



Professional Expertise Distilled

Oracle Advanced PL/SQL Developer Professional Guide

Master advanced PL/SQL concepts along with plenty of example questions for 1Z0-146 examination

Foreword by

Ronald Rood, Oracle ACE, Oracle DBA, OCM

Kamran Agayev A., Oracle ACE, Oracle DBA Expert

Saurabh K. Gupta

[PACKT] enterprise 
PUBLISHING professional expertise distilled


```

Session altered.

/*Alter system to set max cache size*/
ALTER SYSTEM SET RESULT_CACHE_MAX_SIZE = 200M
/

Session altered.

/*Alter system to set max cache results*/
ALTER SYSTEM SET RESULT_CACHE_MAX_RESULT = 20
/

Session altered.

/*Alter system to set cache result retention time*/
ALTER SYSTEM SET RESULT_CACHE_REMOTE_EXPIRATION = 100
/

Session altered.

```

The caching parameter settings can be queried from the V\$PARAMETER dictionary view:

```

SELECT name, value
FROM v$parameter
WHERE name LIKE 'result_cache%'
/

```

NAME	VALUE
result_cache_mode	MANUAL
result_cache_max_size	209715200
result_cache_max_result	100
result_cache_remote_expiration	100

In addition to the initialization parameters shown in the preceding code snippet, Oracle 11g keeps a server process RCBG for Oracle RAC systems. It is used to handle the messages generated by the server processes which are attached to the instances in Oracle RAC architecture:

```

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

/*Query the process details from V$BGPROCESS*/
SQL> SELECT NAME, DESCRIPTION
FROM V$BGPROCESS
WHERE NAME = 'RCBG'
/

```

NAME	DESCRIPTION
RCBG	Result Cache: Background

The DBMS_RESULT_CACHE package

To coordinate the result cache activities at the server, Oracle 11g introduced a new built-in package `DBMS_RESULT_CACHE`. The package is owned by the `SYS` user. The package can be used to perform query result cache maintenance activities such as flushing, invalidating cache results dependent on an object, generating the memory report, and checking the cache status.

The public constants used in the package are as follows:

DBMS_RESULT_CACHE constants (reference: Oracle documentation)

<code>STATUS_BYPS</code>	<code>CONSTANT VARCHAR(10) := 'BYPASS';</code>
<code>STATUS_DISA</code>	<code>CONSTANT VARCHAR(10) := 'DISABLED';</code>
<code>STATUS_ENAB</code>	<code>CONSTANT VARCHAR(10) := 'ENABLED';</code>
<code>STATUS_SYNC</code>	<code>CONSTANT VARCHAR(10) := 'SYNC';</code>

The subprograms used in the package are described in the following table:

DBMS_RESULT_CACHE subprograms (reference: Oracle documentation)

<code>BYPASS</code> procedure	Sets the bypass mode for the result cache
<code>FLUSH</code> function and procedure	Attempts to remove all the objects from the result cache, and depending on the arguments retains or releases the memory and retains or clears the statistics
<code>INVALIDATE</code> functions and procedures	Invalidates all the result-set objects that are dependent upon the specified dependency object
<code>INVALIDATE_OBJECT</code> functions and procedures	Invalidates the specified result-set object(s)
<code>MEMORY_REPORT</code> procedure	Produces the memory usage report for the result cache
<code>STATUS</code> function	Checks the status of the result cache

For illustration, the cache memory report can be generated using the `MEMORY_REPORT` procedure, as shown in the following code snippet. Note that the cache size specifications are expressed as a percentage of shared pool:

```
/*Connect to sysdba*/
SQL> conn sys/system as sysdba
Connected.

/*Enable the serveroutput variable to display the block messages*/
SQL> SET SERVEROUTPUT ON
```

```

/*Generate the cache memory report*/
SQL> exec dbms_result_cache.memory_report
R e s u l t   C a c h e   M e m o r y   R e p o r t
[Parameters]
Block Size           = 1K bytes
Maximum Cache Size  = 200M bytes (200K blocks)
Maximum Result Size = 40M bytes (40K blocks)
[Memory]
Total Memory = 9460 bytes [0.004% of the Shared Pool]
... Fixed Memory = 9460 bytes [0.004% of the Shared Pool]
... Dynamic Memory = 0 bytes [0.000% of the Shared Pool]

PL/SQL procedure successfully completed.

```

Since no results have been cached until now, the report shows 0 bytes for dynamic memory. The cache memory report, shown in the preceding code snippet, also serves as a confirmation for the feature being enabled successfully on the database server. In case the configuration is not proper, the memory report simply displays the following message:

```

SQL> EXEC dbms_result_cache.memory_report
R e s u l t   C a c h e   M e m o r y   R e p o r t
Cache is disabled.

PL/SQL procedure successfully completed.

```

We can check the current status of cache on the server by using the `STATUS` function as follows:

```

SQL> SELECT dbms_result_cache.status FROM DUAL
/

STATUS
-----
ENABLED

```

Implementing the result cache in SQL

As we learned earlier, the database must be configured to enable server-side result caching. Let us now go through illustrations of the result cache in SQL.

Manual result cache

If the result cache operation mode is set as `MANUAL`, the caching feature is known as **manual result cache**. Here, the user has to explicitly specify the `RESULT_CACHE` hint in order to cache the query result. The Oracle server would not automatically cache any result set.

The `RESULT_CACHE_MODE` parameter can be set by the DBA to enable manual result caching:

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

/*Set the parameter as Manual*/
ALTER SYSTEM SET RESULT_CACHE_MODE=MANUAL
/

System altered.
```

We will flush the cache memory and shared pool to clear all the earlier cached results:

```
/*Flush all the earlier cached results*/
SQL> EXEC DBMS_RESULT_CACHE.FLUSH
/

PL/SQL procedure successfully completed.

/*Flush the shared pool*/
SQL> alter system flush shared_pool
/

System altered.
```

Once the cache operation mode is set, the SQL query can be executed using the `RESULT_CACHE` optimizer hint. The hint instructs the server to cache the results of the particular query in the cache component of the memory:

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

/*Execute the query to get SMITH's salary*/
SQL> SELECT /*+RESULT_CACHE*/ sal
        FROM employees
        WHERE EMPNO = 7369
/

      SAL
-----
      800
```

Generate the explain plan for the SQL query:

```
/*Generate the explain plan for the query*/
SQL> EXPLAIN PLAN FOR
      SELECT /*+RESULT_CACHE*/ sal
      FROM employees
```

```

WHERE empno = 7369
/
Explained.

