DYNAMIC SQL IN THE 11G WORLD

Michael Rosenblum, Dulcian, Inc.

It is always difficult to write about a well-known database feature for the following reasons:

1. Almost everyone in the industry already has an opinion about it.
2. No one expects to learn anything new.

Unfortunately this kind of tunnel vision causes many long-term problems in the IT industry because even senior people tend to fall into the "been-there-done-that" mentality. To fight off this attitude, I try to review my old paper and presentation topics to make sure that I am aware of the most recent changes in the landscape. This is especially important considering the fact that *finally*, the majority of database environments are now reliably using Oracle 11gR2, so there is no need for cross-version coverage.

For years, Dynamic SQL was one of my favorite topics that I wrote and spoke about in two presentations at ODTUG-2007/2008, and in entire chapters in *PL/SQL For Dummies* (Wiley, 2006) and *PL/SQL Best Practices* (APress, 2011). However, all of that information was focused on the issue of making rank-and-file developers familiar with the concept of building code on the fly. Unfortunately, the results of my post-factum analysis led me to the following conclusions:

- Too many IT shops are uncomfortable with Dynamic SQL mostly because of organization culture (not technical reasons)
- There are way too many myths and misunderstandings about Dynamic SQL, especially at the management/senior level.
- It is really hard to fight myths against this reasoning.

Overall, there are two opposing groups: proponents and opponents of Dynamic SQL. These groups do not cross at all, so my goal as an educator (at least for now) is to shift from proselytizing Dynamic SQL to improving the effectiveness of its usage (specifically in the Oracle 11g environment). For that reason, this paper will not include any introductory topics. These have already been discussed too often. Also, it will not contain anything already available in earlier versions for exactly the same reasons. However, there are still many good topics to be discussed:

1. Support of CLOB in Native Dynamic SQL
2. Cursor transformation
3. Lifting of datatype restrictions for DBMS_SQL
4. DBMS_ASSERT built-in package
5. Integration of Dynamic SQL and RESULT_CACHE

CLOB as Input

Finally, Oracle has lifted the most irritating restriction of Native Dynamic SQL. It is now possible pass CLOB into EXECUTE IMMEDIATE and OPEN CURSOR FOR, while previously we were restricted by VARCHAR2(32767). An example is shown here:

```
Declare
 V_cl CLOB:='begin null; end;';
Begin
 EXECUTE IMMEDIATE V_CL;
End;

Declare
 V_cl CLOB:='select * from emp';
 V_cur SYS_REFCURSOR;
Begin
 Open v_cur for v_cl;
End;
```

This change makes a lot of difference for anyone using code generators. In many real business cases, about 99.9% of generated code does not exceed the 32K limit. But that extra 0.1% impacted the entire architecture and forced the usage of DBMS_SQL with pretty obscure syntax and extra complexity.

Now the developer's life is significantly simpler in this area, but with a caveat: You need to be comfortable working with CLOB datatypes (see my Kaleidoscope11 presentation, *Managing Unstructured Data: LOBS, SecureFiles, BasicFiles*). The following example illustrates the core ideas:

- CLOB variables use temporary tablespaces, not memory. As a result, they should be properly configured to release all required resources back to the system.
- CLOB operations always imply physical I/O. Therefore, you should access LOB variables as little as possible.
- The best way of operating with CLOBs is by using DBMS_LOB package (please, no concatenation!)

```
function f_BuildQuery_CL (in_tab_tx varchar2) 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('create or replace view V_'||in_tab_tx||' as select ROWNUM num_rows');

  for rec_rep in (select * from my_tab_col where table_name = in_tab_tx) loop
      p_addToClob(','||rec_rep.column_tx);
  end loop;
  ...
  p_addToClob(' from '||in_tab_tx);
  p_flush; -- write leftovers
  return v_out_cl;
end;
```

The concepts mentioned above can be implemented as follows:

- V_OUT_CL variable is initialized with the options CACHE=TRUE and DURATION=DBMS_LOB.CALL. This means that we will not be using Direct I/O. Temporary segments allocated to store the data will be released immediately after the call (and not at the end of the current session, which is the default behavior).
- Data will first be aggregated into the buffer variable V_BUFFER_TX and written to CLOB only when the buffer is full (and at the end of the routine to clear leftovers).

