Dynamic SQL in PL/SQL

Steven Feuerstein
PL/SQL Evangelist, Quest Software
steven.feuerstein@quest.com
www.ToadWorld.com/SF

How to benefit most from this session

- Watch, listen, focus on concepts and principles.
- Download and use any of my the training materials:

PL/SQL Obsession **http://www.ToadWorld.com/SF**

- Download and use any of my scripts (examples, performance scripts, reusable code) from the same location: the demo.zip file.

filename_from_demo_zip.sql

- You have my permission to use *all* these materials to do internal trainings and build your own applications.
  - But remember: they are not production ready.
  - Modify them to fit your needs and then *test them*!

Agenda

- Overview of dynamic SQL
- Dynamic DDL
- Dynamic DML
- Dynamic Queries
- Dynamic PL/SQL
- Advanced topics
- Best Practices

What is Dynamic SQL?

- Dynamic SQL actually refers, in the world of PL/SQL, to two things:
  - SQL statements, such as a DELETE or DROP TABLE, that are constructed and executed at run-time.
  - Anonymous PL/SQL blocks that are constructed, compiled and executed at run-time.

```
'DROP ' ||
   l_type || ' ' || l_name
```

```
'BEGIN ' ||
   l_proc_name || ' (' ||
   l_parameters || '); END;'
```

Some of the possibilities with Dynamic SQL

- Build ad-hoc query and update applications.
  – The user decides what to do and see.
- Execute DDL statements from within PL/SQL.
  – Not otherwise allowed in a PL/SQL block.
- Soft-code your application logic, placing business rules in tables and executing them dynamically.
  – Usually implemented through dynamic PL/SQL

Two Mechanisms Available

- DBMS_SQL
  - A large and complex built-in package that made dynamic SQL possible in Oracle7 and Oracle8.
- Native Dynamic SQL
  - A new (with Oracle8i), native implementation of dynamic SQL that does *almost* all of what DBMS_SQL can do, but much more easily and usually more efficiently.
  - EXECUTE IMMEDIATE
  - OPEN cv FOR 'SELECT ... '

Four Dynamic SQL Methods

- Method 1: DDL or DML without bind variables
  - EXECUTE IMMEDIATE *string*
- Method 2: DML with fixed number of bind variables
  - EXECUTE IMMEDIATE *string* USING
- Method 3: Query with fixed number of expressions in the select list
  - EXECUTE IMMEDIATE *string* INTO
- Method 4: Query with dynamic number of expressions in select list or DML with dynamic number of bind variables.
  - DBMS_SQL is best.

And then there's dynamic PL/SQL....

Method 1: DDL within PL/SQL

- The simplest kind of dynamic SQL.
  - All you can do is pass a string for execution, no values are bound *in*, no values are passed out.
- Always performs an implicit commit.
- Should be used with great care, since a DDL change can cause a ripple effect of invalidating program units.
- Common problem: Insufficient privileges.
  - Directly granted privileges are needed!

dropwhatever.sp
create_index.sp
settrig.sp
create_user.sql
ddl_insuff_privs.sql

Method 2: DML with fixed # of bind variables

- Add the USING clause to EXEC IMMEDIATE to supply bind values for placeholders.
  - Placeholders are strings starting with ":".
- USING elements can include a mode, just like a parameter: IN, OUT or IN OUT.
  - OUT and IN OUT are for dynamic PL/SQL
- Must provide a value for each placeholder.
  - With dynamic SQL, even if the same placeholder is repeated, you must provide the repeat value.

method_2_example.sql
updnval*.*

Dynamic FORALL Method 2 Example

- This example shows the use of bulk binding and collecting, plus application of the RETURNING clause.

```
CREATE TYPE NumList IS TABLE OF NUMBER;
CREATE TYPE NameList IS TABLE OF VARCHAR2(15);

PROCEDURE update_emps (
   col_in IN VARCHAR2, empnos_in IN numList) IS
   enames NameList;
BEGIN
   FORALL indx IN empnos_in.FIRST .. empnos_in.LAST
      EXECUTE IMMEDIATE
        'UPDATE emp SET ' || col_in || ' = ' || col_in
             || ' * 1.1 WHERE empno = :1
         RETURNING ename INTO :2'
         USING empnos_in (indx )
         RETURNING BULK COLLECT INTO enames;
   ...
END;
```

Notice that empnos_in is indexed, but enames is not.