```

Query the explain plan from PLAN_TABLE:

```

/*Check the Explain plan*/
SQL> SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY)
/

```

Refer to the following screenshot for the output of the preceding code block as follows:

```

PLAN_TABLE_OUTPUT
-----
Plan hash value: 833960231
-----
| Id | Operation | Name | Rows | Bytes | Cost (<CPU)| Time |
-----|-----|-----|-----|-----|-----|-----|
| 0 | SELECT STATEMENT | | 1 | 8 | 1 <0> | 00:00:01 |
| 1 | RESULT CACHE | 6jwjx7ap21w71g658hxt5vwng3 | 1 | 8 | 1 <0> | 00:00:01 |
| 2 | TABLE ACCESS BY INDEX ROWID | EMPLOYEES | 1 | 8 | 0 <0> | 00:00:01 |
|* 3 | INDEX UNIQUE SCAN | SYS_C0014587 | 1 | | 0 <0> | 00:00:01 |
-----

```

Note the RESULT_CACHE operation in the explain plan. It implies that the Oracle server has stored the query results in the cache memory with the cache ID 6jwjx7ap21w71g658hxt5vwng3. The same cache ID would be used for the further re-execution of the same SQL.

In addition, we have a new section under PLAN_TABLE_OUTPUT as result cache information (identified by operation ID). This section represents the result cache metadata for this query. The result cache report contains the dependent tables information.

Automatic result cache

If the result cache operation mode is FORCE, the caching feature becomes automatic and the server strictly caches results of all the queries executed. The RESULT_CACHE hint is obsolete and ineffective at the time of automatic result caching. For any query (which is rarely executed), a NO_RESULT_CACHE hint can be specified to override the server operation mode and ignore the query results for caching.

Let us check out how it works:

```

/*Connect as sysdba*/
Conn sys/system as sysdba
Connected

/*Alter the system to set the new cache mode*/
ALTER SYSTEM SET RESULT_CACHE_MODE=FORCE

```

```
/
System altered.
/*Flush all the earlier cached results*/
SQL> EXEC DBMS_RESULT_CACHE.FLUSH
/
PL/SQL procedure successfully completed.
/*Flush the shared pool*/
SQL> alter system flush shared_pool
/
System altered.
```

Now, execute the same SQL query without the RESULT_CACHE hint:

```
/*Connect as USER*/
SQL> CONN ORADEV/ORADEV
Connected.

/*Execute the below SQL to generate the explain plan for the query*/
SQL> EXPLAIN PLAN FOR
  SELECT sal
  FROM employees
  WHERE empno=7369
/
Explained.

/*Query the Explain plan*/
SQL> SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY)
/
```

Refer to the following screenshot for the output of the preceding code block:

```
PLAN_TABLE_OUTPUT
-----
Plan hash value: 833960231

+----+-----+-----+-----+-----+-----+-----+
| Id | Operation              | Name                                     | Rows | Bytes | Cost (%CPU)| Time     |
+----+-----+-----+-----+-----+-----+-----+
|  0 | SELECT STATEMENT        |                                          |      |      |      |      |
|  1 |   RESULT CACHE          | 6jwix7ap21v71g658hxt5vung3            |      |      |      |      |
|  2 |    TABLE ACCESS BY INDEX ROWID | EMPLOYEES                               |      |      |      |      |
|* 3 |      INDEX UNIQUE SCAN   | SYS_C0014587                            |      |      |      |      |
+----+-----+-----+-----+-----+-----+-----+
```

The explain plan shows the RESULT_CACHE component and its cache ID in the cache memory. Note that no hint has been given in the SQL query, but Oracle caches the query results by virtue of its operation mode.

Result cache metadata

Oracle facilitates the monitoring of real-time cache information in the database through result cache dynamic performance views. These views are owned by SYS and content is always in synchronization with the latest database activities.

 Actual dynamic performance views are prefixed with v_\$. Their public synonyms are prefixed with v\$.

The following table enlists the result cache dynamic views along with their purpose:

Synonym	Purpose
V\$RESULT_CACHE_STATISTICS	Records the server cache performance stats, including block count and create count values
V\$RESULT_CACHE_MEMORY	Captures the server cache memory stats (in terms of blocks)
V\$RESULT_CACHE_OBJECTS	Captures the cached result sets information including status
V\$RESULT_CACHE_DEPENDENCY	Captures the dependencies of a result cache

Retrieve the cached result information. The dynamic view V\$RESULT_CACHE_OBJECTS captures the information of the cached results. Some of the accomplishments of this view are as follows:

- A result in the cache memory stores the query result as the `Result` type and caches its dependent object information as the `Dependency` type.
- The results are cached by the user ID as their creator ID.
- The namespace (SQL or PL/SQL) associated with a cached result differentiates the caching from the two components of the server-side result cache.

The `STATUS` column determines the validity status of a cached result. The status of a cached result can be one of the following:

- `NEW`: An under construction cache result
- `PUBLISHED`: A cache result ready to be used
- `INVALID`: An invalid result due to data update or DDL on the dependent object
- `EXPIRED`: An expired cached result that is, the result which has crossed the expiration time

- **BYPASS:** The cached result has been marked for bypass and in bypass mode, the existing cached results are ignored in the query optimizations and new results are not cached

Out of the preceding status list, only the results with the **PUBLISHED** status are the healthiest ones to be used by other SQL queries:

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

```
/*Query the User id of the ORADE user to query its cached results*/
SQL> SELECT user_id
      FROM dba_users
      WHERE username = 'ORADEV'
/
```

```
      USER_ID
-----
          97
```

```
/*Query the cached results*/
SQL> SELECT id, type, status, cache_id
      FROM V$RESULT_CACHE_OBJECTS
      WHERE CREATOR_UID = 97
/
```

```
      ID      TYPE      STATUS      CACHE_ID
-----
      30      Dependency  Published  ORADEV.EMPLOYEES
      31      Result      Published  6jwjx7ap21w71g658hxt5vwng3
```

```
/*Query the SQL associated with the above cached results*/
SQL> SELECT id, name, namespace
      FROM V$RESULT_CACHE_OBJECTS
      WHERE cache_id = '6jwjx7ap21w71g658hxt5vwng3'
/
```

```

ID  NAME
-----
31  SELECT /*+RESULT CACHE*/ SAL FROM EMPLOYEES WHERE EMPNO=7369 SQL
```

Query result cache dependencies

The dependent objects of a given cache result can be queried from the dynamic performance `V$RESULT_CACHE_DEPENDENCY`. The `RESULT_ID` column of the view references the `ID` column of the `Result` type cache entries in `V$RESULT_CACHE_OBJECTS`:

```

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

/*Query result cache dependencies for cache id 31*/
SQL> SELECT *
      FROM V$RESULT_CACHE_DEPENDENCY WHERE
      RESULT_ID = 31
/

  RESULT_ID  DEPEND_ID  OBJECT_NO
-----
           31           30           80571

/*Verify the dependent object in DBA_OBJECTS table*/
SQL> SELECT owner, object_name
      FROM dba_objects
      WHERE object_id = 80571
/

OWNER                                OBJECT_NAME
-----
ORADEV                                EMPLOYEES

```

Cache memory statistics

The current cache memory statistics can be queried from the `V$RESULT_CACHE_STATISTICS` dynamic view. It gives the maximum block count and used block count information.

`Create Count Success` denotes the number of results which are successfully cached in the server result cache.

`Find Count Value` denotes the number of cached results which are successfully used in the repeated executions of the cached queries.

