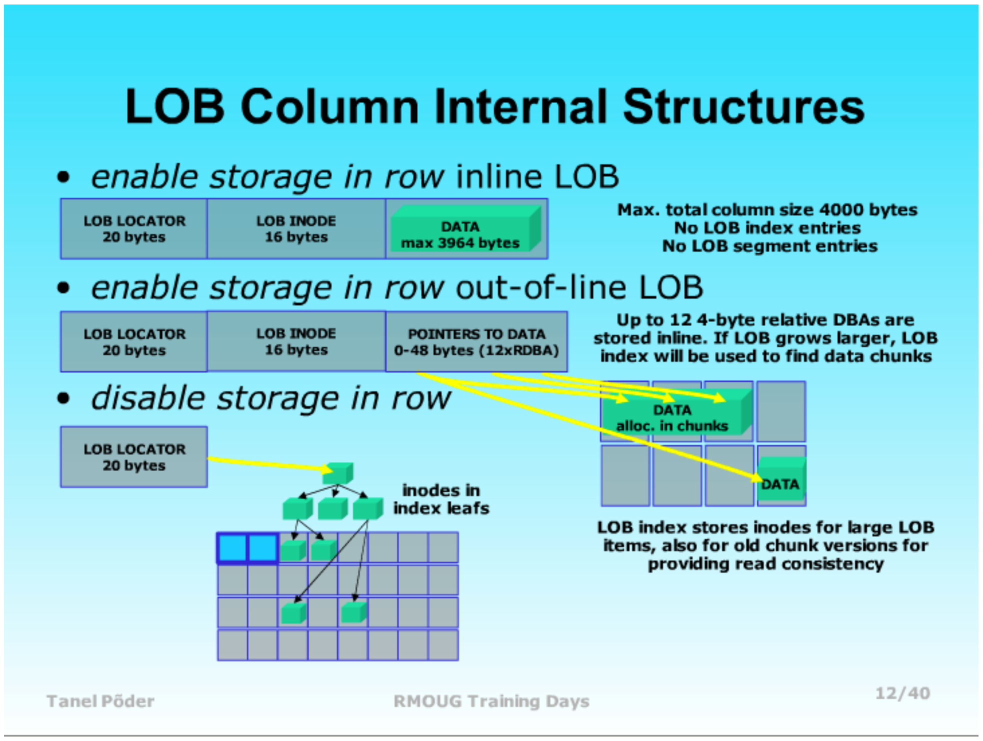
A technical review of LOB segments in Oracle

When Use/ How to Use/LOB Models/ Recommendations

When "DISABLE STORAGE IN ROW" is better than "ENABLE STORAGE IN ROW" on Oracle LOBs?


LOB Column Internal Structures
• enable storage in row inline LOB
LOB LOCATOR 20 bytes
LOB INODE 16 bytes
DATA max 3964 bytes
Max. total column size 4000 bytes
No LOB index entries
No LOB segment entries
• enable storage in row out-of-line LOB
LOB LOCATOR 20 bytes
LOB INODE 16 bytes
POINTERS TO DATA 0-48 bytes (12xRDBA)
Up to 12 4-byte relative DBAs are stored inline. If LOB grows larger, LOB index will be used to find data chunks
• disable storage in row
LOB LOCATOR 20 bytes
inodes in index leafs
DATA alloc. in chunks
DATA
LOB index stores inodes for large LOB items, also for old chunk versions for providing read consistency
Tanel Põder
RMOUG Training Days
12/40

Where Should We Use LOBs?

Large objects are suitable for semi-structured and unstructured data.

Large object features enable you to store the following types of data in the database and also in the operating system files that are accessed from the database.

- Semi-structured data

  Semi-structured data has a logical structure that is not typically interpreted by the database, for example, an XML document that your application or an external service processes. Oracle Database provides features such as Oracle XML DB, Oracle Multimedia, and Oracle Spatial and Graph to help your application work with semi-structured data.

- Unstructured data

  Unstructured data is easily not broken down into smaller logical structures and is not typically interpreted by the database or your application, such as a photographic image stored as a binary file.

Data unsuited for LOBs

- Simple Structured Data

  Simple structured data can be organized into relational tables that are structured based on business rules.

- Complex Structured Data

  Complex structured data is suited for the object-relational features of the Oracle Database such as collections, references, and user-defined types.

Inline and Out-of-Line LOB Storage

LOB columns store locators that reference the location of the actual LOB value.

Actual LOB values are stored either in the table row (inline) or outside of the table row (out-of-line), depending on the column properties you specify when you create the table, and depending the size of the LOB.

LOB values are stored out-of-line when any of the following situations apply:

- If you explicitly specify DISABLE STORAGE IN ROW for the LOB storage clause when you create the table.
- If the size of the LOB is greater than approximately 4000 bytes (4000 minus system control information), regardless of the LOB storage properties for the column.
- If you update a LOB that is stored out-of-line and the resulting LOB is less than approximately 4000 bytes, it is still stored out-of-line.