Method 3: Query with fixed # in select list

- Add the INTO clause to EXEC IMMEDIATE to retrieve values from query.
  - May be in addition to the USING clause.
  - If you don't know the number at compile time, cannot use the INTO clause.
- Usually you are dealing with a dynamic table, column name or WHERE clause.
- The INTO clause can contain a list of variables, a record, a collection, etc.

tabcount_nds.sql
next_key.sf
method_3_examples.sql

Dynamic BULK COLLECT Method 3

- Now you can even avoid the OPEN FOR and just grab your rows in a single pass!

```
CREATE OR REPLACE PROCEDURE fetch_by_loc (loc_in IN VARCHAR2)
IS
   TYPE numlist_t IS TABLE OF NUMBER;
   TYPE namelist_t IS TABLE OF employee.name%TYPE;
   TYPE employee_t IS TABLE OF employee%ROWTYPE;

   emp_cv   sys_refcursor;

   empnos   numlist_t;
   enames   namelist_t;
   l_employees employee_t;
BEGIN
   OPEN emp_cv FOR 'SELECT empno, ename FROM emp_' || loc_in;
   FETCH emp_cv BULK COLLECT INTO empnos, enames;
   CLOSE emp_cv;

   EXECUTE IMMEDIATE 'SELECT * FROM emp_' || loc_in
      BULK COLLECT INTO l_employees;
END;
```

return_nested_table.sf

Quiz!

- What's wrong with this code?
- How would you fix it?

```
PROCEDURE process_lineitem (
   line_in IN PLS_INTEGER)
IS
BEGIN
   IF line_in = 1
   THEN
      process_line1;
   END IF;

   IF line_in = 2
   THEN
      process_line2;
   END IF;
   ...
   IF line_in = 22045
   THEN
      process_line22045;
   END IF;
END;
```

From 22,000 lines of code to 1!

```
PROCEDURE process_lineitem (
   line_in IN INTEGER)
IS
BEGIN
   IF line_in = 1
   THEN
      process_line1;
   END IF;

   IF line_in = 2
   THEN
      process_line2;
   END IF;
   ...
   IF line_in = 22045
   THEN
      process_line22045;
   END IF;
END;
```

```
PROCEDURE process_lineitem (
  line_in IN INTEGER)
IS
BEGIN
   EXECUTE IMMEDIATE
      'BEGIN process_line'||
         line_in ||'; END;';
END;
```

- Identify the pattern and resolve it either with reusable modules or dynamic abstractions.

dynplsql.txt

Dynamic PL/SQL

- Dynamically construct, compile and run an anonymous block with EXECUTE IMMEDIATE.
  - Begins with BEGIN or DECLARE.
  - Ends with END;. The trailing semi-colon is required; otherwise it is parsed as an SQL statement.
- You can only reference globally-accessible data structures (declared in a package specification).
- Exceptions *can* (and should) be trapped in the block from which the dynamic PL/SQL was executed.

dynplsql8i.sp
dynplsql_nolocal.sql

Dynamic PL/SQL Possibilities

- There are so *many* possibilities....some things I have done:
  - Reduce code volume, improve maintainability.
  - Generic string parsing engine: parse any string into your own collection.
  - Generic calculator engine.
  - Implement support for "indirect referencing": read and change values of variables whose names are only determined at run-time.
- And there are also dangers: code injection.

dynvar.pkg
dyncalc.pkg

How to build dynamic PL/SQL code

- 1. Build a static version of the logic you want to execute dynamically.
  - Test it thoroughly.
- 2. Identify the portions of the static code which will need to be made dynamic.
- 3. Convert the block, concatenating or binding those portions which are now dynamic.