Refer to the following code snippet:

```
/*Cache result characteristics*/
SQL> SELECT *
      FROM V$RESULT_CACHE_STATISTICS
/
```

ID	NAME	VALUE
1	Block Size (Bytes)	1024
2	Block Count Maximum	204800
3	Block Count Current	32
4	Result Size Maximum (Blocks)	40960
5	Create Count Success	1
6	Create Count Failure	0
7	Find Count	0
8	Invalidation Count	0
9	Delete Count Invalid	0
10	Delete Count Valid	0
11	Hash Chain Length	1

11 rows selected.

Invalidation of SQL result cache

The result cached in the server cache gets invalidated if the dependent object gets invalidated or the data in the dependent tables gets updated. The result is still cached in the server cache but marked with the `INVALID` status.

First, we will generate a sample cache result at the server cache:

```
/*Connect as SYSDBA*/
CONN sys/system AS SYSDBA
Connected.

/*Flush the cache memory to clear the earlier cached results*/
EXEC DBMS_RESULT_CACHE.flush;
PL/SQL procedure successfully completed.

/*Flush the shared pool*/
SQL> alter system flush shared_pool
/

System altered.

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

```

/*Generate explain plan for a Query with result cache hint*/
Explain plan for select /*+result_cache*/ * from employees
/

Explained.

/*Check the PLAN_TABLE output*/
SELECT * FROM TABLE (DBMS_XPLAN.DISPLAY)
/

```

Refer to the following screenshot for the output of the preceding code block:

Id	Operation	Name	Rows	Bytes	Cost (<%CPU>)	Time
0	SELECT STATEMENT		14	532	3 (<0>)	00:00:01
1	RESULT CACHE	7j5z4u9hzff4036czuj67bwqnt	14	532	3 (<0>)	00:00:01
2	TABLE ACCESS FULL	EMPLOYEES	14	532	3 (<0>)	00:00:01

Now, the server contains a cached result with the cache ID 7j5z4u9hzff4036czuj67bwqnt. Verify it in the following code snippet:

```

/*Connect as SYSDBA*/
CONN sys/system AS SYSDBA
Connected.

/*Query the result cached at the server*/
SELECT id, type, status, namespace, cache_id
FROM V$RESULT_CACHE_OBJECTS
WHERE creator_uid = (SELECT user_id
FROM DBA_USERS
WHERE username='ORADEV')
/

```

ID	TYPE	STATUS	NAMES	CACHE_ID
3	Dependency	Published		ORADEV.EMPLOYEES
4	Result	Published	SQL	7j5z4u9hzff4036czuj67bwqnt

Now, we will update the salary of the employees:

```

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

/*Update salary of employees in EMPLOYEES table*/
SQL> UPDATE employees
SET sal = sal+100

```

```
/

14 rows updated.

/*Commit the transaction*/
SQL> commit;
Commit complete.
```

When we query the cached result in V\$RESULT_CACHE_OBJECTS, the earlier cached results are found to be invalidated:

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

/*Query the result cache objects view to check the INVALID status*/
SELECT id, type, status, namespace, cache_id
FROM V$RESULT_CACHE_OBJECTS
WHERE creator_uid = (SELECT user_id
FROM DBA_USERS
WHERE username='ORADEV')
/
```

ID	TYPE	STATUS	NAMES	CACHE_ID
3	Dependency	Published		ORADEU.EMPLOYEES
4	Result	Invalid	SQL	7j5z4u9hzff4036czuj67bwqnt

Displaying the result cache memory report

The cache memory report can be generated from the DBMS_RESULT_CACHE package using the MEMORY_REPORT procedure. The report displays the following details:

- A single block size
- Maximum cache memory available
- Maximum size of a cached result
- Used and unused portion of the cache memory
- Count of the cached results, invalidated results, and dependent objects

A sample cache memory report looks as follows:

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

```
/*Enable the serveroutput variable to display the block messages*/
SQL> SET SERVEROUTPUT ON

/*Generate the cache memory report*/
SQL> EXEC DBMS_RESULT_CACHE.MEMORY_REPORT
R e s u l t   C a c h e   M e m o r y   R e p o r t
[Parameters]
Block Size = 1K bytes
Maximum Cache Size = 200M bytes (200K blocks)
Maximum Result Size = 40M bytes (40K blocks)
[Memory]
Total Memory = 134992 bytes [0.062% of the Shared Pool]
Fixed Memory = 9460 bytes [0.004% of the Shared Pool]
Dynamic Memory = 125532 bytes [0.058% of the Shared Pool]
Overhead = 92764 bytes
Cache Memory = 32K bytes (32 blocks)
Unused Memory = 30 blocks
Used Memory = 2 blocks
Dependencies = 1 blocks (1 count)
Results = 1 blocks
SQL = 1 blocks (1 count)

PL/SQL procedure successfully completed.
```

Read consistency of the SQL result cache

While the server searches the cached result of a query, the query must ensure read consistency. The clauses given in the following list must hold true for the database to use the cache:

- In a session, if the table (whose earlier query results are cached) is under an uncommitted transaction, the database would not cache its results. The queries using the table fetch the result from the server cache until the transaction is committed and the cached result gets invalidated.
- For the query result to be reusable, the query must make use of flashback to specify the timeframe.

Limitation of SQL result cache

The result of a SQL query will not be cached if the SQL includes dictionary views, temporary tables, SYS-owned tables, sequences, pseudo columns (such as CURRVAL, NEXTVAL, SYSDATE, LEVEL, ROWNUM and so on), or non-deterministic PL/SQL functions.

Implementing result cache in PL/SQL

Result caching in PL/SQL is second component of server-side caching in Oracle 11g. As we discussed briefly in the first section, results of frequently used PL/SQL functions can be retained at the server cache. The PL/SQL result cache feature uses the same infrastructure as the server result cache. When a function marked for result cache is executed, its result is cached at the server cache along with the parameters. The server picks up the result from the cache memory, if the same function is executed with the same parameters. In this way, the server saves a handful of time by bypassing the execution of the function body every time it is invoked, resulting into enhanced performance. The function can be a standalone, packaged, or locally declared (in a subprogram, not in anonymous PL/SQL block) one.

However, the cached result gets invalidated when the function or its referencing tables undergo a structural change followed by recompilation. The cached result also sets to the invalid state when the data in any of the referencing tables is updated.

As a common property of the server-side result cache, the cached result remains available for all connected active sessions of the same database.

The RESULT_CACHE clause

The PL/SQL cache implementation and execution is similar to the RESULT_CACHE hint in SQL. The same hint appears as a keyword in the PL/SQL function definition. A function whose results are to be cached must include the RESULT_CACHE clause in its header. The clause asks the database to cache the results of the function with its actual arguments. Syntax of the new function header looks as follows:

```
CREATE OR REPLACE FUNCTION [FUNCTION NAME]
RETURN [Return data type]
  RESULT_CACHE
  RELIES ON [TABLE NAME] (optional)
AS
BEGIN
...
...
END;
```

The `RELIES_ON` clause was introduced in Oracle 11g Release 1. The clause was used to specify the dependent table or view names, whose state would affect the status of the cached result. The cached result would be invalidated if the data in these tables or views undergo DML transaction. But later, the concept was found redundant when databases were updated to take care of managing dependencies. Therefore, the enhancement was withdrawn by Oracle in its subsequent 11g R2 release. The removal of the `RELIES_ON` clause increased authenticity and capability of the server cache by wiping out chances of error due to dependency.