In addition, while discussing code generators, there is one old but not well-known feature worth mentioning. The built-in package DBMS_DDL allows the creation of wrapped PL/SQL objects on the fly. Unfortunately, this package does not support CLOB as input (only VARCHAR2 or arrays of VARCHAR2). However, it still may be useful if your solution contains wrapped modules. In this way, you can be somewhat sure that no one has touched your generated modules.

```
declare
    v_wrap_tx varchar2(32767);
    v_ddl_tx  varchar2(32767);
begin
    v_ddl_tx:='create or replace procedure p_emp is begin null; end;';

    v_wrap_tx:=dbms_ddl.wrap(v_ddl_tx);
    execute immediate v_wrap_tx;

                or
```

```
    dbms_ddl.create_wrapped(v_ddl_tx);
end;
```

Cursor Transformation

For years, PL/SQL included two mechanisms pointing to conceptually the same object: REF CURSOR and DBMS_SQL cursor. In both cases, it was a reference to the opened dataset. Also, in both cases it was used to pass information around , but there was no way to convert one to another. Until 11g, where two new functions were introduced to DBMS_SQL package:

- DBMS_SQL.TO_CURSOR_NUMBER – takes an opened REF CURSOR as input and transforms it to a DBMS_SQL numeric cursor.
- DBMS_SQL.TO_REFCURSOR – reverse transformation. This time, the restriction is that the DBMS_SQL cursor must be opened, parsed, and executed. Also, only SELECT statements could be processed this way.

It is very important to recognize that both of those transformations preserve the fetching point. If N rows were already fetched, the next one (even using a different mechanism) will be N+1. That's why the other cursor (FROM-side) is automatically closed at the transformation to create a single point of data access. There is another small and less obvious restriction: you cannot transform to REF CURSOR if the %NOTFOUND flag was already raised, but the opposite transformation is still valid.

The cursor transformation feature is very useful in legacy environments where a lot of REF CURSOR variables are being passed. It allows a simple but very efficient, non-intrusive audit of data requests using something similar to the following module, which takes REF CURSOR, reviews it and passes it back.

```
procedure p_expCursor(io_ref_cur IN OUT SYS_REFCURSOR) is
    v_cur     integer;
    v_cols_nr number := 0;
    v_cols_tt dbms_sql.desc_tab;
begin
  v_cur:=dbms_sql.to_cursor_number(io_ref_cur);
  DBMS_SQL.describe_columns (v_cur, v_cols_nr, v_cols_tt);
  for i in 1 .. v_cols_nr loop
     dbms_output.put_line('*'||v_cols_tt (i).col_name);
  end loop;
  io_ref_cur:=dbms_sql.to_refcursor(v_cur);
end;
```

Support of Object Types in DBMS_SQL

Before Oracle 11g, it was always a matter of architectural choice. Either you utilize the high flexibility of DBMS_SQL and limit yourself to basic datatypes, or you go with Native Dynamic SQL, encounter its functional borders, but use any datatype needed. In 11g Oracle finally supports object types/collections in DBMS_SQL, or (to be precise) mostly supports them, because DBMS_SQL.BIND_ARRAY still cannot utilize user-defined collections. (However, there are numerous workarounds to this problem).

To illustrate why this feature is meaningful, review the following business case:

- There is an external module that builds SQL statements. Those SQL statements take the results of multi-select as input. Also, there is a universal value list builder that returns object collections in the format ID/DISPLAY. The same structure (object collection) is also used for multi-select in the UI.
- Each generated SQL statement should be intercepted before execution and its resulting columns should be extracted (and eventually logged)

```
create type lov_oty as object  (id_nr number, display_tx varchar2(4000));
create type lov_nt is table of lov_oty;

Create function f_getlov_nt (i_table_tx varchar2, i_id_tx varchar2, i_display_tx varchar2, i_order_tx
varchar2)
return lov_nt is
    v_out_nt lov_nt := lov_nt();
begin
    execute immediate
     'select lov_oty('||i_id_tx||','||i_display_tx||') from '||i_table_tx||' order by '||i_order_tx
    bulk collect into v_out_nt;
    return v_out_nt;
end;
```

