TALES FROM THE CRYPT: GETTING TO KNOW THE ORACLE SUPPLIED PACKAGES

Rune Mørk, Novo Nordisk Engineering A/S

There exists a number of Oracle supplied packages, that developers in some cases only get to know by chance. In this presentation I will focus on how to get information on the many different oracle supplied packages, and show some of the content in some of the packages with examples of usage from the real life. My aim with this article is to get you guys more interested in the supplied packages, so if you leave this session with the idea that you want to go exploring inside the crypt of oracle, I will have fulfilled my goal.

The title for this article comes from the fact that I here will show packages that, I some cases, has been around since '92 but with lack of public interest, now they are pulled into daylight from the crypt where they we lying hidden in such long time.

The packages I will cover are

- Dbms_application_info
- Dbms_describe
- Dbms_utility
- Dbms_session
- Dbms_ddl
- Dbms_job
- Dbms_ profiler

For each package I will mention the most interesting procedure/function within, give them a line of though and illustrate the use of some of them

The source of information

The first problem you got is how to know that these packages exist, hence if you do not know they are there at all how could you look up information about them. I always peek into user_source to investigate what Oracle decided to put in production for a specific release.

When your attention is drawn to a specific package, then you need to get further information on how to use and abuse it. This info you can get from several places, you could use metalink, buy several books, look into the documentation or peek into the Oracle files.

Using metalink will give you, in some cases, a very short explanation, on how to use these packages, reading books will gain you some knowledge, but in most cases they are not covering all details, hence most books focus on covering all package, and therefore only covers the packages lightly, so you are left with the only option, to look into user_source, test and lean.

There exists several tools, you could use to peek into the crypt. Among them are forms, toad and sql navigator. Looking into the source will reveal that in most cases you are only granted the privilege to see the specification not the body itself, since it is wrapped, meaning it is dianacode made into hex causing it to be unreadable.

Dbms application info

This useful package is one of the best kept secrets of the oracle supplied packages, it enables you to monitor your progress of a long running job without using utl_file, or writing to your own temporary table and committing, by simply querying v$session. You can see the package specification in the file $oracle_home\RDBMS\ADMIN\dbmsapin.sql.

Reading this file reveals that this package has been around since 96, and it even existed earlier on with the name dbms_registration since ‘94. Boy I could have used that package at that time, but sadly it first came to my attention in 1998.

Some of the useful procedures in this package are:

```
procedure set_module(module_name varchar2, action_name varchar2);
procedure read_module(module_name out varchar2, action_name out varchar2);
procedure set_client_into (client_into in varchar2);
procedure get_client_into (client_into out varchar2);
```

The way to use these procedures is to embed it into the code where you want monitor your progress.

The procedures write information to the v$session memory table structure, to which you do not have immediate access, therefore ask the dba to grant you select from v$_session (ie the memory held table) and then create a private synonym v$session for sys.v$_session. You cannot create a public synonym here since sys and system already has a v$session synonym.

The columns in v$session that we are using has limited length, see table 1 for info.

Column name	Length
Module	Varchar2(48)
Action	Varchar2(32)
Client_info	Varchar2(64)

Table 1 Changeable columns in v$session using dbms_application_info

An example of how it works can bee seen in figure 1.

```
PROCEDURE long_running(p_num in number) IS
BEGIN
 dbms_application_info.set_module('Long running','has started');
  for i in 1..p_num loop
     -- do something
    -- monitor your progress here as the last line of code
     if mod(i,1000000) = 0 then
         dbms_application_info.set_module('Long running','Now at '||i);
     end if;

  end loop;
END;
```

Figure 1 Long running procedure

When you run procedure long_running, you can monitor progress by querying v$session, like

```
select module, action from v$session
where module is not null;
```

Looking at the code reveals that you do not need to commit to see the result in v$session, because any change in v$session is held directly in memory, which is of enormous benefit.

Dbms describe

This package is another one of the oldies, dating back to January ’92. The only usage of this package is to describe a procedure within a package. Why would anyone use that feature? Well if you do not have a client side tool that could describe a procedure in a package, then you need to maintain the old version of sql*plus (version 3.x), or fall back on selecting from user_source.

