PL/SQL 101 – DBMS_OUTPUT

Author: BluShadow
Last Updated: 6th January 2017

CONTENTS

1. Introduction

It's been a while since my last PL/SQL 101 article, so this time I've picked something that has come about by popular request (read that as "people are still making the same silly assumptions"). This article is about the DBMS_OUTPUT package which, whilst generally understood by the experts in the community, still frequently comes to light as something that many of the newer developers just don't understand.
For reference you can always look at the documentation, but for some further light-hearted examples, please continue to read on.
Documentation can be found at: http://docs.oracle.com/database/122/ARPLS/DBMS_OUTPUT.htm#ARPLS036

For this article, I'm not going to delve in to the internals of the database or underlying traces of the API's being called. I'll try and keep it to what we see as a user of the database.

2. When Is DBMS_OUTPUT Not An Output?

The simple answer is… **all the time**.

Obviously this may seem a confusing answer, as the name of the package suggests that is can "output" things. So, in our context, what do we mean by "output"?

"Output", in this context, is a term we use when we expect something to be presented to us on the screen. Yes, you can use the term to refer to outputting to a printer, or outputting to a file, but for this package, the common expectation is that the output is going to appear on our screen.

So, let's prove that DBMS_OUTPUT doesn't output anything, by issuing an output statement in SQL*Plus…

```
SQL> exec dbms_output.put_line('I do not output anything');

PL/SQL procedure successfully completed.
```

Now, those of you who are like me (and I know I am), will be saying to yourself "ah! But you haven't turned on the output".

But, hang on, if I write a query…

```
SQL> select * from dual;

D
-
X
```

SQL*Plus is quite happy to "output" the results of that query.
Yet, to get the results of my dbms_output call, I have to explicitly tell SQL*Plus to turn on output…

```
SQL> set serveroutput on
SQL> exec dbms_output.put_line('I do appear to output something');
I do appear to output something

PL/SQL procedure successfully completed.
```

What's happening here? Why do I need to explicitly tell SQL*Plus that I want to see the "output" of the dbms_output package when I don't have to for queries? Let's explore this further…

3. LOST IN SPACE

No, I'm not talking about the T.V. series, I'm talking about where our "output" goes to when we issue a DBMS_OUTPUT.PUT_LINE statement. Let's look at the following code example:

```
SQL> set serveroutput on
SQL> declare
  2    opText varchar2(2000);
  3    status number;
  4  begin
  5    dbms_output.put_line('Help me, I do not know where I am');
  6    dbms_output.get_line(opText, status);
  7    write_to_file(txt => opText);
  8  end;
  9  /

PL/SQL procedure successfully completed.
```

Now hang on just a minute! I had my output turned on, but my output hasn't appeared.
Let's look at line 6 and 7 in that code.

For those who've perused the documentation for the DBMS_OUTPUT package, you may have noticed that it doesn't just have "put" calls, but it also has "get" calls in there. [*Now trust me on this, those "get" calls are NOT there to try and get input from a user via their keyboard, there's no way the database server, where the code is being executed, is going to hack across the network, break in to your client computer's operating system, and read the keyboard input*.]

So, it appears that line 6 is getting the "output" and line 7 is writing that output to a file (for brevity's sake I've left out the code that actually writes data to a file – it's just a simple UTL_FILE package call to open and write text to a default file).

Let's go and have a look at the file:

```
Z:\>type test.txt
Help me, I do not know where I am
```

How about another example:

```
SQL> set serveroutput on
SQL> declare
  2    opText varchar2(2000);
  3    status number;
  4  begin
  5    dbms_output.put_line('I want out first!');
  6    dbms_output.put_line('Ok, me second!');
  7    dbms_output.get_line(opText, status);
  8    write_to_file(txt => opText);
  9  end;
 10  /
Ok, me second!

PL/SQL procedure successfully completed.
```

Now that IS weird. The first "output" hasn't been output, but the second one has.

```
Z:\>type test.txt
I want out first!
```

… but, there's the first one written to the file.

