WHAT DOES MY PL/SQL PROGRAM ACTUALLY DO?

Rune Mørk, Novo Nordisk Engineering A/S

Oracle offers a database package DBMS_PROFILER that can be used for a number of purposes.

In dealing with 3rd party products, it is often used for investigating what the pl/sql program actually does, i.e. reporting on what statements has been executed, but it can also be used for identifying bottlenecks in your pl/sql code .

In this article I will cover the following topics you need to master in order to trace what is happening in a session:

- Introducing dbms_profiler
- Simple use of dbms_profiler
- Using it for identifying a bottleneck.
- Using it for tracing of a pl/sql program.

Along this I will introduce a homegrown tools used for analysing the profiling results, this tool has been created with Oracle Discoverer.

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, furthermore this package is not installed by a standard installation of the RDBMS, so you got to do it by yourself.

This neatly package is used to monitor the usage and timing of pl/sql packages/functions/procedures ant triggers.

Introducing dbms_profiler

The package DBMS_PROFILER can be used to collect information about your pl/sql 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 profiler package to the SYS schema, and the profiler tables and the profiler package to a user schema.

Installing the profiler tables

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

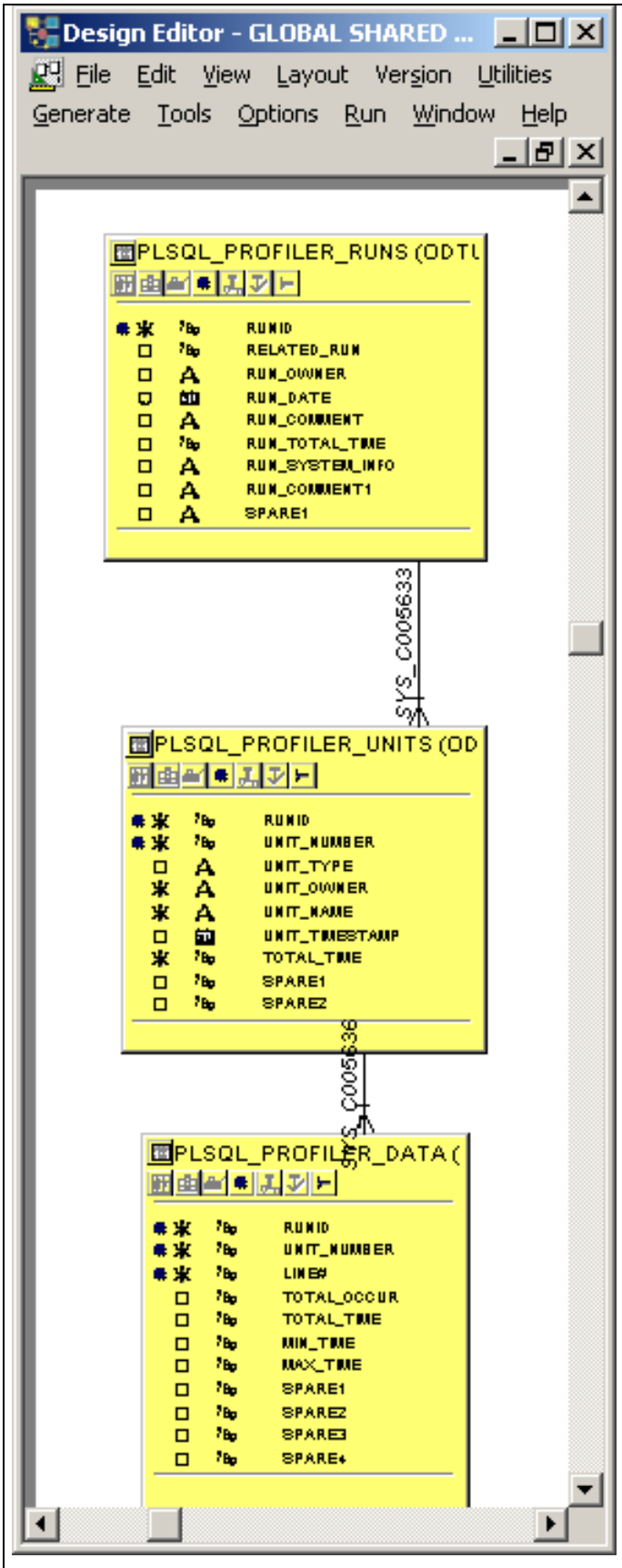
This script installs the following 3 tables:

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

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

Plsql_profiler_data, contains information about which codelines of pl/sql code that has been used, and statistical information about the execution of these.

