The Truth… let’s be fair…

- I’ve never seen:
  - 100% perfect analysis
  - 100% complete set of requirements
  - 100% adequate hardware
Unknowns

- What elements are involved?
- What do you do with the elements you have?
- How you should proceed?
- Do you know whether or not you can proceed?
Dynamic SQL:
- Makes it possible to build and process complete SQL and PL/SQL statements as strings at runtime.
Yes, but…

I’ve never seen:

- 100% perfect analysis
- 100% complete set of requirements
- 100% adequate hardware
- 100% avoidance of Murphy's Law:

IF SOMETHING CAN BE MISUSED
IT WILL BE MISUSED!!!
Back to Basics:
What are we talking about?
Dynamic SQL & PL/SQL

- What does "dynamic" mean?
  - Build a text string on the fly and execute it
  - Very powerful technique
  - Useful in many contexts

- What techniques exist?
  - EXECUTE IMMEDIATE
  - Dynamic cursors
  - DBMS_SQL
About 90% of dynamic SQL is covered by a single command (with variations):

```
declare
    v_variable_tx varchar2(32000);
begin
    v_variable_tx := 'whatever_you_want';
    EXECUTE IMMEDIATE v_variable_tx;
end;
```

OR

```
begin
    EXECUTE IMMEDIATE 'whatever_you_want';
end;
```
Dynamic Cursors

◆ Syntax

```sql
declare
    v_cur SYS_REFCURSOR;
    v_sql_tx varchar2(32000):=...;
    v_rec ...%rowtype; -- or record type
begin
    open v_cur for v_sql_tx;
    fetch v_cur into v_rec;
    close v_cur;
end;
```

◆ Most common use:
  - Processing large datasets with unknown structure
DBMS_SQL package

Predecessor of native dynamic SQL

- **Pros:**
  - Goes above 32K in all versions
  - Separates PARSE and EXECUTE
    - The same query can be reused with different bind variables.
  - Works with unknown number/type of INPUT/OUTPUT values

- **Cons:**
  - Significantly slower (up to 5 times)
  - No user-defined datatypes or output to the record
  - More difficult to use
Things to know (1)

- The code can be passed as a variable/string.
- The variable/string cannot exceed 32K, except
  - In 11g – passing CLOB as input
  - If the database is using fonts where 1 byte=1 char
    – up to 64K when concatenate a number of strings:
      
      ```sql
      execute immediate
      v1_txt || v2_txt || v3_txt || ...;
      ```
  - In 8i and below – no limit on concatenation
    (undocumented feature - not supported by Oracle!)
The variable/string can contain bind variables

- Start with a colon (:)  
- Placeholders for values that will be supplied at runtime (USING and RETURNING clauses)  
- No validation when compiled
  - No check for datatypes
  - No check for passing enough values
- No direct limit on the number of bind variables.
- Like any other variables, they may be IN (default), OUT, or IN/OUT
Things to know (3)

◆ Bind variables are limited:
  - CAN only be used to supply values to be passed to the code
  - CANNOT be used to define the structural elements of queries or PL/SQL blocks.

◆ Special cases:
  - You cannot pass NULL as a literal.
  - If statement has a RETURNING clause, it should also be used in dynamic SQL.

◆ Bind variables may or may not be reusable.
  - ARE NOT reusable in dynamic SQL – the number of variables is equal to the number of parameters.
  - ARE reusable in dynamic PL/SQL – the number of UNIQUE variables is equal to the number of parameters.
1. Use bind variables
2. Use bind variables
3. Use bind variables
4. Use bind variables
5. Use bind variables
6. Use bind variables
7. Use bind variables

© Tom Kyte
function F_GET_col_TX
(i_table_tx,i_showcol_tx,i_pkcol_tx,i_pkValue_nr)
return varchar2 is
    v_out_tx varchar2(4000);
    v_sql_tx varchar2(32000);
Begin
    v_sql_tx:=
        'select to_char('||i_showcol_tx||')'
        ||
        ' from '||i_table_tx||
        ' where '||i_pkcol_tx||'='||:v01';
EXECUTE IMMEDIATE v_sql_tx INTO v_out_tx
USING i_pkValue_nr;

    return v_out_tx;
end;
Security
What privileges do you need to use dynamic SQL?

