USING SQL AS A CODE GENERATOR
(or 'Teaching Oracle to write SQL scripts')

Peter G Robson
British Geological Survey

Summary. Valid SQL statements can be generated from Oracle by embedding SQL syntax, in association with retrieved data, within a 'parent' script. The execution of that parent script will itself generate the new SQL statement.
However, this is a complex process, particularly when multiple hierarchies of scripts are built into one parent script, combined with the retrieval of attributes used as variables in the next embedded script. This latter stage often requires what appear to be confusingly large numbers of single quote differentiators.
The technique is powerful and important. It demonstrates the convergence of true set-based and sequential processing, entirely within the standard SQL environment. Programming problems can be solved which would otherwise have to be implemented in a host language.
The paper will present a series of increasingly complex examples of the technique, while clearly identifying the rules which enable the successful construct of these multi-layer scripts.

Introduction

Experienced Oracle practitioners have often used the embedded SQL script as a means of achieving a particular programming requirement. Although ones encounters occasional references and examples to this technique within the Oracle community, mostly in the Oracle email listings rather than in published texts, no extensive discussion of the technique has been found.

This paper will introduce the technique with examples, while explaining how the constructs are achieved, before moving on to some real world examples of how the technique can be applied in every day situations.

Beginnings

In each of the following example scripts, pagesize has been set to zero, thus removing headings as well. These scripts are presented purely to demonstrate some of the procedures for constructing multi-level nested queries.

The most simple example of the nested script might be as follows:

```
select
   'Select sysdate from dual;'
from dual;
```

this would produce the following:

```
select sysdate from dual;
```

which in its turn would of course present:

25-DEC-1999

(Or whichever date one happened to be running the test on.) To take this example a step further, and instead of selecting sysdate, a literal string retrieval might look like this:

```
select 'Hello' from dual;
```

Now combine it with an attribute COLUMN_A from a table MY_TABLE which has only one row, and the value ‘World’ in the column COLUMN_A in that row:

```
select ‘Hello ‘,COLUMN_A from my_table;
```

which presents:

Hello World

Now nest this query:

```
select
   'select
      ''Hello '',
      column_a
   from
      my_table;'
from dual;
```

and when run, the result is the prviously illustrated script:

```
select
      'Hello ',
      column_a
   from
      my_table;
```

The Points to Note from the above are as follows:

1. Layout is all important for reasons of clarity - things can become very confusing as the number of nested statements increases.
2. The inner ‘select’ statement is enclosed in an opening quote (immediately after the first ‘select’, and just before the second ‘select’), and a closing single quote (after the first ‘;’ and before the last ‘from dual;’)
3. Quotes are doubled where they are already present in the nested select, i.e. ‘Hello ’ becomes ‘‘Hello ‘‘

Taking the above query, a multiple nested example (with comments appended in square brackets) might look like the following:

```
select
   'select               [ x1 quote mark ]
      ''select               [ x2 ]
         ''''select              [ x4 ]
            ''''''''select           [ x8 ]
                  ''''''''''''''''Hello'''''''''''''''', [ x 16 ]
                  column
               from
                  my_table;''''''''
         from dual;''''
      from dual;''
   from dual;'
from dual;
```

The transition from one nested query to the above (a 5-level nested query - there are 5 distinct ‘select’ commands present) is explained and demonstrated in the presentation, but, study the above, and it can be seen how every time a new level of nesting is applied, any quote mark within the nested query (and this applies to any sub-nests, of course) has the incidence of quote marks doubled. Count each instance of the quotes, and it can be seen that they conform very strictly to the above rule.

If the above query is run, the result is a 4-level nested query, which, if run, will itself produce a 3-level query, and so on, until the final result ‘Hello World’ is produced.

Spooling

So far each resultant script has had to be explicitly run in its turn. This now introduces the second stage in making use of this technique, namely spooling a result, and then automatically running the resultant spool file. One might expect that the spool commands should be embedded within the body of the large script as above, such that each ‘spool filename’ is embedded immediately prior to each ‘select’ command, with a complimentary pairing of ‘spool off’ and ‘@filename’ following each ‘from dual;’ command.

