# MANAGING UNSTRUCTURED DATA: LOBS, SECUREFILES, BASICFILES

By Michael Rosenblum, Grigoriy Novikov, Dulcian, Inc

If you start learning about databases, 99% of time the focus is on the three core datatypes (DATE, NUMBER, VARCHAR2). However, in reality, the media (pictures, movies, documents, and sounds) represent the largest and the fastest growing part of any contemporary IT system. As a result, correct handling of such data is as critical to the overall project success as the effective manipulation of financial information or number-crunching. Also, since the total volume of media content is usually distributed across a reasonably small number of attributes, the cost of mishandling of each of them is much higher.

It is common for all database solutions (not only Oracle) to utilize a class of datatypes designed to work with large objects called LOBs. For each version of the Oracle RDBMS, the maximum size may differ (currently, the limit is 8-128TB depending upon the configuration). These LOBs can be divided into two groups based on the way in which the data is stored:

- *Internal large objects* are stored within the database itself and are being accessed by special mechanisms (described below), which are separate from regular table data access. There are three datatypes of internal LOBs (all supported by multiple platforms):
  - » BLOBs are used to store binary information (usually multimedia).
  - » CLOBs are used for textual information.
  - » NCLOB is used to store information in the National Character Set (similar to CLOB).
- *External large objects* are stored in the file system and only the pointer is stored in the database. This pointer is represented via the Oracle-proprietary BFILE datatype. BFILE is used to point to files stored in the operating system and provides *read-only* access to these files. If you need to write to operating system files, you should either use the UTL_FILE package or custom-built Java procedures.

This article describes some important factors to keep in mind when working with large objects in a system. Before jumping into the details, it is worthwhile to mention that in version 11gR1, Oracle introduced an extended internal storage mechanism for handling CLOB/BLOB called *SecureFile* (in beta-releases and in some early papers it was also called *FastFile*) to differentiate it from a traditional *BasicFile* implementation.

- ***BasicFile:***
  - » Exactly the same implementation as in 9*i*/10*g*
  - » Kept for backward compatibility
  - » No extra licenses required
- ***SecureFile:***
  - » Introduced in 11gR1
  - » Already extended in 11gR2
  - » May be extended even further (a number of SecureFile-related errors mention "unimplemented feature," which suggests that some modifications are expected in future releases)
  - » Many interesting features, but some require additional licenses

Since many organizations are still using 10*g* or 11*g* but have not yet switched to SecureFile solutions, this paper will explicitly outline behavior differences (if any exist).

## UNDERLYING CONCEPTS

There are some issues specific to large objects that you should understand before viewing the actual code syntax associated with them. From the average developer's point of view, such low-level details may seem irrelevant (or DBA's business at all), but at Dulcian we strongly believe that the implications of any advanced feature should be clear to all involved parties. We also believe that the "black box" approach to Oracle internals by development teams is completely unproductive and leads to significant problems later in the project lifecycle. That is why, please, do not be surprised if in this article we will be crossing department boundaries (and, yes, you may really *talk* to your DBA afterwards).

### Data Access

Since you may have gigabytes of data in each field of a column in your system, the problem of accessing the data becomes the focus of the whole architecture. Oracle has a fairly elegant solution, namely to separate the content itself from the mechanism of its accessing. This results in two separate entities: LOB data and an LOB *locator* that points to LOB data and allows communication with it.

To understand this data structure, imagine a huge set of barrels with water and a pipe that can take water from the barrel, do something with it, and put it back. The following analogy makes the whole picture clear. If you want to make a barrel (LOB) accessible by a different person (sub-routine), you don't need to extract and pass the whole amount of water (LOB data); you just need to pass the pipe (locator) pointing to the right barrel. But if you need to pour water from one barrel to the other, the same pipe can be used as a tunnel. Using this analogy is helpful when examining the two types of LOB operations:

1. ***Copy semantics*** are used when the data alone is copied from the source to the destination and a new locator is created for the new LOB.
2. ***Reference semantics*** are used when only the locator is copied without any changes to the underlying data.

### Data States

Any attribute may have either a NULL or NOT NULL value by the nature of relational database theory. Because of the existence of locators, from the practical point of view, LOBs have three possible data states:

- ***Null*** – The variable or column in the row exists, but is not initialized.
- ***Empty*** – The variable or column in the row exists and has a locator, but that locator does not point to any data. IS NULL check will return FALSE for an Empty state.
- ***Populated*** –The variable or column in the row exists, has

a locator, and contains data of non-zero length.

The Empty state is very important. Because you can access LOBs only via locators, you must first create them. In some environments, you must have an initial NULL value, but for PL/SQL activities it makes sense to immediately initialize any LOB column as Empty to save extra steps.

