# Large Issues with Large Objects

*Michael Rosenblum, Dulcian, Inc.*

Novice database professionals may assume that three datatypes (DATE, NUMBER, VARCHAR2) are enough to build most systems. However, this is almost never the case. Pictures, movies, documents, and sounds may all need to be stored in newly developed systems. VARCHAR2, the basic Oracle character datatype can only hold 4,000 characters (about one page of text).

As an example, you might be creating an online shopping catalog of electronic goods where each record contained the name of the item, user manual text and front page image, and a link to the original text file with the manual stored on the server.

You can use Oracle's class of datatypes designed to work with large objects called LOBs (up to 8-128TB depending on configuration and environment) of binary/textual information. These LOBs can be divided into two groups based on the way in which the data is stored:

1. *Internal large objects* are stored within the database itself. These objects can be quickly retrieved. You do not have to worry about managing individual files. Using this approach, the database will get very large. Without a good backup utility, it can take hours (or even days) to fully backup the database. There are three datatypes of internal LOBs:
    - 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).
2. *External large objects* are stored in the file system and only the filenames are stored in the database. There are risks associated with storing large objects in the file system. There may be performance issues with some operating systems when thousands of files are placed in a single directory. In addition, you have to worry about people changing the contents of the objects outside your database-based programs. The database is restricted to read-only access to these objects. BFILE is used to point to files stored in the operating system and provide *read-only* access to these files (if you need to write to operating system files – you should either use UTL_FILE package or custom-built JAVA procedures).

## Underlying Concepts to Understand

There are some issues specific to large objects that you need to understand before viewing the actual code syntax associated with them.

### 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 data itself from the mechanism of accessing that data. This results in two separate entities: LOB data and a LOB *locator* that points to LOB data and allows communication with the data.

To understand this data structure, imagine a huge set of cells with water and a tube that can take water from the cell, modify it and put it back. If you want to pass a cell (LOB) to a different place (sub-routine), you don't need to extract and pass the whole amount of water (LOB data), you just need to pass the tube pointing to the area you want to change (locator).

Using locators, there are two types of LOB operations:

- *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.
- *Reference semantics* are used when only the locator is copied without any change to the underlying data.

### Data states

Each attribute of a column can be initialized or not. An attribute may have either a NULL or NOT NULL value. Because of the existence of locators, LOBs have not two, but 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 is not pointing to any data.
- Populated - The variable or column in the row exists, has a locator, and contains real data.

The Empty state is very important. Since 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 an extra step.

## Data storage

LOBs can be tricky from the DBAs point of view. If external LOBs (BFILE) are nothing more than pointers to files stored in the operating system, internal LOBs leave a lot of space for configuration. There are two different kinds of internal LOBs:

- *Persistent LOBs* are represented as a value in the column of the table. As a result, they participate in the transaction (changes could be committed/rolled back) and generate logs (if configured to do so).
- *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.

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 anymore. But with persistent LOBs, each LOB attribute has its own storage structure separate from the table in which it is located.

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 could significantly increase the number of I/O operations.

To navigate chunks, Oracle uses a special *LOB index*. Each LOB column is physically represented by two 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.

Each operation with an LOB chunk requires physical I/O. As a result, you may end up with a high number of wait events in the system. From the other side, placing the whole LOB in the buffer cache could be very expensive. That why Oracle allows you to define the *caching option* in a number of ways:

- NOCACHE is the default option. It should be used only if you are rarely accessing the LOBs or the LOBs are extremely large.
- 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.

One relevant performance question to ask is: Why place data in the special storage structure if in some rows you may only have a small amount of data? Using the online ordering system example, you might have remarks about some goods that only require between 1KB and 1MB of space. Oracle allows you to *store data in the row* (instead of outside of the row) if you have less then 3964 bytes. In that case, all small remarks will 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 even consider disabling this feature since in almost all cases, it is your best option.