By now, hopefully you've had a lightbulb moment, and realised that when we "put" something using dbms_output, that something isn't being "put" to the screen at all, it's going somewhere else. It's not quite Lost-in-Space, but it's a space on the database server that's referred to as the "DBMS_OUTPUT buffer".

The code above demonstrates that, whilst the data remains in the buffer, we can use code to "get" that data back out again, so, in that last example, we put two lines in to the buffer, and got the first of those lines back out (which then got written to a file), just leaving the second line in the buffer, which was displayed by SQL*Plus.

4. So, What Is SQL*Plus Actually Doing?

When we issue the following command in SQL*Plus:

```
SQL> set serveroutput on
SQL>
```

What we are doing is instructing SQL*Plus to read the DBMS_OUTPUT buffer and display it within its own client interface. Other client tools such as SQL*Developer, TOAD, PL/SQL Developer etc. also have mechanism's for "enabling" dbms_output – see their respective documentation/help files. In the examples here, I'm sticking with SQL*Plus as the basic client interface available to most of us, but essentially all client tools are handling the dbms_output buffer in the same way.

When we do the following…

```
SQL> set serveroutput on
SQL> exec dbms_output.put_line('Show Me!');
Show Me!

PL/SQL procedure successfully completed.
```

… what happens appears to be something like …

- Our code is sent from SQL*Plus down to the database server
- The database server executes the code and puts the "output" into the dbms_output buffer
- Upon completion of execution, control is returned to SQL*Plus
- SQL*Plus takes the contents of the dbms_output buffer and displays those to its client interface (your creen)

That's generally the principle of it, though let's take a deeper look.

5. How Long Is the Data is the DBMS_OUTPUT buffer for?

Whatever you put in the buffer exists only for that session, but still, how long is it there for?
Let us test this with some simple examples:

Example 1

```
SQL> set serveroutput off
SQL> exec dbms_output.put_line('Will you ever know me?');

PL/SQL procedure successfully completed.

SQL> set serveroutput on
SQL>
```

In this example, we've told SQL*Plus to turn off the displaying of the dbms_output buffer, executed the statement, and then told SQL*Plus to turn on the displaying of the dbms_output buffer. We didn't get the contents of the buffer, so what does that tell us?
Either:

a) The "set serveroutput on" statement only retrieves "output" put in the buffer AFTER it's been issued, or
b) The "output" was cleared from the buffer after the put_line code block completed, or
c) The "output" of the put_line code block was never put in the buffer, or
d) The "set serveroutput on" statement itself doesn't retrieve the contents of the dbms_output buffer; it's just a switch to tell SQL*Plus to retrieve the contents after a statement has been issued to the database.

To answer this we'll continue with our examples…

EXAMPLE 2

```
SQL> set serveroutput on
SQL> declare
  2    opText varchar2(2000);
  3    status number;
  4  begin
  5    dbms_output.put_line('Lest we forget');
  6    dbms_output.get_line(opText, status);
  7    write_to_file(txt => opText);
  8  end;
  9  /

PL/SQL procedure successfully completed.



Z:\>type test.txt
Lest we forget
```

Ok, so this example we've seen before. The code has put something in to the buffer, read it back out of the buffer and written it to a file, and then SQL*Plus has not had anything in the buffer to display. Let's just try the same thing, but with the server output turned off…

EXAMPLE 3

```
SQL> set serveroutput off
SQL> declare
  2    opText varchar2(2000);
  3    status number;
  4  begin
  5    dbms_output.put_line('Lest we forget');
  6    dbms_output.get_line(opText, status);
  7    write_to_file(txt => opText);
  8  end;
  9  /

PL/SQL procedure successfully completed.


Z:\>type test.txt
```

Now that's odd. The "set serveroutput off" statement is an SQL*Plus command, yet our code executes on the database server, and it appears that the put_line statement didn't actually put any data in the buffer, otherwise our get_line would have read it and it would have got written to our file, but our file is empty!