This package contains one procedure dbms_describe.describe_procedure, so I've written some code for using that, I've included the code for this in figure 2.

```
PROCEDURE my_describe(p_object in varchar2) IS
    v_overload dbms_describe.number_table;
    v_position dbms_describe.number_table;
    v_level dbms_describe.number_table;
    v_argument_name dbms_describe.varchar2_table;
    v_datatype dbms_describe.number_table;
    v_default_value dbms_describe.number_table;
    v_in_out  dbms_describe.number_table;
    v_length dbms_describe.number_table;
    v_precision dbms_describe.number_table;
    v_scale dbms_describe.number_table;
    v_radix dbms_describe.number_table;
    v_spare dbms_describe.number_table;
    v_datatype_name varchar2(30);
BEGIN
  dbms_describe.describe_procedure (p_object,
    null,         null,
    v_overload,   v_position,
    v_level,      v_argument_name,
    v_datatype,   v_default_value,
    v_in_out,     v_length ,
    v_precision,  v_scale,
    v_radix,      v_spare);

  dbms_output.put_line('Overload  ' || 'Position  ' ||
                       'Argument        ' || 'Level     ' ||
                        rpad('Datatype',20)|| 'Length    ' ||
                       'Prec      ' || 'Scale     ' || 'Rad  ');
  dbms_output.put_line(rpad('-',99,'-'));
  for i in 1..v_overload.count loop
      dbms_output.put( rpad(v_overload(i),10) ||
                       rpad(v_position(i),10)) ;
      if v_argument_name(i) is null then
        dbms_output.put(rpad('Return',15));
      else
        dbms_output.put(rpad(v_argument_name(i),15) );
      end if;
      if v_datatype(i) =  1               then v_datatype_name := 'VARCHAR2';
        elsif v_datatype(i) =  2          then v_datatype_name := 'NUMBER';
        elsif v_datatype(i) = 3           then v_datatype_name := 'NATIVE INTEGER';
        elsif v_datatype(i) = 8           then v_datatype_name := 'LONG';
        elsif v_datatype(i) = 11          then v_datatype_name := 'ROWID';
        elsif v_datatype(i) = 12          then v_datatype_name := 'DATE';
        elsif v_datatype(i) = 23          then v_datatype_name := 'RAW';
        elsif v_datatype(i) = 24          then v_datatype_name := 'LONG RAW';
        elsif v_datatype(i) = 96          then v_datatype_name := 'CHAR (ANSI FIXED)';
        elsif v_datatype(i) = 106         then v_datatype_name := 'MLSLABEL';
        elsif v_datatype(i) = 250         then v_datatype_name := 'PL/SQL RECORD';
        elsif v_datatype(i) = 251         then v_datatype_name := 'PL/SQL TABLE';
        elsif v_datatype(i) = 252   then v_datatype_name :='PL/SQL BOOLEAN';
        else  v_datatype_name := 'UNKNOWN';
      end if;

      dbms_output.put_line( rpad(v_level(i),9)             || ' ' ||
                            rpad(v_datatype_name,20)       || ' ' ||
                            rpad(v_length(i),9)            || ' ' ||
                            rpad(v_precision(i),9)         || ' ' ||
                            rpad(v_scale(i),9)             || ' ' ||
                            rpad(v_radix(i),10) );
  end loop;
END;
```

Figure 2 Code for my_describe

There exist a number of problems wit this package. First and foremost the description on metalink is not correct, and even worse the description in the package spec also contains errors.

As you can see in the code in figure 2, the package returns a number indicating the datatype of the parameters. The meaning of these numbers can be seen in table 2 (which is copied from the metalink paper). When you refer to a plsql/table you will get number 123 as the description and not 251 as the table indicates.

Number	Description
1	VARCHAR2
2	NUMBER
3	NATIVE INTEGER (for PL/SQL's BINARY_INTEGER)
8	LONG
11	ROWID
12	DATE
23	RAW
24	LONG RAW
96	CHAR (ANSI FIXED CHAR)
106	MLSLABEL
250	PL/SQL RECORD
251	PL/SQL TABLE
252	PL/SQL BOOLEAN

Table 2 Interpretation of datatype

```
SQL> execute my_describe('pdpack.PDFORMAT_SAGSNR');
Overload  Position  Argument        Level     Datatype            Length    Prec      Scale     Rad
------------------------------------------------------------------------------------------------
0         0         Return          0         VARCHAR2            0         0         0         0
0         1         P_PROJEKTAAR    0         NUMBER              22        0         0         10
0         2         P_NNE_SAGSNUMME 0         NUMBER              22        0         0         10
0         3         P_SAGSNUMMER_2  0         VARCHAR2            0         0         0         0
```

Figure 3 Result from my_describe

Documentation also states that you should be able to issue a command like my_describe('standard.greatest') but that does not work either.