**Data Storage**

LOBs can be tricky from the DBA's point of view. If external LOBs (BFILE) are simply pointers to files stored in the operating system, internal LOBs leave a lot of space for configuration. There are two different types of internal LOBs:

1. ***Persistent LOBs*** are represented as values in the column of a table. As a result, they participate in the transaction (changes could be committed/rolled back) and generate logs (if configured to do so).
2. ***Temporary LOBs*** are created when you instantiate the LOB variable, but when you insert the temporary LOB into the table, it becomes a persistent LOB.

Since, by design, LOBs are created to support large volumes of data, it is completely logical that these datatypes also include extended methods of handling UNDO retention. These methods became even more critical since the introduction of FLASHBACK functionality, because inappropriate generation of UNDO for LOB columns can significantly increase the space requirements in order to guarantee the required *retention period*. Currently, the following options are available:

1. ***BasicFile***
   - a. ***Disabled (default)*** – only support consistent reads and do not participate in the FLASHBACK logic
   - b. ***Enabled*** – the same UNDO_RETENTION parameter should be applied to the LOB column as to regular data
2. ***SecureFile***
   - a. ***Auto (default)*** – only support consistent reads and do not participate in the FLASHBACK logic
   - b. ***None*** – do not generate UNDO at all
   - c. ***MAX <N>*** – keep up to N megabytes of UNDO
   - d. ***MIN <N>*** – guarantee up to N seconds of retention. This allows setting a different value from an overall UNDO_RETENTION setting.

The major difference between LOBs and other datatypes is that even variables are not created in memory. Everything is happening via physical storage. Temporary LOBs are created in the temporary tablespace and released when they are not needed any longer. With persistent LOBs, each LOB attribute has its own storage structure separate from the table in which it is located. As usual in Oracle, each storage structure is represented as a separate *segment*.

If regular table data is stored in blocks, LOB data is stored in *chunks*. Each chunk may consist of one or more database blocks (up to 32KB). Setting the chunk size may have significant performance impacts since Oracle reads/writes one chunk at a time. The wrong chunk size can significantly increase the number of I/O operations. In SecureFile implementation chunks are dynamic (in an attempt to allocate as much continuous space as possible) and cannot be managed manually (at least for now).

To navigate chunks, Oracle uses a special *LOB index* (also physically represented as a separate segment). As a result, each LOB column has two associated segments: one to store data and one to store the index. These segments have the same properties as regular tables: tablespace, initial extend, next extend, etc. The ability to articulate the physical storage properties for each internal LOB column can come in handy for making the database structure more manageable. You can locate a tablespace on a separate drive, set different block size, etc. In some versions of Oracle you can even specify different properties for the index and data segments. Currently, they must be the same, and there are restrictions on what you can do with LOB indexes. For example, you cannot drop or rebuild them.

**Performance Considerations**

Each operation with an LOB chunk requires physical I/O. As a result, you may end up with a high number of I/O-related wait events in the system. But it is reasonable to ask the following question: Why place data in the special storage structure if you only have a small amount of data in some rows? Using an online ordering system as an example, you might have remarks about some goods that only require between 1KB and 1MB of space. To handle such cases, Oracle allows you to *store data in the row* (instead of outside of the row) if you have less than 3964 bytes. This causes all small remarks to be processed as if they are regular VARCHAR2(4000) columns. When their size exceeds this limit, the data will be moved to LOB storage. In some cases, you might consider disabling this feature since, but in almost all cases this is the best option.

Another critical performance question is to figure out how all operations with such large data volumes would impact the buffer cache. Oracle provides enough flexibility to adjust the *caching option* in a number of ways:

- NOCACHE is the default value. It is designed to be used only if you are rarely accessing the LOBs or the LOBs are extremely large. From a physical standpoint, existing implementations are completely different:
  - » ***BasicFile*** – Use DirectRead/DirectWrite. Even these mechanisms allow tunneling to the storage of a lot of data. In the I/O-active system (especially OLTP) they could cause significant "hiccups".
  - » ***SecureFile*** – Utilize a special shared pool area (managed by SHARED_IO_POOL).
- CACHE is the best option for LOBs requiring a lot of read/write activity.
- CACHE READS help when you create the LOB once, read data from it, and the size of LOBs to be read in the system at any time does not take too much space out of the buffer pool. "Write" processes are implemented in the same was as the NOCACHE option.

If your database is running in ARCHIVELOG mode (as the majority of databases are), the problem of generating too many logs becomes a real headache for DBAs. Since LOBs have their own storage segments, from the very beginning it was possible to set up its *logging option* which may be different from the table owning the LOB column. Unfortunately, having NOLOGGING for a column in case of a catastrophic crash meant that whole rows would not be accessible until LOB columns were reset to a stable state. To solve this problem the "SecureFile" mechanism introduced the FILESYSTEM_LIKE_LOGGING option, which preserved all metadata while not logging any changes to the LOB itself. This makes the whole table accessible even in the case of a major failure or switchover to a standby. This option may be viable if the data in CLOB could be easily retrieved from other sources or of a temporal nature. CACHE/CACHE READS options always imply LOGGING option (for all implementations).