As you can see from the specifications, we have to use DBMS_SQL to describe the query, but we need to take an object collection as an input. Doing this in 11g is pretty straightforward because DBMS_SQL.BIND_VARIABLE can now use any datatype needed. Below is an example of a monitoring routine using the previously mentioned REF CURSOR transformation and a real printout of both passing a collection and analyzing the output.

```
PROCEDURE p_prepareSQL (i_sql_tx in varchar2, i_lov_nt in lov_nt,
                        o_cur OUT SYS_REFCURSOR, o_structure_tx OUT varchar2)
is
  v_cur INTEGER;
  v_result_nr integer;

  v_cols_nr number := 0;
  v_cols_tt dbms_sql.desc_tab;
begin
  v_cur:=dbms_sql.open_cursor;
  dbms_sql.parse(v_cur, i_sql_tx, dbms_sql.native);

  dbms_sql.describe_columns(v_cur, v_cols_nr, v_cols_tt);

  for i in 1 .. v_cols_nr loop
    o_structure_tx:=o_structure_tx||'|'||v_cols_tt (i).col_name;
  end loop;

  dbms_sql.bind_variable(v_cur, 'NT1',i_lov_nt);
  v_result_nr:=dbms_sql.execute(v_cur);

  o_cur:=dbms_sql.to_refcursor(v_cur);
end;

-----------------------------------------------------------------
SQL> declare
  2      v_ref_cur SYS_REFCURSOR;
  3      v_columnList_tx varchar2(32767);
  4      v_lov_nt lov_nt:=
  5          f_getlov_nt('DEPT','DEPTNO','DNAME','DEPTNO');
  6      v_sql_tx varchar2(32767):=
  7          'select * '||chr(10)||
  8          'from emp '||chr(10)||
  9          'where deptno in ('||chr(10)||
 10          '    select id_nr'||chr(10)||
 11          '    from table(cast (:NT1 as lov_nt))'||chr(10)||
 12          '    )';
 13  begin
 14      p_prepareSQL(v_sql_tx,
 15                   v_lov_nt,
 16                   v_ref_cur,
 17                   v_columnList_tx);
 18      dbms_output.put_line('Columns:'||v_columnList_tx);
 19      if v_ref_cur%isopen then
 20          dbms_output.put_line('Valid Cursor!');
 21      end if;
 22  end;
 23  /
Columns:|EMPNO|ENAME|JOB|MGR|HIREDATE|SAL|COMM|DEPTNO
Valid Cursor!
PL/SQL procedure successfully completed.
SQL>
```

DBMS_ASSERT

This is not exactly an 11g addition because in 10g, this built-in package existed but was not documented or supported. The main reasons to use this package are to assert that the passed string is what it should be as illustrated in the following available modules:

- SQL_OBJECT_NAME (string) – checks whether or not string is a valid object
- SIMPLE _SQL_NAME – checks whether or not string is a valid simple SQL name

- SCHEMA_NAME – validates that passed string is a valid schema
- ENQUOTE_NAME – adds a second quote to every instance in the name (and double quotes around)
- ENQOUTE_LITERAL – adds single quotes

It is clear that the main goal is to fight off code injections. Although, it is my very strong belief that following two rules can accomplish this goal:

1. Data should be passed only via bind variables of matching datatypes.
2. Structural elements should be selectable only from the repository. People populating the repository and people using the system should be explicitly separated by organizational policies.

