Chapter No. 11
"Profiling and Tracing PL/SQL Code"
In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.11 "Profiling and Tracing PL/SQL Code"

A synopsis of the book’s content

Information on where to buy this book

About the Author

**Saurabh K. Gupta** got introduced to Oracle database around 5 years ago. Since then, he has been synchronizing his on job and off job interests with Oracle database programming. As an Oracle 11g Certified Advanced PL/SQL Professional, he soon moved from programming to database designing, development, and day-to-day database administration activities. He has been an active Oracle blogger and OTN forum member. He has authored and published more than 70 online articles and papers. His work can be seen in RMOUG journal, PSOG, dbanotes, Exforsys, and Club Oracle. He shares his technical experience through his blog: [http://sbhoracle.wordpress.com/](http://sbhoracle.wordpress.com/). He is a member of **All India Oracle Users Group (AIOUG)** and loves to participate in technical meets and conferences.

Besides digging into Oracle, sketching and snooker are other pastimes for him. He can be reached through his blog [sbhOracle](http://sbhOracle) for any comments, suggestions, or feedback regarding this book.

For More Information:

Oracle Database 11g brings in a weighted package of new features which takes the database management philosophy from instrumental to self-intelligence level. The new database features, which are more properly called "advanced", rather than "complex", aim either of the two purposes:

- Replacement of a workaround solution with a permanent one (as an enhancement)
- By virtue of routine researches and explorations, introduce a fresh feature to help database administrators and developers with their daily activities

*Oracle Advanced PL/SQL Professional Guide* focuses on advanced features of Oracle 11g PL/SQL. The areas targeted are PL/SQL code design, measuring and optimizing PL/SQL code performance, and analyzing PL/SQL code for reporting purposes and immunizing against attacks. The advanced programming topics such as usage of collections, implementation of VPD, interaction with external procedures in PL/SQL, performance orientation by caching results, tracing and profiling techniques, and protecting against SQL injection will familiarize you with the latest programming findings, trends and recommendations of Oracle. In addition, this book will help you to learn the latest, best practices of PL/SQL programming in terms of code writing, code analyzing for reporting purposes, tracing for performance, and safeguarding the PL/SQL code against hackers.

*An investment in knowledge pays the best interest.*

- *Benjamin Franklin*
The fact remains that the technical certifications from Oracle Corporation establish a benchmark of technical expertise and credibility, and set the tone of an improved career path for application developers. With the growing market in database development, Oracle introduced Advanced PL/SQL Professional Certification (1Z0-146) in the year 2008. The OCP (1Z0-146) certification exam tests aspirants on knowledge of advanced PL/SQL concepts (validated up to Oracle 11g Release 1). An advanced PL/SQL professional is expected to independently design, develop, and tune the PL/SQL code which can efficiently interface database systems and user applications.

The book, *Oracle Advanced PL/SQL Professional Guide*, is a sure recommendation for the preparation of the OCP certification (1Z0-146) exam. Advanced PL/SQL topics are explained thoroughly with the help of demonstrations, figures, and code examples. The book will not only explain a feature, but will also teach its implementation and application. You can easily pick up the content structure followed in the book. The code examples can be tried on your local database setups to give you a feel of the usage of a specific feature in real time scenarios.

**What This Book Covers**

*Chapter 1, Overview of PL/SQL Programming Concepts,* covers the overview of PL/SQL as the primary database programming language. It describes the characteristics of the language and its strengths in database development. This chapter speeds up with the structure of a PL/SQL block and reviews PL/SQL objects such as procedures, functions, and packages. In this chapter, we will also learn to work with SQL Developer.

*Chapter 2, Designing PL/SQL Code,* discusses the handling of cursors in a PL/SQL program. This chapter helps you to learn the guidelines for designing a cursor, usage of cursor variables, and cursor life cycle.

*Chapter 3, Using Collections,* introduces a very important feature of PL/SQL—collections. A collection in a database is very similar to arrays or maps in other programming languages. This chapter compares collection types and makes recommendations for the appropriate selection in a given situation. This chapter also covers the collection methods which are utility APIs for working with collections.

*Chapter 4, Using Advanced Interface Methods,* teaches how to interact with an external program written in a non-PL/SQL language, within PL/SQL. It demonstrates the execution steps for external procedures in PL/SQL. This steps describe the network configuration on a database server (mounted on Windows OS), library object creation, and publishing of a non-language program as an external routine.

For More Information:
Chapter 5, Implementing VPD with Fine Grained Access Control, introduces the concept of Fine Grained Access in PL/SQL. The working of FGAC as Virtual Private Database is explained in detail along with an insight into its key features. You will find stepwise implementation of VPD with the help of policy function and the DBMS_RLS package. This chapter also describes policy enforcement through application contexts.

Chapter 6, Working with Large Objects, discusses the traditional and conventional way of handling large objects in an Oracle database. This chapter starts with the familiarization of the available LOB data types (BLOB, CLOB, BFILE, and Temporary LOBs) and their significance. You will learn about the creation of LOB types in PL/SQL and their respective handling operations. This chapter demonstrates the management of LOB data types using SQL and the DBMS_LOB package.

Chapter 7, Using SecureFile LOBs, introduces one of the key innovations in Oracle 11g — SecureFiles. SecureFiles are upgraded LOBs which work on an improved philosophy of storage and maintenance. The key improvements of SecureFiles — deduplication, compression, and encryption — are licensed features. This chapter discusses and demonstrates the implementation of these three properties. You will learn how to migrate (or rather upgrade) the existing older LOBs into a new scheme — SecureFiles. The migration techniques covered use an online redefinition method and a partition method.

Chapter 8, Compiling and Tuning to Improve Performance, describes fair practices in effective PL/SQL programming. You will be very interested to discover how better code writing impacts code performance. This chapter explains an important aspect of query optimization — the PLSQL_OPTIMIZE_LEVEL parameter. The code behavior and optimization strategy at each level will help you to understand the language internals. Subsequently, the new PRAGMA feature will give you a deeper insight into subprogram inlining concepts.

Chapter 9, Caching to Improve Performance, covers another hot feature of Oracle 11g Database — server-side result caching. The newly introduced server-side cache component in SGA holds the results retrieved from SQL query or PL/SQL function. This chapter describes the configuration of a database server for caching feature through related parameters, implementation in SQL through RESULT_CACHE hint and implementation in PL/SQL function through the RESULT_CACHE clause. Besides the implementation section, this chapter teaches the validation and invalidation of result cache, using the DBMS_RESULT_CACHE package.