In figure 1 you can see a diagram of these tables along with a brief description of their columns.



Plsql_profiler_runs contains information about the runs that has been performed. The columns in the table is as follows:

Column	Description
Runid	A unique identifier for the run
Related_run	Purpose is unknown, claim to runid of a related run, but I've never seen it used.
Run_owner	User who started the run
Run_date	Timestamp of the date of the run
Run_Comment	User provided text
Run_total_time	Elapsed time for this run

The rest of the columns are reserved for future use.

Plsql_profiler_units, contains information about each library unit in a specific run. The columns in the table is as follows:

Column	Description
Runid	A unique identifier for the run
Unit_number	Internally generated number for use in the primary key.
Unit_type	The type of the unit.
Unit_owner	Library unit owner name
Unit_timestamp	Timestamp of the unit, i.e. the time the unit was last compiled.
Total_time	Elapsed time within this unit, default to 0 must be calculated after profiling

The rest of the columns are reserved for future use.

Plsql_profiler_data, contains information about each line of code in a specific run. The columns in the table is as follows:

Column	Description
Runid	A unique identifier for the run
Unit_number	Internally generated number for use in the primary key.
Line#	The line# from all_source
Total_occur	Number of times a line was executed.
Total_time	Total time spent executing this line.
Min_time	Minimum execution time for this line.
Max_time	Maximum execution time for this line.

The rest of the columns are reserved for future use.

Figure 1 PL/SQL profiler tables

After installation remember to grant all on these tables to public and create public synonyms. If you omit of forget this point, the next step in the installation will not work.

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, that are useful when profiling pl/sql-code .

The first function is used to start the profiling, when is has been executed statistical data is being colleted for all pl/sql 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 := '',
                        run_number   OUT BINARY_INTEGER)
                        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 error code, if you choose to investigate the result of the error code any values different from 0 represents an error, se DBMS_PROFILER documentation for further information.

The package contains several overlaying versions (both procedures and functions) of start and stop profiler, included I guess for you to choose the versions that suits you.

The package also contains other additional procedures and functions, such as:

```
function pause_profiler return binary_integer;
```

used for pausing the profiler, if you so choose,

```
function resume_profiler return binary_integer;
```

used for resuming the profiling whenever you stopped the profiling

```
function flush_data return binary_integer;
```

used for flushing the collected data from the internal storage to the profiler tables.

```
Proceure rollup_unit(run_number in number, unit in number);
```

Used calculating the sums on unit level

```
Procedure rollup_run(run_number);
```

Used calculating the sums on run level

Coming with the installation of the package is also a number of scripts you can run to identify your bottlenecks, but those I'm not covering here, hence I've build by own eul in discoverer for analysing purposes.

Simple use of dbms_profiler

If you are looking for identifying a bottleneck in your pl/sql program then you could use the profiler as shown in the following. To investigate you need to perform these simple 5 steps

1. Starting the profiling
2. Doing profiling
3. Stop profiling
4. Calculate sums.
5. 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
  v_err number;
  v_no binary_integer;
begin
  v_err := dbms_profiler.start_profiler(run_comment  => '&1'
                                        run_comment1 => sysdate
                                        run_number   => v_no);
  dbms_output.put_line('Run no '||v_no||' Error '=> v_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 Calculate sums

Oddly enough dbms_profiler does not calculate the sums when profiling, so you need to do al rollup. This can be done thy the following code:

```
begin
 dbms_profiler.rollup_run(&run_no);
end;
```

again in SQL*PLUS

Where `&run_no` is the run number returned in step 1.