How can you guard against misuse of dynamic SQL?
Granting Privileges (1)

◆ All privileges have to be granted explicitly (Not via ROLES):

➢ System privileges

```sql
create or replace procedure p_makeTable
  (i_name_tx varchar2) is
begin
  execute immediate 'create table ' || i_name_tx || ' (a_tx varchar2(256))';
end;
```

- GRANT DBA to SCOTT … wrong!!!
- GRANT CREATE TABLE to SCOTT – correct!
Granting Privileges (2)

- **Object privileges**

  ```
  function f_getCount_nr (i_user_tx varchar2, i_table_tx varchar2) return number is
  begin
    execute immediate 'select count(*) from ' || i_user_tx || '.' || i_table_tx;
  end;
  ```

  - GRANT DBA to SCOTT … wrong!!!
  - GRANT SELECT on EMPLOYEE to SCOTT – correct!
Fighting Code Injections

◆ DBA protection
  ➢ End users should not see administration tools

◆ UI protection
  ➢ User input should always be passed via bind variables (no concatenation!)
    ▪ Bind variables cannot affect the structure of the query.
  ➢ All structural selections should be made from the limited list of options (repository)
    ▪ Power users/developers populate the repository.
    ▪ End users only access whatever is already in the repository.
Use Case

◆ The problem:
  ➢ Large number of requests from the user interface
    ▪ Take some number of parameters
    ▪ Return something (search, extra info, status, etc.)
  ➢ The whole set of requests is not clear and may change each time.

◆ The solution:
  ➢ Universal wrapper
    ▪ Process all requests dynamically
Single table for both UI and server:

```sql
create table t_extra_ui
(id_nr number primary key,
displayName_tx varchar2(256),
function_tx varchar2(50),
v1_label_tx varchar2(100),
v1_type_tx varchar2(50),
v1_required_yn varchar2(1),
v1_lov_tx varchar2(50),
v1_convert_tx varchar2(50),
... )
```

Example:

```sql
insert into t_extra_ui (...)
values (1, 'Filter Employees', 'f_getEmp_cl',
'Job','TEXT','N',null,null)
```
Published function

- **ID_NR** – unique ID of the function
- **DisplayName_tx** – header of the screen
  - ID and display are shown to users as LOV
- **Function_tx** – real function to be called

Parameters (never needed more than 10)

- **Vx_Label_tx** – label for the parameter
  - if null – parameter is disabled
- **Vx_Type_tx** – helps UI to build the screen – can be:
  - LOV – value list
  - TEXT – free text
  - DATE – attached calendar is needed
- **Vx_Required_yn** – helps UI enforce needed parameters
- **Vx_LOV_tx** – name of the corresponding value list
- **Vx_Convert_Tx** – any expression with one input
  - Example - 'to_date(:1, "YYYYMMDD")' – transformation to the real date
  - Should always use bind variable with correct ID
function \texttt{f\_getEmp\_CL} (i\_job\_tx \texttt{varchar2}) return \texttt{CLOB} is

\texttt{v\_out\_cl} \texttt{CLOB};

procedure \texttt{p\_add} (pi\_tx \texttt{varchar2}) is

begin
  \texttt{dbms\_lob.writeappend(v\_out\_cl,length(pi\_tx),pi\_tx)};
end;

begin
  \texttt{dbms\_lob.createtemporary(v\_out\_cl,true,dbms\_lob.call)};
  \texttt{p\_add('<html><table>')};

for c in (select '<tr>'||'<td>'||empno||'</td>'||
                '<td>'||ename||'</td>'||
            '</tr>' row\_tx
       from emp where job = i\_job\_tx)
loop
  \texttt{p\_add(c.row\_tx)};
end loop;

