11

Performance Guidelines

This section discusses performance guidelines for applications that use LOB data types.

> **Note:**
>
> From release 23ai onwards, you can experience improved read and write performance for LOBs due to the following enhancements:
>
> - Multiple LOBs in a single transaction are buffered simultaneously. This improves performance when you use mixed workload in a transaction. Mixed workload refers to switching between LOBs while writing within a single transaction. Let's consider that you write to LOB1, then you write to LOB2, and then you want to write again to LOB1 in a single transaction. LOB1 and LOB2 are buffered simultaneously, which provides better throughput and minimizes space fragmentation.
> - Various enhancements, such as acceleration of compressed LOB append and compression unit caching, improve the performance of reads and writes to compressed LOBs.
> - The input-output buffer is adaptively resized based on size of the input data for large writes to LOBs with the NOCACHE option. This improves the performance for large direct writes, such as writes to file systems on DBFS and OFS.

11.1 LOB Performance Guidelines

This section provides performance guidelines while using LOBs through Data Interface or LOB APIs.

LOBs can be accessed using the Data Interface or through the LOB APIs.

- Temporary LOBs
  In addition to the performance guidelines applicable to all LOBs described earlier, following are some guidelines for using temporary LOBs:
- Value LOBs
  Value LOBs are temporary LOBs. Hence all Temporary LOB storage guidelines apply to Value LOBs as well.

11.1.1 All LOBs

Learn about the guidelines to achieve good performance while using LOBs in this section.

The following guidelines will help you get the the best performance when using LOBs, and minimize the number of round trips to the server:

- To minimize I/O:
  - Read and write data at block boundaries. This optimizes I/O in many ways, e.g., by minimizing UNDO generation. For temporary LOBs and securefile LOBs, usable data area of the tablespace block size is returned by the following APIs: `DBMS_LOB.GETCHUNKSIZE` in PLSQL, and `OCILobGetChunkSize()` in OCI. When writing in a loop, design your code so that one write call writes everything that needs to go in a database block, thus ensuring that consecutive writes don't write to the same block.
  - Read and write large pieces of data at a time.
  - The 2 recommendations above can be combined by reading and writing in large whole number multiples of database block size returned by the `DBMS_LOB.GETCHUNKSIZE/ OCILobGetChunkSize()` API.
- To minimize the number of round trips to the server:
  - If you know the maximum size of your lob data, and you intend to read or write the entire LOB, use the Data Interface as outlined below. You can allocate the entire size of lob as a single buffer, or use piecewise / callback mechanisms.
    * For read operations, define the LOB as character/binary type using the `OCIDefineByPos()` function in OCI and the `DefineColumnType()` function in JDBC.
    * For write operations, bind the LOB as character/binary type using the `OCIBindByPos()` function in OCI and the `setString()` or `setBytes()` methods in JDBC.
  - Otherwise, use the LOB APIs as follows:
    * Use LOB prefetching for reads. Define the LOB prefetch size such that it can accommodate majority of the LOB values in the column.
    * Use piecewise or callback mechanism while using `OCILobRead2` or `OCILobWrite2` operations to minimize the roundtrips to the server.

> **See Also:**
>
> Data Interface for Persistent LOBs

11.1.2 Performance Guidelines While Using Persistent LOBs

In addition to the performance guidelines applicable to all LOBs described earlier, here are some performance guidelines while using persistent LOBs.

- Maximize writing to a single LOB in consecutive calls within a transaction. Interleaving DML statements prevent caching from reaching its maximum efficiency.
- Avoid taking savepoints or committing too frequently. This neutralizes the advantage of caching while writing.

> **Note:**
>
> Oracle recommends Securefile LOBs for storing persistent LOBs, hence this chapter focuses only on Securefile storage. All mentions of "LOBs" in the persistent LOB context is for Securefile LOBs unless otherwise mentioned.

11.1.3 Temporary LOBs

In addition to the performance guidelines applicable to all LOBs described earlier, following are some guidelines for using temporary LOBs:

- Temporary LOBs reside in the PGA memory or the temporary tablespace, depending on the size. Please ensure that you have a large enough PGA memory and temporary tablespace for the temporary LOBs used by your application.
- Use a separate temporary tablespace for temporary LOB storage instead of the default system tablespace. This avoids device contention when copying data from persistent LOBs to temporary LOBs.