Step 5 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 the figure below.

```
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;
```

Now these 5 steps needs to be repeated every time you find the need to do profiling.

Bottlenecks

When you have identified a pl/sql program where you want do find a specific bottleneck, and you have created the profiling results, you, in most cases, if the program is really big have a abundance of data available to investigate. I found it worthwhile to create a EUL in discoverer where I can investigate my results and find the interesting places. This eul is described in the following.

EUL setup

In the admin tool I've created 4 folders, as seen on the screenshot in figure 2, one folder for each table and a new one based on a view plsql_profiler_view see figure 3 where in include the code from the view is included.

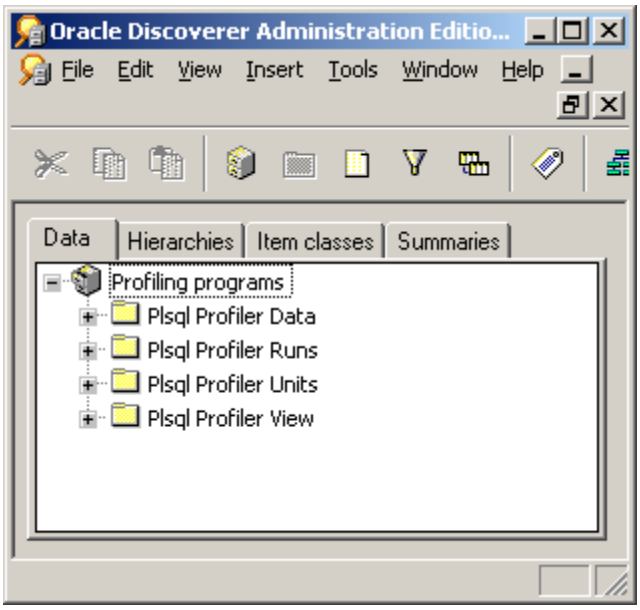


Figure 2 Discoverer admin screenshot

```
CREATE OR REPLACE VIEW PLSQL_PROFILER_VIEW
(RUNID, UNIT_NAME, UNIT_NUMBER, UNIT_TYPE, LINE#,
 TOTAL_OCCUR, TOTAL_TIME, MIN_TIME, MAX_TIME, SOURCE_TEXT)
AS
SELECT
 Ppu.runid, PPU.UNIT_NAME, ppu.unit_number, ppu.unit_type, ppd.line#,
        PPD.TOTAL_OCCUR,
        PPD.TOTAL_TIME,
        PPD.MIN_TIME,
        PPD.MAX_TIME,
        get_text_line(PPU.UNIT_NAME, ppu.unit_type, ppd.line#) source_text
 FROM PLSQL_PROFILER_RUNS PPR,
      PLSQL_PROFILER_UNITS PPU,
PLSQL_PROFILER_DATA PPD
WHERE PPU.RUNID       = PPR.RUNID
   AND PPD.UNIT_NUMBER = PPU.UNIT_NUMBER
   AND PPD.RUNID       = PPU.RUNID
```

Figure 3 Plsql_profiler_view

This view joins together information about pl/sql program units with their metrics.

On top of all this I created item classes for all the based on run_id and unit_number, allowing me to drill from folder to folder, see figure 4.

