUM_QUERIES 2

static text *selstmt[LOB_NUM_QUERIES] = {
    (text *) "select ad_sourcetext from print_media where product_id = 1", /*
0 */
    (text *) "select ad_sourcetext from print_media where product_id = 2 for
update",
};

sword run_query(ub4 index, ub2 dty)
{
  OCILobLocator *c1 = (OCILobLocator *)0;
  OCILobLocator *c2 = (OCILobLocator *)0;

  OCIStmt       *stmthp;
  OCIDefine     *defn1p = (OCIDefine *) 0;
  OCIDefine     *defn2p = (OCIDefine *) 0;
  OCIBind       *bndp1  = (OCIBind *)   0;
  OCIBind       *bndp2  = (OCIBind *)   0;

  ub8            loblen;
  ub1            lbuf[128];
  ub1            inbuf[9] = "modified";
  ub1            inbuf_len = 8;
  ub8            amt = 15;
```

```
  ub8             bamt = 0;
  ub4             csize = 0;
  ub8             slimit = 0;
  boolean         flag = FALSE;
  boolean         boolval = TRUE;
  ub4             id = 10;


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

  /************** Allocate descriptors **********************/
  CHECK_ERROR (OCIDescriptorAlloc((dvoid *) envhp, (dvoid **) &c1,
                                  (ub4)OCI_DTYPE_FILE, (size_t) 0,
                                  (dvoid **) 0));

  CHECK_ERROR (OCIDescriptorAlloc((dvoid *) envhp, (dvoid **) &c2,
                                 (ub4)OCI_DTYPE_FILE, (size_t) 0,

  /********** Execute selstmt[0] to get c1  **********************/
  CHECK_ERROR (OCIStmtPrepare(stmthp, errhp, selstmt[0],
                              (ub4) strlen((char *) selstmt[0]),
                              (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT));

  CHECK_ERROR (OCIDefineByPos(stmthp, &defn1p, errhp, (ub4) 1, (dvoid *) &c1,
                              (sb4) -1, SQLT_CLOB, (dvoid *) 0, (ub2 *) 0,
                              (ub2 *)0, (ub4) OCI_DEFAULT));

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

  /********** Execute selstmt[1] to get c2 **********************/
  CHECK_ERROR (OCIStmtPrepare(stmthp, errhp, selstmt[1],
                              (ub4) strlen((char *) selstmt[1]),
                              (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT));

  CHECK_ERROR (OCIDefineByPos(stmthp, &defn1p, errhp, (ub4) 1, (dvoid *) &c2,
                              (sb4) -1, SQLT_CLOB, (dvoid *) 0, (ub2 *) 0,
                              (ub2 *)0, (ub4) OCI_DEFAULT));

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


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

  CHECK_ERROR (OCILobLocatorIsInit(envhp, errhp, (OCILobLocator *) c1,
                                   &boolval));
  if (boolval)
    printf("LOB locator is initialized! \n");
```

```
else
  printf("LOB locator is NOT initialized \n");


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

/*----------------------- Opening a CLOB ------------------------------*/
CHECK_ERROR (OCILobOpen(svchp, errhp, c1, (ub1)OCI_LOB_READONLY));
printf("OCILobOpen: Works\n");
/*-------------- Determining Whether a CLOB Is Open ----------------*/
CHECK_ERROR (OCILobIsOpen(svchp, errhp, c1,  &boolval));
printf("OCILobIsOpen: %s\n", (boolval)?"TRUE":"FALSE");

/*----------------------- Closing a LOB -------------------------------*/
CHECK_ERROR (OCILobClose(svchp, errhp, c1));
printf("OCILobClose: Works\n");

/*---------------------------------------------------------------------*/
/*--------------------- LOB Read Operations ---------------------------*/
/*---------------------------------------------------------------------*/

printf("OCILobFileOpen: Works\n");

/*----------------- Getting the Length of a LOB --------------------*/
CHECK_ERROR (OCILobGetLength2(svchp, errhp, c1, &loblen));
printf("OCILobGetLength2: loblen: %d \n", loblen);

/*----------------- Getting the Storage Limit of a LOB -------------*/
CHECK_ERROR (OCILobGetStorageLimit(svchp, errhp, c1, &slimit));
printf("OCILobGetStorageLimit: storage limit: %ld \n", slimit);