What that tells us is that there is more to the SQL*Plus "set serveroutput" command than just telling SQL*Plus to display the contents of the buffer. It's actually instructing the database whether to allow output to be put in the buffer in the first place. Can we prove it somehow?...

EXAMPLE 4

```
SQL> set serveroutput off
SQL> declare
  2    opText varchar2(2000);
  3    status number;
  4  begin
  5    dbms_output.enable();
  6    dbms_output.put_line('Lest we forget');
  7    dbms_output.get_line(opText, status);
  8    write_to_file(txt => opText);
  9  end;
 10  /

PL/SQL procedure successfully completed.



Z:\>type test.txt
Lest we forget
```

That just about proves it. Even though SQL*Plus has turned off the dbms_output, by including an "enable" call in our actual code, we have enabled the buffer so our code can "put" to it and "get" from it, and our data ends up in our file.

Does our "enable" statement need to be in the same bit of code?

EXAMPLE 5

```
SQL> set serveroutput off
SQL> exec dbms_output.enable();

PL/SQL procedure successfully completed.

SQL> declare
  2    opText varchar2(2000);
  3    status number;
  4  begin
  5    dbms_output.put_line('Can I please span a call');
  6    dbms_output.get_line(opText, status);
  7    write_to_file(txt => opText);
  8  end;
  9  /

PL/SQL procedure successfully completed.



Z:\>type test.txt
Can I please span a call
```

No, we can enable the dbms_output buffer with a separate call before we call our code.
So is the SQL*Plus "set serveroutput" call just a synonym for executing the dbms_output.enable and dbms_output.disable procedures?

EXAMPLE 6

```
SQL> exec dbms_output.enable();

PL/SQL procedure successfully completed.

SQL> set serveroutput off
SQL> declare
  2    opText varchar2(2000);
  3    status number;
  4  begin
  5    dbms_output.put_line('Can I please span a call');
  6    dbms_output.get_line(opText, status);
  7    write_to_file(txt => opText);
  8  end;
  9  /

PL/SQL procedure successfully completed.


Z:\>type test.txt
```

It would appear to be the case as, if we enable the output through code first and the use the SQL*Plus command to turn it off, our code can no longer use the buffer to "put" and "get", and our file is empty.

But let's just carry out one more test to be sure…

EXAMPLE 7

```
SQL> set serveroutput off
SQL> exec dbms_output.enable();

PL/SQL procedure successfully completed.

SQL> declare
  2    opText varchar2(2000);
  3    status number;
  4  begin
  5    dbms_output.put_line('Show me to the world!');
  6  end;
  7  /

PL/SQL procedure successfully completed.
```

In this example we've issued the SQL*Plus command to turn off server output, but then enabled the dbms_output buffer. In Example 5, we did the same thing, but after putting data in to the buffer we got it back out and wrote it to our file. In this example, we're leaving the data in the buffer and letting the code execution complete. In this case, SQL*Plus has not displayed the contents of the buffer, even though we know from Example 5 that the buffer was enabled within the code.

This therefore leads us to the conclusion that the SQL*Plus "set serveroutput" call has the following two properties:

- It indicates to the database server whether the dbms_output buffer should be enabled or disabled, as per a call to the dbms_output.enable/disable procedures.
- It indicates to SQL*Plus to retrieve the contents of the buffer and display it when any code completes execution.

So, it's does a little bit more than just enabling and disabling the dbms_output buffer on the server.

EXAMPLE 8 – THE FINAL TWIST IN THE TALE?

And just when you thought you finally understood the nuances of it all, what actually did happen to the "output" we put in that dbms_output buffer on the last example? Have we finally proved it gets "lost-in-space"?

```
SQL> set serveroutput on
SQL> exec null;
Show me to the world!

PL/SQL procedure successfully completed.
```

Nope, it's still there, waiting to be retrieved. All we needed to do was to use the second property of the SQL*Plus "set serveroutput on" statement, which tells SQL*Plus we want to retrieve the contents of the buffer after executing any statement, and then execute any statement (in this case I executed a null block of code, but it could have been a query or any other code). The buffer was retrieved and displayed.