Once the server has been configured for caching, PL/SQL function results can be readily cached. Check the following illustration:

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

/*Flush the cache memory to clear the earlier cached results*/
SQL> EXEC dbms_result_cache.flush;

PL/SQL procedure successfully completed.

/*Flush the shared pool*/
SQL> alter system flush shared_pool
/

System altered.
/*connect as ORADEV*/
Conn ORADEV /ORADEV
Connected.

/*Set the SERVEROUTPUT parameter on to display the results*/
SQL> SET SERVEROUTPUT ON

/*Create a function F_GET_SAL*/
CREATE OR REPLACE FUNCTION f_get_sal (P_EMPNO NUMBER)
RETURN NUMBER
RESULT_CACHE
IS
    l_sal NUMBER;
BEGIN
    DBMS_OUTPUT.PUT_LINE(' Function Body execution');
    SELECT sal
    INTO L_SAL
    FROM employees
    WHERE empno=P_EMPNO;
    RETURN l_sal;
END;
/

Function created.

/*Declare a local variable and execute the function*/
SQL> VARIABLE m_sal NUMBER;
```

Caching to Improve Performance

```
SQL> EXEC :m_sal := f_get_sal (7900);
Function Body execution
PL/SQL procedure successfully completed.
SQL> PRINT m_sal
   M_SAL
-----
   1050
```

As soon as the function is executed for an input argument, the server caches the function result for this parameter. Let us now investigate the server cache memory for the function result:

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

/*Query the cached results*/
SELECT id,status,name, type, namespace
FROM v$result_cache_objects
WHERE creator_uid = (SELECT user_id
                     FROM dba_users
                     WHERE username='ORADEV')
/
```

Refer to the following screenshot for the output of the preceding code block:

ID	STATUS	NAME	TYPE	NAMES
2	Published	ORADEV.EMPLOYEES	Dependency	
0	Published	ORADEV.F_GET_SAL	Dependency	
1	Published	"ORADEV"."F_GET_SAL"::8."F_GET_SAL"#{fac892c7867b54c6 #1	Result	PLSQL

The function result has been cached, along with the referenced object that is, the EMPLOYEES table. Next time, if the function is invoked with the same parameter 7900, the F_GET_SAL function would not be executed. In the following repeated execution of F_GET_SAL, note that the display message function body execution has not been printed. It is because the function result has been picked up from the server cache:

```
/*connect as ORADEV*/
Conn ORADEV /ORADEV

/*Enable the serveroutput variable to display the block messages*/
SQL> SET SERVEROUTPUT ON

/*Declare a bind variable to capture the function execution results*/
SQL> VARIABLE m_sal NUMBER;
SQL> EXEC :m_sal := f_get_sal (7900);
PL/SQL procedure successfully completed.
SQL> PRINT m_sal
   M_SAL
-----
   1050
```

For the parameter values different from the 7900, server executes the `F_GET_SAL` function, returns and caches the result at the server. In the following function call for the argument 7844, note that the display message from `dbms_output` has been printed. This implies that the function body has been executed once:

```

/*Enable the serveroutput variable to display the block messages*/
SQL> SET SERVEROUTPUT ON

/*Declare a bind variable to capture the function execution results*/
SQL> variable m_other_sal number;
SQL> EXEC :m_other_sal := f_get_sal (7844);
Function Body execution

PL/SQL procedure successfully completed.

/*Print the results*/
SQL> PRINT m_other_sal;

M_OTHER_SAL
-----
          1600

```

The result from the above function execution for employee, if 7844 is cached and its information can be queried in the `V$RESULT_CACHE_OBJECTS` view, is as follows:

```

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

/*Check the cached results*/
SELECT id, status, type, cache_id
FROM v$result_cache_objects
ORDER BY id
/

```

ID	STATUS	TYPE	CACHE_ID
0	Published	Dependency	ORADEV.F_GET_SAL
1	Published	Result	97wusxcnc35b3053yrt02j5qc7
2	Published	Dependency	ORADEV.EMPLOYEES
3	Published	Result	97wusxcnc35b3053yrt02j5qc7

The PL/SQL caching metadata is stored in the same way as it is done in the SQL result caching. The same type (Result and Dependency) and status (New, Published, Invalid, Expired, or Bypass) appear when querying the cached result information in `V$RESULT_CACHE_OBJECTS`. The dependent object information can be queried from `V$RESULT_CACHE_DEPENDENCY`. Once the function result is cached at the server, the counts in `V$RESULT_CACHE_STATISTICS` would get updated automatically.

Now, let us generate the cache memory report to verify the PL/SQL cache result entries. In the report, note the PL/SQL result's counts, which are cached under used memory:

```
/*Connect to SYSDBA*/
Conn sys/system as sysdba
Connected.

/*Enable the serveroutput variable to display the block messages*/
SQL> SET SERVEROUTPUT ON
SQL> exec dbms_result_cache.memory_report
R e s u l t   C a c h e   M e m o r y   R e p o r t
[Parameters]
Block Size           = 1K bytes
Maximum Cache Size  = 200M bytes (200K blocks)
Maximum Result Size = 100M bytes (100K blocks)
[Memory]
Total Memory = 140616 bytes [0.064% of the Shared Pool]
Fixed Memory = 9460 bytes [0.004% of the Shared Pool]
Dynamic Memory = 131156 bytes [0.060% of the Shared Pool]
Overhead = 98388 bytes
Cache Memory = 32K bytes (32 blocks)
Unused Memory = 28 blocks
Used Memory = 4 blocks
Dependencies = 2 blocks (2 count)
Results = 2 blocks
PLSQL      = 2 blocks (2 count)

PL/SQL procedure successfully completed.
```

Cross-session availability of cached results

Once again, by virtue of common features of the server-side cache, the function results cached in one session would be accessible in all the sessions. The reason for this is the function result which is cached in the SGA and SGA is available for all the sessions.

To be noted, PL/SQL function result caching works only if the formal parameters are passed by reference. This is one of the limitation of the result cache feature in PL/SQL.

Invalidation of PL/SQL result cache

The function result cache gets invalidated if the function or its referencing table undergo a DDL change (alter, modify, or recompilation). In addition, even if the data contained in any of the referencing table gets updated, the server purges the cached result.

Currently, we have two published cache results from the last demonstration:

```

/*Connect to DBA*/
Conn sys/system as sysdba
Connected.

/*Query the cached results*/
SELECT id, status, type, cache_id
FROM v$result_cache_objects
ORDER BY id
/

```

ID	STATUS	TYPE	CACHE_ID
0	Published	Dependency	ORADEV.F_GET_SAL
1	Published	Result	97wusxcnc35b3053yrt02j5qc7
2	Published	Dependency	ORADEV.EMPLOYEES
3	Published	Result	97wusxcnc35b3053yrt02j5qc7

This time, without flushing the results, we will recompile the function to observe the effect on the cached results:

```

/*Connect to the ORADEV user*/
Conn ORADEV/ORADEV
Connected.

/*Compile the function*/
ALTER FUNCTION F_GET_SAL COMPILE
/

Function altered.