/*----------------- Getting the Chunk Size of a LOB -----------------*/
CHECK_ERROR (OCILobGetChunkSize(svchp, errhp, c1, &csize));
printf("OCILobGetChunkSize: storage limit: %d \n", csize);

/*------------------------ Reading LOB Data ------------------------*/
CHECK_ERROR (OCILobRead2(svchp, errhp, c1, &amt,
                         NULL, (oraub8)1, lbuf,
                         (oraub8)sizeof(lbuf), OCI_ONE_PIECE ,(dvoid*)0,
                         NULL, (ub2)0, (ub1)SQLCS_IMPLICIT));
printf("OCILobRead2: buf: %.*s amt: %lu\n", amt, lbuf, amt);

/*---------------------------------------------------------------------*/
/*--------------------- Modifying a LOB -------------------------------*/
/*---------------------------------------------------------------------*/

/*----------------------- Writing Data to LOB ------------------------*/
amt = 8;
CHECK_ERROR (OCILobWrite2(svchp, errhp, c2, &bamt, &amt, 1,
              (dvoid *) inbuf, (ub8)inbuf_len, OCI_ONE_PIECE, (dvoid *)0,
              (OCICallbackLobWrite2)0,
              (ub2) 0, (ub1) SQLCS_IMPLICIT));

/*----------------------- Write Append to a LOB ---------------------*/
```

```
  /* Append 8 characters */
  amt = 8;
  CHECK_ERROR (OCILobWriteAppend2(svchp, errhp, c2, &bamt, &amt,
                (dvoid *) inbuf, (ub8)inbuf_len, OCI_ONE_PIECE, (dvoid *)0,
                (OCICallbackLobWrite2)0,
                (ub2) 0, (ub1) SQLCS_IMPLICIT));

  /*---------------------- Erase part of LOB contents ----------------*/
  /* Erase 5 characters */
  amt = 5;
  CHECK_ERROR (OCILobErase2(svchp, errhp, c2, &amt, 2));

  /*------------------------- Trim a LOB -----------------------------*/
  amt = 1000;
  CHECK_ERROR (OCILobTrim2(svchp, errhp, c2, amt));
  printf("OCILobTrim2  Works! \n");

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

  /*--------------- Check Equality of LOB locators -------------------*/
  CHECK_ERROR ( OCILobIsEqual(envhp, c1, c2, &boolval))
  printf("OCILobIsEqual %s\n", (boolval)?"TRUE":"FALSE");

  /*--------------- Append contents of a LOB to another LOB ----------*/
  CHECK_ERROR(OCILobAppend(svchp, errhp, c2, c1));
  printf("OCILobAppend: Works! \n");

  /*------------------------ LOB Copy --------------------------------*/
  /* Copy 10 characters from offset 1 of source to offset 2 of destination*/
  CHECK_ERROR (OCILobCopy2(svchp, errhp, c2, c1, 10, 2, 1));
  printf("OCILobCopy2: Works! \n");

 /*------------------- LOB Locator Assign ---------------------------*/
  CHECK_ERROR (OCILobLocatorAssign(svchp, errhp, c1, &c2));
  printf("OCILobLocatorAssign: Works! \n");

  /* Free the LOB descriptors which were allocated */
  OCIDescriptorFree((dvoid *) c1, (ub4) SQLT_CLOB);
  OCIDescriptorFree((dvoid *) c2, (ub4) SQLT_CLOB);

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

- Efficiently Reading LOB Data in OCI
  This section describes how to read the contents of a LOB into a buffer.
- Efficiently Writing LOB Data in OCI
  This section describes how to write the contents of a buffer to a LOB.

9.4.1 Efficiently Reading LOB Data in OCI

This section describes how to read the contents of a LOB into a buffer.

Streaming Read in OCI

The most efficient way to read large amounts of LOB data is to use `OCILobRead2()` with the streaming mechanism enabled using polling or callback. To do so, specify the starting point of the read using the `offset` parameter as follows:

```
ub8  char_amt =  0;
ub8  byte_amt =  0;
ub4  offset = 1000;

OCILobRead2(svchp, errhp, locp, &byte_amt, &char_amt, offset, bufp, bufl,
            OCI_ONE_PIECE, 0, 0, 0, 0);
```

When using *polling mode*, be sure to look at the value of the `byte_amt` parameter after each `OCILobRead2()` call to see how many bytes were read into the buffer because the buffer may not be entirely full.

When using *callbacks*, the `lenp` parameter, which is input to the callback, indicates how many bytes are filled in the buffer. Be sure to check the `lenp` parameter during your callback processing because the entire buffer may not be filled with data.

