LOB Performance Guidelines

An Oracle White Paper
May 2004

EXECUTIVE OVERVIEW

This document gives a brief overview of Oracle's LOB data structure, emphasizing various storage parameter options and describes scenarios where those storage parameters are best used. The purpose of the latter is to describe the effects of LOB storage options.

LOBs were designed to efficiently store and retrieve large amounts of data. Small LOBs (< 1MB) perform better than LONGs for inserts, and comparable on selects. Large LOBs perform better than LONGs in general.

Oracle recommends the use of LOBs to store unstructured or semi-structured data, and has provided a LONG API to allow ease of migration from LONGs to LOBs. Oracle plans to de-support LONGs in the future.

LOB OVERVIEW

Whenever a table containing a LOB column is created, two segments are created to hold the specified LOB column. These segments are of type LOBSEGMENT and LOBINDEX. The LOBINDEX segment is used to access LOB chunks/pages that are stored in the LOBSEGMENT segment.

```
CREATE TABLE foo (pkey NUMBER, bar BLOB);

SELECT segment_name, segment_type FROM user_extents;
```

9792 is the object_id of the parent table FOO (if a table has more than one LOB column, LOB segment names are generated differently, use dba|user_lobs view to get parent table association).

SEGMENT_NAME	SEGMENT_TYPE
FOO	TABLE
SYS_IL0000009792C00002$$	LOBINDEX
SYS_LOB0000009792C00002$$	LOBSEGMENT (also referred as LOB chunks/pages)

The LOBSEGMENT and the LOBINDEX segments are stored in the same tablespace as the table containing the LOB, unless otherwise specified.[1]

[1] In Oracle8i, users can specify storage parameters for LOB index, but from Oracle9i onwards, specifying storage parameters for a LOB index is ignored without any error and the index is stored in the same tablespace as the LOB segment, with an Oracle generated index name.

IMPORTANT STORAGE PARAMETERS

This section defines the important storage parameters of a LOB column (or a LOB attribute) - for each definition we describe the effects of the parameter, recommendation for better performance and to avoid errors.

CHUNK

Definition

CHUNK is the smallest unit of LOBSEGMENT allocation. It is a multiple of DB_BLOCK_SIZE.

Points to Note

- For example, if the value of CHUNK is 8K and an inserted LOB is only 1K in size, then 1 chunk is allocated and 7K are wasted in that chunk. The CHUNK option does NOT affect in-line LOBs (see the definition in the next section)
- Choose an appropriate chunk size for best performance also to avoid space wastage. The maximum chunk size is 32K.
- The CHUNK parameter cannot be altered.

Recommendation

Choose a chunk size for optimal performance and minimum space wastage. For LOBs that are less than 32K, a chunk size that is 60% (or more) of the LOB size is a good starting point. For LOBs larger than 32K, choose a chunk size equal to the frequent update size.

In-line and Out-of-Line storage: ENABLE STORAGE IN ROW and DISABLE STORAGE IN ROW

Definition

LOB storage is said to be in-line when the LOB data is stored with the other column data in the row. A LOB can only be stored inline if its size is less than ~4000 bytes. For in-line LOB data, space is allocated in the table segment (the LOBINDEX and LOBSEGMENT segments are empty).

LOB storage is said to be out-of-line when the LOB data is stored, in CHUNK sized blocks in the LOBSEGMENT segment, separate from the other columns' data.

ENABLE STORAGE IN ROW allows LOB data to be stored in the table segment provided it is less than ~4000 bytes.

DISABLE STORAGE IN ROW prevents LOB data from being stored in-line, regardless of the size of the LOB. Instead only a 20-byte LOB locator is stored with the other column data in the table segment.