Dbms utility

Good to know, I've just recently came across it when I had then need to know, inside a plsql-program unit, who actually did invoke the program.

The package consist of many interesting procedure, the first one is used for analyzing all objects belonging to a schema.

```
PROCEDURE analyze_schema(schema VARCHAR2,
                         method VARCHAR2,
                         estimate_rows NUMBER DEFAULT NULL,
                         estimate_percent NUMBER DEFAULT NULL,
                         method_opt VARCHAR2 DEFAULT NULL);
```

There is also one for converting a comma seperated list into a pl/sql table.

```
PROCEDURE comma_to_table(list IN VARCHAR2,
                         tablen OUT BINARY_INTEGER,
                         tab OUT uncl_array);
```

And one for the reverse

```
PROCEDURE table_to_comma(tab IN uncl_array,
                         tablen OUT BINARY_INTEGER,
                         list OUT VARCHAR2);
```

One for compilation of all object in a specific users schema.

```
PROCEDURE compile_schema(schema VARCHAR2);
```

One for formatting the whole error stack into a string.

```
FUNCTION format_error_stack RETURN VARCHAR2
```

Getting the current time with 1/100 of a second precision in a number of the format nnnnnn, where nnnnnn is the number of 1/100th of seconds passed since the last whole hour.

```
FUNCTION get_time RETURN NUMBER;
```

And lastly one for formatting the whole call stack into a string.

```
FUNCTION format_call_stack RETURN VARCHAR2;
```

I've used format call stack in my coding to find the invoker, in this case it is in the context of a web pl/sql program, therefore the strange string handling, of a program, see figure 4.

```
CREATE OR REPLACE PROCEDURE HEGMNMC
 (P_MODULENAME IN OUT VARCHAR2
 ,P_MODULECOMPONENT_NAME IN OUT VARCHAR2
 )
 IS
-- Program Data
V_STRENG VARCHAR2(4000);
V_PLAC PLS_INTEGER;
V_PLAC2 PLS_INTEGER;
V_PLAC3 PLS_INTEGER;

-- PL/SQL Block
BEGIN
  -- the call stack look like
  -- v_streng := 'PL/SQL Call Stack ----- object line object handle number name 62b8ab8 468
  -- package body PYTHIA.WEPACK 5613718 378 package body PYTHIA.HEMODU$MODULE_COMP
  --  5613718 750 package body PYTHIA.HEMODU$MODULE_COMP 5613718 138 package
  --  body PYTHIA.HEMODU$MODULE_COMP 40dcf20 22 package  ';

  v_streng := dbms_utility.format_call_stack;
  --removes new lines
  v_streng := replace(v_streng,chr(10),' ');
  --find the first $
  v_plac := instr(v_streng,'$');
  -- find the fist blankspace after $
  v_plac2 := instr(v_streng,' ',v_plac);
  -- shortens the string so all after the first blankspace after $ is gone
  v_streng := substr(v_streng,1,v_plac2 - 1);
  -- find the first .
  v_plac3 :=  instr(v_streng,'.', -1);
  -- extract the modulename and the modulecomponent name
  p_modulename := substr(v_streng, v_plac3 + 1, v_plac - v_plac3 - 1);
  p_modulecomponent_name := substr(v_streng, v_plac + 1 );
END;
```

Figure 4 Who invoked me?

After the invocation of hegmnnc illustrated in figure 4 we know who the invoker was, as you can see in our case we need a little bit of codeing exercise to extract who the invoker is, so if you decide to use format_call_stack it is a good idea to see the content first of the stack, in order to substring you to the right information you need.

Dbms session

One of the larger packages, it contains a large number of function of which the most interesting ones are:

```
Procedure set_role (role_cmd varchar2);
```

Enabling a specified role equivalent to the sql command set_role, you can use it as seen in figure 5.

```
PROCEDURE aktivate_Write(p_role in varchar2, p_pwd in varchar2) IS
    BEGIN
      sys.dbms_session.set_role(p_role||' identified by '||p_pwd);
 END;
```

Figure 5 Activate a role

Another function is the same area is investigating if a role is enabled.

```
Function is_role_enabled(rolename varchar2) returns boolean;
```

Resetting a package, ie setting package variable to its initial values and closes package cursors.

```
Procedure reset_package;
```

Returning the session id for the current session.

```
Function unique_session_id ;
```

Freeing unused memory after large transactions.

```
Procedure free_unused_user_memory;
```

It is difficult to come show some good examples for this package, since most of the package functions replace sql-commands, but in the next chapter about dbms_ddl I've used one of them.