\texttt{p\_add('</table></html>')};

return \texttt{v\_out\_cl};
end;
function f_wrapper_cl (i_id_nr, 
    v1_tx varchar2:=null,...,v5_tx varchar2:=null)
return CLOB is
    v_out_cl CLOB;
    v_sql_tx varchar2(2000);
    v_rec t_extra_ui%rowtype;
begin
    select * into v_rec from t_extra_ui where id_nr=i_id_nr;

    if v_rec.v1_label_tx is not null then
        v_sql_tx:=nvl(v_rec.v1_convert_tx,'':1''');
    end if;
...
    if v_rec.v5_label_tx is not null then
        v_sql_tx:=v_sql_tx||','||nvl(v_rec.v5_convert_tx,'':5''');
    end if;
v_sql_tx := 
'begin :out:=''v_rec.function_tx'||
'(v_sql_tx||'); end;';

if v5_tx is not null then
    execute immediate v_sql_tx
        using out v_out_cl, v1_tx,...,v5_tx;
...
elsif v1_tx is not null then
    execute immediate v_sql_tx
        using out v_out_cl, v1_tx;
else
    execute immediate v_sql_tx using out v_out_cl;
end if;

return v_out_cl;
end;
11g – Critical Addition

- **DBMS_ASSERT** – validating strings:
  - SQL_OBJECT_NAME (string) – checks whether or not string is a valid object
  - SIMPLE_SQL_NAME – checks whether or not string is a valid SQL name
  - SCHEMA_NAME – validate that passed string is a valid schema
  - ENQUOTE_NAME – add a second quote to every instance in the name (and double quotes around)
  - ENQUOTE_LITERAL – add single quotes

Exists in 10g – but not documented!!!
function F_GET_col_TX (i_table_tx, i_showcol_tx, i_pk_tx, i_pkValue_nr) return varchar2 is
  v_out_tx varchar2(4000);
v_sql_tx varchar2(32000);
Begin
  v_sql_tx :=
    'select to_char(' || 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;
Object Dependencies
Dynamic SQL is executed at runtime, therefore:

- **Bad things**
  - No dependencies to follow up.
  - No way to determine exactly what will be executed.

- **Good things**
  - You can reference objects that may not be in the database (yet).
  - You have a “back door” to resolve logical dead loops or hide existing dependencies.
Con#1: No dependencies

◆ What do to:
  - Use repositories

◆ How:
  - Populate repositories with required information, structured similarly to that of the Oracle data dictionary
  - Generate from the repository using a straightforward, transparent mechanism (so it is clear what becomes what)
  - Compare Oracle data dictionary with your own
Con#2: What is executed?

◆ What to do:
  - Use samplers
◆ How:
  - Generate all possible (or as many as possible) permutations of the code to be executed and create PL/SQL modules with that code.
  - Record all dependencies and keep a simple module that references the same set of objects.
  - If sampler becomes invalid, this means that you have problems.
Pro#1: Reference non-existing objects

- Objects may not exist.
  - Example: Monthly summary table

- Object may be referencing DBLink with a different maintenance cycle.
  - Example: Remote database may be down when you need to recompile a dependent object
Pro#2: Invalidation issues

◆ Code generators:
  - Wrap call of generated modules into dynamic SQL
    - Then you can refresh them without invalidating all dependencies

◆ Logical loops:
  - Sometimes the simplest way out of the dependencies loop is to convert one of the calls to dynamic SQL
Bulk Operations
Supported Features

◆ Native Dynamic SQL supports object collections and all kinds of operations on object collections:
  - FORALL (Currently only in USING clause)
  - BULK COLLECT

◆ DBMS_SQL supports only arrays as bulk list of bind variables
  - Fixed in 11g!
FORALL

FOR ALL – with limitations:

- In USING clause:
  - Right:
    ```sql
    forall i in dept.first..dept.last
    execute immediate
    'delete from emp where deptno=:1' using dept(i);
    ```
  - Wrong:
    ```sql
    forall i in dept.first..dept.last
    execute immediate 'drop table t_'||dept(i);
    ```

- The whole object only (fixed in 11g!)
  - Right - `dept(i)`
  - Wrong - `dept(i).deptno`
BULK COLLECT

◆ Syntax:

EXECUTE IMMEDIATE ... BULK COLLECT INTO v_collect_tt;