More about those rules can be found in my Kaleidoscope 2008 paper. For now, assume that you are not allowed to use a repository-based approach, but would like to close the widest security gaps. In this case, DBMS_ASSERT can be a big help. For example, the following function is reasonably protected against injection because simple SQL names cannot contain any malicious coding patterns:

```
function F_GET_col_TX
 (i_table_tx varchar2,i_showcol_tx varchar2,i_pk_tx varchar2,i_pkValue_nr number)
return varchar2 is
  v_out_tx varchar2(4000);
  v_sql_tx varchar2(32000);
Begin
  v_sql_tx:='select to_char('||dbms_assert.simple_sql_name(i_showcol_tx)||
     ') from '||dbms_assert.simple_sql_name(i_table_tx)||
        ' where '||dbms_assert.simple_sql_name(i_pk_tx)||'=:v01';
  EXECUTE IMMEDIATE v_sql_tx INTO v_out_tx
  USING i_pkValue_nr;
  return v_out_tx;
end;
```

There many different direct examples of DBMS_ASSERT widely available, so I will only discuss a few issues that I personally discovered while working with this feature:

1. Objects on the other side of DB-links are never checked.

```
SQL> select DBMS_ASSERT.SQL_OBJECT_NAME('DBMS_ASSERT@DUMMY_DBLINK') check_yn
  2  from dual;
CHECK_YN
----------------------------
DBMS_ASSERT@DUMMY_DBLINK
```

2. Schema names are sometimes but not always case-sensitive

```
SQL> select DBMS_ASSERT.SCHEMA_NAME('Scott') from dual;
select DBMS_ASSERT.SCHEMA_NAME('Scott') from dual
       *
ERROR at line 1:
ORA-44001: invalid schema
ORA-06512: at "SYS.DBMS_ASSERT", line 266

SQL> select DBMS_ASSERT.SQL_OBJECT_NAME('Scott.emp') check_yn
  2  from dual;
Check_yn
--------------------
Scott.emp
```

3. Object names are case-sensitive only if wrapped in double-quotes. By default, Oracle considers all of its object stored as upper case.

```
SQL> select DBMS_ASSERT.SQL_OBJECT_NAME('DBMS_ASSERT') check_yn from dual;
CHECK_YN
------------------------------
DBMS_ASSERT

SQL> select DBMS_ASSERT.SQL_OBJECT_NAME('"DBMS_ASSERT"')check_yn from dual;
```

```
CHECK_YN
------------------------------
"DBMS_ASSERT"

SQL> select DBMS_ASSERT.SQL_OBJECT_NAME('"dbms_assert"') check_yn from dual;
select DBMS_ASSERT.SQL_OBJECT_NAME('"dbms_assert"') check_yn from dual
       *
ERROR at line 1:
ORA-44002: invalid object name
ORA-06512: at "SYS.DBMS_ASSERT", line 316
```

Integration of Dynamic SQL and RESULT_CACHE

Another good thing about the current implementation of Dynamic SQL is that Oracle's PL/SQL team actively integrates it with other advanced features. For example, the "result cache" feature introduced in 11gR1 (and significantly rewritten in 11gR2) is efficient enough to not only auto-detect hard-coded dependencies, but also recognize and record on-the-fly calls made via Dynamic SQL. The following example includes a function that can get current row counts for a given table:

```
create or replace function f_getCount_nr (i_tab_tx varchar2)
return number
result_cache
is
    v_sql_tx varchar2(256);
    v_out_nr number;
begin
    execute immediate
      'select count(*) from '||i_tab_tx into v_out_nr;
    return v_out_nr;
end;
```

Step #1 would be to confirm that (a) result cache actually works and (b) it recognized on-the-fly dependency.

```
SQL> select f_getCount_nr('EMP') from dual;
F_GETCOUNT_NR('EMP')
--------------------
                  14
SQL> select ro.id,  ro.name,  do.object_name
  2  from   v$result_cache_objects    ro,
  3         v$result_cache_dependency rd,
  4         dba_objects               do
  5  where  ro.id = rd.result_id
  6  and    rd.object_no = do.object_id;
ID NAME                                    OBJECT_NAME
-- ------------------------------------------------------------  -------------
1  "SCOTT"."F_GETCOUNT_NR"::8."F_GETCOUNT_NR"#8440831613f0f5d3 #1  EMP
1  "SCOTT"."F_GETCOUNT_NR"::8."F_GETCOUNT_NR"#8440831613f0f5d3 #1  F_GETCOUNT_NR

SQL>select f_getCount_nr('EMP') from dual;
F_GETCOUNT_NR('EMP')
--------------------
                  14
SQL> select *
  2  from v$result_cache_statistics
  3  where  name in ('Create Count Success','Find Count');
ID  NAME                 VALUE
--- -------------------- ------
5   Create Count Success 1
7   Find Count           1
SQL>
```