For More Information:
Chapter 10, Analyzing PL/SQL Code, helps you to understand and learn code diagnostics tricks and code analysis for reporting purposes. You will learn to monitor identifier usage, about compilation settings, and generate the subsequent reports from SQL Developer. This chapter discusses a very important addition in Oracle 11g—PL/Scope. It covers the explanations and illustrations to generate the structural reports through the dictionary views. In addition, this chapter also demonstrates the use of the DBMS_METADATA package to retrieve and extract metadata of database objects from the database in multiple formats.

Chapter 11, Profiling and Tracing PL/SQL Code, aims to demonstrate the tracing and profiling features in PL/SQL. The tracing demonstration uses the DBMS_TRACE package to trace the enabled or all calls in a PL/SQ program. The PL/SQL hierarchical profiler is a new innovation in 11g to identify and report the time consumed at each line of the program. The biggest benefit is that raw profiler data can be reproduced meaningfully into HTML reports.

Chapter 12, Safeguarding PL/SQL Code against SQL Injection Attacks, discusses the SQL injection as a concept and its remedies. The SQL injection is a serious attack on the vulnerable areas of the PL/SQL code which can lead to extraction of confidential information and many fatal results. You will learn the impacts and precautionary recommendations to avoid injective attacks. This chapter discusses the preventive measures such as using invoker's rights, client input validation tips, and using DBMS_ASSERT to sanitize inputs. It concludes on the testing strategies which can be practiced to identify vulnerable areas in SQL.

Appendix, Answers to Practice Questions, contains the answers to the practice questions for all chapters.
Now that we have stepped out of the code development stage, we are discussing best practices of code management and maintenance. In the last chapter, we walked through the strategies of code tracking, error tracking, and the PL/Scope tool for identifier tracking. We noticed that the PL/Scope tool does static code analysis. In this chapter, we are going to learn two important techniques for measuring code performance. The techniques are known as tracing and profiling. The primary goal of the code tracing and profiling techniques is to identify performance bottlenecks in the PL/SQL code and gather performance statistics at each execution step. We will discuss the tracing and profiling features in PL/SQL in the following topics:

- **Tracing PL/SQL programs**
  - The `DBMS_TRACE` package
  - Viewing trace information
- **Profiling PL/SQL programs**
  - The `DBMS_HPROF` package
  - The `plshprof` utility
  - Generating HTML profiler reports

For More Information:
Tracing the PL/SQL programs

Code tracing is an important technique to measure the code performance during runtime and identify the expensive areas in the code which can be worked upon to improve the performance. The tracing feature shows the code execution path followed by the server and reveals the time consumed at each step. Often developers assume tracing and debugging as one step, but both are distinctive features. Tracing is a one-time activity which analyses the complete code and prepares the platform for debugging. On the other hand, debugging is the bug identification and fixing activity where the trace report can be used to identify and work upon the problematic points.

Oracle offers multiple methods of tracing:

- **DBMS_APPLICATION_INFO**: The `SET_MODULE` and `SET_ACTION` subprograms can be used to register a specific action in a specific module.
- **DBMS_TRACE**: The Oracle built-in package allows tracing of PL/SQL subprograms, exceptions and SQL execution. The trace information is logged into `SYS` owned tracing tables (created by executing `tracetab.sql`).
- **DBMS_SESSION and DBMS_MONITOR**: The package can be employed in parallel to set the client ID and monitor the respective client ID. It is equivalent to a 10046 trace and logs the code diagnostics in a trace file.
- **The trcsess and tkprof utilities**: The `trcsess` utility merges multiple trace files in one and is usually deployed in shared server environments and parallel query sessions. The `tkprof` utility used to be a conventional tracing utility which generated readable output file. It was useful for large trace files and can also be used to load the trace information into a database.

Besides the methods mentioned in the preceding list, there are third-party tools from LOG4PLSQL and Quest which are used to trace the PL/SQL codes. A typical trace flow in a program is demonstrated in the following diagram:

![Trace Flow Diagram]

---

**For More Information:**

In this chapter, we will drill down the `DBMS_TRACE` package to demonstrate the tracing feature in PL/SQL. Further, we will learn the profiling strengths of `DBMS_HPROF` in PL/SQL.

**The DBMS_TRACE package**

`DBMS_TRACE` is a built-in package in Oracle to enable and disable tracing in sessions. As soon as a program is executed in a trace enabled session, the server captures and logs the information in trace log tables. The `dbmspbt.sql` and `prvtpbt.sql` table scripts are available in the database installation folder. The trace tables can be analysed to review the execution flow of the PL/SQL program and take decisions in accordance.

**Installing DBMS_TRACE**

If the `DBMS_TRACE` package is not installed at the server, it can be installed by running the following scripts from the database installation folder:

- `$ORACLE_HOME/rdbms/admin/dbmspbt.sql`: This script creates the `DBMS_TRACE` package specification
- `$ORACLE_HOME/rdbms/admin/prvtpbt.plb`: This script creates the `DBMS_TRACE` package body

The scripts must be executed as the `sys` user and in the same order as mentioned.

**DBMS_TRACE subprograms**

The `DBMS_TRACE` subprograms deal with the setting of the trace, getting the trace information, and clearing the trace. While configuring the database for the trace, the trace level must be specified to signify the degree of tracing in the session. The trace level majorly deals with two levels. The first level traces all the events of an action while the other level traces only the actions from those program units which have been compiled with the debug and trace option.

The `DBMS_TRACE` constants are used for setting the trace level. Even the numeric values are available for all the constants, but still the constant names are used in the programs.