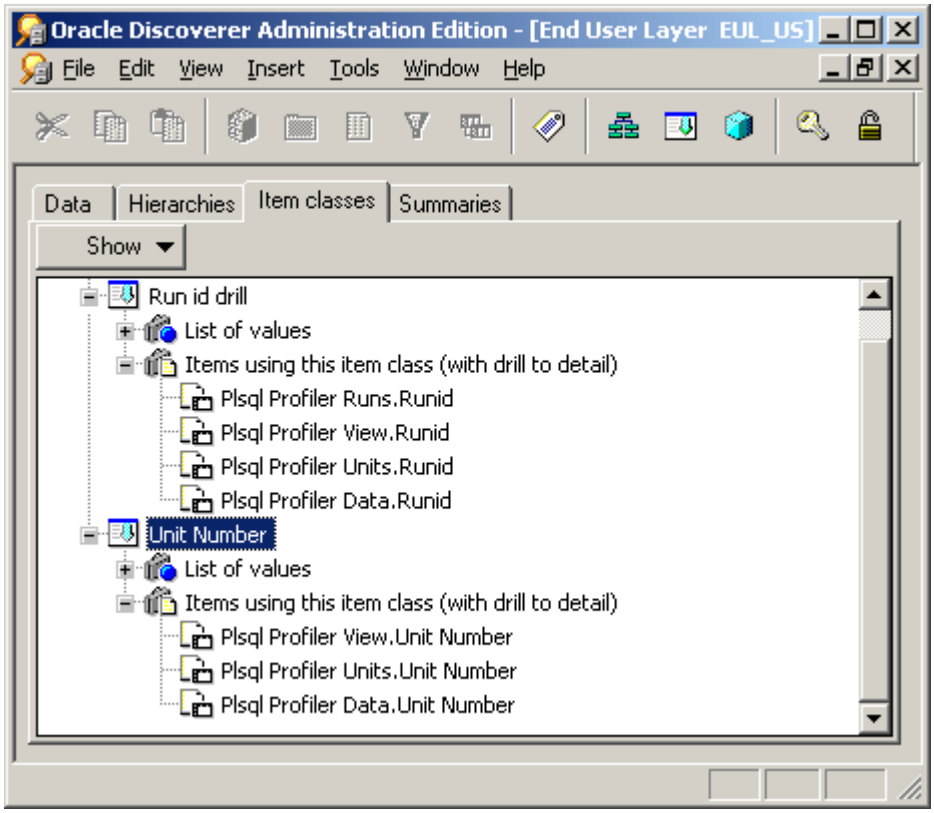


Figure 4 Discoverer admin item classes

With all this in place in now have an environment where i easily can navigate my profiling results, and create customized reports, as seen in the next section.

End user tool

In the end user tool, all i have to start up is a simple report with one tab page based on the folder plsql_profiler_runs as seen in figure 5, where run no 122 seems interesting.