LOB values are stored inline when any of the following conditions apply:

- When the size of the LOB stored in the given row is small, approximately 4000 bytes or less, and you either explicitly specify ENABLE STORAGE IN ROW or the LOB storage clause when you create the table, or when you do not specify this parameter (which is the default).
- When the LOB value is NULL (regardless of the LOB storage properties for the column).

Using the default LOB storage properties (inline storage) can allow for better database performance; it avoids the overhead of creating and managing out-of-line storage for smaller LOB values. If LOB values stored in your database are frequently small in size, then using inline storage is recommended.

A technical review of LOB segments in Oracle

Note:

- LOB locators are always stored in the row.
- A LOB locator always exists for any LOB instance regardless of the LOB storage properties or LOB value - NULL, empty, or otherwise.
- If the LOB is created with DISABLE STORAGE IN ROW properties and the BasicFiles LOB holds any data, then a minimum of one CHUNK of out-of-line storage space is used; even when the size of the LOB is less than the CHUNK size.
- If a LOB column is initialized with EMPTY_CLOB() or EMPTY_BLOB(), then no LOB value exists, not even NULL. The row holds a LOB locator only. No additional LOB storage is used.
- LOB storage properties do not affect BFILE columns. BFILE data is always stored in operating system files outside the database.

Defining Tablespace and Storage Characteristics for Persistent LOBs

When defining LOBs in a table, you can explicitly indicate the tablespace and storage characteristics for each *persistent LOB* column.

To create a BasicFiles LOB, the BASICFILE keyword is optional but is recommended for clarity, as shown in the following example:

```
CREATE TABLE ContainsLOB_tab (n NUMBER, c CLOB)
    lob(c) STORE AS BASICFILE
    segname (TABLESPACE lobtbs1 CHUNK 4096
          PCTVERSION 5
          NOCACHE LOGGING
          STORAGE (MAXEXTENTS 5));
```

For SecureFiles, the SECUREFILE keyword is necessary, as shown in the following example (assuming TABLESPACE lobtbs1 is ASSM):

```
CREATE TABLE ContainsLOB_tab1 (n NUMBER, c CLOB)
    lob(c) STORE AS SECUREFILE
     sfsegname (TABLESPACE lobtbs1
            RETENTION AUTO
            CACHE LOGGING
            STORAGE (MAXEXTENTS 5));
```

Maximum Size of a LOB

The maximum permissible LOB size for your configuration depends on the block size setting of the tablespace. It is calculated as (4 gigabytes - 1)*(space usable for data in the LOB block). For example, if a LOB is stored in a tablespace of block size 8K, then the approximate maximum LOB size is about 32 terabytes.

A LOB can be up to 8 terabytes or more in size depending on your block size.

A LOB can be up to 128 terabytes or more in size depending on your block size.

First Extent of a SecureFile LOB Segment

A SecureFile LOB segment can only be created in Locally Managed Tablespace with Automatic Segment Space Management (ASSM). The number of blocks required in the first extent depends on the release. Before 21c, the first extent requires at least 16 blocks. After 21c, the number is 32 if the compatible parameter is greater than or equal to 20.1.0.0.0. Segments created in the previous release will continue to work in the new release. However, they will not be automatically upgraded.

The actual size of the first extent depends on the database block_size. If the tablespace is configured to use uniform extent, the extent must be bigger than the aforementioned number. For example, with block_size = 8k, the uniform extent size must be at least 128K pre-21c, or 256K on 21c with compatible parameter set. If the tablespace is configured to use uniform extent that is less than this number, the LOB segment creation will fail.

ENABLE or DISABLE STORAGE IN ROW:

LOB columns store locators that reference the location of the actual LOB value.

Actual LOB values are stored either in the table row (inline) or outside of the table row (out-of-line), depending on the column properties you specify when you create the table, and depending the size of the LOB.

The ENABLE | DISABLE STORAGE IN ROW clause is used to indicate whether the LOB should be stored inline (in the row) or out-of-line.

The default is ENABLE STORAGE IN ROW because it provides a performance benefit for small LOBs.

ENABLE STORAGE IN ROW:

If ENABLE STORAGE IN ROW is set, the maximum amount of LOB data stored in the row is 4000 bytes. This includes the control information and the LOB value.

If the LOB is stored IN ROW,

- Exadata pushdown is enabled for LOBs, including when using securefile compression and encryption.

- In-Memory is enabled for LOBs without securefile compression and encryption.

LOBs larger than approximately 4000 bytes are stored out-of-line. However, the control information is still stored in the row, thus enabling us to read the out-of-line LOB data faster.

DISABLE STORAGE IN ROW:

In some cases DISABLE STORAGE IN ROW is a better choice. This is because storing the LOB in the row increases the size of the row. This impacts performance if you are doing a lot of base table processing, such as full table scans, multi-row accesses (range scans), or many UPDATE/SELECT to columns other than the LOB columns.