> **See Also:**
>
> *Oracle Call Interface Programmer's Guide*

LOB Array Read

This section describes how to read LOB data for multiple locators in one round trip, using `OCILobArrayRead()`.

For an OCI application example, assume that the program has a prepared SQL statement such as:

```
SELECT lob1 FROM lob_table;
```

where `lob1` is the LOB column and `lob_array` is an array of define variables corresponding to a LOB column:

```
OCILobLocator * lob_array[10];

...
 for (i=0; i<10, i++)        /* initialize array of locators */
    lob_array[i] = OCIDescriptorAlloc(..., OCI_DTYPE_LOB, ...);

...

OCIDefineByPos(..., 1, (dvoid *) lob_array, ... SQLT_CLOB, ...);

/* Execute the statement with iters = 10 to do an array fetch of 10 locators. */
OCIStmtExecute ( <service context>, <statement handle>, <error handle>,
                 10,    /* iters  */
                 0,     /* row offset */
```

```
                NULL,  /* snapshot IN */
                NULL,  /* snapshot out */
                OCI_DEFAULT /* mode */);
...

  ub4 array_iter = 10;
  char  *bufp[10];
  oraub8 bufl[10];
  oraub8 char_amtp[10];
  oraub8 offset[10];

 for (i=0; i<10; i++)
  {
    bufp[i] = (char *)malloc(1000);
    bufl[i] = 1000;
    offset[i] = 1;
    char_amtp[i] = 1000;  /* Single byte fixed width char set. */
  }

/* Read the 1st 1000 characters for all 10 locators in one
 * round trip. Note that offset and amount need not be
 * same for all the locators. */

OCILobArrayRead(<service context>, <error handle>,
                &array_iter, /* array size */
                lob_array,   /* array of locators */
                NULL,        /* array of byte amounts */
                char_amtp,   /* array of char amounts */
                offset,      /* array of offsets */
       (void **)bufp,        /* array of read buffers */
                bufl,        /* array of buffer lengths */
                OCI_ONE_PIECE,  /* piece information */
                NULL,           /* callback context */
                NULL,           /* callback function */
                0,              /* character set ID - default */
                SQLCS_IMPLICIT);/* character set form */

 ...

for (i=0; i<10; i++)
  {
    /* Fill bufp[i] buffers with data to be written */
    strncpy (bufp[i], "Test Data------", 15);
    bufl[i] = 1000;
    offset[i] = 50;
    char_amtp[i] = 15;  /* Single byte fixed width char set. */
  }

/* Write the 15 characters from offset 50 to all 10
 * locators in one round trip. Note that offset and
 * amount need not be same for all the locators. */
 */

OCILobArrayWrite(<service context>, <error handle>,
                 &array_iter, /* array size */
                 lob_array,   /* array of locators */
                 NULL,        /* array of byte amounts */
                 char_amtp,   /* array of char amounts */
                 offset,      /* array of offsets */
            (void **)bufp,    /* array of read buffers */
                 bufl,        /* array of buffer lengths */
                 OCI_ONE_PIECE,  /* piece information */
```

```
                NULL,           /* callback context */
                NULL,           /* callback function */
                0,              /* character set ID - default */
                SQLCS_IMPLICIT);/* character set form */
...
```

LOB Array Read with Streaming

LOB array APIs can be used to read/write LOB data in multiple pieces. This can be done by using polling method or a callback function.Here data is read/written in multiple pieces sequentially for the array of locators. For polling, the API would return to the application after reading/writing each piece with the `array_iter` parameter (OUT) indicating the index of the locator for which data is read/written. With a callback, the function is called after reading/writing each piece with `array_iter` as IN parameter.

Note that:

- It is possible to read/write data for a few of the locators in one piece and read/write data for other locators in multiple pieces. Data is read/written in one piece for locators which have sufficient buffer lengths to accommodate the whole data to be read/written.
- Your application can use different amount value and buffer lengths for each locator.
- Your application can pass zero as the amount value for one or more locators indicating pure streaming for those locators. In the case of reading, LOB data is read to the end for those locators. For writing, data is written until `OCI_LAST_PIECE` is specified for those locators.