### Enabling SecureFile

It must be mentioned that Oracle also introduced a special parameter to handle usage of the SecureFile storage mechanism – DB_SECUREFILE. This parameter has one of the following values:

- ***Permitted (Default)*** – BasicFile is created unless SecureFile is explicitly specified
- ***Always*** – SecureFile is created unless BasicFile is explicitly specified
- ***Force*** – always create SecureFile
- ***Ignore*** – allow creation of SecureFile attributes, but treat them as BasicFile (all SecureFile features are disabled)
- ***Never*** – raise exceptions if SecureFile storage mechanism is selected

Selection of the appropriate parameter should be driven by your system policies, but we recommend to use either FORCE or NEVER to preserve code consistency.

## STANDARD USE OF LOBS

The following code assumes the example of an online shopping catalog of electronic goods where each record contains the name of the item, user manual text, front page image, and a link to the original text file, with the manual stored on the server. Because of the nature of the required data, LOB datatypes are clearly unavoidable so, the table code might look like the following:

```
CREATE TABLE goods_tab
  (item_id      NUMBER PRIMARY KEY,
   name_tx      VARCHAR2(256),
   remarks_cl   CLOB DEFAULT EMPTY_CLOB(),
   manual_cl    CLOB DEFAULT EMPTY_CLOB(),
   firstpage_bl BLOB DEFAULT EMPTY_BLOB(),
   mastertxt_bf BFILE)
LOB(remarks_cl) STORE AS SecureFile remarks_seg(
   TABLESPACE USERS
   ENABLE STORAGE IN ROW
   CACHE)
LOB(manual_cl) STORE AS SecureFile manual_seg(
   TABLESPACE LOBS_BIG
   DISABLE STORAGE IN ROW
   NOCACHE
   FILESYSTEM_LIKE_LOGGING)
LOB(firstpage_bl) STORE AS BasicFile firstpage_seg(
   TABLESPACE LOBS_BIG
   DISABLE STORAGE IN ROW
   CHUNK 32768
   CACHE READS)
```

This example includes all three datatypes: CLOB, BLOB, BFILE and two implementations (BasicFile, SecureFile). Also, each internal LOB has its own storage block at the end of the table definition. There are a number of factors to consider when using this approach:

1. Each internal LOB has explicit segment names (REMARKS_SEG, MANUAL_SEG, FIRSTPAGE_SEG) instead of system-generated ones. This is done for the convenience of working with the USER_SEGMENTS dictionary view.
2. Since we are planning to work with LOBs in PL/SQL, all internal LOBs are initialized to empty values (so they now contain a locator that could be retrieved) via special functions - EMPTY_CLOB() and EMPTY_BLOB( ).
3. The column REMARK_CL is accessed and modified very often, but the amount of data is not very large. Therefore, the best option is to place the column in the same tablespace as the main data and enable storage "in row" since a significant number of values could be less than 4000 characters. The cache option should also be enabled for performance optimization. Since a lot of people will be working with that column, you do not want to generate extra wait events because of direct read/direct write operations.
4. The column MANUAL_CL is accessed not very often and always can be reloaded from master files. That's why the independent tablespace, no storage "in row," and no caching options are appropriate here. FILESYSTEM_LIKE_LOGGING is also safe to use because the data could be easily reproduced.
5. The difference between FIRSTPAGE_BL and MANUAL_CL is that although this column will never be updated, it could be read by different users often enough. This is the reason why you should enable caching on reads.

This table also illustrates that both BasicFile and SecureFile storage mechanisms can be used in the same table simultaneously, although, for the sake of code maintenance, this situation should be avoided unless there are good reasons.

***NOTE:*** As of version 11.2.0.2, Oracle Datapump utility has issues with tables containing LOB columns with different types of CACHE setting (bug #1313537.1). The data load crashes with ORA-07445 on the kernel level. It should be fixed in 11.2.0.3, but currently the only workaround is to use EXP/IMP for such tables.