Although this approach can be followed, and indeed will be briefly demonstrated later, it is in fact quite unnecessary, and indeed is far more complex that the preferred solution. Consider the following demonstration using an outline structure:

```
spool A1.SQL
      [ the total nested SQL script as above ]
spool off
  spool A2.SQL
  @A1.SQL
    spool off
    spool A3.SQL
  @A2.SQL
      spool off
      spool A4.SQL
  @A3.SQL
      spool off
  @A4.SQL
```

This is something to demonstrate to oneself, as it shows that the final result is actually produced by the last script file, A4.SQL. Each one of these spool files can be individually examined, to see how each generation of the nested script develops. So, for a 5-level nested query, four intermediate spool files are used. However, one need never take any interest in any of these spool files, as their job is simply to present the data back to SQL*Plus, or finally back to the user. To be complete, any command-line script employing this technique should delete these spool files once processing is over.

The above method of spooling is by far the easiest way to achieve multi-nested spooling. However, it is instructive to look at the alternative approach of embedding the spool commands inside the nested script, as this introduces a new and important aspect of controlling nested scripts. Consider the following example:

```
spool a1.sql
select
  'spool a2.sql',
  'select ',
  ‘Sysdate ’,
  'from dual;',
  'spool off',
```

```
  '@a2.sql'
from dual;
```

If this is submitted to SQL just as it appears above, the following is the results:

spool a2.sql select sysdate from dual; spool off @a2.sql

which is no good at all. The line feeds must be retained if the next spool is to work. Therefore, an explicit line feed has to be introduced into the script. This is done using the ASCII line feed control character chr(10), which amends the first script to the following:

```
spool a1.sql
select
  'spool a2.sql' || chr(10) ||
  'select '      || chr(10) ||
  'Sysdate '     || chr(10) ||
  'from dual;'   || chr(10) ||
  'spool off'    || chr(10) ||
  '@a2.sql'
from dual;
```

this results in:

spool a2.sql
select
sysdate
from dual;
spool off
@a2.sql

which is exactly what is required. When run in its turn, this provides the correct result. Note that both the chr(10) control character, as well as the '||' concatenation character have been used. This latter feature is used repeatedly in this technique of nested queries, and is most useful in truncating trailing space from field values which don't extend to the full width of the field definition. Frequent use of these two features will be made in later examples.

The above queries are only really of use to demonstrate the syntax required to construct nested queries. A simple real world application of a nested query will now be examined.

Imagine that one has just created, for example, 20 new tables. It is then required to assign identical access privileges to each one of them for a category of Oracle users. One could submit a single grant command for each and every table. However, this task can be more simply achieved using just one submission, utilising a 2-level nested script, as follows:

```
spool A1.sql
select
   'Grant select on '||table_name||' to public;'
from user_tables where created > sysdate -1;
spool off
@A1.SQL
```

It is assumed that these twenty tables are the only tables created today, and that 'public' is the target community to be provided with select access. The result of the above script would be to generate twenty instances of the 'grant select on table_01 to public;' within the spool file A1.SQL, which would then be executed when that spool file was itself executed (and assuming the first table name to be 'table_01').

Of course, there are many variations on the above theme which might be used. The tables could be identified, not on date of creation, but on a naming characteristic, such as all beginning or ending with

a certain set of characters.

Looking at a Table Description

Consider another example, where one wants to generate a description of several tables which can all be identified in some way, and also pull back information regarding the number of rows in that table. The presentation will work through the means of achieving this end, but for reasons of brevity, the final solution script will be presented here:

```
set verify off
spool a.sql
select
'set heading on'|| chr(10) ||
'select column_name "Name"
 ,decode(nullable,''N'',''NOT NULL'',''Y'','' '','' '') "Null?"
 ,data_type "Type"
 from user_tab_columns
 where table_name='||''''||table_name||''''|| chr(10) ||
 'order by column_id;' || chr(10) ||
 'select count(*) "No of Rows in '
       ||table_name||'" from'||table_name||';'|| chr(10)
from user_tables;
spool off
@a.sql
```

This is a 2-level nested query, but for every row returned from the control table (in this case `user_tables`), two new queries are built, one of which presents the familiar output from the SQL*Plus ‘Describe’ command, and the second is a count of the table, the name of which is retrieved. However, this last ‘count’ query has a twist - notice how the heading for the count is now switched to dynamically give the name of the table being counted. Once the main script has run and generated the spool file ‘a.sql’, this latter will produce the following type output for every table found in the master ‘where’ clause:

```
Name                           Null?    Type
------------------------------ -------- ---------
OWNER                          NOT NULL VARCHAR2
TNAME                                   VARCHAR2
STREET                                  VARCHAR2
AREA                                    VARCHAR2
CITY                                    VARCHAR2
COUNTRY                                 VARCHAR2

6 rows selected.

No of Rows in DATA_TABLE
------------------------
                       7
```

The presentation will develop this script further.

