155
DBMS_PROFILER

The package provides an interface to profile existing PL/SQL applications and identify performance bottlenecks. You can then collect and persistently store the PL/SQL profiler data.

This chapter contains the following topics:

DBMS_PROFILER Overview

This package enables the collection of profiler (performance) data for performance improvement or for determining code coverage for PL/SQL applications. Application developers can use code coverage data to focus their incremental testing efforts.

With this interface, you can generate profiling information for all named library units that are executed in a session. The profiler gathers information at the PL/SQL virtual machine level. This information includes the total number of times each line has been executed, the total amount of time that has been spent executing that line, and the minimum and maximum times that have been spent on a particular execution of that line.

> **Note:**
>
> It is possible to infer the code coverage figures for PL/SQL units for which data has been collected.

The profiling information is stored in database tables. This enables querying on the data: you can build customizable reports (summary reports, hottest lines, code coverage data, and so on. And you can analyze the data.

The `PROFTAB.SQL` script creates tables with the columns, datatypes, and definitions as shown in the following tables.

Table 155-1 Columns in Table PLSQL_PROFILER_RUNS

Column	Datatype	Definition
`runid`	`NUMBER PRIMARY KEY`	Unique run identifier from plsql_profiler_runnumber
`related_run`	`NUMBER`	Runid of related run (for client/server correlation)
`run_owner`	`VARCHAR2(128)`	User who started run
`run_date`	`DATE`	Start time of run

Table 155-1 (Cont.) Columns in Table PLSQL_PROFILER_RUNS

Column	Datatype	Definition
run_comment	VARCHAR2(2047)	User provided comment for this run
run_total_time	NUMBER	Elapsed time for this run in nanoseconds
run_system_info	VARCHAR2(2047)	Currently unused
run_comment1	VARCHAR2(2047)	Additional comment
spare1	VARCHAR2(256)	Unused

Table 155-2 Columns in Table PLSQL_PROFILER_UNITS

Column	Datatype	Definition
runid	NUMBER	Primary key, references plsql_profiler_runs,
unit_number	NUMBER	Primary key, internally generated library unit #
unit_type	VARCHAR2(128)	Library unit type
unit_owner	VARCHAR2(128)	Library unit owner name
unit_name	VARCHAR2(128)	Library unit name timestamp on library unit
unit_timestamp	DATE	In the future will be used to detect changes to unit between runs
total_time	NUMBER	Total time spent in this unit in nanoseconds. The profiler does not set this field, but it is provided for the convenience of analysis tools.
spare1	NUMBER	Unused
spare2	NUMBER	Unused

Table 155-3 Columns in Table PLSQL_PROFILER_DATA

Column	Datatype	Definition
runid	NUMBER	Primary key, unique (generated) run identifier
unit_number	NUMBER	Primary key, internally generated library unit number
line#	NUMBER	Primary key, not null, line number in unit
total_occur	NUMBER	Number of times line was executed
total_time	NUMBER	Total time spent executing line in nanoseconds
min_time	NUMBER	Minimum execution time for this line in nanoseconds
max_time	NUMBER	Maximum execution time for this line in nanoseconds
spare1	NUMBER	Unused
spare2	NUMBER	Unused
spare3	NUMBER	Unused
spare4	NUMBER	Unused

With Oracle database version 8.x, a sample textual report writer(profrep.sql) is provided with the PL/SQL demo scripts.

Note that prior to Oracle Database 10*g*, the DBMS_PROFILER package was not automatically loaded when the database was created, and the Oracle-supplied PROFLOAD.SQL script was used to create it. In 10*g* and beyond, the DBMS_PROFILER package is loaded automatically when the database is created, and PROFLOAD.SQL is no longer needed.

DBMS_PROFILER Security Model

The profiler only gathers data for units for which a user has CREATE privilege; you cannot use the package to profile units for which EXECUTE ONLY access has been granted. In general, if a user can debug a unit, the same user can profile it. However, a unit can be profiled whether or not it has been compiled DEBUG. Oracle advises that modules that are being profiled should be compiled DEBUG, since this provides additional information about the unit in the database.

> **Note:**
>
> DBMS_PROFILER treats any program unit that is compiled in NATIVE mode as if you do not have CREATE privilege, that is, you will not get any output.

DBMS_PROFILER Operational Notes

These notes describe a typical run, how to interpret output, and two methods of exception generation.

Typical Run

Improving application performance is an iterative process. Each iteration involves the following steps:

1. Running the application with one or more benchmark tests with profiler data collection enabled.
2. Analyzing the profiler data and identifying performance problems.
3. Fixing the problems.