## SAMPLE BASIC LOB OPERATION

For the sake of saving the space this article does not include a lot of basic LOB examples since it was not designed to be a complete reference, although, the most popular one will be used, because many developers eventually think about a way of loading a binary file in the database. Here is a code snippet that loads a picture *frontPage_PLSQL_Exper.jpg* (stored in the directory I/O) into a BLOB column of the table GOODS_TAB

```
DECLARE
   v_file_bf  BFILE:= BFILENAME ('IO','frontpage_PLSQL_
Expert.jpg');
   v_firstpage_bl   BLOB;
   src_offset       NUMBER := 1;
   dst_offset       NUMBER := 1;
BEGIN
   SELECT firstpage_bl
   INTO v_firstpage_bl
   FROM goods_tab
   WHERE item_id = 1
   FOR UPDATE OF firstpage_bl;
```

```
      DBMS_LOB.FILEOPEN (v_file_bf, DBMS_LOB.FILE_RE-
ADONLY);
      DBMS_LOB.LOADBLOBFROMFILE (v_firstpage_bl,
      v_file_bf,
     DBMS_LOB.GETLENGTH (v_file_bf),
      dst_offset, src_offset);
      DBMS_LOB.FILECLOSE (v_file_bf);
   END;
```

This code illustrates the real meaning of locators. There is no UPDATE in the block, but the value in the table will be changed. Using SELECT… INTO… FOR UPDATE locks the record and returns the locator back to the LOBs. Because of the lock, this locator contains the ID of the current transaction (more about transaction issues a bit later). This means that you cannot only read data from the LOB, but also write to the LOB. Using the "barrel and pipe" analogy, you have your own tube and your own barrel to do whatever you want. The way to read data is very straightforward: open the file via locator, read the data, close the file. Source and destination offset parameters are also very interesting. They are of type IN/OUT and originally specify the starting points for reading and writing. But when the procedure call is completed, they are set to the ending points. That way you always know how many bytes (for BLOB) and characters (for CLOB) were read, and how many of them were written.

Another useful thing to notice is a BFILENAME constructor. It takes a directory and file name as input and generates a BFILE pointer. Be careful with this feature since this constructor does NOT check whether the file really exists. In the example, the real check would happen only when DBMS_LOB.FILEOPEN is called.

## SECUREFILES EXTRAS (FOR EXTRA MONEY)

Since different production companies have different IT budgets, this article focuses on the "common denominator" (features available in all editions and for all possible installation types). But it is worth mentioning a number of advanced options introduced with SecureFile storage implementation despite the additional licensing fees involved:

- "Oracle Advanced Compression Option" gives you access to:
  - » De-duplication – preservation of only one copy of LOB in the same table if values match exactly
  - » Compression (High/Medium/Low) – built-in basic archive utility to compress the data – technically, a trade-off between CPU and extra storage
- "Oracle Advanced Security Option" gives you access to:
  - » Encryption – direct implementation of Transparent Data Encryption per LOB column

Of course, results may vary, but our opinion about these extra options is as follows:

- ***De-duplication*** – useless for smaller systems. Could save some space for a large system where the same file could be sent to hundreds of people.
- ***Encryption*** – it is always nice to have higher granularity of what you can and what you cannot encrypt.
- ***Compression*** – definitely makes sense in a lot of cases, but should not be applied blindly because of CPU cost.

The following example illustrates the impact of both compression and de-duplication on the space allocation. First, it is necessary to set up the test case and populate it with data. Note that sample data is generated by DBMS_RANDOM.STRING and may be too chaotic compared with the real documents, but as a worst case scenario, it should work just fine.

```
   -- create empty table with one CLOB column
   CREATE TABLE secure_tab (demo_cl CLOB)
   LOB(demo_cl) STORE AS SecureFile demo_seg(
     COMPRESS HIGH
     DEDUPLICATE)
   -- load 20 exactly the same CLOB (each has 1MB of data)
   DECLARE
     v_cl CLOB;
   BEGIN
    DBMS_LOB.CREATETEMPORARY(v_cl,true,dbms_lob.
call);
     FOR i IN 1..250 LOOP
       DBMS_LOB.WRITEAPPEND(v_cl,4000,DBMS_RAN-
DOM.STRING('x',4000));
     END LOOP;

     FOR i IN 1..20 LOOP
       INSERT INTO secure_tab VALUES (v_cl);
     END LOOP;

     COMMIT;
   END;
```