As long as our database session remains connected and the dbms_output buffer remains enabled, we can get the contents of it… with just one ultimate condition…

EXAMPLE 9 – LOSING THE PLOT

If you read the documentation for DBMS_OUTPUT.get_line or get_lines, you'll find a small statement made in the Usage Notes:

> ***After calling GET_LINE or GET_LINES, any lines not retrieved before the next call to PUT, PUT_LINE, or NEW_LINE are discarded to avoid confusing them with the next message.***

So, what is this saying?

It's telling us that calls that put data into the dbms_output buffer are considered as creating an individual "message", even if that is multiple calls/lines that are put in to it. When you call get_line or get_lines, the package considers that you are now reading that "message", so whether you read all of it or just part of it, it's considered as "read", and then if you subsequently try and put more data in to the dbms_output buffer, the existing "message" is discarded, and a new "message" is started. Let's see that in action:

```
SQL> set serveroutput on
SQL> declare
  2    opText varchar2(2000);
  3    status number;
  4  begin
  5    dbms_output.put_line('I am the first line in Message 1');
  6    dbms_output.put_line('I am the second line in Message 1');
  7    -- Get a line from the Message 1
  8    dbms_output.get_line(opText, status);
  9    write_to_file(txt => opText);
 10    -- At this point there is still the second line in the buffer
 11    -- But, let us put another line in the buffer
 12    dbms_output.put_line('I am the first line in Message 2');
 13  end;
 14  /
I am the first line in Message 2

PL/SQL procedure successfully completed.


Z:\>type test.txt
I am the first line in Message 1
```

This shows us that even though the second line of our first message was still in the buffer, it was discarded the moment we issued the put_line statement for the second message, simply because we had used a get_line call, essentially indicating that we had completed any creation of our first message.

This is something to bear in mind if you are writing your own code to read from the dbms_output buffer. If you do write your own code, I would suggest using the get_lines (plural) call to read all the lines from the buffer and then process all of those, unless you have a very specific reason for just reading the first N lines of a message.

6. WHAT IS SQL*PLUS ACTUALLY DOING NOW?

We need to revise our earlier list, to really reflect what SQL*Plus is doing, based on our observations from the previous examples. So, when we issue "set serveroutput on" the following occurs…

- **"set serveroutput on" indicates to the server to enable the dbms_output buffer, and indicates to the SQL*Plus client to retrieve the buffer upon completion of any executed statements.**
- Our code is sent from SQL*Plus down to the database server
- The database server executes the code and puts the "output" into the dbms_output buffer
- Upon completion of execution, control is returned to SQL*Plus
- **SQL*Plus retrieves the dbms_output buffer**
- SQL*Plus takes the contents of the dbms_output buffer and displays those to its client interface (your screen)

7. Taking It Further, DBMS_OUTPUT For Debugging?

If we read the documentation for DBMS_OUTPUT we can see it says

> ***The package is typically used for debugging***

But just how good is it for this purpose?

Let's simulate some code that takes a while to run, and periodically writes content in to our dbms_output buffer. When we execute this code we'll ensure we have the SQL*Plus serveroutput turned on so we retrieve the contents of the buffer.

```
SQL> set serveroutput on
SQL> begin
  2    for i in 1 .. 3
  3    loop
  4      dbms_output.put_line('['||i||'] '||to_char(sysdate,'HH24:MI:SS'));
  5      dbms_lock.sleep(3);
  6    end loop;
  7  end;
  8  /
[1] 14:13:20
[2] 14:13:23
[3] 14:13:26

PL/SQL procedure successfully completed.
```

Now, I can't demonstrate this in a static document (you can try the same code yourself and see what happens), but essentially the 3 lines of "output" are only displayed to the client screen, once the code has completed execution. You may have been expecting that you would see "output" being displayed whilst the code was executing, but that's not the case.