---

For More Information:

The summary of `DBMS_TRACE` constants is as follows (refer to the Oracle documentation for more details). Note that all constants are of the `INTEGER` type:

<table>
<thead>
<tr>
<th><code>DBMS_TRACE</code> constant</th>
<th>Default</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>TRACE_ALL_CALLS</code></td>
<td>1</td>
<td>Traces all calls</td>
</tr>
<tr>
<td><code>TRACE_ENABLED_CALLS</code></td>
<td>2</td>
<td>Traces calls which are enabled for tracing</td>
</tr>
<tr>
<td><code>TRACE_ALL_EXCEPTIONS</code></td>
<td>4</td>
<td>Traces all exceptions</td>
</tr>
<tr>
<td><code>TRACE_ENABLED_EXCEPTIONS</code></td>
<td>8</td>
<td>Traces exceptions which are enabled for tracing</td>
</tr>
<tr>
<td><code>TRACE_ALL_SQL</code></td>
<td>32</td>
<td>Traces all SQL statements</td>
</tr>
<tr>
<td><code>TRACE_ENABLED_SQL</code></td>
<td>64</td>
<td>Traces SQL statements which are enabled for tracing</td>
</tr>
<tr>
<td><code>TRACE_ALL_LINES</code></td>
<td>128</td>
<td>Traces each line</td>
</tr>
<tr>
<td><code>TRACE_ENABLED_LINES</code></td>
<td>256</td>
<td>Traces lines which are enabled for tracing</td>
</tr>
<tr>
<td><code>TRACE_PAUSE</code></td>
<td>4096</td>
<td>Pauses tracing (controls tracing process)</td>
</tr>
<tr>
<td><code>TRACE_RESUME</code></td>
<td>8192</td>
<td>Resume tracing (controls tracing process)</td>
</tr>
<tr>
<td><code>TRACE_STOP</code></td>
<td>16384</td>
<td>Stops tracing (controls tracing process)</td>
</tr>
<tr>
<td><code>TRACE_LIMIT</code></td>
<td>16</td>
<td>Limits the trace information (controls tracing process)</td>
</tr>
<tr>
<td><code>TRACE_MINOR_VERSION</code></td>
<td>0</td>
<td>Administer tracing process</td>
</tr>
<tr>
<td><code>TRACE_MAJOR_VERSION</code></td>
<td>1</td>
<td>Administer tracing process</td>
</tr>
<tr>
<td><code>NO_TRACE_ADMINISTRATIVE</code></td>
<td>32768</td>
<td>Prevents tracing of administrative events such as:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• PL/SQL Trace Tool started</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Trace flags changed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• PL/SQL Virtual Machine started</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• PL/SQL Virtual Machine stopped</td>
</tr>
<tr>
<td><code>NO_TRACE_HANDLED_EXCEPTIONS</code></td>
<td>65536</td>
<td>Prevents tracing of handled exceptions</td>
</tr>
</tbody>
</table>

The subprograms contained in the `DBMS_TRACE` package are as follows:

<table>
<thead>
<tr>
<th><code>DBMS_TRACE</code> subprogram</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CLEAR_PLSQL_TRACE</code> procedure</td>
<td>Stops trace data dumping in session</td>
</tr>
<tr>
<td><code>GET_PLSQL_TRACE_LEVEL</code> function</td>
<td>Gets the trace level</td>
</tr>
<tr>
<td><code>GET_PLSQL_TRACE_RUNNUMBER</code> function</td>
<td>Gets the current sequence of execution of trace</td>
</tr>
</tbody>
</table>

[ 342 ]

For More Information:
DBMS_TRACE subprogram | Remarks
---|---
PLSQL_TRACE_VERSION procedure | Gets the version number of the trace package
SET_PLSQL_TRACE procedure | Starts tracing in the current session
COMMENT_PLSQL_TRACE procedure | Includes comment on the PL/SQL tracing
INTERNAL_VERSION_CHECK function | Has a value as 0, if the internal version check has not been done
LIMIT_PLSQL_TRACE procedure | Sets limit for the PL/SQL tracing
PAUSE_PLSQL_TRACE procedure | Pauses the PL/SQL tracing
RESUME_PLSQL_TRACE procedure | Resumes the PL/SQL tracing

In the preceding list, the key subprograms are:

- **SET_PLSQL_TRACE**: It kicks off the PL/SQL tracing session. For example, `DBMS_TRACE.SET_PLSQL_TRACE (DBMS_TRACE.TRACE_ALL_SQL)` traces all SQL in the program.

- **CLEAR_PLSQL_TRACE**: It stops the tracing session.

`PLSQL_TRACE_VERSION` returns the current trace version as the `OUT` parameter value.

--- Trace level that controls the tracing process (stop, pause, resume, and limit) cannot be used in combination with other trace levels

### The PLSQL_DEBUG parameter and the DEBUG option

As a prerequisite, a subprogram can be enabled for tracing only if it is compiled in the debug mode. The `PLSQL_DEBUG` parameter is used to enable a database, session, or a program for debugging. The compilation parameter can be set at `SYSTEM`, `SESSION`, or any specific program level. When set to `TRUE`, the program units are compiled in the interpreted mode for debug purpose. The Oracle server explicitly compiles the program in interpreted mode to use the strengths of a debugger. However, debugging of a natively compiled program unit is not yet supported in the Oracle database. For this reason, native compilation of program units is less preferable than interpreted mode during development.

```
ALTER [SYSTEM | SESSION] SET PLSQL_DEBUG= [TRUE | FALSE]
```

---

For More Information:  
The trace can be enabled at the subprogram level (not for anonymous blocks):

```
ALTER [Procedure | Function | Package] [Name] 
  COMPILE PLSQL_DEBUG= [TRUE | FALSE]
/
```

Or

```
ALTER [Procedure | Function | Package] [Name] COMPILE DEBUG [BODY]
/
```

Enabling tracing at the subprogram level is usually preferred to avoid dumping of huge volume of trace data.

The PLSQL_DEBUG parameter has been devalued in Oracle 11g. When a subprogram is compiled with the PLSQL_DEBUG option set to TRUE in a warning enabled session, the server records the following two warnings:

- PLW-06015: parameter PLSQL_DEBUG is deprecated; use PLSQL_OPTIMIZE_LEVEL = 1
- PLW-06013: deprecated parameter PLSQL_DEBUG forces PLSQL_OPTIMIZE_LEVEL <= 1

### Viewing the PL/SQL trace information

Oracle provides no built-in data dictionary view to query the trace session information. Instead, the trace information is logged into the trace tables. These trace tables can be created by running the `$ORACLE_HOME/rdbms/admin/tracetab.sql` script as SYS user. The script creates the following two tables:

- **PLSQL_TRACE_RUNS**: This table stores execution-specific information. The following structure shows that the table contains the trace header information such as RUNID and comments:
  ```
  /*Describe the PLSQL_TRACE_RUNS table structure*/
  SQL> DESC plsql_trace_runs
  Name                                      Null?    Type
  ----------------------------------------- -------- --------------
  RUNID                                     NOT NULL NUMBER
  RUN_DATE                                           DATE
  RUN_OWNER                                          VARCHAR2(31)
  RUN_COMMENT                                        VARCHAR2(2047)
  RUN_COMMENT1                                       VARCHAR2(2047)
  ```

For More Information:

In the preceding table, **RUNID** is the unique run identifier which derives its value from a sequence, **PLSQL_TRACE_RUNNUMBER**. The **RUN_DATE** and **RUN_END** columns specify the start and end time of the run respectively. The **RUN_SYSTEM_INFO** and **SPARE1** columns are the currently unused columns in the table.