LOB Array Read with Callback

The following example reads 10Kbytes of data for each of 10 locators with 1Kbyte buffer size. Each locator needs 10 pieces to read all the data. The callback function is called 100 (10*10) times to return the pieces sequentially.

```
/* Fetch the locators */
...
     ub4    array_iter = 10;
     char  *bufp[10];
     oraub8 bufl[10];
     oraub8 char_amtp[10];
     oraub8 offset[10];
     sword  st;

     for (i=0; i<10; i++)
     {
       bufp[i] = (char *)malloc(1000);
       bufl[i] = 1000;
       offset[i] = 1;
       char_amtp[i] = 10000;       /* Single byte fixed width char set. */
      }

      st =  OCILobArrayRead(<service context>, <error handle>,
                        &array_iter, /* array size */
                        lob_array,   /* array of locators */
                        NULL,        /* array of byte amounts */
                        char_amtp,   /* array of char amounts */
                        offset,      /* array of offsets */
                  (void **)bufp,     /* array of read buffers */
                        bufl,        /* array of buffer lengths */
                        OCI_FIRST_PIECE,  /* piece information */
                        ctx,              /* callback context */
                        cbk_read_lob,     /* callback function */
```

```
                        0,                /* character set ID - default */
                        SQLCS_IMPLICIT);
...
/* Callback function for LOB array read. */
sb4 cbk_read_lob(dvoid *ctxp, ub4 array_iter, CONST dvoid *bufxp, oraub8 len,
                 ub1 piece, dvoid **changed_bufpp, oraub8 *changed_lenp)
{
   static ub4 piece_count = 0;
   piece_count++;
   switch (piece)
   {
    case OCI_LAST_PIECE:
      /*--- buffer processing code goes here ---*/
      (void) printf("callback read the %d th piece(last piece) for %dth locator \n\n",
                piece_count, array_iter );
      piece_count = 0;
      break;
    case OCI_FIRST_PIECE:
      /*--- buffer processing code goes here ---*/
      (void) printf("callback read the 1st piece for %dth locator\n",
                    array_iter);
      /* --Optional code to set changed_bufpp and changed_lenp if the buffer needs
         to be changed dynamically --*/
      break;
    case OCI_NEXT_PIECE:
      /*--- buffer processing code goes here ---*/
      (void) printf("callback read the %d th piece for %dth locator\n",
                    piece_count, array_iter);
      /* --Optional code to set changed_bufpp and changed_lenp if the  buffer
           must be changed dynamically --*/
      break;
      default:
      (void) printf("callback read error: unkown piece = %d.\n", piece);
      return OCI_ERROR;
   }
    return OCI_CONTINUE;
}
...
```

LOB Array Read in Polling Mode

The following example reads 10Kbytes of data for each of 10 locators with 1Kbyte buffer size. Each locator needs 10 pieces to read the complete data. `OCILobArrayRead()` must be called 100 (10*10) times to fetch all the data.First we call `OCILobArrayRead()` with `OCI_FIRST_PIECE` as `piece` parameter. This call returns the first 1K piece for the first locator.Next `OCILobArrayRead()` is called in a loop until the application finishes reading all the pieces for the locators and returns `OCI_SUCCESS`. In this example it loops 99 times returning the pieces for the locators sequentially.

```
/* Fetch the locators */
...

     /* array_iter parameter indicates the number of locators in the array read.
      * It is an IN parameter for the 1st call in polling and is ignored as IN
      * parameter for subsequent calls. As OUT parameter it indicates the locator
      * index for which the piece is read.
      */

     ub4    array_iter = 10;
     char  *bufp[10];
     oraub8 bufl[10];
     oraub8 char_amtp[10];
```

```
oraub8 offset[10];
sword  st;

for (i=0; i<10; i++)
{
  bufp[i] = (char *)malloc(1000);
  bufl[i] = 1000;
  offset[i] = 1;
  char_amtp[i] = 10000;       /* Single byte fixed width char set. */
}

st =  OCILobArrayRead(<service context>, <error handle>,
                    &array_iter, /* array size */
                    lob_array, /* array of locators */
                    NULL,      /* array of byte amounts */
                    char_amtp, /* array of char amounts */
                    offset,    /* array of offsets */
            (void **)bufp,     /* array of read buffers */
                    bufl,      /* array of buffer lengths */
                    OCI_FIRST_PIECE, /* piece information */
                    NULL,           /* callback context */
                    NULL,           /* callback function */
                    0,              /* character set ID - default */
                    SQLCS_IMPLICIT); /* character set form */

/* First piece for the first locator is read here.
 * bufp[0]          => Buffer pointer into which data is read.
 * char_amtp[0 ]    => Number of characters read in current buffer
 *
 */

While ( st == OCI_NEED_DATA)
{
     st =  OCILobArrayRead(<service context>, <error handle>,
                     &array_iter, /* array size */
                     lob_array, /* array of locators */
                     NULL,      /* array of byte amounts */
                     char_amtp, /* array of char amounts */
                     offset,    /* array of offsets */
             (void **)bufp,     /* array of read buffers */
                     bufl,      /* array of buffer lengths */
                     OCI_NEXT_PIECE, /* piece information */
                     NULL,           /* callback context */
                     NULL,           /* callback function */
                     0,              /* character set ID - default */
                     SQLCS_IMPLICIT);

  /* array_iter returns the index of the current array element for which
   * data is read. for example, aray_iter = 1 implies first locator,
   * array_iter = 2 implies second locator and so on.
   *
   * lob_array[ array_iter - 1]=> Lob locator for which data is read.
   * bufp[array_iter - 1]      => Buffer pointer into which data is read.
   * char_amtp[array_iter - 1] => Number of characters read in current buffer
   */

...
    /* Consume the data here */
...
}
```