If you look at the steps in section 6, you'll see that SQL*Plus retrieves the contents of the buffer only when the execution of the code has completed.

This isn't just a feature of SQL*Plus, this is the same regardless of which client tool you choose to use, and relates to the way your session communicates through SQL*Net with the database, and the way the database server supplies the buffer back to the client through that session connection.

So, if you're just wanting to see "how far did my code get" after the event, sure you can use DBMS_OUTPUT for debugging.

8. But I Want To Trace My Code In Real-Time!

If that's what you want, then DBMS_OUTPUT is not really the way to achieve your requirement. However, here's a couple of options for you…

1) Write your own version of the DBMS_OUTPUT package for your own schema, which will take precedence over the system owned one. You can then have your package put the contents that are "put" directly in to a file on the server, or output via whatever means you choose to get it to you in real-time. However, if you go down this route, ensure you include functionality to replicate ALL the functions/procedures of the original dbms_output package, otherwise some code may break, including SQL*Plus's "set serveroutput" command e.g.

```
SQL> set serveroutput on
ERROR:
ORA-06550: line 1, column 19:
PLS-00302: component 'ENABLE' must be declared
ORA-06550: line 1, column 7:
PL/SQL: Statement ignored


SQL> set serveroutput off
ERROR:
ORA-06550: line 1, column 19:
PLS-00302: component 'DISABLE' must be declared
ORA-06550: line 1, column 7:
PL/SQL: Statement ignored
```

(*NB. This further shows that "set serveroutput" is issuing enable/disable calls to the database as discussed*)

This something that I would NOT recommend doing, unless you are really competent in understanding the functionality you need to implement. It is generally a bad idea to override system packages, as it may lead to security issues and other functionality may break. As such you should go with the second option…

2) The other option is to not rely on DBMS_OUTPUT, but to write your own 'trace' package. A very simple example would be something like:

```
create sequence traceSeq
/
create table myCodeTrace (
  pk   number,
  txt  varchar2(4000),
  dt   date
)
/
create or replace trigger trgMyCodeTrace
before insert on myCodeTrace
for each row
begin
  :new.pk := traceSeq.nextval;
  :new.dt := sysdate;
end;
/
create or replace procedure codeTrace(txt in varchar2) as
  pragma autonomous_transaction;
begin
  insert into mycodetrace(txt) values(codeTrace.txt);
  commit;
end;
/
```

Wherever you want to trace your code you call codeTrace with your text and it will be inserted in to the myCodeTrace table, which you can monitor from any other session. The autonomous transaction ensures that it doesn't rely or interfere with any transactions in your main code, so even rolling back a transaction will not

remove the trace you've made. Of course, this example can be greatly expanded upon, to allow tracing to be turned on or off at a session level, or to allow different granularities of tracing, or for tracing specific sections of code, for debugging purposes. The possibilities are all open to you; just use your imagination. You can even have it write to a table, to a file, and optionally put the results in the dbms_output buffer.

SUMMARY

In summary then, what we've seen is that DBMS_OUTPUT doesn't "output" anything, it just allows us to "put" things in to a buffer on the server (specific to the connected session) and "get" things back out of it if we desire. Client tools have the ability to enable and disable that buffer, and to retrieve it and display the contents, but they can only retrieve it following execution of a statement, and not during execution, meaning that DBMS_OUTPUT is ok for basic debugging purposes, but if you really want to be able to trace execution of your code in (near) real-time, then using DBMS_OUTPUT isn't the right tool for the job – but there are alternative methods to achieve that.

With certainty, the one thing I will say about DBMS_OUTPUT is, if you haven't already figured it out…

Don't rely on DBMS_OUTPUT to produce "output" – It is NOT an appropriate means of outputting data in a production system.

If you're not familiar with the DBMS_OUTPUT package and the functionality it provides, take some time to read through the documentation and have a practice to see for yourself how it works. As you've seen from this document, testing simple examples is not difficult.

http://docs.oracle.com/database/122/ARPLS/DBMS_OUTPUT.htm#ARPLS036