The test itself will gradually decrease the compression level and finally disable de-duplication.

```
SQL> DECLARE
   v_tx VARCHAR2(99):='ALTER TABLE secure_tab'||
                 ' MODIFY LOB(demo_cl) ';
   PROCEDURE p_print (pi_type_tx VARCHAR2) IS
     v_seg_blocks_nr       NUMBER;
     v_seg_bytes_nr        NUMBER;
     v_used_blocks_nr      NUMBER;
     v_used_bytes_nr       NUMBER;
     v_expired_blocks_nr   NUMBER;
     v_expired_bytes_nr    NUMBER;
     v_unexpired_blocks_nr NUMBER;
     v_unexpired_bytes_nr  NUMBER;
   BEGIN
     DBMS_SPACE.SPACE_USAGE(
      user,'DEMO_SEG','LOB',
      partition_name     => NULL,
      segment_size_blocks=> v_seg_blocks_nr,
      segment_size_bytes => v_seg_bytes_nr,
      used_blocks        => v_used_blocks_nr,
      used_bytes         => v_used_bytes_nr,
      expired_blocks     => v_expired_blocks_nr,
      expired_bytes      => v_expired_bytes_nr,
      unexpired_blocks   => v_unexpired_blocks_nr,
      unexpired_bytes    => v_unexpired_bytes_nr);
     DBMS_OUTPUT.PUTLINE (pi_type_tx||':seg-'||
      v_seg_bytes_nr||'/used-'|| v_used_bytes_nr);
```

```
  END;
BEGIN
  p_print('High Compress');
  EXECUTE IMMEDIATE v_tx||'(COMPRESS MEDIUM)';
  p_print('Medium Compress');
  EXECUTE IMMEDIATE v_tx||'(COMPRESS LOW)';
  p_print('Low Compress');
  EXECUTE IMMEDIATE v_tx||'(NOCOMPRESS)';
  p_print('No Compress');
  EXECUTE IMMEDIATE v_tx||'(KEEP_DUPLICATES)';
  p_print('Keep dups');
END;
/
High Compress:    seg-1245184/used-688128
Medium Compress:  seg-2293760/used-712704
Low Compress:     seg-3342336/used-1024000
No Compress:      seg-3342336/used-1032192
Keep dups:        seg-22216704/used-20488192
```

The results of the test clearly show that both of Oracle's new features work exactly as specified:

- The highest level of compression provides the most space saving (688,128 bytes vs 1,032,192 bytes) while lower levels are less efficient.
- De-duplication keeps only one copy of the data.

Although, it should be mentioned that segment allocation patterns may differ from space usage patterns (as shown in the example), on average, they are closely related. This is a topic for a much more detailed discussion and outside of the scope of this article.

## SPECIAL CASES AND PROBLEMS

The level of complexity introduced by LOBs requires a number of restrictions to be placed on their use, even though later versions of Oracle attempt to remove as many of them as possible. There are three major areas of concern: generic restrictions, string processing problems, and transaction limitations which are discussed here.

### Generic Restrictions

From Oracle 10g onwards, there are things that just cannot be done with LOB datatypes at all:

***1. SQL activity restrictions:***

a. You cannot have LOB columns in ORDER BY or GROUP BY clauses or any aggregate functions.

b. You cannot have an LOB column in a SELECT DISTINCT statement.

c. You cannot join two tables using LOB columns.

d. Direct binding of string variables is limited to 4000 characters if you are passing a string into the CLOB column. This restriction is a bit tricky and requires an example. In the following code, the first output will return 4000 (because string was directly passed into the UPDATE statement), but in the second case, the output will be 6000 (because the string was passed via PL/SQL variable).

```
DECLARE
   v_tx VARCHAR2(6000):=LPAD('*',6000,'*');
   v_count_nr NUMBER;
BEGIN
   UPDATE goods_tab
   SET remarks_cl =LAPD('*',6000,'*')
   WHERE item_id = 1
   RETURNING LENGTH(remarks_cl) INTO v_count_nr;
   DBMS_OUTPUT.PUT_LINE('Length:'||v_count_nr);

   UPDATE goods_tab
   SET remarks_cl =v_tx
   WHERE item_id = 1
   RETURNING LENGTH(remarks_cl) INTO v_count_nr;
   DBMS_OUTPUT.PUT_LINE('Length:'||v_count_nr);
END;
```

***2. DDL restrictions:***

a. LOB columns cannot be a part of a primary key.

b. LOB columns cannot be a part of an index (unless you are using a domain index, Oracle Text or Function-Based index).

c. You cannot specify an LOB column in the trigger clause FOR UPDATE OF.

d. If you change LOBs using the locator with the DBML_LOB package, no update trigger is fired on the table. This is extremely critical to know if your system audit is based on triggers. Although this practice is not considered very efficient, it seems to be in use quite often. Be sure that your DBAs and management are aware of it.

***3. DBLink restrictions:***

a. You can only use CREATE TABLE AS SELECT and INSERT AS SELECT if the remote table contains LOBs. No other activity is permitted.

***4. Administration restrictions:***

a. Only a limited number of BFILEs can be opened at the same time. The maximum number is set up by the initialization parameter SESSION_MAX_OPEN_FILES. The default value is 10, but it can be modified by the DBA.

b. Once a table with an internal LOB is created, only some LOB parameters can be modified. You can change the tablespace, storage properties, caching options, but you cannot modify the chunk size, or storage-in-the-row option.