1. Write and verify the static block code.

- Here is a static program to parse a string of directories for the path list.

```
PROCEDURE setpath (str IN VARCHAR2, delim IN VARCHAR2 := c_delim)
IS
   v_loc         PLS_INTEGER;
   v_startloc    PLS_INTEGER     := 1;
   v_item        VARCHAR2 (2000);
BEGIN
   dirs.DELETE;
   LOOP
      v_loc := INSTR (str, delim, v_startloc);

      IF v_loc = v_startloc
      THEN
         v_item := NULL;
      ELSIF v_loc = 0
      THEN
         v_item := SUBSTR (str, v_startloc);
      ELSE
         v_item := SUBSTR (str, v_startloc, v_loc - v_startloc);
      END IF;

      dirs (dirs.COUNT + 1) := v_item;

      IF v_loc = 0
      THEN
         EXIT;
      ELSE
         v_startloc := v_loc + 1;
      END IF;
   END LOOP;
END set_path;
```

filepath.pkg

2. Identify the dynamic elements of the block.

🟥 **Dynamic code**

🟦 **Bind variable**

```
PROCEDURE setpath (str IN VARCHAR2, delim IN VARCHAR2 := c_delim)
IS
   v_loc         PLS_INTEGER;
   v_startloc    PLS_INTEGER     := 1;
   v_item        VARCHAR2 (2000);
BEGIN
   dirs.DELETE;
   LOOP
      v_loc := INSTR (str, delim, v_startloc);

      IF v_loc = v_startloc
      THEN
         v_item := NULL;
      ELSIF v_loc = 0
      THEN
         v_item := SUBSTR (str, v_startloc);
      ELSE
         v_item := SUBSTR (str, v_startloc, v_loc - v_startloc);
      END IF;

      dirs (dirs.COUNT + 1) := v_item;

      IF v_loc = 0
      THEN
         EXIT;
      ELSE
         v_startloc := v_loc + 1;
      END IF;
   END LOOP;
END set_path;
```

3a. Convert from static to dynamic block.

- Assign the complex string to a variable.
- Makes it easier to report errors and debug.

```
dynblock :=
  'DECLARE
      v_loc PLS_INTEGER;
      v_start PLS_INTEGER := 1;
      v_item ' || datatype || ';
   BEGIN ' ||
      collname || '.DELETE;
      IF :str IS NOT NULL
      THEN
         LOOP
             v_loc := INSTR (:str, :delim, v_start);
             IF v_loc = v_startloc
             THEN
                v_item := NULL;
             ELSIF v_loc = 0
             THEN
                v_item := SUBSTR (:str, v_start);
             ELSE
                v_item := SUBSTR (:str, v_start, v_loc - v_start);
             END IF;' ||
             collname || '(' || nextrowstring || ') := v_item;

             IF v_loc = 0 THEN EXIT;
             ELSE v_start := v_loc + 1;
             END IF;
          END LOOP;
       END IF;
    END;';
```

str2list.pkg

3b. Execute the dynamic block.

- With dynamic PL/SQL, even if you reference the same bind variable more than once, you only specify it once in the USING clause.
  - In other words, PL/SQL is using a variation of "named notation" rather than the default positional notation for dynamic SQL statements.

```
EXECUTE IMMEDIATE dynblock
        USING str, delim;
```

Advanced Topics

- Dynamic SQL method 4
  - Most generic and challenging scenario
- Parsing very long strings
- Describe columns in query
- The problem of SQL injection
- Oracle11g enhancements

Method 4 Dynamic SQL with DBMS_SQL

- Method 4 dynamic SQL is the most generalized and most complex - by far!
  - You don't know at compile time either the number of columns or the number of bind variables.
  - With DBMS_SQL, you must put calls to DBMS_SQL.DEFINE_COLUMN and/or DBMS_SQL.BIND_VARIABLE into loops.
- With NDS, you must shift from dynamic SQL to dynamic PL/SQL.
  - How else can you have a variable INTO or USING clause?

Dynamic "SELECT * FROM <table>" in PL/SQL

Method 4

- You provide the table and WHERE clause. I display all the data.
  - I don't know in advance which or how many rows to query.
- I can obtain the column information from ALL_TAB_COLUMNS...and from there the fun begins!
- A relatively simple example to use as a starting point.

intab_dbms_sql.sp - uses DBMS_SQL
intab_nds.sp - uses NDS
intab.tst

Pseudo-code flow for DBMS_SQL implementation

Build the SELECT list

Parse the variable SQL

Define each column

Execute the query

Extract each value

```
BEGIN
   FOR each-column-in-table LOOP
      add-column-to-select-list;
   END LOOP;

   DBMS_SQL.PARSE (cur, select_string, DBMS_SQL.NATIVE);

   FOR each-column-in-table LOOP
      DBMS_SQL.DEFINE_COLUMN (cur, nth_col, datatype);
   END LOOP;

   fdbk := DBMS_SQL.EXECUTE (cur);

   LOOP
      fetch-a-row;
      FOR each-column-in-table LOOP
         DBMS_SQL.COLUMN_VALUE (cur, nth_col, val);
      END LOOP;
   END LOOP;
END;
```