Points to Note

- In-line LOBs are subject to normal chaining and row migration rules within Oracle. If you store a 3900 byte LOB in a row with 2K block size then the row will be chained across two or more blocks. Both REDO and UNDO are written for in-line LOBs as they are part of the normal row data. The CHUNK option does not affect in-line LOBs.
- With out-of-line storage, UNDO is written only for the LOB locator and LOBINDEX changes. No UNDO is generated for chunks/pages in the LOBSEGMENT. Consistent Read is achieved by using page versions (see the RETENTION or PCTVERSION options).
- DML operations on out-of-line LOBs can generate high amounts of redo information, because redo is generated for the entire chunk. For example, in the extreme case, 'DISABLE STORAGE IN ROW CHUNK 32K' would write redo for the whole 32K even if the LOB changes were only 5 bytes.
- When in-line LOB data is updated, and if the new LOB size is greater than 3964 bytes, then it is migrated and stored out-of-line. If this migrated LOB is updated again and its size becomes less than 3964 bytes, it is not moved back in-line (except when we use LONG API for update).
- ENABLE|DISABLE STORAGE IN ROW parameters cannot be altered.

Recommendation

Use ENABLE STORAGE IN ROW, except in cases where the LOB data is not retrieved as much as other columns' data. In this case, if the LOB data is stored out-of-line, the biggest gain is achieved while performing full table scans, as the operation does not retrieve the LOB's data.

CACHE, NOCACHE

Definition

The CACHE storage parameter causes LOB data blocks to be read/written via the buffer cache.

With the NOCACHE storage parameter, LOB data is read/written using direct reads/writes. This means that the LOB data blocks are never in the buffer cache and the Oracle server process performs the reads/writes.

Points to Note

- With the CACHE option, LOB data reads show up as wait event 'db file sequential read', writes are performed by the DBWR process. With the NOCACHE option, LOB data reads/writes show up as wait events 'direct path read (lob)'/'direct path write (lob)'. Corresponding statistics are 'physical reads direct (lob)' and 'physical writes direct (lob)'.

- In-line LOBs are not affected by the CACHE option as they reside with the other column data, which is typically accessed via the buffer cache.
- The CACHE option gives better read/write performance than the NOCACHE option.
- The CACHE option for LOB columns is different from the CACHE option for tables. This means that caution is required otherwise the read of a large LOB can effectively flush the buffer cache.
- The CACHE|NOCACHE option can be altered.

Recommendation

Enable caching, except for cases where caching LOBs would severely impact performance for other online users, by forcing these users to perform disk reads rather than getting cache hits.

Consistent Reads on LOBs: RETENTION and PCTVERSION

Consistent Read (CR) on LOBs uses a different mechanism than that used for other data blocks in Oracle. Older versions of the LOB are retained in the LOB segment and CR is used on the LOB index to access these older versions (for in-line LOBs which are stored in the table segment, the regular UNDO mechanism is used). There are two ways to control how long older versions are maintained.

Definition

- RETENTION – time-based: this specifies how long older versions are to be retained.
- PCTVERSION – space-based: this specifies what percentage of the LOB segment is to be used to hold older versions.

Points to Note

- RETENTION is a keyword in the LOB column definition. No value can be specified for RETENTION. The RETENTION value is implicit. If a LOB is created with database compatibility set to 9.2.0.0 or higher, undo_management=TRUE and PCTVERSION is not explicitly specified, time-based retention is used. The LOB RETENTION value is always equal to the value of the UNDO_RETENTION database instance parameter.
- You cannot specify both PCTVERSION and RETENTION.
- PCTVERSION is applicable only to LOB chunks/pages allocated in LOBSEGMENTS. Other LOB related data in the table column and the LOBINDEX segment use regular undo mechanism.
- PCTVERSION=0: the space allocated for older versions of LOB data in LOBSEGMENTS can be reused by other transactions and can cause "snapshot too old" errors.

- PCTVERSION=100: the space allocated by older versions of LOB data can never be reused by other transactions. LOB data storage space is never reclaimed and it always increases.
- RETENTION and PCTVERSION can be altered

Recommendation

Time-based retention using the RETENTION keyword is preferred.

A high value for RETENTION or PCTVERSION may be needed to avoid "snapshot too old" errors in environments with high concurrent read/write LOB access.

LOGGING, NOLOGGING

Definition

LOGGING: enables logging of LOB data changes to the redo logs.