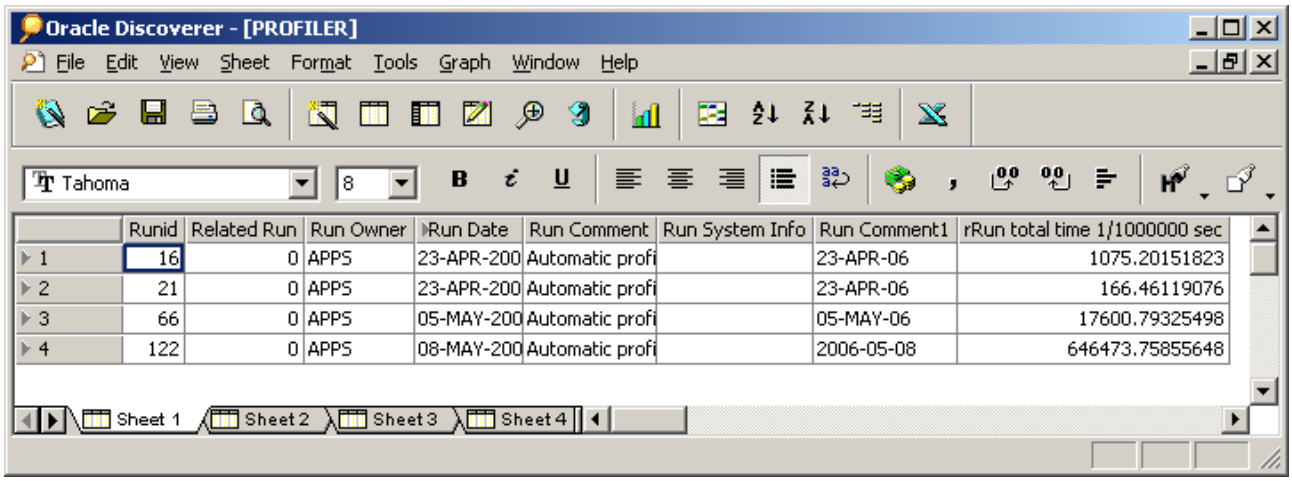
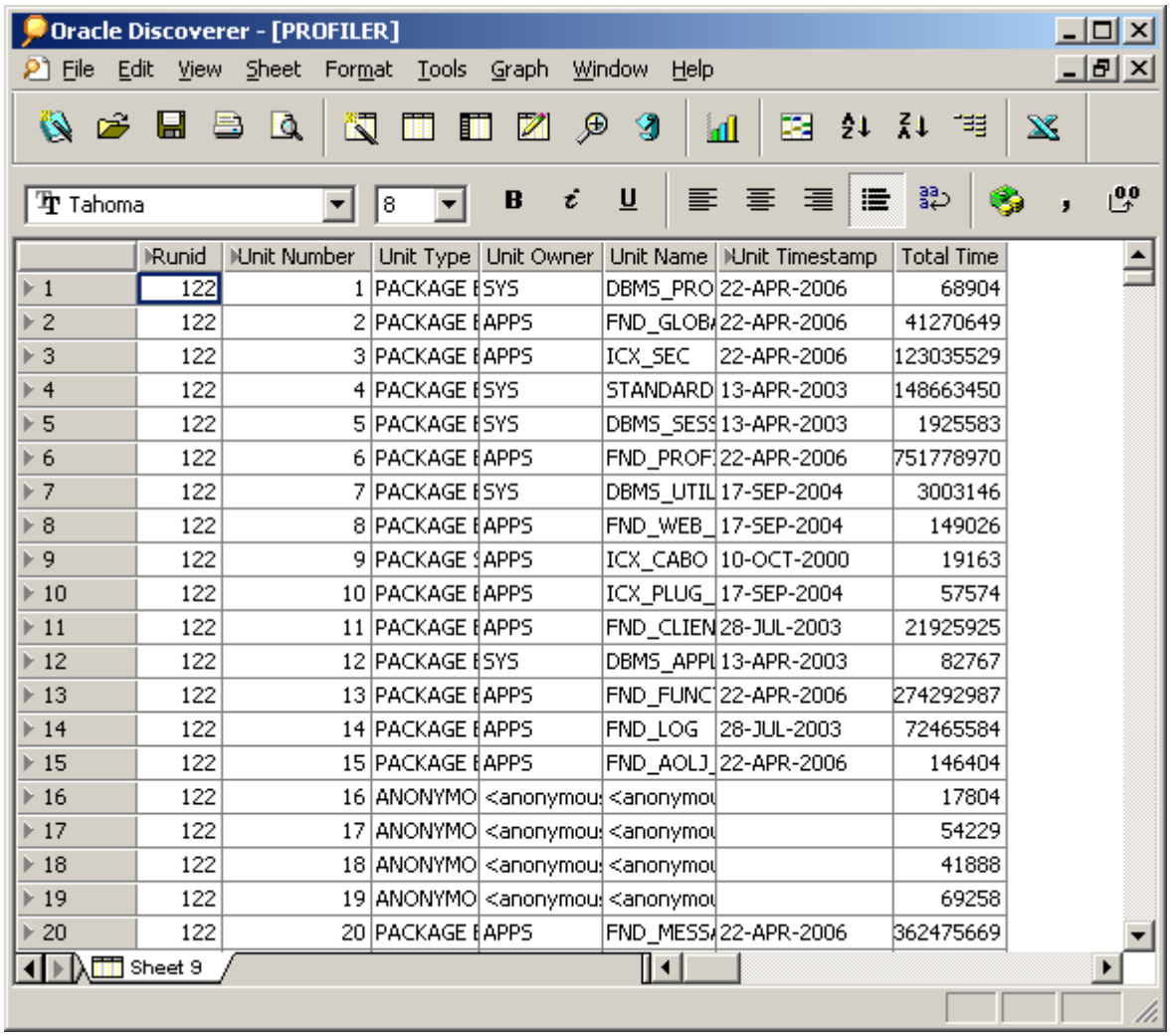


Figure 5 Displaying the different runs

Now it is simple click and double click to navigate around my information, with a double click I get information about the units involved in the run 122, as seen in figure 6.