◆ Major advantage

- Even Dynamic SQL does not support any PL/SQL datatypes.
  - You can use RECORD as an output of a dynamic query.
- Dynamic SQL does support all user-defined SQL datatypes.
 Constructor should be used INSIDE of Dynamic SQL

define lov_oty is object (id_nr ..., disp_tx...);
define lov_nt as table of lov_oty;

function f_getLov_nt
    (i_table_tx,i_id_tx,i_disp_tx)
return lov_nt is
    v_out_nt lov_nt := lov_nt();
begin
    execute immediate 'select lov_oty('||i_id_tx||','||i_disp_tx||')'||
        ' from '||i_table_tx
    bulk collect into v_out_nt;
return v_out_nt;
end;
Transformation of cursors for DBMS_SQL
11g Additions

- Conversion between REF_CURSOR and DBMS_SQL cursor
  - DBMS_SQL.TO_REFCURSOR
  - DBMS_SQL.TO_CURSOR_NUMBER
The problem:
- A lot of REF Cursors in the system with no clear way of figuring out what exactly they are

The solution:
- Generic routine to describe REF Cursor with minimal impact on the system
procedure p_expCursor
  (io_ref_cur IN OUT SYS_REFCURSOR) is
  v_cur integer := dbms_sql.open_cursor;
  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;
Performance
and
Resource Utilization
Keep in mind (Very Important!)

- **EXECUTE IMMEDIATE** and dynamic cursors:
  - If you use bind variables – 1 hard + N soft parses
  - If you don’t use bind variables – N hard parses

- **DBMS_SQL** – the same plus:
  - Extra option – only 1 parse
Use Case

The problem:
- Users upload CSV-files
  - Name of file defines type
  - Column headers map directly to table columns
  - 1 row of file = 1 logical group (1..N real rows)
  - Group-level validation

The solution:
- Universal CSV-loader
  - Build all inserts on the fly
Declare
    type integer_tt is table of integer;
    v_cur_tt integer_tt;
Begin
for r in v_groupRow_tt.first..v_groupRow_tt.last loop
    v_cur_tt(r):=DBMS_SQL.OPEN_CURSOR;
    for c in c_cols(v_mapRows_tt(r)) loop
        for i in v_header_tt.first..v_header_tt.last loop
            if v_header_tt(i).text=c.name_tx then
                v_col_tt(i):=c;
                v_col_tx:=v_col_tx||','||v_col_tt(i).viewcol_tx;
                v_val_tx:=v_val_tx||',:'||v_col_tt(i).viewcol_tx;
            end if;
        end loop;
    end loop;
    v_sql_tx:='insert into '||v_map_rec.view_tx||
        ' (''||v_col_tx||'' ) values(''||v_value_tx||'' );
    DBMS_SQL.PARSE(v_cur_tt(r),v_sql_tx,DBMS_SQL.NATIVE);
end loop;
for i in 2..v_row_tt.count loop
  for r in v_groupRow_tt.first..v_groupRow_tt.last loop
    for c in v_col_tt.first..v_col_tt.last loop
      if v_col_tt(c).id = v_mapRows_tt(r) then
        DBMS_SQL.BIND_VARIABLE(v_cur_tt(r),
                                ':'.||v_col_tt(c).viewcol_tx,
                                v_data_tt(c).text);
      end if;
    end loop;
    v_nr:=dbms_sql.execute(v_cur_tt(r));
  end loop;
end loop;
Best Practices

◆ Whenever possible:
  ➢ Use BULK operations
  ➢ Minimize parsing
  ➢ Use bind variables

◆ Options to consider:
  ➢ Build a code repository
  ➢ Use generated code
Dynamic SQL does:
- Significantly extend the list of available options for resolving many problems
- Provide extra maneuvering room in production environments

Dynamic SQL should NOT:
- be considered a substitute for good analysis
- be used where “regular” solutions are valid
Contact Information

- Michael Rosenblum – mrosenblum@dulcian.com
- Dulcian website - www.dulcian.com

Latest book:
Oracle PL/SQL for Dummies