Dbms ddl

Very useful package.

In an environment that often changes it is bound that objects gets invalidated, or that we often need to compute statistics for objects in order to get the costbased optimizer to work properly, this is what this package is ment for.

The interesting part of this package is the following 2 procedures:

```
Procedure alter_compile( type varchar2, schema varchar2, name varchar2);

procedure analyze_object
   (type varchar2, schema varchar2,
   name varchar2,
   method varchar2,
   estimate_rows number default null,
   estimate_percent number default null,
   method_opt varchar2 default null,
   partname varchar2 default null);
```

There exist 2 purposes for using these, i've written the code for, first one is a batch recompiler of all invalid objects owned by a specific user, the code for this can bee seen in figure 6

```
PROCEDURE batch_compile IS
cursor sel_obj is
select object_name, object_type, rownum
from USER_objects
where status = 'INVALID'
and object_type in ('TRIGGER','PROCEDURE','FUNCTION','PACKAGE','PACKAGE BODY', 'VIEW')
order by decode(object_type,'VIEW',1,'PACKAGE',2,3);
v_number pls_integer;
v_run pls_integer := 0;
BEGIN
 select count(*)
 into v_number
 from USER_objects
 where status = 'INVALID'
   and object_type in ('TRIGGER','PROCEDURE','FUNCTION','PACKAGE','PACKAGE BODY', 'VIEW');

 FOR my_obj IN sel_obj LOOP
         v_run := v_run + 1;
    dbms_application_info.set_module('Batch compile '||v_number||' objects' ,
     'Compiling('||v_run ||') '|| my_obj.object_name);
     if my_obj.object_type = 'VIEW' then
          execute immediate('Alter View '||my_obj.object_name||' compile');
     else

      dbms_ddl.alter_compile(my_obj.object_type, USER, my_obj.object_name);
     end if;
 END LOOP;
 dbms_session.free_unused_user_memory;
END;
```

Figure 6 Batch recompilation

A funny point in this, as you can see in the code, is that i've used user_object to select from, in my testing I also tried to use dba and all_objects, and strange enough, all_object did not at all behave the same when used in the pl/sql engine, as it did in sql, here is room for a more through investigation.

You might wonder if it is possible to create a batch compiler of all invalid objects, but sadly I havn't found a way through this yet. You could grant all on all your objects to one schema, and let this schema the DBA privilege, or at least the alter any procedure privilege plus access to dba_objects. But I recon that this is not an archichecture that is desireable.

The other usage of the package is a batch analyzer of objects, for which I included the code in figure 7.

```
PROCEDURE batch_analyze IS
cursor sel_a is
select object_name, object_type
from user_objects
where object_type in ('TABLE','INDEX')
order by object_type desc, object_name;
BEGIN

 FOR a_rec IN sel_a LOOP
     dbms_ddl.analyze_object(a_rec.object_type,
                    USER, a_rec.object_name,
                        'ESTIMATE',        NULL, 10, NULL, NULL);
    dbms_application_info.set_module('Batch analysing','Analyzing '|| a_rec.object_name);
 END LOOP;
 dbms_session.free_unused_user_memory;
END;
```

Figure 7 Batch analyzing

As you can see I've used dbms_session.free_unused_user_memory after the analyzing and the batch compilation.

Dbms job

This package might bee useful to you, the purpose of it is to manage batch processing of jobs. It is used for setting jobs in queue and to monitor the progress of such jobs.

The package has been around for a while, but is not very widely used in all communities, since most operating systems offers a service similar to the one offered by dbms_job, and my experience with dba's has revealed that they tent to let the operating system handle such batch jobs instead of letting the oracle kernel.

Oracle itself uses for instance this package for updating materialized views and handling preaggregated summaries is discoverer.