	Runid	Unit Number	Unit Type	Unit Owner	Unit Name	Unit Timestamp	Total Time
1	122	1	PACKAGE B	SYS	DBMS_PRO	22-APR-2006	68904
2	122	2	PACKAGE B	APPS	FND_GLOB/	22-APR-2006	41270649
3	122	3	PACKAGE B	APPS	ICX_SEC	22-APR-2006	123035529
4	122	4	PACKAGE B	SYS	STANDARD	13-APR-2003	148663450
5	122	5	PACKAGE B	SYS	DBMS_SESS	13-APR-2003	1925583
6	122	6	PACKAGE B	APPS	FND_PROF	22-APR-2006	751778970
7	122	7	PACKAGE B	SYS	DBMS_UTIL	17-SEP-2004	3003146
8	122	8	PACKAGE B	APPS	FND_WEB_	17-SEP-2004	149026
9	122	9	PACKAGE S	APPS	ICX_CABO	10-OCT-2000	19163
10	122	10	PACKAGE B	APPS	ICX_PLUG_	17-SEP-2004	57574
11	122	11	PACKAGE B	APPS	FND_CLIEN	28-JUL-2003	21925925
12	122	12	PACKAGE B	SYS	DBMS_APPL	13-APR-2003	82767
13	122	13	PACKAGE B	APPS	FND_FUNC	22-APR-2006	274292987
14	122	14	PACKAGE B	APPS	FND_LOG	28-JUL-2003	72465584
15	122	15	PACKAGE B	APPS	FND_AOLJ_	22-APR-2006	146404
16	122	16	ANONYMO	<anonymous	<anonymou		17804
17	122	17	ANONYMO	<anonymous	<anonymou		54229
18	122	18	ANONYMO	<anonymous	<anonymou		41888
19	122	19	ANONYMO	<anonymous	<anonymou		69258
20	122	20	PACKAGE B	APPS	FND_MESS/	22-APR-2006	362475669

Figure 6 Units executed

And here line 13 looks interesting, so by double clicking I get the information about the specific lines executed along with the metrics for them, as seen in figure 7.