Inserting Data into a Table

Often, one may want to enter one or two new rows into an Oracle table, but not want to bother pulling up a screen editor such as TOAD to do the job. (There is an underlying assumption here that one has SQL*Plus open all the time, which is almost certainly the case for a DBA...) Normally, one would have to check the table attributes with the ‘describe’ command, then construct an insert

command along the lines of:

```
insert into table_name (column1, column2,...) Values
('value1','value2',...);
```

Using the following script, the entire process can be automated. This script uses a simple combination of embedded scripts, combined into one larger script. It introduces a further few issues which need to be considered when running multiple queries through spool files.

The characteristics of the script will be commented on as the script is displayed. Each of the 9 Parts should be appended in the order in which they occur, to produce the final script, which can then be run.

First, the SQL*Plus environment needs to be managed to stop headings, feedback data, page throws, and so on, from being incorporated into the generated script files. These can all be eliminated using a few lines of 'settings' commands. These commands could be placed into a small file, which itself would be called at the start of the main script file.

Part 1:

```
set echo off
set sqlprompt ""
set trims on
set pagesize 0
set feedback off
set verify off
undefine table
```

By setting the pagesize to zero, headings are automatically suppressed. The value of the variable 'Table', about to be used, is undefined in case any value remains in it from a previous run. Prompt and Accept commands are used to take the name of the table to operate on.

Part 2:

```
prompt Enter name of table to populate:
accept table prompt ' Table: '
```

The spool file, which is going to collect the SQL script to do the work, is called insert.sql. The first data to go into it is the basic 'insert' command:

Part 3:

```
spool insert.sql

select
'insert into &&table',
' (' from dual;
```

The next task is to generate a complete list of the attributes owned by the table, with the exception of the last attribute. There are various ways of building this script, using 'prompt' for example, but the writer prefers to use 'select ... from dual;' whenever there is an alternative.

Part 4:

```
select
column_name||' ,'
from user_tab_columns
where table_name = upper('&&table')
and column_id <
( select max(column_id)
  from user_tab_columns
```

```
  where table_name = upper('&&table'));
```

The purpose of the above construct is to build a list of all table attributes followed by a comma - of course, the last attribute is followed by a closing round bracket ‘)’, and has to be treated differently. Similarly, one could have treated the first attribute differently, and prefaced all the remaining columns with a comma.

Part 5:

```
select
column_name
from  user_tab_columns
where column_id =
( select max(column_id)
  from user_tab_columns
  where table_name = upper('&&table'))
and  table_name = upper('&&table') ;
```

The intermediate ‘values’ syntax word is inserted here, together with its surrounding round brackets:

Part 6:

```
select ') values (' from dual;
```

followed by the script which prompts the user for input to every field in turn, again with the exception of the very last attribute:

Part 7:

```
select
'''&'||column_name||''' ,'
from user_tab_columns
where table_name = upper('&&table')
and column_id <
( select max(column_id)
  from user_tab_columns
  where table_name = upper('&&table'));
```

The last attribute is now attached:

Part 8:

```
select
'''&'||column_name||'''   )'|| chr(10) ||
'/ '
from  user_tab_columns
where column_id =
( select max(column_id)
  from user_tab_columns
  where table_name = upper('&&table'))
  and table_name =upper('&&table')
;
spool off
undefine table
```

The build file ‘insert.sql’ is complete, and some of the SQL*Plus settings can be turned back on before finally starting up ‘insert.sql’ itself:

Part 9:

```
set sqlprompt "SQL>"
set feedback on
set pagesize 24
set trims on
@insert.sql
```

All nine parts should be assembled into one script file, and run. At this point, one will be asked to name the table into which rows are to be inserted.

Once that script has completed, a new script ‘insert.sql’ will have been generated, and will then automatically run, prompting for each attribute in turn for one row to be inserted. Subsequent rows can be inserted into the same table by entering the ‘/’ command at the SQL> prompt, as many times as rows are to be inserted.

There are several ways to both construct and enhance this script, for example by informing a user when a column is ‘Not Null’, and therefore demands a data item, but this is something for the reader to experiment with. Note that ‘set verify on’ has not been restored – one could choose to insert this control at the end of the script ‘insert.sql’ as it is being created.

The above is presented as an example of ‘process’, rather than as a definitive solution in its own right.

Counting table rows in a pre-defined suite of tables.

Finally, the last example to be illustrated shows two very compact, 3-level nested scripts which achieves a solution that would previously have required a 3GL or PL/SQL solution. The problem is that a number of tables have to be counted, and the values of those counts inserted into another table. The original list of tables to be counted is held in a reference table. There are several distinct stages to this which map naturally into a 3GL solution context. The challenge is to accomplish the entire process in SQL.