### String Restrictions

Oracle tries to simplify string activities for CLOBs by including overloads of standard built-in functions to support larger amounts of data. You can now also use explicit conversions of datatypes. For example, you can assign a CLOB column to a VARCHAR2 PL/SQL variable as long as it can hold all of the data from the CLOB. Conversely, you can initialize a CLOB variable with a VARCHAR2 value. As a result, there are some activities that could be done using SQL semantics (built-in functions) or API semantics (DBMS_LOB package).

Although SQL semantics are much easier to work with, there are still some issues with such overloads. Recently we discovered the following bug with the specified set of actions:

- REPLACE command is fired against a CLOB variable when trying to change 'T_' to 'ZZ'.
- In the CLOB, there is a part of text that looks like 'TA'

Surprisingly, Oracle replaces both 'T_' and 'TA' with 'ZZ': it appears that the underscore is being interpreted as a wild-card! Eventually we discovered that it is a known bug #4598943 first detected in 9.2.0.6 and only fixed in 11g:

```
DECLARE
  v1_cl CLOB := 'T_TABLE(a NUMBER)';
BEGIN
  v1_cl := replace (v1_cl, 'T_', 'ZZ');
  DBMS_OUTPUT.PUT_LINE(v1_cl);
END;

Received result: ZZZZBLE(a NUMBER)
Expected result: ZZTABLE(a NUMBER)
```

There are also some differences between PL/SQL code and SQL statements (even inside of PL/SQL routines) from the perspective of what you can and cannot do with LOBs. You can compare LOBs (>,!=, between) only as a part of a PL/SQL routine as shown here:

```
DECLARE
 v_remarks_cl CLOB;
 v_manual_cl CLOB;
BEGIN
 SELECT remarks_cl, manual_cl
 INTO v_remarks_cl, v_manual_cl
 FROM goods_tab
 WHERE item_id = 1;
 --AND remarks_cl!=manual_cl -- INVALID

 IF v_remarks_cl!=v_manual_cl THEN -- VALID
   DBMS_OUTPUT.PUT_LINE('Compared');
 END IF;
END;
```

Using SQL semantics could get you into a lot of trouble with some built-in functions. INITCAP, SOUNDEX, TRANSLATE, DECODE and some other functions will process only the first 4K of your data if used in embedded sQL. In the following example, the second statement will raise the exception because TRANSLATE could not process 6000 characters:

```
DECLARE
 v_tx VARCHAR2(6000):=LPAD('a',6000,'a');
 v_count_nr NUMBER;
BEGIN
 UPDATE goods_tab
 SET remarks_cl =v_tx
 WHERE item_id = 1
 RETURNING LENGTH (remarks_cl) INTO v_count_nr;
 DBMS_OUTPUT.PUT_LINE('Length:'||v_count_nr);
 -- this will fail!
 UPDATE goods_tab
 SET remarks_cl = TRANSLATE (remarks_cl,'a','A')
 WHERE item_id = 1
 RETURNING LENGTH(remarks_cl) INTO v_count_nr;
 DBMS_OUTPUT.PUT_LINE('Length:'||v_count_nr);
END;
```

**Transaction Restrictions**

There are a number of restrictions when using LOBs for transaction control:

1. Each locator may or may not contain a transaction ID.
   - If you already started a new transaction (SELECT FOR UPDATE, INSERT/UPDATE/DELETE, PRAGMA AUTONOMOUS TRANSACTION), your locator will contain the transaction ID.
   - If you use SELECT FOR UPDATE of an LOB column, the transaction is started implicitly and your locator will contain the transaction ID.
2. You cannot read using the locator when it contains an old transaction ID (for example, you made a number of data changes and committed them) and your session parameter TRANSACTION LEVEL is set to SERIALIZABLE. This is a very rare case.
3. First write using the locator:
   - You need to have a lock on the record containing the LOB that you are updating at the point when you are trying to perform the update (not necessarily at the point of acquiring of the locator). That lock could be the result of SELECT FOR UPDATE, INSERT, or UPDATE. (It is enough to update any column in the record to create the lock.)

```
DECLARE
 v_manual_cl   CLOB;
 v_add_tx      VARCHAR2(2000) :=
  'Loaded: '||TO_CHAR(SYSDATE,'mm/dd/yyyy');
BEGIN
 SELECT manual_cl
 INTO v_manual_cl
 FROM goods_tab
 WHERE item_id = 1;

 UPDATE goods_tab
 SET name_tx = '<'||name_tx||'>'
 WHERE item_id = 1;

 DBMS_LOB.WRITEAPPEND (v_manual_cl,
   LENGTH (v_add_tx), v_add_tx);
END;
```