- **PLSQL_TRACE_EVENTS**: This table displays accumulated results from trace executions and captures the detailed trace information:

```sql
/*Describe the PLSQL_TRACE_EVENTS table structure*/
SQL> desc plsql_trace_events
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Null?</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>RUNID</td>
<td>NOT NULL</td>
<td>NUMBER</td>
</tr>
<tr>
<td>EVENT_SEQ</td>
<td>NOT NULL</td>
<td>NUMBER</td>
</tr>
<tr>
<td>EVENT_TIME</td>
<td></td>
<td>DATE</td>
</tr>
<tr>
<td>RELATED_EVENT</td>
<td></td>
<td>NUMBER</td>
</tr>
<tr>
<td>EVENT_KIND</td>
<td></td>
<td>NUMBER</td>
</tr>
<tr>
<td>EVENT_UNIT_DBLINK</td>
<td></td>
<td>VARCHAR2(4000)</td>
</tr>
<tr>
<td>EVENT_UNIT_OWNER</td>
<td></td>
<td>VARCHAR2(31)</td>
</tr>
<tr>
<td>EVENT_UNIT</td>
<td></td>
<td>VARCHAR2(31)</td>
</tr>
<tr>
<td>EVENT_UNIT_KIND</td>
<td></td>
<td>VARCHAR2(31)</td>
</tr>
<tr>
<td>EVENT_LINE</td>
<td></td>
<td>NUMBER</td>
</tr>
<tr>
<td>EVENT_PROC_NAME</td>
<td></td>
<td>VARCHAR2(31)</td>
</tr>
<tr>
<td>STACK_DEPTH</td>
<td></td>
<td>NUMBER</td>
</tr>
<tr>
<td>PROC_NAME</td>
<td></td>
<td>VARCHAR2(31)</td>
</tr>
<tr>
<td>PROC_DBLINK</td>
<td></td>
<td>VARCHAR2(4000)</td>
</tr>
<tr>
<td>PROC_OWNER</td>
<td></td>
<td>VARCHAR2(31)</td>
</tr>
<tr>
<td>PROC_UNIT</td>
<td></td>
<td>VARCHAR2(31)</td>
</tr>
<tr>
<td>PROC_UNIT_KIND</td>
<td></td>
<td>VARCHAR2(31)</td>
</tr>
<tr>
<td>PROC_LINE</td>
<td></td>
<td>NUMBER</td>
</tr>
<tr>
<td>PROC_PARAMS</td>
<td></td>
<td>VARCHAR2(2047)</td>
</tr>
<tr>
<td>ICD_INDEX</td>
<td></td>
<td>NUMBER</td>
</tr>
<tr>
<td>USER_EXCP</td>
<td></td>
<td>NUMBER</td>
</tr>
<tr>
<td>EXCP</td>
<td></td>
<td>NUMBER</td>
</tr>
<tr>
<td>EVENT_COMMENT</td>
<td></td>
<td>VARCHAR2(2047)</td>
</tr>
<tr>
<td>MODULE</td>
<td></td>
<td>VARCHAR2(4000)</td>
</tr>
<tr>
<td>ACTION</td>
<td></td>
<td>VARCHAR2(4000)</td>
</tr>
<tr>
<td>CLIENT_INFO</td>
<td></td>
<td>VARCHAR2(4000)</td>
</tr>
</tbody>
</table>

For More Information:
CLIENT_ID                                    VARCHAR2 (4000)
ECID_ID                                      VARCHAR2 (4000)
ECID_SEQ                                     NUMBER
CALLSTACK                                    CLOB
ERRORSTACK                                   CLOB

The following points can be noted about this table:

° The RUNID column references the RUNID column of the PLSQL_TRACE_RUNS table
° EVENT_SEQ is the unique event identifier within a single run
° The EVENT_UNIT, EVENT_UNIT_KIND, EVENT_UNIT_OWNER, and EVENT_LINE columns capture the program unit information (such as name, type, owner, and line number) which initiates the trace event
° The PROC_NAME, PROC_UNIT, PROC_UNIT_KIND, PROC_OWNER, and PROC_LINE columns capture the procedure information (such as name, type, owner, and line number) which is currently being traced
° The EXCP and USER_EXCP columns apply to the exceptions occurring during the trace
° The EVENT_COMMENT column gives user defined comment or the actual event description
° The MODULE, ACTION, CLIENT_INFO, CLIENT_ID, ECID_ID, and ECID_SEQ columns capture information about the session running on a SQL*Plus client
° The CALLSTACK and ERRORSTACK columns store the call stack information

Once the script has been executed, the DBA should create public synonyms for the tables and sequence in order to be accessed by all users.

/*Connect as SYSDBA*/
Conn sys/system as SYSDBA
Connected.

/*Create synonym for PLSQL_TRACE_RUNS*/
CREATE PUBLIC SYNONYM plsql_trace_runs FOR plsql_trace_runs
/

Synonym created.

/*Create synonym for PLSQL_TRACE_EVENTS*/
CREATE PUBLIC SYNONYM plsql_trace_events FOR plsql_trace_events
/

For More Information:
Synonym created.

/*Create synonym for PLSQL_TRACE_RUNNUMBER sequence*/
CREATE PUBLIC SYNONYM plsql_trace_runnumber FOR plsql_trace_runnumber
/

Synonym created.

/*Grant privileges on the PLSQL_TRACE_RUNS*/
GRANT select, insert, update, delete ON plsql_trace_runs TO PUBLIC
/

Grant succeeded.

/*Grant privileges on the PLSQL_TRACE_EVENTS*/
GRANT select, insert, update, delete ON plsql_trace_events TO PUBLIC
/

Grant succeeded.

/*Grant privileges on the PLSQL_TRACE_RUNNUMBER*/
GRANT select ON plsql_trace_runnumber TO PUBLIC
/

Grant succeeded.

Demonstrating the PL/SQL tracing

PL/SQL tracing is demonstrated in the following steps:

1. The F_GET_LOC function looks as follows (this function has been already created in the schema):

/*Connect as ORADEV user*/
Conn ORADEV/ORADEV
Connected.

/*Create the function*/
CREATE OR REPLACE FUNCTION F_GET_LOC (P_EMPNO NUMBER)
RETURN VARCHAR2
IS

For More Information:
/*Cursor select location for the given employee*/
CURSOR C_DEPT IS
  SELECT d.loc
  FROM employees e, departments d
  WHERE e.deptno = d.deptno
  AND e.empno = P_EMPNO;
  l_loc VARCHAR2(100);
BEGIN
  /*Cursor is open and fetched into a local variable*/
  OPEN C_DEPT;
  FETCH C_DEPT INTO l_loc;
  CLOSE C_DEPT;

  /*Location returned*/
  RETURN l_loc;
END;
/

Function created.

We will trace the execution path for the preceding function.

2. Recompile the F_GET_LOC function for tracing:
   /*Compile the function in debug mode*/
   SQL> ALTER FUNCTION F_GET_LOC COMPILE DEBUG
   /
   Function altered.

3. Start the tracing session to trace all calls:
   BEGIN
   /*Enable tracing for all calls in the session*/
   DBMS_TRACE.SET_PLSQL_TRACE(DBMS_TRACE.TRACE_ALL_CALLS);
   END;
   /

   Specify additional trace levels using the + sign as:
   DBMS_TRACE.SET_PLSQL_TRACE (tracelevel1 + tracelevel2 ...)

For More Information:
4. Execute the function and capture the result into a bind variable:

/*Declare a SQLPLUS environment variable*/
SQL> VARIABLE M_LOC VARCHAR2(100);

/*Execute the function and assign the return output to the variable*/
SQL> EXEC :M_LOC := F_GET_LOC (7369);

PL/SQL procedure successfully completed.

/*Print the variable*/
SQL> PRINT M_LOC

M_LOC
-----------------------------------------
DALLAS

5. Stop the trace session:

BEGIN
/*Stop the trace session*/