A reference table contains two attributes, holding the owner and name of the tables to be counted. The result of this count is to be inserted into a second table, (Monitor_Count, with columns: owner, table_name, table_count, count_date), together with the current date to indicate when the count was carried out. Attribute format and length are not important here.

To demonstrate the script construct, the presentation will work backwards from the final result. The last stage is, therefore, the insertion into the data file of the name of the owner of the table, the table_name currently being counted, the rows counted for that table, and the current date:

Script 1:

```
insert into  monitor_count
(owner,table_name,table_count,count_date) values (
   'PGR','TEST_1',
   6,sysdate);
```

The script will appear for as many different tables as appear in the reference table (named `tables_to_count`), with the owner, table_name, and count result. Notice that this script contains the result of the count of rows in table TEST_1 (result being 6 rows), owned by PGR. The script which generated this must therefore have been of the type ‘select count(*)’. This is it, with the above Script 1 wrapped within it:

Script 2:

```
select
```

```
'insert into monitor_count
      (owner,table_name,table_count,count_date) values (
   ''PGR'',''TEST_1'''||',
   '||count(*)||',sysdate);' from TEST_1;
```

Notice how this select returns a string variable containing the name of the table which is itself the subject of the count(*). The result of this script is of course Script 1, above. Script 2 is generated by querying the table `table_to_count`, a query which will generate as many examples of the above script as there are tables cited in the reference table `tables_to_count`. It appears as follows:

Script 3:

```
select
  'select',
  '''insert into  monitor_count
     (owner,table_name,table_count,count_date) values (
  '''||''''||owner||''''''||',''
   ||''''''||tname||''''''''||'''||''',
'''||'''||'||'count(*)'||'||'',sysdate);'' from '||tname||';'
 from tables_to_count;
```

This is the final and complete script, embodying the previous two scripts. To avoid confusion, the name of the attribute in the table `tables_to_count` which holds the name of the table to count is named ‘tname’. Script 3 is the owner query from which the complete result is generated. Again, the spooling is achieved by wrapping this script in the following spool commands:

```
spool a1.sql
   [ the command script - Script 3 ]
 spool off
spool a2.sql
@a1.sql
spool off
@a2
```

Running the query then results in the reference table being interrogated - every table which is listed there will be counted. The result of that count is then inserted into a data table holding the count information. The query is a 3-level nested example, in which the outer nest is the controlling ‘select’. This is followed by the first inner nest which is another ‘select’. The innermost nested query is the one which inserts the row count information into the data table.

A further variation of this approach can be seen in the next complete script, which does not insert data into a data table, but rather updates a single table, the same table holding a list of all rows to be counted. This is actually a neater solution, as only one reference table is being used to both pass back the names of each table to be counted, and to subsequently receive the number of rows counted. The difference here is that the count data persists only until the next count is carried out.

```
select
  'select',
  '''update monitor_count set mhdb_count=''||count(*)||'|| chr(10) ||
  ''',mhdb_date=sysdate''|| chr(10) ||'|| chr(10) ||
  ''' where owner=''||''''''''||OWNER||''''''''||'|| chr(10) ||
  ''' and table_name=''||''''''''||TABLE_NAME||''''''''||'''||''';'''
|| chr(10) ||
  'from '||owner||'.'||tname||';'
```

```
from tables_to_count;
```

Rather than go through the inner workings of this script, readers are invited to apply the lessons learned from the previous examples, and work out the processing themselves.

Conclusions

This paper has presented what may appear to be a somewhat esoteric aspect of SQL. It may be argued that the problems presented can be solved in a variety of other ways. This is certainly true, but it is not the point - the exercise has been to demonstrate how flexible and powerful SQL*Plus can be. In this case that power is demonstrated when used to nest several quite distinct scripts within one larger whole.

It has been shown how the values retrieved from one query can be transparently passed to another query - all within one owner query operating within SQL*Plus. A variety of SQL commands can be embedded within one such sequence of scripts, including not just ‘select’, update’, delete’ and ‘insert’, but also the full suite of DDL commands as well.

This approach frees one from any dependence on either PL/SQL or any other 3GL language skill, or indeed dependency on any particular compiler being present on the current platform. In short, this technique emphasises the platform independence of Oracle, while retaining both procedural and set-based programming logic.

The author claims that with a thorough and deep understanding of SQL, one may be surprised at just how much can be achieved within that medium alone using techniques such as those described in this paper.