   - If your locator did not contain the transaction ID, but was used to update the LOB, now it will contain the transaction ID (as in the previous example). But if your locator already contained the transaction ID, nothing will change.
4. Consecutive write using the locator:
   - If your locator contains a transaction ID that differs from the current one, the update will always fail because locators cannot span transactions as shown here:

```
DECLARE
 v_manual_cl   CLOB;
 v_add_tx      VARCHAR2(2000):=
  'Loaded: '||TO_CHAR(SYSDATE,'mm/dd/yyyy');
BEGIN
 SELECT manual_cl
 INTO v_manual_cl
 FROM goods_tab
 WHERE item_id = 1;
```

```
      UPDATE goods_tab
      SET name_tx = name_tx||'>'
      WHERE item_id = 1;

      DBMS_LOB.WRITEAPPEND
       (v_manual_cl,LENGTH(v_add_tx),v_add_tx);--OK
      ROLLBACK; -- end of transaction
      DBMS_LOB.WRITEAPPEND
       (v_manual_cl,LENGTH(v_add_tx),v_add_tx);--FAIL!
    END;
```

This information can be simplified into three rules:

1. You can perform read operations using locators as much as you want unless your TRANSACTION LEVEL is set to SERIALIZABLE.
2. If you want to write using a locator, you need to have a lock on the record.
3. If you want to write using the same locator multiple times, it must be done within the same transaction.

### 11g Additions (SecureFiles only)

In terms of pure PL/SQL, SecureFile storage implementation included a number of interesting features. The first one continues the recent thread of providing extra granularity for the developer. Now you can set advanced parameters not only per column, but also per instance (LOB object in every row). The following example illustrates the rule that the compression should be disabled for all LOBs below 100,000 KB of size:

```
PROCEDURE p_loadLob(i_id NUMBER, i_cl CLOB)IS
  v_cl CLOB;
BEGIN
  INSERT INTO goods_tab (item_id, manual_cl)
  VALUES (i_id, i_cl)
  RETURNING manual_cl into v_cl;

  IF LENGTH(v_cl)<100000 THEN
    DBMS_LOB.SETOPTIONS
      (v_cl,DBMS_LOB.OPT_COMPRESS,0);
  END IF;
END;
```

The other set of new tools is the result of advanced automatic control of data chunks written to the tablespace. Since Oracle manages chunks on the fly, it seems logical, that it should be possible to directly access and modify any data up to the highest valid size (32K). Indeed, the DBMS_LOB package now includes the following functions – FRAGMENT_INSERT, FRAGMENT_DELETE, FRAGMENT_MOVE, FRAGMENT_REPLACE with the logic as described here:

```
DECLARE
    v_cl CLOB;
BEGIN
    SELECT manual_cl
    INTO v_cl
    FROM goods_tab
    WHERE item_id = pkg_global.gv_current_id
    FOR UPDATE;
    DBMS_OUTPUT.PUT_LINE(v_cl);

    DBMS_LOB.FRAGMENT_INSERT(v_cl, 5, 4, '[111]');
    DBMS_OUTPUT.PUT_LINE (v_cl);

    DBMS_LOB_FRAGMENT_MOVE(v_cl, 5, 4, 2);
    DBMS_OUTPUT.PUT_LINE (v_cl);

    DBMS_LOB.FRAGMENT_REPLACE(v_cl,5,6,2,'[9999]');
    DBMS_OUTPUT.PUT_LINE (v_cl);

    DBMS_LOB.FRAGMENT_DELETE(v_cl,6,2);
    DBMS_OUTPUT.PUT_LINE (v_cl);
END;
```

***NOTE:*** As of 11.2.0.2 all FRAGMENT_* operations are not available for SecureFile LOBs with de-duplication turned on.

## REAL-WORLD EXAMPLES

There are dozens of "war-stories," but we decided to select one developer-oriented and one DBA-oriented case. One thing both cases had in common was the fact that the resulting impact was very significant.

### HTML on the Fly

Since most modern front-end development environments now support CLOBs, this datatype is very useful as a way of communicating large amounts of read-only information to the client via manually generated HTML page. Here is an outline for such a module:

```
FUNCTION f_getEmp_CL (...) RETURN CLOB IS
  v_out_cl CLOB;
  v_break_tx VARCHAR2(4):='<BR>';
  v_hasErrors_yn VARCHAR2(1):='N';
  v_buffer_tx VARCHAR2(32767);

  PROCEDURE p_flush IS
  BEGIN
    DBMS_LOB.WRITEAPPEND(v_out_cl,
      LENGTH(v_buffer_tx), v_buffer_tx);
      v_buffer_tx:=NULL;
  END;

  PROCEDURE p_addToClob (in_tx VARCHAR2) IS
  BEGIN
    IF LENGTH(in_tx)+length(v_buffer_tx)>32767 THEN
      p_flush;
    END IF;
    v_buffer_tx:= v_buffer_tx||in_tx;
  END;
BEGIN
  DBMS_LOB.CREATETEMPORARY
   (v_out_cl,true,DBMS_LOB.CALL);

  p_addToClob('--Employee review--'||v_break_tx);
  FOR rec_emp IN (SELECT * FROM emp) LOOP
    IF emp_rec.bonus IS NULL THEN
        p_addToClob(' * '||rec_emp.ename||
         ' has no bonuses!'||v_break_tx);
        v_hasErrors_yn:='Y';
      END IF;
    END LOOP;
    ...
    IF v_hasErrors_yn='Y' THEN
      p_addToClob(v_break_tx||
         '<font color="red">***Errors!***</font>');
    END IF;

    p_flush; -- write leftovers
    RETURN v_out_cl;
  EXCEPTION
  WHEN OTHERS THEN
    RETURN '<font color="red">***Errors!***</font>'
           ||v_break_tx||SQLERRM;
  END;
```