    DBMS_TRACE.CLEAR_PLSQL_TRACE;
END;
/

6. Query the trace log tables.

Query the PLSQL_TRACE_RUNS table to retrieve the current RUNID:

/*Query the PLSQL_TRACE_RUNS table*/
SELECT runid, run_owner, run_date
FROM plsql_trace_runs
ORDER BY runid
/

<table>
<thead>
<tr>
<th>RUNID</th>
<th>RUN_OWNER</th>
<th>RUN_DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ORADEV</td>
<td>29-JAN-12</td>
</tr>
</tbody>
</table>

Query the PLSQL_TRACE_EVENTS table to retrieve the trace events for the RUNID as 1.

The highlighted portion shows the tracing of execution of the F_GET_LOC function. The trace events appearing before and after the highlighted portion represent the starting and stopping of the trace session.

/*Query the PLSQL_TRACE_EVENTS table*/
SELECT runid,

For More Information:
event_comment,  
    event_unit_owner,  
    event_unit,  
    event_unit_kind,  
    event_line  
FROM plsql_trace_events  
WHERE runid = 1  
ORDER BY event_seq
/

The output of the preceding query is shown in the following screenshot:

<table>
<thead>
<tr>
<th>RUNID</th>
<th>EVENT_COMMENT</th>
<th>EVENT_UNIT</th>
<th>EVENT_UNIT</th>
<th>EVENT_UNIT_KIND</th>
<th>EVENT_LINE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PL/SQL Trace Tool started</td>
<td>EVC</td>
<td>DBMS_TRACE</td>
<td>PACKAGE BODY</td>
<td>21</td>
</tr>
<tr>
<td>1</td>
<td>Trace flags changed</td>
<td></td>
<td></td>
<td></td>
<td>76</td>
</tr>
<tr>
<td>1</td>
<td>Return from procedure call</td>
<td>EVC</td>
<td>DBMS_TRACE</td>
<td>PACKAGE BODY</td>
<td>81</td>
</tr>
<tr>
<td>1</td>
<td>Return from procedure call</td>
<td>EVC</td>
<td>DBMS_TRACE</td>
<td>PACKAGE BODY</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>PL/SQL Virtual Machine started</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Procedure Call</td>
<td></td>
<td>&lt;anonymous&gt;</td>
<td>ANONYMOUS BLOCK</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>Procedure Call</td>
<td></td>
<td>&lt;anonymous&gt;</td>
<td>ANONYMOUS BLOCK</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>Procedure Call</td>
<td>ORDEU</td>
<td>F_GET_LOC</td>
<td>FUNCTION</td>
<td>13</td>
</tr>
<tr>
<td>1</td>
<td>Procedure Call</td>
<td>ORDEU</td>
<td>F_GET_LOC</td>
<td>FUNCTION</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>Procedure Call</td>
<td>ORDEU</td>
<td>F_GET_LOC</td>
<td>FUNCTION</td>
<td>19</td>
</tr>
<tr>
<td>1</td>
<td>PL/SQL Virtual Machine stopped</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Procedure Call</td>
<td>&lt;anonymous&gt;</td>
<td>ANONYMOUS</td>
<td>BLOCK</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>Procedure Call</td>
<td>&lt;anonymous&gt;</td>
<td>ANONYMOUS</td>
<td>BLOCK</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>Procedure Call</td>
<td>EVC</td>
<td>DBMS_TRACE</td>
<td>PACKAGE BODY</td>
<td>94</td>
</tr>
<tr>
<td>1</td>
<td>Procedure Call</td>
<td>EVC</td>
<td>DBMS_TRACE</td>
<td>PACKAGE BODY</td>
<td>72</td>
</tr>
<tr>
<td>1</td>
<td>Procedure Call</td>
<td>EVC</td>
<td>DBMS_TRACE</td>
<td>PACKAGE BODY</td>
<td>12</td>
</tr>
<tr>
<td>1</td>
<td>Procedure Call</td>
<td>EVC</td>
<td>DBMS_TRACE</td>
<td>PACKAGE BODY</td>
<td>57</td>
</tr>
<tr>
<td>1</td>
<td>Procedure Call</td>
<td>EVC</td>
<td>DBMS_TRACE</td>
<td>PACKAGE BODY</td>
<td>75</td>
</tr>
<tr>
<td>21 rows selected.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The query output shows the F_GET_LOC function execution flow starting from the time the trace session started (EVENT_COMMENT = PL/SQL Trace Tool started) till the trace session was stopped (EVENT_COMMENT = PL/SQL trace stopped).

Profiling the PL/SQL programs

We just saw tracing capabilities in PL/SQL programs. It presents the execution flow of the program in an interactive format with clear comments at each stage. But it doesn’t provide the execution statistics of the program which prevents the user from determining the performance of a program. The user never comes to know about the time consumed at each step or process.

Before the release of Oracle 11g, DBMS_PROFILER was used as the primary tool for profiling PL/SQL programs.
Oracle hierarchical profiler—the DBMS_HPROF package

Oracle introduced the PL/SQL hierarchical profiler in Oracle 11g release 1. The profiling was restructured as hierarchical profiling. The hierarchical profiling could profile even the subprogram calls made in the PL/SQL code. It fills the gap between tracing loopholes and the expectations of performance tracing. The hierarchical profiler creates the dynamic execution profile of a PL/SQL program. The efficiencies of the hierarchical profiler are as follows:

- Distinct reporting for SQL and PL/SQL time consumption.
- Reports count of distinct subprograms calls made in the PL/SQL code and the time spent with each subprogram call.
- Multiple interactive analytics reports in HTML format using the command line utility.
- More efficient than other tracing utilities and offers more powerful profiling than a conventional profiler. The conventional DBMS_PROFILER tracks the performance at a lower level (individual line of programs) while DBMS_HPROF tracks the cumulative performance of a program unit.

The DBMS_HPROF package implements hierarchical profiling. It is a SYS owned Oracle built-in package whose subprograms profile the PL/SQL code execution.

The PL/SQL hierarchical profiler consists of two subcomponents. The two components—Data collector and Analyzer—are indicative of the two-step hierarchical profiling process.

The Data collector component is the "worker" component which initiates the profiling process, collects all the raw profiler data from the PL/SQL code execution, and stops. The raw profiler data is dumped into a system-based text file for further analysis. In simple words, it stakes itself to prepare the stage for the Analyzer component.