9.4.2 Efficiently Writing LOB Data in OCI

This section describes how to write the contents of a buffer to a LOB.

Streaming Write in OCI

The most efficient way to write large amounts of LOB data is to use `OCILobWrite2()` with the streaming mechanism enabled, and using polling or a callback. If you know how much data is written to the LOB, then specify that amount when calling `OCILobWrite2()`. This ensures that LOB data on the disk is contiguous. Apart from being spatially efficient, the contiguous structure of the LOB data makes reads and writes in subsequent operations faster.

LOB Array Write with Callback

The following example writes 10Kbytes of data for each of 10 locators with a 1K buffer size. A total of 100 pieces must be written (10 pieces for each locator). The first piece is provided by the `OCILobArrayWrite()` call. The callback function is called 99 times to get the data for subsequent pieces to be written.

```
/* Fetch the locators */
...

    ub4    array_iter = 10;
    char  *bufp[10];
    oraub8 bufl[10];
    oraub8 char_amtp[10];
    oraub8 offset[10];
    sword  st;

    for (i=0; i<10; i++)
    {
      bufp[i] = (char *)malloc(1000);
      bufl[i] = 1000;
      offset[i] = 1;
      char_amtp[i] = 10000;       /* Single byte fixed width char set. */
    }

 st =  OCILobArrayWrite(<service context>, <error handle>,
                        &array_iter, /* array size */
                        lob_array,   /* array of locators */
                        NULL,        /* array of byte amounts */
                        char_amtp,   /* array of char amounts */
                        offset,      /* array of offsets */
                (void **)bufp,       /* array of write buffers */
                        bufl,        /* array of buffer lengths */
                        OCI_FIRST_PIECE,  /* piece information */
                        ctx,              /* callback context */
                        cbk_write_lob     /* callback function */
                        0,                /* character set ID - default */
                        SQLCS_IMPLICIT);

...

/* Callback function for LOB array write. */
sb4 cbk_write_lob(dvoid *ctxp, ub4 array_iter, dvoid *bufxp, oraub8 *lenp,
                  ub1 *piecep, ub1 *changed_bufpp, oraub8 *changed_lenp)
{
 static ub4 piece_count = 0;
 piece_count++;
```

```
 printf (" %dth piece written  for %dth locator \n\n", piece_count, array_iter);

 /*-- code to fill bufxp with data goes here. *lenp should reflect the  size and
  *   should be less than or equal to MAXBUFLEN -- */
 /* --Optional code to set changed_bufpp and changed_lenp if the buffer must
  *   be changed dynamically --*/

  if (this is the last data buffer for current locator)
     *piecep = OCI_LAST_PIECE;
  else if (this is the first data buffer for the next locator)
     *piecep = OCI_FIRST_PIECE;
     piece_count = 0;
  else
     *piecep = OCI_NEXT_PIECE;

     return OCI_CONTINUE;
    }
...
```