Oracle successfully recognized the EMP table as cache dependency and was able to return a value from the cache when the function was called a second time. Now to test cache invalidation, I will insert a new row to EMP table and re-fire the function F_GETCOUNT_NR

```
SQL> insert into emp(empno) values (100);
1 row created.
SQL> commit;
```

```
Commit complete.
SQL> select f_getCount_nr('EMP') from dual;
F_GETCOUNT_NR('EMP')
--------------------
                  15
SQL> select id, name, value
  2  from v$result_cache_statistics
  3  where  name in ('Create Count Success',
  4                  'Find Count','Invalidation Count');

ID  NAME                 VALUE
--- -------------------- ------
5   Create Count Success 2
7   Find Count           1
8   Invalidation Count   1
SQL>
```

This time, Oracle successfully detected data changes and invalidated the previously cached information. Now let's introduce a new dynamic dependency (to DEPT table) and see whether the resulting cache would successfully recognize the difference.

```
SQL> select f_getCount_nr('DEPT') from dual;
F_GETCOUNT_NR('DEPT')
---------------------
                    4
SQL> select id,  name,  object_name
  2  from   v$result_cache_objects    ro,
  3         v$result_cache_dependency rd,
  4         dba_objects               do
  5  where  ro.id = rd.result_id
  6  and    rd.object_no = do.object_id;

ID  NAME                                                                OBJECT_NAME
--- ------------------------------------------------------------------- -------------
3   "SCOTT"."F_GETCOUNT_NR"::8."F_GETCOUNT_NR"#8440831613f0f5d3 #1  EMP
3   "SCOTT"."F_GETCOUNT_NR"::8."F_GETCOUNT_NR"#8440831613f0f5d3 #1  F_GETCOUNT_NR
4   "SCOTT"."F_GETCOUNT_NR"::8."F_GETCOUNT_NR"#8440831613f0f5d3 #1  DEPT
4   "SCOTT"."F_GETCOUNT_NR"::8."F_GETCOUNT_NR"#8440831613f0f5d3 #1  F_GETCOUNT_NR

SQL> select id, name, value
  2  from v$result_cache_statistics
  3  where  name in ('Create Count Success',
  4                  'Find Count','Invalidation Count');

ID  NAME                 VALUE
--- -------------------- ------
5   Create Count Success 3
7   Find Count           1
8   Invalidation Count   1
SQL>
```

As you can see, a variation of the resulting cache with the dependency on DEPT (rather than EMP) was immediately recognized. This means that Dynamic SQL is indeed fully integrated into the overall on-the-fly caching mechanism.

Summary

This paper is much closer to a set of observations about Dynamic SQL than true deep-dive research. However, every feature, including Dynamic SQL, constantly evolves. This means that you need to keep your head above water and recognize all of the subtle nuances because, at some point, those details could save real projects. It is my strong belief that only the process of constant learning can keep contemporary IT specialists from losing connections with reality. How often have once great ideas become dangerous a couple of versions later? This is a never ending process. Rumors about Oracle 12c are already well-spread, so, TO BE CONTINUED…

About the Author

Michael Rosenblum is a Software Architect/Development DBA at Dulcian, Inc. where he is responsible for system tuning and application architecture. Michael supports Dulcian developers by writing complex PL/SQL routines and researching new features. He is the co-author of *PL/SQL for Dummies* (Wiley Press, 2006), contributing author of *Expert PL/SQL Practices* (APress, 2011), and author of a number of database-related articles (IOUG Select Journal, ODTUG Tech Journal) and conference papers. Michael is an Oracle ACE, a frequent presenter at various Oracle user group conferences (Oracle OpenWorld, ODTUG, IOUG Collaborate, RMOUG, NYOUG, etc), and winner of the ODTUG Kaleidoscope 2009 Best Speaker Award. In his native Ukraine, he received the scholarship of the president of Ukraine, a Master of Science degree in Information Systems, and a diploma with honors from the Kiev National University of Economics.