```

Now, when we check the cache object information in V\$RESULT_CACHE_OBJECTS, the Published results will be invalidated:

```

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

/*Query the cached results*/
SELECT id, status, type, cache_id
FROM v$result_cache_objects
ORDER BY id
/

```

ID	STATUS	TYPE	CACHE_ID
0	Published	Dependency	ORADEV.F_GET_SAL
1	Invalid	Result	97wusxcnc35b3053yrt02j5qc7
2	Published	Dependency	ORADEV.EMPLOYEES
3	Invalid	Result	97wusxcnc35b3053yrt02j5qc7

The observation deduces that the cached result is highly sensitive to the changes occurring on the objects with which it shares direct and indirect dependencies.

Limitations of PL/SQL function result cache

The efficiencies and accomplishments of the server result cache are beyond doubts. However, in some exceptional cases the caching feature is automatically ignored by the database server. We will discuss certain limitations of the result caching feature in Oracle.

Argument and return type restrictions

The argument and return type limitations are as follows:

- Functions with pass by value parameters (OUT or IN OUT)
- Functions with CLOB, NCLOB, or BLOB arguments
- Functions with arguments of a user-defined object or collection type
- Function return type is LOB

Function structural restrictions

The function structural limitations are as follows:

- Functions which are created with its invoker's rights.
- The function declared locally in an anonymous PL/SQL block. Oracle raises the following PLS-00999 exception:

```
PLS-00999: implementation restriction (may be temporary) RESULT_
CACHE is disallowed on subprograms in anonymous blocks
```

It appears as a temporary restriction and might be seen in the forthcoming releases. However, a function declared locally in a subprogram (procedure or function) can still be stored in the server cache.

- Pipelined table functions

Summary

In this chapter, we learned how server result caching can dramatically improve performance in SQL and PL/SQL applications. We understood the database configuration required to enable the caching feature at the server. We learned the SQL result caching and PL/SQL function result caching through demonstrations.

In the next chapter, we will learn the PL/SQL analysis steps and techniques.

Practice exercise

1. The initialization parameter settings for your database are as follows:

```
MEMORY_TARGET = 500M
RESULT_CACHE_MODE = MANUAL
RESULT_CACHE_MAX_SIZE = 0
```

You execute a query by using the `RESULT_CACHE` hint. Which statement is true in this scenario?

- a. The query results are not stored in the cache because no memory is allocated for the result cache.
 - b. The query results are stored in the cache because Oracle implicitly manages the cache memory.
 - c. The query results are not stored in the cache because `RESULT_CACHE_MODE` is `MANUAL`.
 - d. The query results are stored in the cache automatically when `RESULT_CACHE_MODE` is `MANUAL`.
2. You set the following initialization parameter settings for your database:

```
MEMORY_TARGET = 500M
RESULT_CACHE_MODE = FORCE
RESULT_CACHE_MAX_SIZE = 200M
```

You execute the following query:

```
SELECT /*+RESULT_CACHE*/ ENAME, DEPTNO
FROM EMPLOYEES
WHERE EMPNO = 7844
/
```

Which of the following statements are true?

- a. The query results are cached because the SQL uses the `RESULT_CACHE` hint.
 - b. The query results are cached because the result cache mode is `FORCE`.
 - c. The query results are not cached because the SQL uses the `RESULT_CACHE` hint.
 - d. The `RESULT_CACHE` hint is ignored when result cache mode is `FORCE`.
3. The cached query result becomes invalid when the data accessed by the query gets modified.
 - a. True
 - b. False
 4. The SQL query result cache is persistent only for the current session.
 - a. True
 - b. False
 5. Which of the following PL/SQL objects' results cannot be cached?
 - a. Standalone function
 - b. Procedure
 - c. A function local to a procedure
 - d. Packaged function
 6. The `RELIES_ON` clause in the PL/SQL function result cache can be used to specify the dependent tables or views whose state would affect the cached result.
 - a. True
 - b. False
 7. Server settings are as follows:

```
MEMORY_TARGET = 500M
RESULT_CACHE_MODE = FORCE
RESULT_CACHE_MAX_SIZE = 200M
```

Identify the SQL queries whose results cannot be cached by the server.

- a. `SELECT ename, sal FROM employees WHERE empno = 7900;`
- b. `SELECT seq_empid.nextval FROM DUAL;`
- c. `SELECT ename, sysdate, hiredate FROM employees;`
- d. `SELECT dname, loc FROM departments WHERE deptno = 10;`

8. Identify the correct statements about the PL/SQL function result cache.
 - a. PL/SQL function result cache requires additional server configuration.
 - b. PL/SQL function result cache cannot be operated on procedures.
 - c. PL/SQL function result cache works with all categories of functions.
 - d. PL/SQL function cache features can work with the function which take collection type arguments.

9. Identify the admissible value of the `STATUS` column in `V$RESULT_CACHE_OBJECTS`.
 - a. PUBLISHED
 - b. INVALID
 - c. USED
 - d. UNUSED

10. Choose the correct statement about the following sample cache memory statistics report:

ID	NAME	VALUE
1	Block Size (Bytes)	1024
2	Block Count Maximum	204800
3	Block Count Current	32
4	Result Size Maximum (Blocks)	40960
5	Create Count Success	1
6	Create Count Failure	0
7	Find Count	0
8	Invalidation Count	0
9	Delete Count Invalid	0
10	Delete Count Valid	0
11	Hash Chain Length	1

- a. Create Count Success is the count of successfully cached results.
- b. Find Count is the count of the successfully cached results found and used in the queries.
- c. Invalidation Count is the count of the invalidated cached results.
- d. Block Count Maximum is the static value of total blocks available in the cache memory.

10

Analyzing PL/SQL Code

Code writing and tuning is the first stage of application life cycle development. As this life cycle matures and grows, the maintenance of code base becomes mandatory for code analysis and forecasts. The code management strategy aims at code testing, tracing, profiling, and reporting the coding information. This chapter covers some recommended techniques to analyze PL/SQL code through Oracle-supplied resources such as data dictionary views, initialization parameters, and built-in packages. Within the scope of the chapter, we will cover the following topics:

- Tracing and generating reports on PL/SQL source code
- Reporting usage of identifiers in PL/SQL source code
- Extracting schema object definitions using `DBMS_METADATA`

Track coding information

Once the development stage of the code base is over, it might be required to track through the code for search operation or to extract some crucial information for analysis or maintenance purposes. Such scenarios do not require thorough line-by-line digging as might seem to be the case. The line-by-line or code-by-code approach not only eats up a lot of time and resource but also ends up in a huge effort with tiny result. For this reason, Oracle supplies a set of dictionary views which make the life of analysts easy. The Oracle-supplied dictionary views are proven metadata sources of Oracle to provide accurate and detailed end results. The dictionary views used for tracking PL/SQL code information are `ALL_ARGUMENTS`, `ALL_OBJECTS`, `ALL_SOURCE`, `ALL PROCEDURES`, and `ALL_DEPENDENCIES`.