The PL/SQL profiler supports this process using the concept of a *"run".* A run involves running the application through benchmark tests with profiler data collection enabled. You can control the beginning and the ending of a run by calling the START_PROFILER and STOP_PROFILER functions.

The user must first create database tables in the profiler user's schema to collect the data. The PROFTAB.SQL script creates the tables and other data structures required for persistently storing the profiler data.

Note that running PROFTAB.SQL drops the current tables. The PROFTAB.SQL script is in the RDBMS/ADMIN directory. Some PL/SQL operations, such as the first execution of a PL/SQL unit, may involve I/O to catalog tables to load the byte code for the PL/SQL unit being executed. Also, it may take some time executing package initialization code the first time a package procedure or function is called.

To avoid timing this overhead, "*warm up"* the database before collecting profile data. To do this, run the application once without gathering profiler data.

You can allow profiling across all users of a system, for example, to profile all users of a package, independent of who is using it. In such cases, the `SYSADMIN` should use a modified `PROFTAB.SQL` script which:

- Creates the profiler tables and sequence
- Grants `SELECT`/`INSERT`/`UPDATE` on those tables and sequence to all users
- Defines public synonyms for the tables and sequence

> **Note:**
>
> Do not alter the actual fields of the tables.

A typical run then involves:

- Starting profiler data collection in the run.
- Executing PL/SQL code for which profiler and code coverage data is required.
- Stopping profiler data collection, which writes the collected data for the run into database tables

> **Note:**
>
> The collected profiler data is not automatically stored when the user disconnects. You must issue an explicit call to the `FLUSH_DATA` or the `STOP_PROFILER` function to store the data at the end of the session. Stopping data collection stores the collected data.

As the application executes, profiler data is collected in memory data structures that last for the duration of the run. You can call the `FLUSH_DATA` function at intermediate points during the run to get incremental data and to free memory for allocated profiler data structures. Flushing the collected data involves storing collected data in the database tables created earlier.

> **See Also:**
>
> "FLUSH_DATA Function and Procedure".

Interpreting Output

The table `plsql_profiler_data` contains one row for each line of the source unit for which code was generated. The line# value specifies which source line. If the row exists, and the total_occur value in that row is > 0, some code associated with that line was executed. If the row exists, and total_occur value is 0, no code associated with that line was executed. If the row doesn't exist in the table, no code was generated for that line, and therefore it should not be mentioned in reports

If the source of a single statement is on a single line, any code generated for that statement will be attributed to that line number. (In some cases, such as a simple declaration, or because of optimization, no code will be needed). To get coverage information, units should be compiled with `PLSQL_OPTIMIZE_LEVEL=1`.

If a statement spans multiple lines, any code generated for that statement will be attributed to some line in the range, but it is not guaranteed that every line in the range will have code attributed to it. In such a case there will be gaps in the set of line# values. In particular, multi-line SQL-related statements may appear to be on a single line (usually the first). This is because PL/SQL passes the processed text of the cursor to the SQL engine; therefore, as far as PL/SQL is concerned, the entire SQL statement is a single indivisible operation.

When multiple statements are on the same line, the profiler will combine the occurrences for each statement. This may be confusing if a line has embedded control flow. For example, if `'then ...'` and `'else ...'` are on the same line, it will not be possible to determine whether the `'then'` or the `'else'` was taken.

In general, profiler and coverage reports are most easily interpreted if each statement is on its own line.

Two Methods of Exception Generation

Each routine in this package has two versions that allow you to determine how errors are reported.

- A function that returns success/failure as a status value and will never raise an exception
- A procedure that returns normally if it succeeds and raises an exception if it fails

In each case, the parameters of the function and procedure are identical. Only the method by which errors are reported differs. If there is an error, there is a correspondence between the error codes that the functions return, and the exceptions that the procedures raise.

To avoid redundancy, the following section only provides details about the functional form.

DBMS_PROFILER Exceptions

DBMS_PROFILER throws the exceptions described in this topic.

Table 155-4 DBMS_PROFILER Exceptions

Exception	Description
`version_mismatch`	Corresponds to error_version.
`profiler_error`	Corresponds to either "error_param" or "error_io".