## STANDARD USE OF LOBS

Using the example of an online shopping catalog for electronic goods, you can use LOBs to create a table as shown here:

```
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 remarks_seg(
  tablespace USERS
  enable storage in row
  chunk 8192
  cache)
LOB(manual_cl) store as manual_seg(
  tablespace LOBS_BIG
  disable storage in row
  chunk 32768
  nocache)
LOB(firstpage_bl) store as firstpage_seg(
  tablespace LOBS_BIG
  disable storage in row
  chunk 32768
  cache reads)
```

This example includes all three datatypes: CLOB, BLOB and BFILE. 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:

- Each internal LOB has explicit segment names (remarks_seq, manual_seq, firstpage_seq) instead of system-generated ones. This is done for the convenience of working with the `user_segments` dictionary view.
- 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()`.
- The column `remarks_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 so that if you have joins to other tables Oracle needs to do less work. This enables storage within the row for small amounts of data. The cache option should also be enabled for performance optimizations. Since a lot of people will be working with that column, you definitely don't want to generate extra wait events because of direct read/direct write operations.
- The column `manual_cl` is accessed extremely rarely. That's why the independent tablespace, large chunk size, no storage in the row, and no caching options are appropriate here.
- 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. Everything else follows the same logic.

At this point, you have configured the physical storage configured, but somewhere on the server are a number of manuals. It is necessary to get the information in the manuals to the database. If you have had some experience with the UTL_FILE, you know that starting with version 9*i* the only way that PL/SQL can access operating system files is via directories. You can build a directory using the following code:

```
create directory IO as 'C:\IO';
grant read, write on directory IO to public;
```

## LOAD BFILE

Now you can move the required files into that directory and start loading them using this code:

```
declare
  v_bf BFILE := BFILENAME ('IO', 'book.txt');
begin
  insert into  goods_tab
   (item_id, name_tx, mastertxt_bf)
  values (object_seq.nextval, 'The New Book', v_bf)
  returning item_id into pkg_global.gv_current_id;
end;
```

In order to show that you can use the same datatypes in PL/SQL, you can create a variable of type BLOB using the special built-in function `BFILENAME` that takes the directory and file name and returns a temporary locator. After that the code inserted a newly created locator to the table and made it permanent. In the current version of Oracle, it does not matter whether or not the specified file really exists; it is your responsibility to check before using it.

## Load CLOB and BLOB from files

Assuming that there is a file with the book in the folder IO, there is an easy way of loading its contents into the corresponding CLOB using the special PL/SQL APIs provided by built-in package `DBMS_LOB` shown here:

```
declare
  v_file_bf     BFILE;
  v_manual_cl   CLOB;
  lang_ctx      NUMBER := DBMS_LOB.default_lang_ctx;
  charset_id    NUMBER := 0;
  src_offset    NUMBER := 1;
  dst_offset    NUMBER := 1;
  warning       NUMBER;
begin
  select mastertxt_bf, manual_cl
  into v_file_bf, v_manual_cl
  from goods_tab
  where item_id = pkg_global.v_current_id
  for update of manual_cl;

  DBMS_LOB.fileopen (v_file_bf, DBMS_LOB.file_readonly);
  DBMS_LOB.loadclobfromfile (v_manual_cl,
             v_file_bf,
           DBMS_LOB.getlength (v_file_bf),
           src_offset, dst_offset,
           charset_id, lang_ctx,
           warning);
 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 locators back to the LOBs. But these special locators contain the ID of the current transaction (more about transaction issues a bit later). This means that you can not only read data from the LOB, but also write to the LOB. Using the cell and tube analogy, you have your own tube and your own cell to do whatever you want. The way to read data is very straightforward: open the file via locator, read the data, close the file. Because of possible language issues, the reading of textual information to the CLOB is a bit tricky. That is why you have the extra options of specifying the language and character set. 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.

The process of reading the image of the first page is even simpler since it is binary information you don't need to worry about languages as shown here:

```
declare
  v_file_bf  BFILE:= BFILENAME ('IO','picture.gif');
  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 = pkg_global.v_current_id
  for update of firstpage_bl;

