

PL/SQL Profiling & Test Coverage

|  |  |
| --- | --- |
| Date | September 23, 2008 |
| Author | Eduardo Morelli  Data Architect  Sakonnet Technology, LLC. |
| Document Version | Open source release |

**Table of Contents**

[1 Overview 4](#_Toc211832047)

[2 Before Profiling 4](#_Toc211832048)

[3 PKG\_PROFILER 5](#_Toc211832049)

[4 How to Query Results 6](#_Toc211832050)

[5 Profiling Activated for Every Session 7](#_Toc211832051)

[6 Test Coverarge 7](#_Toc211832052)

[7 Reporting 9](#_Toc211832053)

Document History

|  |  |  |
| --- | --- | --- |
| **Version** | **Date** | **Change** |
| Open Source | Feb 23, 2012 | Sanitized for public release |
| 1.3 | Nov 25, 2008 | Test Coverage Wrappers |
| 1.2 | Oct 15,2008 | Added XML reporting facilities for Test Coverage |
| 1.1 | Sep 30,2008 | Added comments regarding PROF.ZIP and reporting |
| 1 | Sep 23, 2008 | Document created |

# Overview

This document explains how to use PKG\_PROFILER. It has two main purposes: PL/SQL test coverage and performance bottleneck identification.

# Before Profiling

PKG\_PROFILER wraps DBMS\_PROFILER:

SQL> DESC DBMS\_PROFILER

FUNCTION FLUSH\_DATA RETURNS BINARY\_INTEGER

PROCEDURE FLUSH\_DATA

PROCEDURE GET\_VERSION

Argument Name Type In/Out Default?

------------------------------ ----------------------- ------ --------

MAJOR BINARY\_INTEGER OUT

MINOR BINARY\_INTEGER OUT

FUNCTION INTERNAL\_VERSION\_CHECK RETURNS BINARY\_INTEGER

FUNCTION PAUSE\_PROFILER RETURNS BINARY\_INTEGER

PROCEDURE PAUSE\_PROFILER

FUNCTION RESUME\_PROFILER RETURNS BINARY\_INTEGER

PROCEDURE RESUME\_PROFILER

PROCEDURE ROLLUP\_RUN

Argument Name Type In/Out Default?

------------------------------ ----------------------- ------ --------

RUN\_NUMBER NUMBER IN

PROCEDURE ROLLUP\_UNIT

Argument Name Type In/Out Default?

------------------------------ ----------------------- ------ --------

RUN\_NUMBER NUMBER IN

UNIT NUMBER IN

FUNCTION START\_PROFILER RETURNS BINARY\_INTEGER

Argument Name Type In/Out Default?

------------------------------ ----------------------- ------ --------

RUN\_COMMENT VARCHAR2 IN DEFAULT

RUN\_COMMENT1 VARCHAR2 IN DEFAULT

RUN\_NUMBER BINARY\_INTEGER OUT

PROCEDURE START\_PROFILER

Argument Name Type In/Out Default?

------------------------------ ----------------------- ------ --------

RUN\_COMMENT VARCHAR2 IN DEFAULT

RUN\_COMMENT1 VARCHAR2 IN DEFAULT

RUN\_NUMBER BINARY\_INTEGER OUT

FUNCTION START\_PROFILER RETURNS BINARY\_INTEGER

Argument Name Type In/Out Default?

------------------------------ ----------------------- ------ --------

RUN\_COMMENT VARCHAR2 IN DEFAULT

RUN\_COMMENT1 VARCHAR2 IN DEFAULT

PROCEDURE START\_PROFILER

Argument Name Type In/Out Default?

------------------------------ ----------------------- ------ --------

RUN\_COMMENT VARCHAR2 IN DEFAULT

RUN\_COMMENT1 VARCHAR2 IN DEFAULT

FUNCTION STOP\_PROFILER RETURNS BINARY\_INTEGER

PROCEDURE STOP\_PROFILER

If it is not available, please contact the DBA in order to create it.

Before any invocation, you must assure that 3 tables exist in your local schema:

|  |  |
| --- | --- |
| **Table Name** | **Comments** |
| PLSQL\_PROFILER\_RUNS | Every time profiling is started, a new row is inserted in this table. Primary key: RUNID |
| PLSQL\_PROFILER\_UNITS | Stores units being executed for each RUNID. A unit can be a package specification, package body, procedure, trigger or even an anonymous code. Primary key: RUNID, UNIT\_NUMBER |
| PLSQL\_PROFILER\_DATA | For each unit executed, every line of code is registered having total amount of time spent during all executions. Primary key: RUNID, UNIT\_NUMBER, LINE# |

Table 1 – DBMS\_PROFILER tables.