LOB Array Write in Polling Mode

The following example writes 10Kbytes of data for each of `10` locators with a 1K buffer size. `OCILobArrayWrite()` has to be called `100` (`10` times `10`) times to write all the data. The function is used in a similar manner to `OCILobWrite2()`.

```
/* Fetch the locators */
...

/* array_iter parameter indicates the number of locators in the array read.
 * It is an IN parameter for the 1st call in polling and is ignored as IN
 * parameter for subsequent calls. As an OUT parameter it indicates the locator
 * index for which the piece is written.
 */

ub4    array_iter = 10;
char  *bufp[10];
oraub8 bufl[10];
oraub8 char_amtp[10];
oraub8 offset[10];
sword  st;
int    i, j;

for (i=0; i<10; i++)
{
  bufp[i] = (char *)malloc(1000);
  bufl[i] = 1000;
  /* Fill bufp here. */
...
  offset[i] = 1;
  char_amtp[i] = 10000;       /* Single byte fixed width char set. */
}

for (i  = 1; i <= 10; i++)
{
 /* Fill up bufp[i-1] here.  The first piece for ith locator would be written from
    bufp[i-1] */
...
    st =  OCILobArrayWrite(<service context>, <error handle>,
                      &array_iter, /* array size */
                      lob_array,   /* array of locators */
                      NULL,        /* array of byte amounts */
                      char_amtp,   /* array of char amounts */
```

```
                    offset,      /* array of offsets */
              (void **)bufp,     /* array of write buffers */
                    bufl,        /* array of buffer lengths */
                    OCI_FIRST_PIECE, /* piece information */
                    NULL,            /* callback context */
                    NULL,            /* callback function */
                    0,               /* character set ID - default */
                    SQLCS_IMPLICIT); /* character set form */

 for ( j = 2; j < 10; j++)
 {
 /* Fill up bufp[i-1] here.  The jth piece for ith locator would be written from
    bufp[i-1] */
...
 st =  OCILobArrayWrite(<service context>, <error handle>,
                        &array_iter, /* array size */
                        lob_array,   /* array of locators */
                        NULL,        /* array of byte amounts */
                        char_amtp,   /* array of char amounts */
                        offset,      /* array of offsets */
                (void **)bufp,       /* array of write buffers */
                        bufl,        /* array of buffer lengths */
                        OCI_NEXT_PIECE, /* piece information */
                        NULL,           /* callback context */
                        NULL,           /* callback function */
                        0,              /* character set ID - default */
                        SQLCS_IMPLICIT);

    /* array_iter returns the index of the current array element for which
     * data is being written. for example, aray_iter = 1 implies first locator,
     * array_iter = 2 implies second locator and so on. Here i = array_iter.
     *
     * lob_array[ array_iter - 1] => Lob locator for which data is written.
     * bufp[array_iter - 1]       => Buffer pointer from which data is written.
     * char_amtp[ array_iter - 1] => Number of characters written in
     * the piece just written
     */
}

/* Fill up bufp[i-1] here.  The last piece for ith locator would be written from
   bufp[i -1] */
...
 st =  OCILobArrayWrite(<service context>, <error handle>,
                        &array_iter, /* array size */
                        lob_array,   /* array of locators */
                        NULL,        /* array of byte amounts */
                        char_amtp,   /* array of char amounts */
                        offset,      /* array of offsets */
                (void **)bufp,       /* array of write buffers */
                        bufl,        /* array of buffer lengths */
                        OCI_LAST_PIECE,  /* piece information */
                        NULL,            /* callback context */
                        NULL,            /* callback function */
                        0,               /* character set ID - default */
                        SQLCS_IMPLICIT);
}

...
```

9.5 ODP.NET API for LOBs

Oracle Data Provider for .NET (ODP.NET) is an ADO.NET provider for the Oracle Database.

ODP.NET offers fast and reliable access to Oracle data and features from any .NET Core or .NET Framework application. ODP.NET also uses and inherits classes and interfaces available in the Microsoft .NET Class Library. The ODP.NET supports the following LOBs as native data types with .NET: `BLOB`, `CLOB`, `NCLOB`, and `BFILE`.