The following diagram lists the dictionary views along with a brief description. Note that only ALL_* views are listed in the chart but, nevertheless, the same purpose is achieved by the other [USER | DBA] flavors too:

ALL_ARGUMENTS	■ Stores information about member subprograms and its arguments for a PL/SQL subprogram
ALL_OBJECTS	■ Stores the objects created in the database
ALL_SOURCE	■ Stores the source code of the compilation and stores program units in a schema
ALL_PROCEDURES	■ Stores the package, procedure, and function information
ALL_DEPENDENCIES	■ Stores the dependencies of an object

When a schema object is compiled and created in the database, SYS-owned tables capture the relevant information about the PL/SQL object. Dictionary views are built on top of the SYS-owned tables to present the information in a meaningful way. These views exist in the following three flavors:

- USER: Contains metadata of the objects whose owner is the current user
- ALL: Contains metadata of the objects accessible by the current user
- DBA: Contains metadata of all objects

Dictionary views are accessed by prefixing their scope with the name. It is not mandatory that a view must exist in all three flavors. For example, the DBA_GLOBAL_CONTEXT view exists, but the USER_GLOBAL_CONTEXT and ALL_GLOBAL_CONTEXT views do not exist.

 All dictionary views along with their description can be queried from an Oracle-supplied view `DICTIONARY`.

Let us create a small procedure and a function to understand how the preceding dictionary views present meaningful metadata information. The `P_PRINT_NAME` procedure accepts a parameter and prints it in uppercase. A similar result is achieved by the `F_GET_NAME` function too:

```

/*Connect to ORADEV user*/
SQL> conn ORADEV/ORADEV
Connected.

/*Enable the serveroutput to display the error messages*/
SQL> SET SERVEROUTPUT ON

/*Create the procedure*/
SQL> CREATE OR REPLACE PROCEDURE p_print_name (p_name VARCHAR2)
  IS
    l_name VARCHAR2(255);
  BEGIN

/*Convert the input string case to upper*/
    l_name := UPPER(p_name);

/*Print the input string in upper case*/
    DBMS_OUTPUT.PUT_LINE(l_name);
  END;
/

Procedure created.

/*Create the function*/
SQL> create or replace function f_print_name (p_name varchar2)
  return VARCHAR2
  IS
  begin

/*Return the string in upper case*/
    return UPPER(p_name);
  END;
/

Function created.

```

[DBA | ALL | USER]_ARGUMENTS

Now, let us query each of the dictionary views to check the information collected by them. We will start with the `USER_ARGUMENTS` view. The view contains object properties such as the object ID, its name, its parent package name (if any), and argument information such as arguments of the subprogram, its sequence, data type, and parameter passing mode.

The structure of the view is as follows:

```
/*Print the structure of USER_ARGUMENTS*/  
SQL> DESC USER_ARGUMENTS
```

Name	Null?	Type
OBJECT_NAME		VARCHAR2 (30)
PACKAGE_NAME		VARCHAR2 (30)
OBJECT_ID	NOT NULL	NUMBER
OVERLOAD		VARCHAR2 (40)
SUBPROGRAM_ID		NUMBER
ARGUMENT_NAME		VARCHAR2 (30)
POSITION	NOT NULL	NUMBER
SEQUENCE	NOT NULL	NUMBER
DATA_LEVEL	NOT NULL	NUMBER
DATA_TYPE		VARCHAR2 (30)
DEFAULTED		VARCHAR2 (1)
DEFAULT_VALUE		LONG
DEFAULT_LENGTH		NUMBER
IN_OUT		VARCHAR2 (9)
DATA_LENGTH		NUMBER
DATA_PRECISION		NUMBER
DATA_SCALE		NUMBER
RADIX		NUMBER
CHARACTER_SET_NAME		VARCHAR2 (44)
TYPE_OWNER		VARCHAR2 (30)
TYPE_NAME		VARCHAR2 (30)
TYPE_SUBNAME		VARCHAR2 (30)
TYPE_LINK		VARCHAR2 (128)
PLS_TYPE		VARCHAR2 (30)
CHAR_LENGTH		NUMBER
CHAR_USED		VARCHAR2 (1)

The view columns can be described with comments using the DICT_COLUMNS dictionary view:

```
/*Query the view columns*/  
SELECT column_name, comments  
FROM dict_columns  
WHERE table_name='USER_ARGUMENTS'  
/
```

Refer to the following screenshot for the output:

COLUMN_NAME	COMMENTS
OBJECT_NAME	Procedure or function name
PACKAGE_NAME	Package name
OBJECT_ID	Object number of the object
OVERLOAD	Overload unique identifier
SUBPROGRAM_ID	Unique sub-program Identifier
ARGUMENT_NAME	Argument name
POSITION	Position in argument list, or null for function return value
SEQUENCE	Argument sequence, including all nesting levels
DATA_LEVEL	Nesting depth of argument for composite types
DATA_TYPE	Datatype of the argument
DEFAULTED	Is the argument defaulted?
DEFAULT_VALUE	Default value for the argument
DEFAULT_LENGTH	Length of default value for the argument
IN_OUT	Argument direction (IN, OUT, or IN/OUT)
DATA_LENGTH	Length of the column in bytes
DATA_PRECISION	Length: decimal digits (NUMBER) or binary digits (FLOAT)
DATA_SCALE	Digits to right of decimal point in a number
RADIX	Argument radix for a number
CHARACTER_SET_NAME	Character set name for the argument
TYPE_OWNER	Owner name for the argument type in case of object types
TYPE_NAME	Object name for the argument type in case of object types
TYPE_SUBNAME	Subordinate object name for the argument type in case of object types
TYPE_LINK	Database link name for the argument type in case of object types
PLS_TYPE	PL/SQL type name for numeric arguments
CHAR_LENGTH	Character limit for string datatypes
CHAR_USED	Is the byte limit (B) or char limit (C) official for this string?

26 rows selected.

The ARGUMENT_NAME view column denotes the actual argument name as given in the program header (DATA_LEVEL = 0). If it is NULL, it signifies the return type of the function to be in OUT mode (DATA_LEVEL = 0). For DATA_LEVEL > 0, the argument is of object type or composite data type.

 The USER_ARGUMENTS view contains only the argument name, type, passing mode, and default value. However, the view does not maintain any information about the NOCOPY hint, if used with the OUT or IN OUT arguments.

The argument contained in the P_PRINT_NAME procedure and the F_PRINT_NAME function can be queried from the view as shown in the following code snippet. Observe the record entry with the NULL argument name which denotes the return type of the function:

```
/*Query the arugment for the procedure P_PRINT_NAME*/
SELECT object_name, subprogram_id,argument_name, data_type, in_out
FROM user_arguments
WHERE object_name IN ('P_PRINT_NAME','F_PRINT_NAME')
/
```

OBJECT_NAME	SUBPROGRAM_ID	ARGUMENT	DATA_TYPE	IN_OUT
F_PRINT_NAME	1	P_NAME	VARCHAR2	IN
F_PRINT_NAME	1		VARCHAR2	OUT
P_PRINT_NAME	1	P_NAME	VARCHAR2	IN

[DBA | ALL | USER]_OBJECTS

The USER_OBJECTS view simply stores the metadata information of the schema objects. Apart from storing basic information such as the object ID, name, or creation timestamp, it collects the information about the object type, status, and namespace details.