The package consist of many procedures and function, the most interesting ones are:

Submitting a job for a queue, where job is an identifier, what is the job, next_date is the date to perform the job (default sysdate), interval is the interval for the next job, here it is ment to be an expression that could evaluate to a date expression(default null), parse is a boolean indicating if a job should be passed at submit time or passed at execution time (default false).

```
DBMS_JOB.SUBMIT(job OUT BINARY_INTEGER,
                what IN VARCHAR2,
                next_date IN DATE,
                interval IN VARCHAR2,
                no_parse IN BOOLEAN)
```

Removes a job from the queue.

```
DBMS_JOB.REMOVE(job  IN  BINARY_INTEGER);
```

Changes parameters to an already scheduled job , if you want to alter just a single change of parameters can be done by separate procedure not revealed here.

```
DBMS_JOB.CHANGE(job IN BINARY_INTEGER,
                what IN VARCHAR2,
                next_date IN DATE,
                interval IN VARCHAR2);
```

Used for marking a job as broken or unbroken.

```
DBMS_JOB.BROKEN(job IN BINARY_INTEGER,
                broken IN BOOLEAN,
                next_date IN DATE);
```

Forces a job to be run now.

```
DBMS_JOB.RUN(job IN BINARY_INTEGER);
```

In order of getting dbms_job to work properly, there are 2 init.ora parameters you might need to consider.

The first one is JOB_QUEUE_PROCESSES, which indicates how many processes to start. If set to zero, no jobs are executed, and default is 0, range is 0..10. (At least that what documentation says, reality is at default is 4)

The other one is JOB_QUEUE_INTERVAL, which is deciding how long an interval the process will sleep before checking for a new job, default is 60 sec, range is 1..3600 sec. (And here, default is 10).

I've included a small demo of how it works. Imagine I want to run a procedure every minute and thereafter every second minute, I would use a command like:

```
dbms_job.submit(v_job, 'xxx;', sysdate +120/(24*60*60),'SYSDATE + 60/(24*60*60)');
```

Now if I want to inspect when I actually is going to run the job I could investigate the user_job table, with a query like:

```
select job, to_char(next_date,'ddmmyyyy hh24:MI:SS') next,
```

```
to_char(SYSDATE,'ddmmyyyy hh24:MI:SS') now, what
from user_jobs
```

That would give me a result like:

```
       JOB NEXT              NOW               WHAT
--------- ----------------- ----------------- --------------------------------------------------
       29 19042002 18:27:08 19042002 18:26:04 xxx;
```

Now if I want to alter anything about this job, I would use one of the above-mentioned functions, I my case I would like to remove the job so I issue:

```
exec dbms_job.remove(29);
```

So that was a brief intro to dbms_job, and how to use it.

Dbms profiler

Officially this package has been around since 8i, but never the less it existed already for 8.0.4, but, off course, it wasn't documented, neither did the script work properly and had to bee modified by hand to get it installed, hence this package is not installed by a standard installation of the RDBMS.

This neatly package is used to monitor the usage and timing of plsql packages/functions/procedures. For a very thorough description of the package see my article for last years conference.

To install this package you need to run the script Profload.sql in the RDBMS- catalog of your Oracle_home.

Introducing dbms_profiler

The package DBMS_PROFILER can be used to collect information about your plsql program units and how well or poor they perform, the package is not default installed in your database, to get it to work you need to install both the profiler tables and the profiler package to the SYS account.

Installing the profiler tables

To install the profiler tables, sequences and grant access to them you need to run the script ORACLE_HOME\rdbms\admin\proftab.sql.

Plsql_profiler_runs, that contains information about the different profiler runs that has been run.

Plsql_profiler_units, that contains information about witch programunits that has been executed in a specific profile run.

Plsql_profiler_data, that contains information about which codelines of plsql code that has been used and statistical information about the execution of these (number of executions, total number of runs, min execution time and max execution time).

After installation remember to grant all on them and create public synonyms.

Installing the profiler packages

To install the DBMS_PROFILER-package you need to run the script ORACLE_HOME/rdbms/admin/profload.sql.

This package contains a number of procedures and functions. In this article I will only present those that is necessary to make profiling work:

The first function is used to start the profiling, when is has been executed statistical data is beeing colleted for all plsql program units executed in the current session, until you explicit pause or stop the profiling.

```
function start_profiler(run_comment IN varchar2 := sysdate,
                        run_comment1 IN varchar2 := '')
                        return binary_integer;
```

To stop profiling you need to know the function:

```
function stop_profiler return binary_integer;
```

Both function return a binary integer that is an errorcode, if you choose to investigate the result of the errorcode any values different from 0 represents an error, se DBMS_PROFILER documentation for further information.