A `0` return value from any function denotes successful completion; a nonzero return value denotes an error condition. The possible errors are as follows:

- 'A subprogram was called with an incorrect parameter.'

  ```
  error_param constant binary_integer := 1;
  ```

- 'Data flush operation failed. Check whether the profiler tables have been created, are accessible, and that there is adequate space.'

  ```
  error_io    constant binary_integer := 2;
  ```

- There is a mismatch between package and database implementation. Oracle returns this error if an incorrect version of the `DBMS_PROFILER` package is installed, and if the version of the profiler package cannot work with this database version. The only recovery is to install the correct version of the package.

```
error_version constant binary_integer := -1;
```

Summary of DBMS_PROFILER Subprograms

This table lists the DBMS_PROFILER subprograms and briefly describes them.

Table 155-5 DBMS_PROFILER Package Subprograms

Subprogram	Description
FLUSH_DATA Function and Procedure	Flushes profiler data collected in the user's session
GET_VERSION Procedure	Gets the version of this API
INTERNAL_VERSION_CHECK Function	Verifies that this version of the DBMS_PROFILER package can work with the implementation in the database
PAUSE_PROFILER Function and Procedure	Pauses profiler data collection
RESUME_PROFILER Function and Procedure	Resumes profiler data collection
START_PROFILER Functions and Procedures	Starts profiler data collection in the user's session
STOP_PROFILER Function and Procedure	Stops profiler data collection in the user's session

FLUSH_DATA Function and Procedure

This function flushes profiler data collected in the user's session. The data is flushed to database tables, which are expected to preexist.

> **Note:**
>
> Use the PROFTAB.SQL script to create the tables and other data structures required for persistently storing the profiler data.

Syntax

```
DBMS_PROFILER.FLUSH_DATA
  RETURN BINARY_INTEGER;

DBMS_PROFILER.FLUSH_DATA;
```

GET_VERSION Procedure

This procedure gets the version of this API.

Syntax

```
DBMS_PROFILER.GET_VERSION (
   major  OUT BINARY_INTEGER,
   minor  OUT BINARY_INTEGER);
```

Parameters

Table 155-6 GET_VERSION Procedure Parameters

Parameter	Description
major	Major version of DBMS_PROFILER.
minor	Minor version of DBMS_PROFILER.

INTERNAL_VERSION_CHECK Function

This function verifies that this version of the DBMS_PROFILER package can work with the implementation in the database.

Syntax

```
DBMS_PROFILER.INTERNAL_VERSION_CHECK
  RETURN BINARY_INTEGER;
```

PAUSE_PROFILER Function and Procedure

This function pauses profiler data collection.

Syntax

```
DBMS_PROFILER.PAUSE_PROFILER
  RETURN BINARY_INTEGER;

DBMS_PROFILER.PAUSE_PROFILER;
```

RESUME_PROFILER Function and Procedure

This function resumes profiler data collection.

Syntax

```
DBMS_PROFILER.RESUME_PROFILER
  RETURN BINARY_INTEGER;

DBMS_PROFILER.RESUME_PROFILER;
```

START_PROFILER Functions and Procedures

This function starts profiler data collection in the user's session.

There are two overloaded forms of the START_PROFILER function; one returns the run number of the started run, as well as the result of the call. The other does not return the run number. The first form is intended for use with GUI-based tools controlling the profiler.

Syntax

```
DBMS_PROFILER.START_PROFILER(
   run_comment    IN VARCHAR2 := sysdate,
   run_comment1   IN VARCHAR2 :='',
   run_number     OUT BINARY_INTEGER)
 RETURN BINARY_INTEGER;
```

```
DBMS_PROFILER.START_PROFILER(
   run_comment IN VARCHAR2 := sysdate,
   run_comment1 IN VARCHAR2 :='')
RETURN BINARY_INTEGER;

DBMS_PROFILER.START_PROFILER(
   run_comment   IN VARCHAR2 := sysdate,
   run_comment1  IN VARCHAR2 :='',
   run_number    OUT BINARY_INTEGER);

DBMS_PROFILER.START_PROFILER(
   run_comment IN VARCHAR2 := sysdate,
   run_comment1 IN VARCHAR2 :='');
```

Parameters

Table 155-7 START_PROFILER Function Parameters

Parameter	Description
run_comment	Each profiler run can be associated with a comment. For example, the comment could provide the name and version of the benchmark test that was used to collect data.
run_number	Stores the number of the run so you can store and later recall the run's data.
run_comment1	Allows you to make interesting comments about the run.

STOP_PROFILER Function and Procedure

This function stops profiler data collection in the user's session.

This function has the side effect of flushing data collected so far in the session, and it signals the end of a run.

Syntax

```
DBMS_PROFILER.STOP_PROFILER
  RETURN BINARY_INTEGER;

DBMS_PROFILER.STOP_PROFILER;
```