NOLOGGING: changes to LOB data (stored in LOBSEGMENTs) are not logged into the redo logs, however in-line LOB changes are still logged as normal.

Points to Note

- The CACHE option implicitly enables LOGGING.
- If NOLOGGING was set, and if you have to recover the database, then sections of the LOBSEGMENT will be marked as corrupt during recovery (LOBINDEX changes are logged to redo logs and are recovered, but the corresponding LOBSEGMENTs are not logged for recovery).
- LOGGING|NOLOGGING can be altered. The NOCACHE option is required to turn off LOGGING, e.g. (NOCACHE NOLOGGING).

Recommendation

Use NOLOGGING only when doing bulk loads or migrating from LONG to LOB. Backup is recommended after bulk operations.

PERFORMANCE GUIDELINE – LOB LOADING

In the rest of the document, you will notice LOB API and LONG API methods being referenced many times. The difference between these APIs is as follows:

LOB API: the LOB data is accessed by first selecting the LOB locator.

LONG API: the LOB data is accessed without using the LOB locator.

Points to Note

Use array operations for LOB inserts

APPENDIX A

LONG API access to LOB datatype

Oracle provides transparent access to LOBs from applications that use LONG and LONG RAW datatypes. If your application uses DML (INSERT, UPDATE, DELETE) statements from OCI or PL/SQL (PRO*C etc) for LONG or LONG RAW data, no application changes are required after the column is converted to a LOB.

For example, you can SELECT a CLOB into a character variable, or a BLOB into a RAW variable. You can define a CLOB column as SQLT_CHR or a BLOB column as SQLT_BIN and select the LOB data directly into a CHARACTER or RAW buffer **without selecting out the locator first.**

The following example demonstrates this concept:

```
create table foo (pkey number(10) not null, bar long raw) ;

set serveroutput on

declare
  in_buf           raw(32767);
  out_buf          raw(32767);
  out_pkey         number;
begin
in_buf := utl_raw.cast_to_raw (rpad('FF', 32767, 'FF'));

for j in 1..10 loop
  insert into foo values (j, in_buf) ;
  commit;
end loop;
dbms_output.put_line ('Write test finished ');

for j in 1..10 loop
  select pkey, bar into  out_pkey, out_buf from foo where pkey=j ;
end loop;
dbms_output.put_line ('Read test finished ');

end;
/
```

Now migrate LONG RAW column to BLOB column

```
alter table foo modify (bar blob);
```

When doing the LONG to LOB migration. The alter table migration statement runs serially in 9i. Indexes need to be rebuilt and statistics recollected.

After the LONG to LOB migration, the above PL/SQL block will work without any modifications.

Advanced LOB features may require the use of the LOB API, described in the Oracle Documentation[2]

[2] Large Objects (LOBs) in Oracle9*i* Application Developer's Guide, DBMS_LOB package in Oracle9*i* Supplied PL/SQL Packages and Types Reference, LOB and FILE Operations in Oracle Call Interface Programmer's guide

APPENDIX B

Migration from in-line to out-of-line (and out-of-line to in-line) storage

This section explains one major difference between the LOB API and LONG API methods.

If a change to the in-line LOB data makes it larger than 3964 bytes, then it is automatically moved out of table segment and stored out-of-line. If during future operations, the LOB data shrinks to under 3964 bytes, it will remain out-of-line.

In other words, once a LOB is migrated out, it is always stored out-of-line irrespective of its size, with the following exception scenario.

Consider a scenario where you used the LONG API to update the LOB datatype

```
[..]
begin
   in_buf := utl_raw.cast_to_raw (rpad('FF', 3964, 'FF'));
   insert into foo values (1, in_buf) ;
   commit;
[..]
```

Above LOB is stored in-line, update the LOB to a size more than 3964 bytes

```
[..]
   in_buf := utl_raw.cast_to_raw (rpad('FF', 4500, 'FF'));
   update foo set bar=buffer where pkey=1;
   commit;
[..]
```

After the update LOB is stored out-of-line, now update the LOB to a size smaller than 3964 bytes

```
[..]
   in_buf := utl_raw.cast_to_raw (rpad('FF', 3000, 'FF'));
   update foo set bar=buffer where pkey=1;
   commit;
[..]
```

LOB is stored in-line again.