You can find current Lob setting by:

```
SQL> select dbms_metadata.get_ddl( 'TABLE', 'T' ) from dual;
```

If you create the LOB column with DISABLE STORAGE IN ROW, then the LOB data is always stored out-of-line. LOB index is always used. Only the LOB ID is stored inline, and the ID is looked up in LOB index, where you'll get the pointers to actual LOB chunk blocks.

If you create the LOB column with ENABLE STORAGE IN ROW, then the LOB data may be stored in-line or out-of-line.

If the total LOB data + overhead <= 4000 bytes, then the LOB item will be stored in-line. No LOB index is used, even if you modify the LOB later on as everything is stored in-line with the row and versioning/rollback is achieved with undo data.

If the total LOB data + overhead > 4000 bytes, then the LOB item will be stored out-of-line. If the LOB fits into 12 x LOB_chunk_size, then no LOB index entries are created, because the in-line LOB locator can store up to 12 pointers to the LOB chunk blocks for each lob item. So if your LOB chunk size is 8kB, you can store LOB items up to 96kB in size without inserting anything to LOB index. However if the LOB item is bigger, then no pointers are stored in-row and all pointers will be put to the LOB index.

Note that once you modify an existing LOB item (which is bigger than 4000 bytes with its overhead), but smaller than 12 x chunk_size, then LOB index will still be used after the first LOB change operation as pointers to the old LOB chunk versions have to be stored in it (LOB segments don't rely on undo for rollback & consistency, but just use LOB chunk versioning managed by LOB index).

The "overhead" of an in-line LOB item is 36 bytes, so the actual LOB data must be 4000 – 36 = 3964 bytes or less in order to fully fit in-row. And note that we are talking about bytes here, not characters. With multi-byte character sets a character in a CLOB may take multiple bytes.

Setting a Persistent LOB to NULL

You may want to set a persistent LOB value to NULL upon inserting the row in cases where you do not have the LOB data at the time of the INSERT or if you want to use a SELECT statement, such as the following, to determine whether the LOB holds a NULL value:

```
SELECT COUNT (*) FROM TABLE WHERE LOB_COL IS NOT NULL;

SELECT COUNT (*) FROM TABLE WHERE LOB_COL IS NULL;
```

Note that you cannot call OCI or DBMS_LOB functions on a NULL LOB, so you must then use an SQL UPDATE statement to reset the LOB column to anon-NULL (or empty) value.

The point is that you cannot make a function call from the supported programmatic environments on a LOB that is NULL. These functions only work with a locator, and if the LOB column is NULL, then there is no locator in the row.

Setting a Persistent LOB to Empty

You can initialize a persistent LOB to EMPTY rather that NULL. Doing so, enables you to obtain a locator for the LOB instance without populating the LOB with data.

To set a persistent LOB to EMPTY, use the SQL function EMPTY_BLOB() or EMPTY_CLOB() in the INSERT statement:

```
SQL>INSERT INTO TABLE1 VALUES (EMPTY_BLOB());
```

As an alternative, you can use the RETURNING clause to obtain the LOB locator in one operation rather than calling a subsequent SELECT statement:

```
DECLARE

  Lob_loc  BLOB;

BEGIN

  INSERT INTO TABLE1 VALUES (EMPTY_BLOB()) RETURNING blob_col INTO Lob_loc;

   /*Now use the locator Lob_loc to populate the BLOB with data */

END;
```

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.

All LOBs

The following guidelines will help you get 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.

Performance Guidelines While Using Persistent LOBs

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

- Maximize writing to a single LOB in consecutive calls within a transaction. Frequently switching across LOBs or having interleaving DML statements prevent caching from reaching its maximum efficiency.
- Avoid taking savepoints or commiting too frequently. This neutralizes the advantage of caching while writing.

Note: Oracle recommends Securefile LOBs for storing persistent LOBs.

CACHE / NOCACHE / CACHE READS

When creating tables that contain LOBs, use the cache options according to the guidelines in below:

When to Use CACHE, NOCACHE, and CACHE READS

Cache Mode	Read	Write
CACHE READS	Frequently	Once or occasionally
CACHE	Frequently	Frequently
NOCACHE (default)	Once or occasionally	Never

CACHE / NOCACHE / CACHE READS: LOB Values and Buffer Cache

- CACHE: LOB pages are placed in the buffer cache for faster access.
- NOCACHE: As a parameter in the STORE AS clause, NOCACHE specifies that LOB values are not brought into the buffer cache.
- CACHE READS: LOB values are brought into the buffer cache only during read and not during write operations.

NOCACHE is the default for both SecureFiles and BasicFiles LOBs.

Note:

Using the CACHE option results in improved performance when reading and writing data from the LOB column. However, it can potentially age other non-LOB pages out of the buffer cache prematurely.

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 CLOBs/BLOBs, 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.
- 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

- 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.
- 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.

Moving Data to LOBs in a Threaded Environment

In this section I present 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.

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.

****

Regards,

Alireza Kamrani

Senior Database Consultant.