The Analyzer component takes the raw profiler data and loads it into the profiler log tables. The effort of the component lies in understanding the raw profiler data and placing it correctly in the profiler tables. Conceptually, the Analyzer component lives the same life cycle as that of an ETL (Extraction, Transformation, and Loading) process.

For More Information:
The following table shows the `DBMS_HPROF` subprograms:

<table>
<thead>
<tr>
<th>Subprogram</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANALYZE function</td>
<td>Analyzes the raw profiler output and produces hierarchical profiler information in database tables</td>
</tr>
<tr>
<td>START_PROFILING</td>
<td>Starts hierarchical profiler data collection in the user's session</td>
</tr>
<tr>
<td>STOP_PROFILING</td>
<td>Stops profiler data collection in the user's session</td>
</tr>
</tbody>
</table>

In the preceding subprograms list, the `START_PROFILING` and `STOP_PROFILING` procedures come under the Data collector component while the subprogram `ANALYZE` is a sure selection under the Analyzer component.

The DBA must grant the `EXECUTE` privilege to the user who intends to perform profiling activity.

**View profiler information**

Similar to the trace log tables, Oracle 11g has facilitated the profiler with relational tables to log the analyzed profiler data. The profiler log tables can be created by running the `$ORACLE_HOME/rdbms/admin/dbmshptab.sql` script. On execution of this script, the following three tables are created:

- **DBMSHP_RUNS**: This table maintains the flat information about each command executed during profiling
- **DBMSHP_FUNCTION_INFO**: This table contains information about the profiled function
- **DBMSHP_PARENT_CHILD_INFO**: This table contains parent-child profiler information

The script execution might raise some exceptions which can be ignored for the first time. Once the script is executed and tables are created, the DBA must grant a `SELECT` privilege on these tables to the users.

**Demonstrating the profiling of a PL/SQL program**

The following steps demonstrate the profiling of a PL/SQL stored function, `F_GET_LOC`:

1. Create a directory to create a trace file for raw profiler data:
   ```sql
   /*Connect as sysdba*/
   Conn sys/system as sysdba  
   Connected.
   ```

For More Information:
/*Create directory where raw profiler data would be stored*/
SQL> CREATE DIRECTORY PROFILER_REP AS 'C:\PROFILER\'
/
Directory created.

/*Grant read, write privilege on the directory to ORADEV*/
SQL> GRANT READ, WRITE ON DIRECTORY PROFILER_REP TO ORADEV
/
Grant succeeded.

/*Grant execute privilege on DBMS_HPROF package to ORADEV*/
SQL> GRANT EXECUTE ON DBMS_HPROF TO ORADEV
/
Grant succeeded.

/*Grant SELECT privilege on DBMSHP_RUNS to ORADEV*/
SQL> GRANT select on DBMSHP_RUNS to ORADEV
/
Grant succeeded.

/*Grant SELECT privilege on DBMSHP_FUNCTION_INFO to ORADEV*/
SQL> GRANT select on DBMSHP_FUNCTION_INFO to ORADEV
/
Grant succeeded.

/*Grant SELECT privilege on DBMSHP_PARENT_CHILD_INFO to ORADEV*/
SQL> GRANT select on DBMSHP_PARENT_CHILD_INFO to ORADEV
/
Grant succeeded.

2. Start the profiling:
   /*Connect to ORADEV*/
Conn ORADEV/ORADEV
Connected.

BEGIN
/*Start the profiling*/
/*Specify the directory and file name*/
Profi ling and Tracing PL/SQL Code

        DBMS_HPROF.START_PROFILING ('PROFILER_REP', 'F_GET_LOC.TXT');
        END;
        /

        PL/SQL procedure successfully completed.

        max_depth is the third parameter of the START_PROFILING
subprogram which can be used to limit recursive subprogram
 calls. By default, it is NULL.

3. Execute the F_GET_LOC function:
   /*Declare a SQLPLUS environment variable*/
   SQL> VARIABLE M_LOC VARCHAR2(100);
   /*Execute the function and assign the return output to the
variable*/
   SQL> EXEC :M_LOC := F_GET_LOC (7369);

   PL/SQL procedure successfully completed.

   /*Print the variable*/
   SQL> PRINT M_LOC
   M_LOC
   -------------------------------
   DALLAS

4. Stop the profiling
   BEGIN
     /*Stop the profiling */
     DBMS_HPROF.STOP_PROFILING;
     END;
     /

     PL/SQL procedure successfully completed.

5. Check the PROFILER_REP database directory. A text file, F_GET_LOC.txt, has
been created with the raw profiler content. A small screen cast of the raw
profiler data is as follows:

P#V PLSHPROF Internal Version 1.0
P#! PL/SQL Timer Started
P#C PLSQL."".""."".__plsql_vm"
P#X 7
P#C PLSQL."".""."".__anonymous_block"

For More Information:
From the preceding sample of raw profiler data, one can get clear indications for the following:

- **Namespace distinction at each line as SQL or PLSQL**
- **Operations captured by the hierarchical profiler as follows:**
  - `_anonymous_block` indicates anonymous block execution
  - `_dyn_sql_exec_line#` indicates dynamic SQL statement execution at line#
  - `_pkg_init` indicates PL/SQL package initialization
  - `_plsql_vm` indicates PL/SQL virtual machine call
  - `_sql_fetch_line#` indicates fetch operation at line#
  - `_static_sql_exec_line#` indicates static SQL execution at line#
- **Each line starts with an encrypted indication as P#X, P#C. Let us briefly understand what they indicate:**
  - P#C is the call event which indicates a subprogram call
  - P#R is the return event which indicates a "return" from a subprogram
  - P#X shows the time consumed between the two subprogram calls
  - P#! is the comment which appears in the analyzer's output

However, the raw profile doesn't appear to be a comprehensive one which can be interpreted fast and easily. This leads to the need for an analyzer which can translate the raw data into a meaningful form. The Analyzer component of HPROF can reform the raw profiler data into accessible form. The raw profiler text file would be interpreted and loaded into profiling log tables.

Note that until Step 5, the Data collector component of the hierarchical profiler was active. The raw profiler data has been collected and recorded in a text file.

---

**For More Information:**

6. Execute the `ANALYZE` subprogram to insert the data into profiler tables.