See Also:

- LOB Support
- Obtaining LOB Data

Table 9-7 ODP.NET methods in OracleClob and OracleBlob classes

Category	Function/Procedure	Description
Open/Close	`BeginChunkWrite`	Open a LOB
	`EndChunkWrite`	Close a LOB
	`IsInChunkWriteMode`	Check if a LOB is open
Read Operations	`Length`	Get the length of the LOB
	`OptimumChunkSize`	Get the optimum read/write size
	`Value`	Returns the entire LOB data as a string for CLOB and a byte array for BLOB
	`Read`	Read data from the LOB starting at the specified offset
	`Search`	Return the matching position of a pattern in a LOB using `INSTR`
Modify Operations	`Write`	Write data to the LOB at a specified offset
	`Erase`	Erase part of a LOB, starting at a specified offset
	`SetLength`	Trim the LOB value to the specified shorter length
Operations involving multiple locators	`Compare`	Compare all or part of the value of two LOBs
	`IsEqual`	Check if two LOBs point to the same LOB data
	`Append`	Append a LOB value to another LOB, or append a byte array, string, or character array to an existing LOB
	`CopyTo`	Copy all or part of a LOB to another LOB
	`Clone`	Assign LOB locator `src` to LOB locator `dst`

9.6 OCCI API for LOBs

OCCI provides a seamless interface to manipulate objects of user-defined types as C++ class instances.

Oracle C++ Call Interface (OCCI) is a C++ API for manipulating data in an Oracle database. OCCI is organized as an easy-to-use set of C++ classes that enable a C++ program to connect to a database, run SQL statements, insert/update values in database tables, retrieve results of a query, run stored procedures in the database, and access metadata of database schema objects.

Oracle C++ Call Interface (OCCI) is designed so that you can use OCI and OCCI together to build applications.

The OCCI API provides the following advantages over JDBC and ODBC:

- OCCI encompasses more Oracle functionality than JDBC. OCCI provides all the functionality of OCI that JDBC does not provide.
- OCCI provides *compiled* performance. With compiled programs, the source code is written as close to the computer as possible. Because JDBC is an *interpreted* API, it cannot provide the performance of a compiled API. With an interpreted program, performance degrades as each line of code must be interpreted individually into code that is close to the computer.
- OCCI provides memory management with smart pointers. You do not have to be concerned about managing memory for OCCI objects. This results in robust higher performance application code.
- Navigational access of OCCI enables you to intuitively access objects and call methods. Changes to objects persist without writing corresponding SQL statements. If you use the client side cache, then the navigational interface performs better than the object interface.
- With respect to ODBC, the OCCI API is simpler to use. Because ODBC is built on the C language, OCCI has all the advantages C++ provides over C. Moreover, ODBC has a reputation as being difficult to learn. The OCCI, by contrast, is designed for ease of use.

You can use OCCI to perform random and piecewise operations on LOBs, which means that you specify the offset or amount of the operation to read or write a part of the LOB value.

OCCI provides these classes that allow you to use different types of LOB instances as objects in your C++ application:

- `Clob` class to access and modify data stored in persistent `CLOB`s and `NCLOB`s
- `Blob` class to access and modify data stored in persistent `BLOB`s

> **See Also:**
>
> Syntax information on these classes and details on OCCI in general is available in the*Oracle C++ Call Interface Developer's Guide*.

Clob Class

The Clob driver implements a `CLOB` object using an SQL LOB locator. This means that a CLOB object contains a logical pointer to the SQL `CLOB` data rather than the data itself.

The CLOB interface provides methods for getting the length of an SQL CLOB value, for materializing a CLOB value on the client, and getting a substring. Methods in the ResultSet and Statement interfaces such as getClob() and setClob() allow you to access SQL CLOB values.

Blob Class

Methods in the ResultSet and Statement interfaces, such as getBlob() and setBlob(), allow you to access SQL BLOB values. The Blob interface provides methods for getting the length of a SQL BLOB value, for materializing a BLOB value on the client, and for extracting a part of the BLOB.

Fixed-Width Character Set Rules

In OCCI, for *fixed-width* client-side character sets, these rules apply:

- Clob: offset and amount parameters are always in characters
- Blob: offset and amount parameters are always in bytes

The following rules apply only to *varying-width* client-side character sets:

- **Offset parameter**: Regardless of whether the client-side character set is varying-width, the offset parameter is always as follows:
  - Clob(): in characters
  - Blob(): in bytes
- **Amount parameter**: The amount parameter is always as indicated:
  - Clob: in characters, when referring to a server-side LOB
  - Blob: in bytes, when referring to a client-side buffer
- **length()**: Regardless of whether the client-side character set is varying-width, the output length is as follows:
  - Clob.length(): in characters
  - Blob.length(): in bytes
- **Clob.read() and Blob.read()**: With client-side character set of varying-width, CLOBs and NCLOBs:
  - ***Input amount*** is in characters. Input amount refers to the number of characters to read from the server-side CLOB or NCLOB.
  - ***Output amount*** is in bytes. Output amount indicates how many bytes were read into the OCCI buffer parameter, buffer.
- **Clob.write() and Blob.write()**: With client-side character set of varying-width, CLOBs and NCLOBs:
  - **Input amount** is in bytes. Input amount refers to the number of bytes of data in the OCCI input buffer, buffer.
  - **Output amount** is in characters. Output amount refers to the number of characters written into the server-side CLOB or NCLOB.
- **Amount Parameter for Other OCCI Operations:** For the OCCI LOB operations Clob.copy(), Clob.erase(), Clob.trim() irrespective of the client-side character set, the *amount parameter* is in characters for CLOBs and NCLOBs. All these operations refer to the amount of LOB data on the server.