Normal profiling

If you already have an idea about what plsql program that is causing problems for your application, you could use the profiler as intended. To do so you need to perform 4 steps

1. Starting the profiling
2. Doing profiling
3. Stop profiling
4. See the results

Step 1 Starting the profiling

In order to start the profiling you need to tell the profiling utility to start collecting data, which could be done by issuing:

```
declare
  err number;
begin
  err := dbms_profiler.start_profiler('&1');
  dbms_output.put_line(err);
end;
```

in SQL*PLUS

Step 2 Doing profiling

I've created 2 sample pl/sql programs in order to demonstrate the profiling they look like the following:

```
CREATE OR REPLACE procedure give_all_raise is
cursor sel_dept is
select deptno
from dept
order by deptno;
begin
  for i in 1 .. 2000 loop
    for r in sel_dept loop
      give_raise(r.deptno,i/1000);
    end loop;
  end loop;
  end;

procedure       give_raise (
```

```
    p_deptno in number,
    p_raise_percent in number )
as
begin
   update emp set sal = sal + (sal * p_raise_percent * .01)
   where deptno = p_deptno;
   commit;
end give_raise;
```

These programs are really nonsense, but in order to be able to demonstrate then ...

So now I execute the procedure give_all_raise, the execution will be slightly slower in order to collect the statistics.

Step 3 Stop profiling

After the program give_all_raise has been executed then I need to stop the profiling. This can be done by issuing the following:

```
declare
  err number;
begin
  err := dbms_profiler.stop_profiler;
end;
```

again in SQL*PLUS.

Step 4 Viewing the result.

Now it is fairly easy to investigate the profiling result, by joining the profiler tables with user_source, you can get a accurate picture of what the pl/sql program unit actually did spent its time on. The select statement looks like:

```
SELECT  SUBSTR(PPU.UNIT_NAME,1,10) UNAME,
        PPD.TOTAL_OCCUR,
        PPD.TOTAL_TIME,
        PPD.MIN_TIME,
        PPD.MAX_TIME,
        US.TEXT
 FROM PLSQL_PROFILER_DATA PPD,
      PLSQL_PROFILER_RUNS PPR,
      PLSQL_PROFILER_UNITS PPU,
      USER_SOURCE US
 WHERE PPU.RUNID       = PPR.RUNID
   AND PPD.UNIT_NUMBER = PPU.UNIT_NUMBER
   AND PPD.RUNID       = PPU.RUNID
   AND US.NAME         = PPU.UNIT_NAME
   AND US.LINE         = PPD.LINE#
   AND US.TYPE         = PPU.UNIT_TYPE
   AND PPU.RUNID = &1
 ORDER BY PPU.UNIT_NAME, PPD.LINE#
```

After executing this you would get a result like the one in figure 10.

```
UNAME      TOTAL_OCCUR TOTAL_TIME  MIN_TIME  MAX_TIME TEXT
---------- ----------- ---------- --------- --------- ------------------------------------------
GIVE_ALL_R        2000  2.677E+10  11414858 356021784 select deptno
GIVE_ALL_R        2001  1.153E+09    404520  24447241   for i in 1 .. 2000 loop
GIVE_ALL_R       12000  1.835E+11    300596 1.819E+09     for r in sel_dept loop
GIVE_ALL_R       20001  3.413E+10     72355 196875580       give_raise(r.deptno,i/1000);
```

```
GIVE_RAISE        8000  1.518E+12  35737502 7.828E+10    update emp set sal = sal + (sal *
p_raise_percent *
GIVE_RAISE        8000  1.849E+11   5190045 2.019E+10    commit;
```

Figure 10 Profiling result

Now these 4 steps needs to be repeated every time you find the need to do profiling, I've found it worth while to invest a little time in order to make this much easier.

Conclusion

My aim with this article was to take you on a journey into the Oracle crypt, letting you have a peek into what interesting stuff lays buried there.

During the preparation for this article I took a closer look on STANDARD package, and came across the clip in figure 11

```
function 'IS NOT NULL'(b BOOLEAN) return BOOLEAN is
  begin
    return (NOT b IS NULL);
  end 'IS NOT NULL';
```

Figure 11 Function is not null

That gave me the idea to create a function like the one in figure 12

```
create or replace function "Wery strange" return varchar2 is
begin
  return 'Wery strange ideed';
end;
```

Figure 12 The strange function

A even more strange thing about this function is that it is case sensitive and I need to use double quotes where oracle seem like to get along with single quotes.

So you see there is plenty to investigate in the Oracle crypt, the list for my next expedition involve packages like

1. DBMS_SPACE (Space management)
2. STANDARD (The standard package)
3. UTL_INADDR (Gives opportunity to find ipadress of local host)
4. OUTLN_PKG (Drop outlines)
5. DBMS_RLS (Row level security)

Thank you wery much, you can contact me on runm@nne.dk.