```sql
/*Connect as DBA*/
Conn sys/system as sysdba
Connected.

/*Start the PL/SQL block*/
DECLARE
  l_runid  NUMBER;
BEGIN

/*Invoke the analyzer API*/
  l_runid := DBMS_HPROF.analyze
  (location    => 'PROFILER_REP',
   FILENAME     => 'F_GET_LOC.txt',
   run_comment  => 'Analyzing the execution of F_GET_LOC');

  DBMS_OUTPUT.put_line('l_runid=' || l_runid);
END;
/
PL/SQL procedure successfully completed
```

If profiling is enabled for a session and the trace file contains a huge volume of raw profiler data, you can analyze only selected subprograms by specifying the `TRACE` parameter in the `ANALYZE` API. The following example code snippet shows the usage of the `TRACE` parameter in the `ANALYZER` subprogram. The `MULTIPLE_RAW_PROFILES.txt` trace file contains raw profiler data from multiple profiles. But only the profiles of `F_GET_SAL` and `F_GET_JOB` can be analyzed as follows:

```sql
DECLARE
  l_runid NUMBER;
BEGIN
  l_runid := dbms_hprof.analyze
             (location => 'PROFILER_REP',
              filename=> 'MULTIPLE_RAW_PROFILES.txt',
              trace => '"F_GET_SAL"."F_GET_JOB"');
end;
/
```

For More Information:
7. Query the profiling log tables

/*Query the DBMSHP_RUNS table*/
SELECT runid, total_elapsed_time, run_comment
FROM dbmshp_runs
ORDER BY runid
/

<table>
<thead>
<tr>
<th>RUNID</th>
<th>TOTAL_ELAPSED_TIME</th>
<th>RUN_COMMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>106407</td>
<td>Analyzing the execution of F_GET_LOC</td>
</tr>
</tbody>
</table>

In the preceding query result, note that TOTAL_ELAPSED_TIME is the total execution time (in micro seconds) for the procedure. The run comment appears as per the input given during analysis.

/*Query the DBMSHP_FUNCTION_INFO table*/
SELECT runid, owner, module, type, function, namespace, function_elapsed_time, calls
FROM dbmshp_function_info
WHERE runid = 1

The output of the preceding query is shown in the following screenshot:

Here, we see how the analyzer output clearly indicates the step-by-step execution profile of a PL/SQL program. It shows which engine (namespace) was employed on which call event along with the time consumed at each event.

The plshprof utility

The analyzer component simplifies much of the problem by interpreting the raw profiler data and loading it into the database tables. What more can one expect? But the services of hierarchical profiler don't end here. The correct analysis of the profiler data is as important as the interpretation of data. For this purpose, a command-line tool has been provided which can generate multiple reports in HTML format.

plshprof is a command-line utility which reads the raw profiler data and generates multiple HTML reports. Each report builds up and showcases a new frame of analysis and offers better statistical foresight to the user. The sixteen reports generated can be navigated from the main report page.
The plshprof utility can be executed as follows:

```
C:\Profiler path> plshprof –output [HTML FILE] [RAW PROFILER DATA]
```

Let us now generate the HTML report of the profiler data which we derived above:

```
C:\>cd profiler

C:\profiler>plshprof -output F_GET_LOC F_GET_LOC.TXT
PLSHPROF: Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Production
[7 symbols processed]
[Report written to 'F_GET_LOC.html']

C:\profiler>
```

As soon as the plshprof utility process is over, the following HTML files are generated at the directory location:

- F_GET_LOC.html
- F_GET_LOC_2c.html
- F_GET_LOC_2f.html
- F_GET_LOC_2n.html
- F_GET_LOC_fn.html
- F_GET_LOC_md.html
- F_GET_LOC_md.html
- F_GET_LOC_ms.html
- F_GET_LOC_nsc.html
- F_GET_LOC_nsf.html
- F_GET_LOC_nsp.html
- F_GET_LOC_pc.html
- F_GET_LOC_tc.html
- F_GET_LOC_td.html
- F_GET_LOC_tf.html
- F_GET_LOC_ts.html

For More Information:

Chapter 11

Here, `F_GET_LOC.html` is the main index file which contains navigational links to all other reports. The main index page is shown in the following screenshot:

**PL/SQL Elapsed Time (microsecs) Analysis**

106407 microseconds (elapsed time) & 9 function calls

The PL/SQL Hierarchical Profiler produces a collection of reports that present information derived from the profiler's output log in a variety of forms. The following reports have been found to be the most generally useful as starting points for browsing:

- Function Elapsed Time (microsecs) Data sorted by Total Subtree Elapsed Time (microsecs)
- Function Elapsed Time (microsecs) Data sorted by Total Function Call Count
- Function Elapsed Time (microsecs) Data sorted by Total Function Call Size
- Function Elapsed Time (microsecs) Data sorted by Most Function Elapsed Time (microsecs)
- Function Elapsed Time (microsecs) Data sorted by Most Function Call Count
- Function Elapsed Time (microsecs) Data sorted by Most Function Call Size
- Function Elapsed Time (microsecs) Data sorted by Most Function Call Size
- Function Elapsed Time (microsecs) Data sorted by Most Function Call Size
- Function Elapsed Time (microsecs) Data sorted by Most Function Call Size

In addition, the following reports are also available:

- Function Elapsed Time (microsecs) Data sorted by Function Name
- Function Elapsed Time (microsecs) Data sorted by Total Subtree Elapsed Time (microsecs)
- Function Elapsed Time (microsecs) Data sorted by Total Function Call Count
- Function Elapsed Time (microsecs) Data sorted by Total Function Call Size
- Function Elapsed Time (microsecs) Data sorted by Most Subtree Elapsed Time (microsecs)
- Function Elapsed Time (microsecs) Data sorted by Most Function Elapsed Time (microsecs)
- Function Elapsed Time (microsecs) Data sorted by Most Function Call Count
- Function Elapsed Time (microsecs) Data sorted by Most Function Call Size
- Function Elapsed Time (microsecs) Data sorted by Most Function Call Size
- Function Elapsed Time (microsecs) Data sorted by Most Function Call Size

Sample reports

In this section, we will overview some important reports:

- **Function Elapsed Time (microsecs) Data sorted by Total Subtree Elapsed Time (microsecs):** The report provides the flat view of raw profiler data. It includes total call count, self time, subtree time, and descendants of each function:

```
<table>
<thead>
<tr>
<th>Subtree</th>
<th>Function</th>
<th>Procedure</th>
<th>Total</th>
<th>Subtree Elapsed Time (microsecs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>106407</td>
<td>106408</td>
<td></td>
<td>106408</td>
<td>106407</td>
</tr>
<tr>
<td>106408</td>
<td>106409</td>
<td></td>
<td>106409</td>
<td>106408</td>
</tr>
<tr>
<td>106409</td>
<td>106410</td>
<td></td>
<td>106410</td>
<td>106409</td>
</tr>
<tr>
<td>106410</td>
<td>106411</td>
<td></td>
<td>106411</td>
<td>106410</td>
</tr>
<tr>
<td>106411</td>
<td>106412</td>
<td></td>
<td>106412</td>
<td>106411</td>
</tr>
</tbody>
</table>
```

For More Information:

Profiling and Tracing PL/SQL Code

- Function Elapsed Time (microsecs) Data sorted by Total Function Elapsed Time (microsecs): This is the module-level summary report which shows the total time spent in each module and the total calls to the functions in the module:

  ![Function Elapsed Time Data](image)

- Namespace Elapsed Time (microsecs) Data sorted by Namespace: This report provides the distribution of time spent by the PL/SQL engine and SQL engine separately. SQL and PLSQL are the two namespace categories available for a block. It is very useful in reducing the disk I/O and hence enhancing the block performance. The net sum of the distribution is always 100 percent:

  ![Namespace Elapsed Time Data](image)

Likewise, other reports also reveal and present some important statistics for the PL/SQL code execution.

For More Information:
Summary
In this chapter, we learned the tracing and profiling features of Oracle 11g. While the tracing feature tracks the execution path of PL/SQL code, the profiling feature reports the time consumed at each subprogram call or line number. We demonstrated the implementation and analysis of tracing and profiling features.

In the next chapter, we will see how to identify vulnerable areas in a PL/SQL code and safeguard them against injective attacks.

Practice exercise
1. Which component of the PL/SQL hierarchical profiler uploads the result of profiling into database tables?
   a. The Profiler component
   b. The Analyzer component
   c. The shared library component
   d. The Data collector component

2. The plshprof utility is a SQL utility to generate a HTML profiler report from profiler tables in the database.
   a. True
   b. False

3. Suppose that you are using Oracle 11g Release 2 express edition and you issue the following command:
   
   ALTER SESSION SET PLSQL_WARNINGS = 'ENABLE:ALL'
   /
   Session altered.
   ALTER FUNCTION FUNC COMPILE PLSQL_DEBUG=TRUE
   /
   Function altered.
Determine the output of the following SELECT statement

```
SELECT * FROM USER_ERRORS
/
```

a. No output
b. PLW-06015: parameter PLSQL_DEBUG is deprecated; use
   PLSQL_OPTIMIZE_LEVEL = 1

c. PLW-06013: deprecated parameter PLSQL_DEBUG forces
   PLSQL_OPTIMIZE_LEVEL <= 1

d. Both b and c

4. Identify the trace log tables:
   a. PLSQL_TRACE
   b. PLSQL_TRACE_ACTIONS
   c. PLSQL_TRACE_EVENTS
   d. PLSQL_TRACE_INFO

5. Identify the correct trace level combination from the following options
   a. DBMS_TRACE.SET_PLSQL_TRACE
      (DBMS_TRACE.TRACE_ALL_CALLS+DBMS_TRACE.TRACE_ALL_EXCEPTIONS);
   b. DBMS_TRACE.SET_PLSQL_TRACE
      (DBMS_TRACE.TRACE_ALL_SQL+DBMS_TRACE.TRACE_ALL_EXCEPTIONS);
   c. DBMS_TRACE.SET_PLSQL_TRACE
      (DBMS_TRACE.TRACE_ALL_LINES+DBMS_TRACE.TRACE_PAUSE);
   d. DBMS_TRACE.SET_PLSQL_TRACE
      (DBMS_TRACE.TRACE_ALL_EXCEPTIONS+DBMS_TRACE.TRACE_STOP);

6. From the following options, choose the correct statements about the
   plshprof utility:
   a. It is a command line utility.
   b. It generates the HTML reports from the raw profiler data.
   c. It is a SQL command to load the raw profiler data into profiler log tables.
   d. The utility was available with DBMS_PROFILER.

For More Information:
Chapter 11

7. You issue the following command to analyze the profiler output:

```sql
begin
  :r := dbms_hprof.analyze(
      location=> 'DIR',
      filename=> 'xyz.trc',
      trace => '"FUNC1"."FUNC2"."FUNC3"'
  );
end;
```

Choose the correct option:

a. The Analyzer component cannot trace multiple subprograms.
b. The Analyzer component can trace only text (.txt) files.
c. The Analyzer component analyzes the raw profiler data in xyz.trc and loads the data into profiler tables.
d. The trace file can contain profile information of only one subprogram.

8. The `max_depth` parameter specified the limit of recursive calls in `START_PROFILING`.

   a. True
   b. False

For More Information:
Where to buy this book


Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.