If those tables are not created, the following script must be executed locally:

**proftab.sql**

It is located in the Oracle installation directory:

**/Oracle/product/10.2.0/db\_1/RDBMS/ADMIN/**

If you cannot execute it, please ask the DBA for that.

Actually, if new fresh data is needed, this script should be re-executed.

CAUTION: never, ever issue a delete against those tables, because profiling code will fail.

# PKG\_PROFILER

PKG\_PROFILER has a package level variable, P\_ACTIVE\_PROFILING, which accepts two possible values, either 1 (on) or 0 (off). If it is turned on, profiling can be started executing PRC\_START\_PROFILING (see Table 2).

There two ways of changing P\_ACTIVE\_PROFILING. It can be modified directly in PKG\_PROFILER package body and then recompile it. Alternatively, PRC\_SET\_PROFILING can be executed, but this change would be disregarded at the end of current session.

PKG\_PROFILER has two functions and three procedures:

|  |  |
| --- | --- |
| **Unit** | **Comments** |
| PRC\_SET\_PROFILING | Allows profiling at session level. Its argument only accepts two values: 0 or 1. |
| PRC\_START\_PROFILING | Starts profiling. Its argument is a comment for identifying a run id. If this procedure is executed while profiling is being run, an error will be raised. |
| PRC\_END\_PROFILING | Ends profiling. If this procedure is executed while profiling is not being run, an error will be raised. |
| FNC\_RETURN\_CURR\_ACTIVE\_PROF | Returns either ENABLED or DISABLED. |
| FNC\_RETURN\_CURR\_ACTIVE\_RUN\_ID | Returns current Run Id, if there is one being executed. Otherwise, it returns -1. |

Table 2 – PKG\_PROFILER units

For example, let’s assume we want to profile PKG\_MKT\_COPY\_MDS. See below a hypothetical usage:

SQL> select pkg\_profiler.FNC\_RETURN\_CURR\_ACTIVE\_PROF from dual;

FNC\_RETURN\_CURR\_ACTIVE\_PROF

---------------------------------------------------------------------

ENABLED

SQL> execute pkg\_profiler.PRC\_START\_PROFILING ('Copy Mds')

PL/SQL procedure successfully completed.

SQL> execute pkg\_mkt\_copy\_mds.prc\_mkt\_copy\_mds(1,21)

PL/SQL procedure successfully completed.

SQL> execute pkg\_profiler.PRC\_END\_PROFILING

PL/SQL procedure successfully completed.

# How to Query Results

Once a profiling session is ended, we can easily query what was collected. For example, the following query would display which lines took more time to be completed:

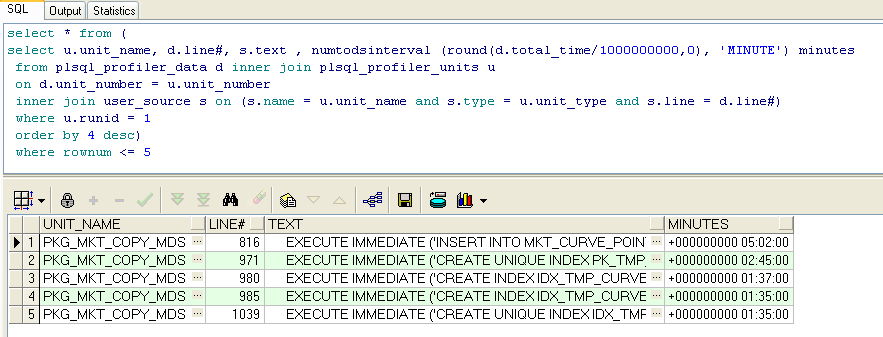


Figure 1 – Top 5 most consuming statements

From Figure 1, we can conclude that line 816 in PKG\_MKT\_COPY\_MDS package body took 5 minutes and 2 seconds to be completed.

Here comes the copyable SQL text:

select \* from (

select u.unit\_name, d.line#, s.text , numtodsinterval (round(d.total\_time/1000000000,2), 'MINUTE') minutes

from plsql\_profiler\_data d inner join plsql\_profiler\_units u

on d.unit\_number = u.unit\_number

inner join user\_source s on (s.name = u.unit\_name and s.type = u.unit\_type and s.line = d.line#)

where u.runid = 1

order by 4 desc)

where rownum <= 5

# Profiling Activated for Every Session

Every profiling activity only last for a session. If we are interested in keep profiling activities for more time, we should automate PRC\_START\_PROFILING and PRC\_END\_PROFILING calls. This could be done using schema triggers:

CREATE OR REPLACE TRIGGER On\_Logon

AFTER LOGON ON schema

DECLARE

V\_WHO VARCHAR2(200);

BEGIN

SELECT sys\_context('USERENV', 'OS\_USER') || '\_' ||

sys\_context('USERENV', 'SID') || '\_' ||

sys\_context('USERENV', 'TERMINAL')