The code is very straightforward. First, you create a temporary CLOB. Because the resulting size is not very large, you can make it cached (second Boolean parameter set to TRUE). The third parameter is set to DBMS_LOB.CALL (another option is DBMS_LOB.SESSION or DBMS_LOB.TRANSACTION in 11gR2). Since you are not planning to reuse the LOB, it makes sense to mark it ready to be released immediately after the function finishes its execution. Now you have initialized the CLOB so you can start writing error messages, remarks, headers, etc. This example only includes two HTML tags <BR> and <FONT>, but the idea is clear.

What may not be clear is why we use a PL/SQL variable V_BUFFER_TX instead of directly writing to the CLOB. The answer is very simple – performance considerations! As mentioned previously, temporary CLOBs do not reside in memory; they have allocated space in the temporary tablespace. That is why any DBMS_LOB.WRITEAPPEND call causes a physical I/O operation, which is expensive. By introducing a buffer, the total number of such calls could be decreased up to 100 times writing an average of 32 bytes per request.

**CLOBs and XML**

One of the most critical lessons learned over years of working with Oracle's XMLTYPE is that internally (unless you specify otherwise – you could use object-relational of binary XML) it contains a CLOB column, created with the default settings. As already described, the default setting may or may not be the best option, depending upon your implementation. For example, if the column is very actively accessed, having NOCACHE is a really bad idea. This is exactly what happened to one of our production environments. At one point, the system started to register an abnormally high number of wait events related to Direct I/O operations, and the suspect was obvious. What was not obvious was how to correctly adjust the parameters, because we needed to adjust an internal part of Oracle's own datatype. The good news is that data dictionary views are rich enough to solve this problem. The bad news is that some guessing is required. The following example illustrates the whole chain of events:

```
    CREATE TABLE goods_xml(id NUMBER, a_xml XMLTYPE)
    -- find a column, internally created for XMLTYPE
    SELECT column_name -- will look like SYS_***
    FROM user_lobs
    WHERE table_name = 'GOODS_XML'
    -- Alter the column
    ALTER TABLE goods_xml MODIFY LOB(<column>)
(CACHE);
```

The reason why guessing is required is that if you have multiple XMLTYPE columns in the same table there is no clean way of identifying what CLOB segment belongs to what column. It looks like the order of these SYS_*** names matches the order of columns in the table, but there is no guarantee. Our suggestion would be to add all XMLTYPE columns one by one using the following statement and explicitly name the storage segment (to solve the identification problem):

```
    -- Explicitly specify CLOB storage parameters
    ALTER TABLE goods_xml ADD review_xml XMLTYPE
    XMLTYPE COLUMN review_xml
    STORE AS SecureFile CLOB review_seg(
        ENABLE STORAGE IN ROW
        CACHE)
```

## SUMMARY

Large objects can be very useful in current system development environments because most information can now be stored in the database. But, as with any advanced feature, you need a thorough understanding of the core mechanisms, associated ideas, and principles; otherwise you can do more harm than good to your system. Don't ever try to use new features in production systems before they have gone through a full testing cycle. Also, don't believe everything you read without testing it for yourself (not even this article)!

Oracle Development Tools Member

## About The Authors

Michael Rosenblum is a software architect/development DBA at Dulcian, Inc. He supports Dulcian developers by writing complex PL/SQL routines and researching new features. He is the co-author of *PL/SQL for Dummies*, contributing author of Expert PL/SQL Practices, and author of a number of database-related articles. Michael is an Oracle ACE, a frequent presenter at various regional and national Oracle user group conferences, and winner of the ODTUG Kaleidoscope 2009 Best Speaker Award.

Grigoriy Novikov is a senior developer at Dulcian, Inc. He specializes in the development of information systems for federal and state government organizations as well as private industry clients. He is also responsible for database design, data modeling, researching new features and tuning.