  If you use SQL or PL/SQL semantics for LOBs in your applications, then many temporary LOBs are created silently. Ensure that PGA memory and temporary tablespace for storing these temporary LOBs is large enough for your applications. In particular, these temporary LOBs are silently created when you use the following:

  - SQL functions on LOBs
  - PL/SQL built-in character functions on LOBs
  - Variable assignments from `VARCHAR2`/`RAW` to `CLOB`s/`BLOB`s, respectively.
  - Perform a `LONG`-to-`LOB` migration
- Free up temporary LOBs returned from SQL queries and PL/SQL programs

  In PL/SQL, C (OCI), Java and other programmatic interfaces, SQL query results or PL/SQL program executions return temporary LOBs for operation/function calls on LOBs. For example:

  ```
  SELECT substr(CLOB_Column, 4001, 32000) FROM ...
  ```

If the query is executed in PL/SQL, then the returned temporary LOBs are automatically freed at the end of a PL/SQL program block. You can also explicitly free the temporary LOBs at any time. In OCI and Java, the returned temporary LOB must be explicitly freed.

Without proper deallocation of the temporary LOBs returned from SQL queries, you may observe performance degradation.

- In PL/SQL, use NOCOPY to pass temporary LOB parameters by reference whenever possible.

> **See Also:**
>
> *Oracle Database PL/SQL Language Reference*for more information on passing parameters by reference and parameter aliasing

- Temporary LOBs created with the CACHE parameter set to true move through the buffer cache and avoid the disk access.
- Oracle provides `v$temporary_lobs` view to monitor the use of temporary LOBs across all open sessions. Here is an example:

```
SQL> select * from v$temporary_lobs;

       SID CACHE_LOBS NOCACHE_LOBS ABSTRACT_LOBS    CON_ID
---------- ---------- ------------ ------------- ----------
       141          2            3             4          0
       146          0            0             1          0
       148          0            0             1          0
```

Following is the interpretation of output:

  - The `SID` column is the session ID.
  - The `CACHE_LOBS` column shows that session 141 currently has 2 temporary lobs in the temporary tablespace with CACHE turned on.
  - The `NOCACHE_LOBS` column shows that session 141 currently has 3 temporary lobs in the temporary tablespace with CACHE turned off.
  - The `ABSTRACT_LOBS` column shows that session 141 currently has 4 temporary lobs in the PGA memory.
  - The `CON_ID` column is the pluggable database container ID.

- For optimal performance, temporary LOBs use reference on read, copy on write semantics. When a temporary LOB locator is assigned to another locator, the physical LOB data is not copied. Subsequent READ operations using either of the LOB locators refer to the same physical LOB data. On the first WRITE operation after the assignment, the physical LOB data is copied in order to preserve LOB value semantics, that is, to ensure that each locator points to a unique LOB value.

  In PL/SQL, reference on read, copy on write semantics are illustrated as follows:

```
LOCATOR1 BLOB;
LOCATOR2 BLOB;
DBMS_LOB.CREATETEMPORARY (LOCATOR1,TRUE,DBMS_LOB.SESSION);

-- LOB data is not copied in this assignment operation:
```

```
LOCATOR2 := LOCATOR;
-- These read operations refer to the same physical LOB copy:
DBMS_LOB.READ(LOCATOR1, ...);
DBMS_LOB.GETLENGTH(LOCATOR2, ...);

-- A physical copy of the LOB data is made on WRITE:
DBMS_LOB.WRITE(LOCATOR2, ...);
```

In OCI, to ensure value semantics of LOB locators and data, `OCILobLocatorAssign()` is used to copy temporary LOB locators and the LOB Data. `OCILobLocatorAssign()` does not make a round trip to the server. The physical temporary LOB copy is made when LOB updates happen in the same round trip as the LOB update API as illustrated in the following:

```
OCILobLocator *LOC1;
OCILobLocator *LOC2;
OCILobCreateTemporary(... LOC1, ... TRUE,OCI_DURATION_SESSION);

/* No round-trip is incurred in the following call. */
OCILobLocatorAssign(... LOC1, LOC2);

/* Read operations refer to the same physical LOB copy. */
OCILobRead2(... LOC1 ...)

/* One round-trip is incurred to make a new copy of the
 * LOB data and to write to the new LOB copy.
 */
OCILobWrite2(... LOC1 ...)

/* LOC2 does not see the same LOB data as LOC1. */
OCILobRead2(... LOC2 ...)
```