INTO V\_WHO

FROM dual;

PKG\_PROFILER.PRC\_START\_PROFILING('STARTING PROFILER FOR (OS\_USER, SID, TERMINAL): ' || V\_WHO);

END;

/

CREATE OR REPLACE TRIGGER On\_Logoff

BEFORE LOGOFF

ON schema

DECLARE

BEGIN

PKG\_PROFILER.PRC\_END\_PROFILING;

END;

/

# Test Coverage

PKG\_PROFILER can be used as test coverage tool. For example, after creating those triggers, ON\_LOGON and ON\_LOGOFF, we could run all fit tests:

Cd E:\rdbms\_trade\rdbms.query

Maven fit

Cd E:\rdbms\_trade\domain

Maven fit4dbs

Once all fits have been executed, we can run this query in order to discover which packages hadn’t been executed:

select DISTINCT p.object\_name

from plsql\_profiler\_units u

right outer

join user\_procedures p on p.object\_name = u.unit\_name

where u.unit\_name is null

ORDER BY 1;

PKG\_PROFILER has a nice procedure named PRC\_TEST\_COVERAGE, which generates a report with all packages executed in a certain range of run ids. For example:

SQL> EXECUTE PKG\_PROFILER.PRC\_TEST\_COVERAGE (1,862);

1. PKG\_EOD\_PAL\_INTERFACE ------------- 100.00
2. PKG\_EOD\_RUN ------------- 8.35
3. PKG\_MKT\_CALC\_HIST\_FIXING ------------- 33.33
4. PKG\_MKT\_COPY\_MDS ------------- 47.89
5. PKG\_MKT\_CURVE\_PT\_OUTPUT\_CALC ------------- 16.29
6. PKG\_MKT\_DATA\_VALIDATION ------------- 18.03
7. PKG\_MKT\_DENORMALIZE\_DATA ------------- 42.48
8. PKG\_MKT\_GET\_PRECALC\_VAL\_PS ------------- 12.50
9. PKG\_MKT\_GET\_VAL\_EOD ------------- 46.43
10. PKG\_MKT\_GET\_VAL\_P ------------- 4.04
11. PKG\_MKT\_GET\_VAL\_PS ------------- 30.65
12. PKG\_MKT\_GET\_VAL\_TRADE\_DES ------------- 15.14
13. PKG\_MKT\_HIST\_FIXINGS ------------- 93.83
14. PKG\_MKT\_SHATTER\_UNSHATTER ------------- 33.33
15. PKG\_MKT\_UTILS ------------- 48.29
16. PKG\_MKT\_VAL\_UTILS ------------- 15.67
17. PKG\_MKT\_VAR\_PS ------------- 83.33
18. PKG\_OBJ\_IS\_LATEST\_VERSION\_CK ------------- 79.24
19. PKG\_PROFILER ------------- 12.50
20. PKG\_UTILS ------------- 9.34
21. PKG\_VAL\_AGG\_PHYSICAL\_FLOW ------------- 9.76
22. PKG\_VAL\_EOD\_REPORT\_ERROR ------------- 66.67
23. PKG\_VAL\_EOD\_REPORT\_NUON\_OTC ------------- 50.00
24. PKG\_VAL\_EOD\_REPORT\_PBP\_CHANGE ------------- 100.00
25. PKG\_VAL\_EOD\_REPORT\_PNLBYPERIOD------------- 66.67
26. PKG\_VAL\_EOD\_REPORT\_POSITION ------------- 66.67
27. PKG\_VAL\_EOD\_REPORT\_POS\_CHANGE ------------- 100.00
28. PKG\_VAL\_EXCEL\_PRICING ------------- 100.00
29. PKG\_VAL\_MAT\_DASHBOARD\_PRICING ------------- 23.24
30. PKG\_VAL\_PHYSICAL\_FLOW\_REPORT ------------- 12.79
31. PKG\_VAL\_SCALE\_P ------------- 47.62
32. PKG\_VAL\_SCALE\_PCPSTT ------------- 47.62
33. PKG\_VAL\_SCALE\_PRC ------------- 47.62
34. PKG\_VAL\_SCALE\_PS ------------- 41.27
35. PKG\_VAL\_SCALE\_PTT ------------- 47.62

Triggers not being executed can be displayed executing the following query:

select DISTINCT t.trigger\_name object\_name

from plsql\_profiler\_units u

right outer join user\_triggers t on t.trigger\_name = u.unit\_name

where u.unit\_name is null

ORDER BY 1;

Trigger percent coverage can be generated executing:

declare

CURSOR CUNITS IS

SELECT DISTINCT UNIT\_NAME

FROM PLSQL\_PROFILER\_UNITS

WHERE UNIT\_NAME LIKE 'T\_%'