  DBMS_LOB.fileopen (v_file_bf, DBMS_LOB.file_readonly);
  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;
```

In this case, instead of using the existing locator to the external file, you can create a temporary one on the fly and process it using the same locking logic and DBMS_LOB package.

## Simple Operations

From the developer's point of view, BLOBs are significantly less interesting. They are just used to store any binary information required for the application (pictures, streams, videos, etc.). It is just unstructured data, and there is nothing much you can do with it. But CLOBs are used mostly to store textual data, which is semi-structured by definition (character string is already a structure). Therefore there is a lot of extra activity that may occur when accessing CLOBs. In recent versions, Oracle tries to make life a bit easier with many standard string built-in functions (search for the patterns, get length, get part of the code, and so on) that support CLOBs. There are some caveats which will be covered later. You can implement a search or indexing routine for all large text files loaded in the database exactly the same way as you would for regular strings as shown here:

```
declare
  v_manual_cl   CLOB;
  v_nr          NUMBER;
  v_tx          VARCHAR2 (2000);
  v_add_tx      VARCHAR2 (2000)
   :='Loaded: '||TO_CHAR(SYSDATE,'mm/dd/yyyy hh24:mi');
begin
  select manual_cl
   into v_manual_cl
   from goods_tab
  where item_id = pkg_global.v_current_id
  for update;

  DBMS_LOB.writeappend (v_manual_cl, LENGTH (v_add_tx), v_add_tx);

  v_nr := INSTR (v_manual_cl, 'Loaded:', -1);
  v_tx := SUBSTR (v_manual_cl, v_nr);
  DBMS_OUTPUT.put_line (v_tx);
END;
```

## Special cases and special problems

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

### Generic restrictions

As of Oracle 10*g* release 2, there are couple of restriction sets that you need to be aware of.

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). In case you need to write more than 32K of information, you must use DBMS_LOB package.

```
declare
  v_tx varchar2(6000):=lpad('*',6000,'*');
  v_count_nr number;
begin
  update goods_tab
  set remarks_cl =lpad('*',6000,'*')
  where item_id = pkg_global.v_current_id
  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 = pkg_global.v_current_id
  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 or Oracle Text)
    c. You cannot specify a 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
3. DBLink restrictions:
    a. You can only use CREATE TABLE AS SELECT and INSERT AS SELECT if 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).

Even though SQL semantics are much easier to work with, there are some reasons for when not to use them:

- If you are working with more than 100K of data for each CLOB, APIs handle caching significantly better.
- If there is a great deal of random access to the data in the CLOB, APIs utilize LOB indexes much more efficiently.

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 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 = pkg_global.v_current_id
  --and remarks_cl!=manual_cl -- INVALID
  ;

  if v_remarks_cl!=v_manual_cl -- VALID
  then
    dbms_output.put_line('Compared');
  end if;
end;
```

Furthermore, using SQL semantics could get you into a lot of troubles with some built-in functions. INITCAP, SOUNDEX, TRANSLATE, DECODE and some other functions will process only the first 4K (for SQL statements) and 32K (for PL/SQL code) of your data. There will be no notifications, and no errors, just part of the data will be removed. This "feature" is not well known, but easily cause problems. In the following example even when creating a CLOB with a length of 6K, after the second update, it only contains 4K:

```
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 = pkg_global.v_current_id
  returning length (remarks_cl) into v_count_nr;
  dbms_output.put_line('Length:'||v_count_nr);

  update goods_tab
  set remarks_cl = translate (remarks_cl,'a','A')
  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), but 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 moment you are trying to perform the update (not necessarily at the moment 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.)

```
  v_manual_cl   CLOB;
  v_add_tx      VARCHAR2 (2000)
   :='Loaded: '||TO_CHAR(SYSDATE,'mm/dd/yyyy hh24:mi');
begin
  select manual_cl
   into v_manual_cl
   from goods_tab
  where item_id = pkg_global.v_current_id;

  update goods_tab
  set name_tx = '<'||name_tx||'>'
  where item_id = pkg_global.v_current_id;

  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 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 for it.
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 hh24:mi');
begin
  select manual_cl
   into v_manual_cl
   from goods_tab
  where item_id = pkg_global.v_current_id;

  update goods_tab
  set name_tx = name_tx||'>'
```

```
    where item_id = pkg_global.v_current_id;

    DBMS_LOB.writeappend (v_manual_cl, LENGTH (v_add_tx), v_add_tx);

    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.
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, you have to do it in the same transaction.

## Transaction control and some advanced features

Rule #3 is critical nowadays because of the wide use of two very powerful Oracle features: autonomous transactions and dynamic SQL.

One big problem when coding dynamic SQL is that programmers often forget that all DDL commands fire implicit COMMITs. As a result, even if your code does not appear to be initializing a new transaction, it actually is. The following code will have the same problem of the locator spanning multiple transactions as if you used a COMMIT or ROLLBACK:

```
declare
  v_manual_cl   CLOB;
  v_add_tx      VARCHAR2 (2000)
   :='Loaded: '||TO_CHAR(SYSDATE,'mm/dd/yyyy hh24:mi');
  v_string_tx varchar2(2000);
begin
  select manual_cl
   into v_manual_cl
   from goods_tab
  where item_id = pkg_global.v_current_id;

  update goods_tab
  set name_tx = name_tx||'>'
  where item_id = pkg_global.v_current_id;

  DBMS_LOB.writeappend (v_manual_cl, LENGTH (v_add_tx), v_add_tx);

  v_string_tx:='create table goods_tab_'||to_char(sysdate,'YYYYMMDD')||
       ' as select * from goods_tab';
  execute immediate v_string_tx; -- DDL causes new transaction to start

  DBMS_LOB.writeappend (v_manual_cl, LENGTH (v_add_tx), v_add_tx); -- FAILS!
end;
```

Autonomous transactions have more direct problems. If you locked the record in the parent transaction, you cannot lock it in the child one (otherwise deadlock will occur), so you cannot do the first update as shown here:

```
declare
  v_manual_cl   CLOB;
  v_add_tx      VARCHAR2 (2000)
   :='Loaded: '||TO_CHAR(SYSDATE,'mm/dd/yyyy hh24:mi');
  v_string_tx varchar2(2000);

  procedure p_update (pi_cl IN OUT CLOB, pi_tx varchar2) is
```

```
    pragma autonomous_transaction;
  begin
    DBMS_LOB.writeappend (pi_cl, LENGTH (pi_tx), pi_tx); -- NO LOCK!
    commit;
  end;
begin
  select manual_cl
   into v_manual_cl
   from goods_tab
  where item_id = pkg_global.v_current_id;

  update goods_tab
  set name_tx = name_tx||'>'
  where item_id = pkg_global.v_current_id; -- locked the record

  p_update (v_manual_cl, v_add_tx);
end;
```

If you locked the record in the parent transaction and already did the first update (so the locator contains transaction ID), the update in the subroutine will fail because its transaction ID is different from the one passed with the locator as shown here:

```
declare
  v_manual_cl   CLOB;
  v_add_tx      VARCHAR2 (2000)
   :='Loaded: '||TO_CHAR(SYSDATE,'mm/dd/yyyy hh24:mi');
  v_string_tx varchar2(2000);

  procedure p_update (pi_cl IN OUT CLOB, pi_tx varchar2) is
    pragma autonomous_transaction;
  begin
    DBMS_LOB.writeappend (pi_cl, LENGTH (pi_tx), pi_tx); -- wrong trans ID!
    commit;
  end;
begin
  select manual_cl
   into v_manual_cl
   from goods_tab
  where item_id = pkg_global.v_current_id;

  update goods_tab
  set name_tx = name_tx||'>'
  where item_id = pkg_global.v_current_id; -- locked

  DBMS_LOB.writeappend (v_manual_cl, LENGTH (v_add_tx), v_add_tx);-- set trans ID

  p_update (v_manual_cl, v_add_tx);
end;
```

If you locked the record in the child transaction, you have to either commit or rollback the changes to finish it. As a result, the locator will contain the wrong transaction ID as shown here:

```
declare
  v_manual_cl   CLOB;
  v_add_tx      VARCHAR2 (2000)
   :='Loaded: '||TO_CHAR(SYSDATE,'mm/dd/yyyy hh24:mi');
  v_string_tx varchar2(2000);
```

```
  procedure p_update (pi_cl IN OUT CLOB, pi_tx varchar2) is
    pragma autonomous_transaction;
  begin
    update goods_tab
    set name_tx = name_tx||'>'
    where item_id = pkg_global.v_current_id; -- locked

    DBMS_LOB.writeappend (pi_cl, LENGTH (pi_tx), pi_tx); -- set transaction ID
    commit;
  end;
begin
  select manual_cl
   into v_manual_cl
   from goods_tab
  where item_id = pkg_global.v_current_id;

  p_update (v_manual_cl, v_add_tx);

  DBMS_LOB.writeappend (v_manual_cl, LENGTH (v_add_tx), v_add_tx); -- wrong ID!
end;
```

## Case Study

The following information is based on an actual project that the author worked on, in which handling large objects was particularly relevant.

### HTML on the fly

Since most modern development environments now support CLOBs, they are a very useful as a way of communicating large amounts of read-only information to the client. Assume that you have a large organizational structure that could be versioned; however, before doing the versioning, you must validate the new structural model before rolling it over the old one. There may be no errors, a few errors, or many errors. The solution options are as follows:

1. Populate a temporary table where one row represents one error.
    a. **Advantages of this approach:** No clean up is needed.
    b. **Disadvantages of this approach:** It is a two-step process (populate and query) requiring the same session, which may not be possible in the web environment. Formatting the result must be hard-coded on the client side, so there is no way to change it.
2. Populate (and commit) a permanent table and clean up after the user confirms that he/she saw the report.
    a. **Advantages of this approach:** It resolves the problem of session-dependency.
    b. **Disadvantages of this approach:** What if the user's connection was broken? You could have a lot of records that will not be cleaned. Formatting of the result has to be hard-coded on the client side, so there is no way to change it.
3. Create a function that returns an object collection.
    a. **Advantages of this approach:** It is a one-step process and no clean up is required.
    b. **Disadvantages of this approach:** Formatting the result has to be hard-coded on the client side, so there is no way to change it.

In using HTML on the fly, the issues to deal with include session-dependency, clean up and formatting. CLOBs allow you to solve all of them at once because:

- A function takes a parameter and returns a CLOB in one round-trip.
- Temporary CLOBs are released automatically.

- Full formatting can be done in the CLOB itself. Since most current development environments understand HTML, it is the best choice.

An example of a routine to handle these issues is as follows:

```
function f_validateOrgStruct_CL (...) return CLOB
is
  v_out_cl CLOB;
  v_break_tx varchar2(4):='<BR>';
  v_hasErrors_yn varchar2(1):='N';
  ...
  <<number of cursors>>
  ...
  procedure p_addToClob (in_tx varchar2) is
  begin
    dbms_lob.writeappend(v_out_cl,length(in_tx),in_tx);
  end;
begin
  dbms_lob.createtemporary(v_out_cl,true,dbms_lob.Call);

  p_addToClob('------------------VALIDATE RECRUITERS-------------'||v_break_tx);
  for rec_recrtr in c_recrtr loop
    if rec_recrtr.ric_tx is null  then
      p_addToClob('  * '||rec_recrtr.rc_recrtr_dsp||
                         ' - missing code'||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 ARE DETECTED! ***</font>');
  end if;

  return v_out_cl;
exception
when others then
  return '<font color="red">*** FATAL ERRORS! ***</font>'||v_break_tx||sqlerrm;
end;
```

The code is very straightforward. First, you create a temporary CLOB. Considering that 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). 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. If, at some point, you need to change the formatting you can simply alter the function without touching either the middle-tier or the client code.

## XML-BASED FORMS

The next problem is similar to the previous one. The data is stored in the relational database, but it needs to be passed to an environment similar to XML (not exactly XML, which is why you cannot use standard Oracle features). As a result, a special mapping routine is built to have a CLOB with XML as an output. The advantages are the same: free formatting, easy clean up, one roundtrip. In general, any time you need to pass structured data, an XML-based format can be very useful. Oracle also uses CLOB as the storage mechanism for its XMLType datatype.

## Emails from the database

Another area where large objects can be very useful is if you need to send email directly from the database using the UTL_SMTP package. Sending attachments is particularly challenging. Due to space limitations, this code is based on the well-known Oracle package DEMO_MAIL (found in manuals or on OTN):