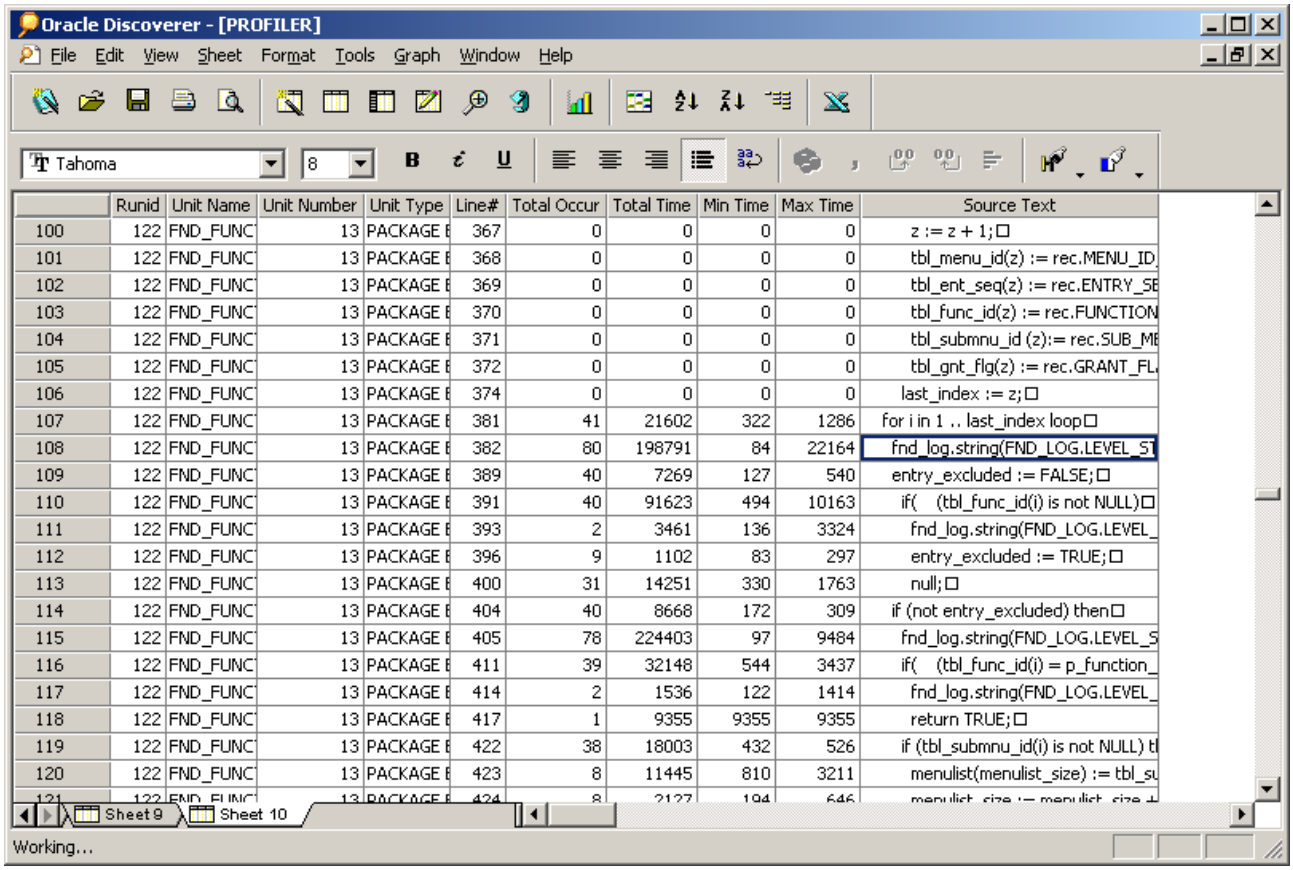


Figure 7 Metrics pr unit

Now what does my program do?

When extending a 3rd party environment, like the Oracle E*business suite, you often need to investigate what the underlying code is actually doing in order to have your extension work properly.

You first need to identify the pl/sql program in play, but when that is done you can use the same simple steps as mentioned above to create a report on the code executed.

During the preparation of this article I've spend some time investigating the e*business suite to see if I could build in the use of dbms_profiler in their standard code, and sadly i had to conclude that it was not possible, even though that I was convinced that it should be possible.

I did mange to get it working for the forms part of E*bus, but i was unable to get it to work for the self-service part of it, the part building on J2EE. The reason for this is that selfservice does not have persistent connections, and oracle did not create a hook (or trigger) where you could direct any code executed to stop the profiling. So starting the profiling was possible to extending the fnd_global package, but stopping the profiling proved not possible due to spawning of processes.

Newer the less I used the profiling method in order to follow a specific package because i needed to know what exactly happens when a resource is allocated to a project. By using the internal trace mechanism in E*bus, I identified that the procedure used for this was pa_assigments_pub.create_assignments, so therefore i could start the profiling, execute the package, and stop it again, and then inverstigate what happend.

The result of this cannot be included in this article, hence the number of lines executed is well above 10000, but during the presentation in will include a live demonstration.

Conclusion

In this article I've shown that concepts of dbms_profiler, and how is used. I also presented a discoverer framework that can be used for analyzing profiling results.

Thank you very much; any questions or ideas feel free to contact me at runm@nne.dk.