ORDER BY 1;

V\_COVERAGE NUMBER(5, 2);

BEGIN

FOR REG IN CUNITS LOOP

SELECT ROUND(EXEC.NBR / TOTAL.NBR \* 100, 2) COVERAGE

INTO V\_COVERAGE

FROM (SELECT COUNT(\*) NBR

FROM PLSQL\_PROFILER\_DATA D, PLSQL\_PROFILER\_UNITS U

WHERE D.RUNID = U.RUNID

AND D.UNIT\_NUMBER = U.UNIT\_NUMBER

AND U.UNIT\_NAME = REG.UNIT\_NAME) TOTAL,

(SELECT COUNT(\*) NBR

FROM PLSQL\_PROFILER\_DATA D, PLSQL\_PROFILER\_UNITS U

WHERE D.RUNID = U.RUNID

AND D.UNIT\_NUMBER = U.UNIT\_NUMBER

AND U.UNIT\_NAME = REG.UNIT\_NAME

AND D.TOTAL\_OCCUR > 0) EXEC;

DBMS\_OUTPUT.PUT\_LINE(RPAD(REG.UNIT\_NAME, 30, ' ') || '-------------' ||

TO\_CHAR(V\_COVERAGE, '999.99'));

END LOOP;

end;

/

Note that every run id is being taken into account. In the future, we might standardize the method we get those percent coverages: either using a procedure or a script.

At Sakonnet, a table was used, TC\_NOT\_TOUCHED\_HISTORICAL which stores object names and dates. Basically, it must be updated every time a new full test coverage report is generated, in order to compare with previous executions.

# Reporting

MetaLink note 243775.1, “Implementing and Using the PL/SQL Profiler,” contains a downloadable ZIP file (PROF.ZIP), with a SQL script to take a run and generate HTML output. You can run profiler.sql as is and enter a run ID, or pass the ID in as a parameter. For example:

SQL> @profiler 7

RUNID RUN\_DATE RUN\_COMMENT

---------- --------------------------- --------------------

1 30-SEP-08 14:21:11 30-09-2008 14:21:11

2 30-SEP-08 14:21:13 30-09-2008 14:21:13

3 30-SEP-08 14:21:13 30-09-2008 14:21:13

4 30-SEP-08 14:21:13 30-09-2008 14:21:13

5 30-SEP-08 14:21:18 30-09-2008 14:21:18

6 30-SEP-08 14:21:18 30-09-2008 14:21:18

7 30-SEP-08 14:21:19 30-09-2008 14:21:19

8 30-SEP-08 14:21:23 30-09-2008 14:21:23

9 30-SEP-08 14:21:23 30-09-2008 14:21:23

10 30-SEP-08 14:21:23 30-09-2008 14:21:23

11 30-SEP-08 14:21:28 30-09-2008 14:21:28

12 30-SEP-08 14:21:28 30-09-2008 14:21:28

...

858 30-SEP-08 14:33:12 30-09-2008 14:33:12

859 30-SEP-08 14:33:12 30-09-2008 14:33:12

860 30-SEP-08 14:33:12 30-09-2008 14:33:12

861 30-SEP-08 14:33:14 30-09-2008 14:33:14

RUNID RUN\_DATE RUN\_COMMENT

---------- --------------------------- --------------------

862 30-SEP-08 14:33:14 30-09-2008 14:33:14

862 rows selected.

Usage:

sqlplus apps/<pwd>

SQL> START profiler.sql <runid>

A new HTML file (profiler\_7.html) will be generated having several interesting information regarding Run Id 7.

# Test Coverage XML Reporting

The following is a brief description of the scripts needed to convert the profiling data into a coverage report:

The main script is **Oracle\_TC.sql**. First, it invokes **Load\_PLSQL\_Profiler\_  
tables.sql**, which must be adjusted to reference correct source (test schema).

In **Load\_PLSQL\_Profiler\_tables.sql**, proftab.sql is invoked in order to create local copies of profiler internal tables. It is identical to a homonymous script find in <ORACLE\_HOME>\RDBMS\ADMIN.

After loading XML\_STAGE, an internal table for reporting purposes, several *wrappers* are invoked. They will stay there until there will be any fit to touch correspondant packages. Thus WRP\_ERR.sql will be executed until any fit forces its execution.

**Triggers\_Not\_Being\_Touched.sql** acts as a wrapper for all non touched triggers.

As previously stated, all non touched packages and triggers names are stored in tc\_not\_touched\_historical, the aforementioned historical results table.

As a final step, Load\_And\_Export\_CLOBS.sql is invoked to generate all XML files, each one for exery package, procedure and trigger. Finally, in order to generate all files properly, it must be created a local directory, E:\TestCoverage\XML.