If LOB value semantics are not intended, then you can use C pointer assignment so that both locators point to the same data as illustrated in the following code snippet:

```
OCILobLocator *LOC1;
OCILobLocator *LOC2;
OCILobCreateTemporary(... LOC1, ... TRUE,OCI_DURATION_SESSION);

/* Pointer is copied. LOC1 and LOC2 refer to the same LOB data. */
LOC2 = LOC1;

/* Write to LOC2. */
OCILobWrite2(...LOC2...)

/* LOC1 sees the change made to LOC2. */
OCILobRead2(...LOC1...)
```

- Use OCI_OBJECT mode for temporary LOBs

  To improve the performance of temporary LOBs on LOB assignment, use `OCI_OBJECT` mode for `OCILobLocatorAssign()`. In `OCI_OBJECT` mode, the database tries to minimize the number of deep copies to be done. Hence, after `OCILobLocatorAssign()` is done on a source temporary LOB in `OCI_OBJECT` mode, the source and the destination locators point to the same LOB until any modification is made through either LOB locator.

11.1.4 Value LOBs

Value LOBs are temporary LOBs. Hence all Temporary LOB storage guidelines apply to Value LOBs as well.

On the client side, Oracle recommends that you set the LOB prefetch size large enough to accommodate at least 80% of your LOB read size for Value LOBs.

11.2 Moving Data to LOBs in a Threaded Environment

Learn about the recommended procedure to follow while moving data to LOBs in this section.

There are two possible procedures that you can use to move data to LOBs in a threaded environment, one of which should be avoided.

Recommended Procedure

The recommended procedure is as follows:

1. `INSERT` an empty LOB, `RETURNING` the LOB locator.
2. Move data into the LOB using this locator.
3. `COMMIT`. This releases the ROW locks and makes the LOB data persistent.

Alternatively, you can use Data Interface to insert character data or raw data directly for the LOB columns or LOB attributes.

Procedure to Avoid

The following sequence requires a new connection when using a threaded environment, adversely affects performance, and is not recommended:

1. Create an empty (non-`NULL`) LOB
2. Perform `INSERT` using the empty LOB
3. `SELECT-FOR-UPDATE` of the row just entered
4. Move data into the LOB
5. `COMMIT`. This releases the `ROW` locks and makes the LOB data persistent.

11.3 LOB Access Statistics

Three session-level statistics specific to LOBs are available to users: LOB reads, LOB writes, and LOB writes unaligned.

Session statistics are accessible through the `V$MYSTAT`, `V$SESSTAT`, and `V$SYSSTAT` dynamic performance views. To query these views, the user must be granted the privileges `SELECT_CATALOG_ROLE`, `SELECT ON SYS.V_$MYSTAT` view, and `SELECT ON SYS.V_$STATNAME` view.

LOB reads is defined as the number of LOB API read operations performed in the session/system. A single LOB API read may correspond to multiple physical/logical disk block reads.

LOB writes is defined as the number of LOB API write operations performed in the session/system. A single LOB API write may correspond to multiple physical/logical disk block writes.

LOB writes unaligned is defined as the number of LOB API write operations whose start offset or buffer size is not aligned to the LOB block boundary. Writes aligned to block boundaries are the most efficient write operations. The usable LOB block size of a LOB is available through the LOB API (for example, using PL/SQL, by `DBMS_LOB.GETCHUNKSIZE()`).

It is important to note that session statistics are aggregated across operations to all LOBs accessed in a session; the statistics are not separated or categorized by objects (that is, table, column, segment, object numbers, and so on). Oracle recommends that you reconnect to the database for each demonstration to clear the `V$MYSTAT`. This enables you to see how the lob statistics change for the specific operation you are testing, without the potentially obscuring effect of past LOB operations within the same session.