The structure of the dictionary view looks as shown in the following code snippet:

```
/*Display the structure of USER_OBJECTS*/  
SQL> DESC USER_OBJECTS
```

Name	Null?	Type
OBJECT_NAME		VARCHAR2 (128)
SUBOBJECT_NAME		VARCHAR2 (30)
OBJECT_ID		NUMBER
DATA_OBJECT_ID		NUMBER
OBJECT_TYPE		VARCHAR2 (19)
CREATED		DATE
LAST_DDL_TIME		DATE
TIMESTAMP		VARCHAR2 (19)
STATUS		VARCHAR2 (7)
TEMPORARY		VARCHAR2 (1)
GENERATED		VARCHAR2 (1)
SECONDARY		VARCHAR2 (1)
NAMESPACE		NUMBER
EDITION_NAME		VARCHAR2 (30)

The view columns with comments can be queried from the DICT_COLUMNS view:

```
/*Query the view columns*/  
SELECT column_name, comments  
FROM dict_columns  
WHERE table_name='USER_OBJECTS'  
/
```

Refer to the following screenshot for the output:

COLUMN_NAME	COMMENTS
OBJECT_NAME	Name of the object
SUBOBJECT_NAME	Name of the sub-object (for example, partition)
OBJECT_ID	Object number of the object
DATA_OBJECT_ID	Object number of the segment which contains the object
OBJECT_TYPE	Type of the object
CREATED	Timestamp for the creation of the object
LAST_DDL_TIME	Timestamp for the last DDL change (including GRANT and REVOKE) to the object
TIMESTAMP	Timestamp for the specification of the object
STATUS	Status of the object
TEMPORARY	Can the current session only see data that it place in this object itself?
GENERATED	Was the name of this object system generated?
SECONDARY	Is this a secondary object created as part of icreate for domain indexes?
NAMESPACE	Namespace for the object
EDITION_NAME	Name of the edition in which the object is actual

14 rows selected.

In the preceding `USER_OBJECTS` view structure, it is important to understand the behavior of the date type columns — `CREATED`, `LAST_DDL_TIME` and `TIMESTAMP`. The `CREATED` column stores the fixed value as the date when the object was created for the first time. The `LAST_DDL_TIME` column stores the date when the object was recompiled last time. The `TIMESTAMP` column stores the date when the source code of the object was modified. If the object is recompiled, only the `LAST_DDL_TIME` column gets updated. But if the source code of the object undergoes a modification, both `TIMESTAMP` and `LAST_DDL_TIME` get updated.

The `EDITION_NAME` column stores the actual edition name of the object. Editions are non-schema objects which are used to maintain versions of schema objects. A new edition inherits all objects from the latest edition. Oracle 11g R2 brings in a mandatory default edition `ORA$BASE` for all databases. Further reading can be continued at Oracle documentation (http://docs.oracle.com/cd/E11882_01/appdev.112/e10471/adfns_editions.htm).

The object properties of the `P_PRINT_NAME` procedure, as schema objects, can be retrieved as follows:

```
/Query the object properties of P_PRINT_NAME*/
SELECT object_id, object_type, status, namespace
FROM user_objects
WHERE object_name='P_PRINT_NAME'
/
```

OBJECT_ID	OBJECT_TYPE	STATUS	NAMESPACE
81410	PROCEDURE	VALID	1

[DBA | ALL | USER]_SOURCE

The USER_SOURCE dictionary view should give you the complete source code for the object which you request. Here is the structure of the view:

```
/*Display the structure of USER_SOURCE*/
SQL> DESC USER_SOURCE
```

Name	Null?	Type
NAME		VARCHAR2 (30)
TYPE		VARCHAR2 (12)
LINE		NUMBER
TEXT		VARCHAR2 (4000)

The columns of the USER_SOURCE view, along with their comments, can be queried from the DICT_COLUMNS view:

```
/*Query the view columns*/
SELECT column_name, comments
FROM dict_columns
WHERE table_name='USER_SOURCE'
/
```

COLUMN_NAME	COMMENTS
NAME	Name of the object
TYPE	Type of the object: "TYPE", "TYPE BODY", "PROCEDURE", "FUNCTION", "PACKAGE", "PACKAGE BODY", "LIBRARY" or "JAVA SOURCE"
LINE	Line number of this line of source
TEXT	Source text

The source code of the P_PRINT_NAME program unit can be queried as follows:

```
/*Query the source code of P_PRINT_NAME*/
SELECT *
FROM user_SOURCE
WHERE name='P_PRINT_NAME'
/
```

NAME	TYPE	LINE	TEXT
P_PRINT_NAME	PROCEDURE	1	PROCEDURE p_print_name (p_name VARCHAR2)
P_PRINT_NAME	PROCEDURE	2	IS
P_PRINT_NAME	PROCEDURE	3	l_name VARCHAR2(255);
P_PRINT_NAME	PROCEDURE	4	BEGIN
P_PRINT_NAME	PROCEDURE	5	/*Convert the input string case to upper*/
P_PRINT_NAME	PROCEDURE	6	l_name := UPPER(p_name);
P_PRINT_NAME	PROCEDURE	7	/*Print the input string in upper case*/
P_PRINT_NAME	PROCEDURE	8	DBMS_OUTPUT.PUT_LINE(l_name);
P_PRINT_NAME	PROCEDURE	9	END;

9 rows selected.

[DBA | ALL | USER]_PROCEDURES

The USER_PROCEDURES dictionary view stores the subprogram properties of an object. Unlike its name, it stores the details for a procedure, function, or packages contained in the database schema. The structure of the dictionary view looks as follows:

```

/*Display the structure of USER_PROCEDURES*/
SQL> DESC USER_PROCEDURES

```

Name	Null?	Type
OBJECT_NAME		VARCHAR2 (128)
PROCEDURE_NAME		VARCHAR2 (30)
OBJECT_ID		NUMBER
SUBPROGRAM_ID		NUMBER
OVERLOAD		VARCHAR2 (40)
OBJECT_TYPE		VARCHAR2 (19)
AGGREGATE		VARCHAR2 (3)
PIPELINED		VARCHAR2 (3)
IMPLTYPEOWNER		VARCHAR2 (30)
IMPLTYPENAME		VARCHAR2 (30)
PARALLEL		VARCHAR2 (3)
INTERFACE		VARCHAR2 (3)
DETERMINISTIC		VARCHAR2 (3)
AUTHID		VARCHAR2 (12)

The columns of the USER_PROCEDURES view can be queried from the DICT_COLUMNS view:

```

/*Query the view columns*/
SELECT column_name, comments
FROM dict_columns
WHERE table_name='USER_PROCEDURES'
/

```

COLUMN_NAME	COMMENTS
OBJECT_NAME	Name of the object: top level function/procedure/package/type/trigger name
PROCEDURE_NAME	Name of the package or type subprogram
OBJECT_ID	Object number of the object
SUBPROGRAM_ID	Unique sub-program identifier
OVERLOAD	Overload unique identifier
OBJECT_TYPE	The typename of the object

```

AGGREGATE           Is it an aggregate function ?
PIPELINED           Is it a pipelined table function ?
IMPLTYPEOWNER       Name of the owner of the implementation type (if
any)
IMPLTYPENAME        Name of the implementation type (if any)
PARALLEL            Is the procedure parallel enabled ?
INTERFACE
DETERMINISTIC
AUTHID

```

14 rows selected.