When using the LONG API for update, the **older LOB is deleted** (or space is reclaimed as per RETENTION or PCTVERSION setting) and a **new LOB is created, with a new LOB locator**. This is different from using LOB API, where DML on LOB is possible only using the LOB locator (the LOB locator doesn’t change)

APPENDIX C

How LOB data is stored

The purpose of this section is to differentiate how the ENABLE STORAGE IN ROW option is different from the DISABLE STORAGE IN ROW option for LOB data size greater than 3964 bytes. It also highlights when LOBINDEX is actually used (following example scenarios assume Solaris OS and Oracle 9204 – 32 bit version).

In-line LOB – LOB size less than 3964 bytes

LOB can be NULL, EMPTY_BLOB, and actual LOB data

```
create table foo (pkey number(10) not null, bar BLOB)
lob (bar) store as (enable storage in row chunk 2k);

declare
   inbuf     raw(3964);

begin
   inbuf := utl_raw.cast_to_raw(rpad('FF', 3964, 'FF'));
   insert into foo values (1, NULL);
   insert into foo values (2, EMPTY_BLOB() );
   insert into foo values (3, inbuf );
   commit;
end;
/
```

Foo table rows

Pkey=1	Bar=0 byte (nothing is stored)
Pkey=2	Bar=36 byte
Pkey=3	Bar=4000 byte (36 byte + 3964 byte of LOB data, nothing stored in LOBINDEX and LOBSEGMENT

LobId – LOB Locator

In-line LOB – LOB size = 3965 bytes (1 byte greater than 3964)

LOB is defined as in-line, but actual data is greater than 3964 bytes, so moved out – please note this is different from LOB being defined as out-of-line.

```
[..]
   inbuf := utl_raw.cast_to_raw(rpad('FF', 3965, 'FF'));
   insert into foo values (4, inbuf );
[..]
```

Foo table row

Pkey=4	Bar=40 bytes. Oracle directly accesses LOB data in LOBSEGMENT. Nothing stored in LOBINDEX

In-line LOB – LOB size greater than 12 chunk addresses

With in-line LOB option, we store the first 12 chunk addresses in the table row. This takes 84 bytes of size in table row. LOBs that are less than 12 chunks in size will not have entries in the LOBINDEX if ENABLE STORAGE IN ROW is used

```
[..]
    inbuf := utl_raw.cast_to_raw(rpad('FF', 32767, 'FF'));
    insert into foo values (5, inbuf );
[..]
```

Here, we are inserting 32767 bytes of LOB data, given our chunk size of 2k, we need approximately 16 blocks (32767/2048). So we store first 12 chunk addresses in table row and the rest in LOBINDEX

Foo table row

Pkey=5	Bar=84 bytes. Oracle directly accesses 12 LOB chunks in LOBSEGMENT. Then Oracle looks up LOBINDEX to get the rest of the LOB chunk addresses.

Out-of-line LOBs – All LOB sizes

With out-of-line LOB option, only LOB locator is stored in table row. Using LOB locator, we lookup LOBINDEX and find the range of chunk addresses, using these addresses we read LOB data from the LOBSEGMENT

```
create table foo (pkey number(10) not null, bar BLOB)
lob (bar) store as (disable storage in row chunk 2k);

[..]
    inbuf := utl_raw.cast_to_raw(rpad('FF', 20, 'FF'));
    insert into foo values (6, inbuf );
[..]
```

Foo table rows

Pkey=6	Bar=20 bytes. All chunk addresses are stored in LOBINDEX.

LOB Performance Guidelines
May 2004
Author: V. Jegraj
Contributing authors: Geoff Lee, Vishy Karra, Krishna Kunchithapadam, Cecilia Gervasio

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
www.oracle.com