Lots of code, but relatively straightforward

Also:
dyn_placeholder.*

Parsing very long strings

- One problem with EXECUTE IMMEDIATE is that you pass it a single VARCHAR2 string.
  - Maximum length 32K.
  - Very likely to happen when you are generating SQL statements based on tables with many columns.
  - Also when you want to dynamically compile a program.
- So what do you do when your string is longer?
  - In Oracle11g, can pass CLOBs...
  - Prior to 11g, time to switch to DBMS_SQL!

DBMS_SQL.PARSE overloading for collections

- Oracle offers an overloading of DBMS_SQL.PARSE that accepts a collection of strings, rather than a single string.
- DBMS_SQL offers two different array types:
  - DBMS_SQL.VARCHAR2S - max 255 bytes.
  - DBMS_SQL.VARCHAR2A - max 32,767 bytes
- New in Oracle11g: both NDS and DBMS_SQL accept CLOBs.

exec_ddl_from_file.sql

Describe columns in a query

- DBMS_SQL offers the ability to "ask" a cursor to describe the columns defined in that cursor.
- By using the DESCRIBE_COLUMNS procedure, you can sometimes avoid complex parsing and analysis logic.
  – Particularly useful with method 4 dynamic SQL.

desccols.pkg
desccols.tst

SQL (code) Injection

- "Injection" means that unintended and often malicious code is inserted into a dynamic SQL statement.
  - Biggest risk occurs with dynamic PL/SQL, but it is also possible to subvert SQL statements.
- Best ways to avoid injection:
  - Restrict privileges tightly on user schemas.
  - Use bind variables whenever possible.
  - Check dynamic text for dangerous text.
  - Use DBMS_ASSERT to validate object names, like tables and views.
  - Preface all built-in packages with "SYS."

code_injection.sql
sql_guard.*
dbms_assert_demo.sql

Oracle11g Enhancements

- EXECUTE IMMEDIATE a CLOB.
- Interoperability
  - Convert DBMS_SQL cursor to cursor variable
  - Convert cursor variable to DBMS_SQL cursor
- Improved security
  - Random generation of DBMS_SQL cursor handles
  - Denial of access/use of DBMS_SQL with invalid cursor or change of effective user.

Interoperability

- DBMS_SQL.TO_REFCURSOR
  - Cursor handle to cursor variable
  - Useful when you need DBMS_SQL to bind and execute, but easier to fetch through cursor variable.
- DBMS_SQL.TO_CURSOR
  - Cursor variable to cursor handle
  - Binding is static but SELECT list is dynamic

11g_to_cursorid.sql
11g_to_refcursor.sql

Best Practices for Dynamic SQL

- Stored programs with dynamic SQL should be defined as AUTHID CURRENT_USER.
- Remember that dynamic DDL causes an implicit commit.
  - Consider making all DDL programs autonomous transactions.
- Always EXECUTE IMMEDIATE a variable, so that you can then display/log/view that variable's value in case of an error.
- Avoid concatenation;
  bind whenever possible.

dropwhatever.sp
usebinding.sp
toomuchbinding.sp
useconcat*.*
ultrabind.*

NDS or DBMS_SQL: Which should you use?

- Reasons to go with NDS:
  - Ease of use
  - Works with all SQL datatypes (including user-defined object and collection types)
  - Fetch into records and collections of records
  - Usually faster runtime performance

- Why You'd Use DBMS_SQL:
  - Method 4 Dynamic SQL
  - DESCRIBE columns of cursor
  - SQL statements larger than 32K (prior to 11g)
  - Better reuse of parsed SQL statements -- persistent cursor handles!

Bottom line: NDS should be your first choice.

Dynamic SQL Conclusions

- Dynamic SQL is needed in most applications.
- Native dynamic SQL makes it easy.
- Increased complexity means you need to take more care to write code that is easy to understand and maintain.

- And now...a demonstration of the Oracle evaluation website!