> **See also:**
>
> *Oracle Database Globalization Support Guide*

Table 9-8 OCCI Methods for LOBs

Category	Function/Procedure	Description
Sanity Checking	`Clob/Blob.isInitialized`	Checks whether a LOB locator is initialized.
Open/Close	`Clob/Blob.Open()`	Open a LOB
	`Clob/Blob.isOpen()`	Check if a LOB is open
	`Clob/Blob.Close()`	Close the LOB
Read Operations	`Blob/Clob.length()`	Get the length of the LOB
	`Blob/Clob.getChunkSize()`	Get the optimum read or write size
	`Blob/Clob.read()`	Read data from the LOB starting at the specified offset
	`Clob.getCharSetId()`	Return the character set ID of a LOB
	`Clob.getCharSetForm()`	Return the character set form of a LOB.
Modify Operations	`Blob/Clob.write()`	Write data to the LOB at a specified offset
	`Blob/Clob.trim()`	Trim the LOB value to the specified shorter length
Operations involving multiple locators	`Clob/Blob.operator == and !=`	Checks whether two LOB locators refer to the same LOB.
	`Blob/Clob.append()`	Append a LOB value to another LOB
	`Blob/Clob.copy()`	Copy all or part of a LOB to another LOB, or load from a BFILE into a LOB
	`Clob/Blob.operator =`	Assign one LOB to another
Operations specific to securefiles	`Blob/Clob.getOptions()`	Returns options (deduplication, compression, encryption) for SecureFiles.
	`Blob/Clob.setOptions()`	Sets LOB features (deduplication and compression) for SecureFiles
	`Blob/Clob.getContentType()`	Gets the content string for a SecureFiles
	`Blob/Clob.setContentType()`	Sets a content string in a SecureFiles

9.7 Pro*C/C++ and Pro*COBOL API for LOBs

This section describes the mapping of Pro*C/C++ and Pro*COBOL locators to locator pointers to access a LOB value.

Embedded SQL statements enable you to access data stored in `BLOB`s, `CLOB`s, and `NCLOB`s.

> **See Also:**
>
> *Pro*C/C++ Programmer's Guide* and *Pro*COBOL Programmer's Guide* for detailed documentation, including syntax, host variables, host variable types and example code.

Unlike locators in PL/SQL, locators in Pro*C/C++ and Pro*COBOL are mapped to locator pointers which are then used to refer to the LOB value. To successfully complete an embedded SQL LOB statement you must do the following:

1. Provide an allocated input locator pointer that represents a LOB that exists in the database tablespaces or external file system before you run the statement.
2. SELECT a LOB locator into a LOB locator pointer variable.
3. Use this variable in the embedded SQL LOB statement to access and manipulate the LOB value.

Table 9-9 Pro*C/C++ and Pro*COBOL Embedded SQL Statements for LOBs

Category	Function/Procedure	Description
Open/Close	`OPEN`	Open a LOB
	`DESCRIBE[ISOPEN]`	Check is a LOB is open
	`CLOSE`	Close the LOB
Read Operations	`DESCRIBE[LENGTH]`	Get the length of the LOB
	`DESCRIBE[CHUNKSIZE]`	Get the optimum read or write size
	`READ`	Read data from the LOB starting at a specified offset
Modify Operations	`WRITE`	Write data to the LOB at a specified offset
	`WRITE APPEND`	Write data to the end of the LOB
	`ERASE`	Erase part of a LOB, starting at a specified offset
	`TRIM`	Trim the LOB value to the specified shorter length
Operations involving multiple locators	`APPEND`	Append a LOB value to another LOB
	`COPY`	Copy all or part of a LOB to another LOB
	`ASSIGN`	Assign one LOB to another
	`LOAD FROM FILE`	Load `BFILE` data into a LOB