> **See also:**
>
> *Oracle Database Reference*, appendix E, "Statistics Descriptions"

This example demonstrates how LOB session statistics are updated as the user performs read or write operations on LOBs.

```
rem
rem Set up the user
rem

CONNECT / AS SYSDBA;
SET ECHO ON;
GRANT SELECT_CATALOG_ROLE TO pm;
GRANT SELECT ON sys.v_$mystat TO pm;
GRANT SELECT ON sys.v_$statname TO pm;

rem
rem Create a simplified view for statistics queries
rem

CONNECT pm/pm;
SET ECHO ON;

DROP VIEW mylobstats;
CREATE VIEW mylobstats
AS
SELECT  SUBSTR(n.name,1,20) name,
        m.value             value
FROM    v$mystat    m,
        v$statname  n
WHERE   m.statistic# = n.statistic#
    AND n.name LIKE 'lob%';

rem
rem Create a test table
rem

DROP TABLE t;
CREATE TABLE t (i NUMBER, c CLOB)
    lob(c) STORE AS (DISABLE STORAGE IN ROW);

rem
rem Populate some data
rem
rem This should result in unaligned writes, one for
```

```
rem each row/lob populated.
rem

CONNECT pm/pm
SELECT * FROM mylobstats;
INSERT INTO t VALUES (1, 'a');
INSERT INTO t VALUES (2, rpad('a',4000,'a'));
COMMIT;
SELECT * FROM mylobstats;

rem
rem Get the lob length
rem
rem Computing lob length does not read lob data, no change
rem in read/write stats.
rem

CONNECT pm/pm;
SELECT * FROM mylobstats;
SELECT LENGTH(c) FROM t;
SELECT * FROM mylobstats;

rem
rem Read the lobs
rem
rem Lob reads are performed, one for each lob in the table.
rem

CONNECT pm/pm;
SELECT * FROM mylobstats;
SELECT * FROM t;
SELECT * FROM mylobstats;

rem
rem Read and manipulate the lobs (through temporary lobs)
rem
rem The use of complex operators like "substr()" results in
rem the implicit creation and use of temporary lobs. operations
rem on temporary lobs also update lob statistics.
rem

CONNECT pm/pm;
SELECT * FROM mylobstats;
SELECT substr(c, length(c), 1) FROM t;
SELECT substr(c, 1, 1) FROM t;
SELECT * FROM mylobstats;

rem
rem Perform some aligned overwrites
rem
rem Only lob write statistics are updated because both the
rem byte offset of the write, and the size of the buffer
rem being written are aligned on the lob block size.
rem

CONNECT pm/pm;
SELECT * FROM mylobstats;
DECLARE
    loc     CLOB;
    buf     LONG;
    chunk   NUMBER;
BEGIN
```

```
    SELECT c INTO loc FROM t WHERE i = 1
        FOR UPDATE;

    chunk := DBMS_LOB.GETCHUNKSIZE(loc);
    chunk = chunk * floor(32767/chunk); /* integer multiple of chunk */
    buf   := rpad('b', chunk, 'b');

    -- aligned buffer length and offset
    DBMS_LOB.WRITE(loc, chunk, 1, buf);
    DBMS_LOB.WRITE(loc, chunk, 1+chunk, buf);
    COMMIT;
END;
/
SELECT * FROM mylobstats;

rem
rem Perform some unaligned overwrites
rem
rem Both lob write and lob unaligned write statistics are
rem updated because either one or both of the write byte offset
rem and buffer size are unaligned with the lob's chunksize.
rem

CONNECT pm/pm;
SELECT * FROM mylobstats;
DECLARE
    loc CLOB;
    buf LONG;
BEGIN
    SELECT c INTO loc FROM t WHERE i = 1
        FOR UPDATE;

    buf := rpad('b', DBMS_LOB.GETCHUNKSIZE(loc), 'b');

    -- unaligned buffer length
    DBMS_LOB.WRITE(loc, DBMS_LOB.GETCHUNKSIZE(loc)-1, 1, buf);

    -- unaligned start offset
    DBMS_LOB.WRITE(loc, DBMS_LOB.GETCHUNKSIZE(loc), 2, buf);

    -- unaligned buffer length and start offset
    DBMS_LOB.WRITE(loc, DBMS_LOB.GETCHUNKSIZE(loc)-1, 2, buf);

    COMMIT;
END;
/
SELECT * FROM mylobstats;
DROP TABLE t;
DROP VIEW mylobstats;

CONNECT / AS SYSDBA
REVOKE SELECT_CATALOG_ROLE FROM pm;
REVOKE SELECT ON sys.v_$mystat FROM pm;
REVOKE SELECT ON sys.v_$statname FROM pm;

QUIT;
```