```
procedure p_attach_file (conn in out nocopy utl_smtp.connection,
  i_filename_tx varchar2,
  i_directory_tx varchar2)
is
  v_file_bl     BLOB;
  v_file_bf     BFILE:=BFILENAME (i_filename_tx, i_directory_tx);
  src_offset    pls_integer := 1;
  dst_offset    pls_integer := 1;
  v_length_nr   pls_integer;

  v_mod_nr     pls_integer;
  v_pieces_nr  pls_integer;
  v_counter_nr pls_integer         := 1;

  v_amt_nr     binary_integer      := 54; -- amount of bytes to be read at once
  v_filepos_nr pls_integer         := 1;

  v_buf_raw    raw (100); -- buffer to communicate with UTL_SMTP

begin
  dbms_lob.createtemporary (v_file_bl, true, dbms_lob.call );
  dbms_lob.fileopen (v_file_bf, dbms_lob.file_readonly);
  v_length_nr:=dbms_lob.getlength(v_file_bl);
  dbms_lob.loadblobfromfile(v_file_bl, v_file_bf, v_length_nr, dst_offset, src_offset);
  dbms_lob.fileclose (v_file_bf);

  demo_mail.begin_attachment (conn,'application/octet-stream', true,
              i_filename_tx, 'base64');

  v_mod_nr    := mod (v_length_nr, v_amt_nr);
  v_pieces_nr := trunc (v_length_nr / v_amt_nr);

  while (v_counter_nr <= v_pieces_nr) loop
    dbms_lob.read (v_file_bl, v_amt_nr, v_filepos_nr, v_buf_raw);
    demo_mail.write_raw (conn, utl_encode.base64_encode (v_buf_raw));
    v_filepos_nr := v_counter_nr * v_amt_nr + 1;
    v_counter_nr := v_counter_nr + 1;
  end loop;

  if (v_mod_nr <> 0) then
    -- read a piece of original file
    dbms_lob.read (v_file_bl, v_mod_nr, v_filepos_nr, v_buf_raw);
    -- the best way to send binary data is via BASE64 encoding
    demo_mail.write_raw (conn, utl_encode.base64_encode (v_buf_raw));
  end if;

  demo_mail.end_attachment (conn);
end;
```

Assuming that you have already have established a connection to an SMTP server, the logic of the current routine is as follows:

1. Load the file you are planning to attach to the temporary LOB. This step is needed for performance reasons. Of course, you can read data directly using BFILE, but in that case, each operation will cause a direct read (and a large number of wait events with any significant number of users). Even loading a lot of data to the temporary tablespace is not the best idea. However, since you never send more than a few megabytes, the database can easily handle it.
2. The second step is to attach the data to the email. By SMTP protocol standards, you cannot just pour binary data into the body of the email. The data should be encoded into the special BASE64 format (textual representation of binary data) and you should only send a limited number of bytes at a time. This is the reason why you need to use a buffer. The amount of data retrieved at one step could be tuned depending upon your environment (up to 2000 bytes at once). Common knowledge recommends not setting this value higher than 100.

## Summary

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

## About the Author

Michael Rosenblum is a Development DBA at Dulcian, Inc. He is responsible for system tuning and application architecture. He supports Dulcian developers by writing complex PL/SQL routines and researching new features. Mr. Rosenblum is the co-author of PL/SQL for Dummies (Wiley Press, 2006). Michael is a frequent presenter at various regional and national Oracle user group conferences. In his native Ukraine, he received the scholarship of the President of Ukraine, a Masters Degree in Information Systems, and a Diploma with Honors from the Kiev National University of Economics.