From the above column description, it is important to understand a few columns such as IMPLTYPEOWNER, IMPLTYPENAME, and AUTHID. Implementation type and owner values interpret any object type association of the program. The AUTHID column shows the authorization holder during invocation – possible values can be DEFINER, if the program has to be invoked by its owner's rights, and CURRENT_USER, if the program has to be invoked by its invoker.

The procedural properties of the P_PRINT_NAME subprogram can be queried from the view as follows:

```

/*Query the subprogram properties of P_PRINT_NAME*/
select object_id,object_type, overload, authid
FROM user_procedures
WHERE object_name='P_PRINT_NAME'
/

```

OBJECT_ID	OBJECT_TYPE	OVERLOAD	AUTHID
81410	PROCEDURE		DEFINER

[DBA | ALL | USER]_DEPENDENCIES

The USER_DEPENDENCIES dictionary view reveals very important information about the dependencies shared by the object. In many cases, the dependency shared by an object is decisive over its validity status. The view contains the details of the objects which are referenced within the definition of a particular object:

```

/*Display the structure of USER_DEPENDENCIES*/
SQL> DESC USER_DEPENDENCIES

```

Name	Null?	Type
NAME	NOT NULL	VARCHAR2 (30)
TYPE		VARCHAR2 (18)

REFERENCED_OWNER	VARCHAR2 (30)
REFERENCED_NAME	VARCHAR2 (64)
REFERENCED_TYPE	VARCHAR2 (18)
REFERENCED_LINK_NAME	VARCHAR2 (128)
SCHEMAID	NUMBER
DEPENDENCY_TYPE	VARCHAR2 (4)

The columns of the preceding view structure can be explained from the DICT_COLUMNS view:

```

/*The view columns with comments*/
SELECT column_name, comments
FROM dict_columns
WHERE table_name='USER_DEPENDENCIES'
/

```

Refer to the following screenshot for the output:

```

COLUMN_NAME      COMMENTS
-----
NAME              Name of the object
TYPE              Type of the object
REFERENCED_OWNER  Owner of referenced object (remote owner if remote object)
REFERENCED_NAME   Name of referenced object
REFERENCED_TYPE   Type of referenced object
REFERENCED_LINK_NAME Name of dblink if this is a remote object
SCHEMAID
DEPENDENCY_TYPE
8 rows selected.

```

The dependency shared by the P_PRINT_NAME procedure can be queried as per the following query:

```

/*Query the dependent objects of P_PRINT_NAME*/
select type, referenced_owner, referenced_name, referenced_type
FROM user_dependencies
WHERE name='P_PRINT_NAME'
/

```

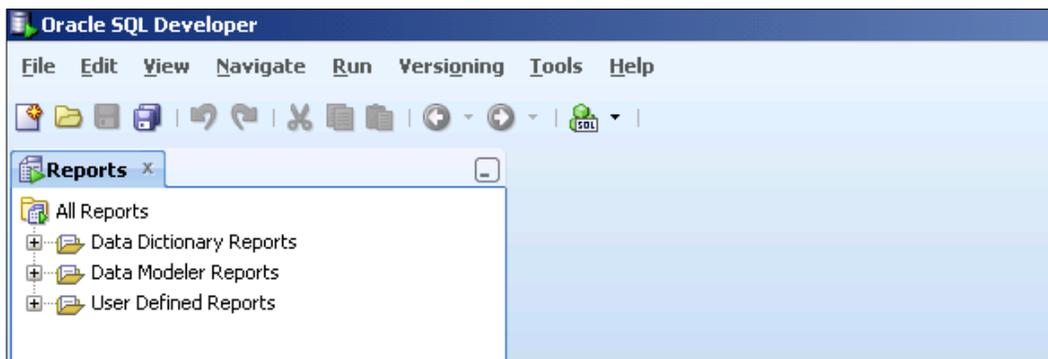
TYPE	REFERENCED_OWNER	REFERENCED_NAME	REFERENCED_TYPE
PROCEDURE	SYS	STANDARD	PACKAGE
PROCEDURE	SYS	DBMS_OUTPUT	PACKAGE
PROCEDURE	SYS	SYS_STUB_FOR_PURITY_ANALYSIS	PACKAGE

Using SQL Developer to find coding information

The object metadata information retrieved from the dictionary views is a conventional way to track code information. But these days, the IDE have been made self-sufficient to generate some vital predefined reports. The metadata information demonstrated in the last section using dictionary views can also be generated from SQL Developer. **SQL Developer** is a free UI based interactive IDE tool which boards multiple database utilities.

Here, we will demonstrate the tracking of code through SQL Developer:

1. Go to **View | Reports**:

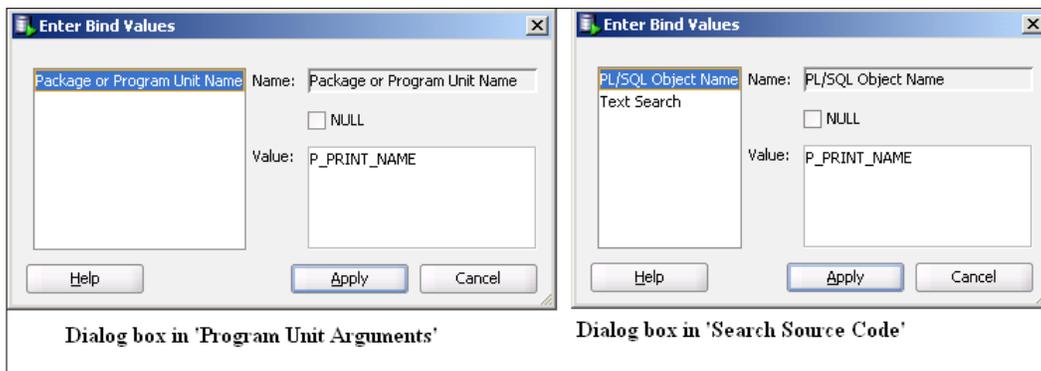


2. Go to **All Reports | Data Dictionary Reports | PLSQL**:
 - Under **PLSQL**, you find three options. These options are analogous to the dictionary views which we queried in the preceding section.
 - The **Program Unit Arguments** option queries the `USER_ARGUMENTS` dictionary view. The **Search Source Code** and **Unit Line Counts** options query the `USER_SOURCE` view.

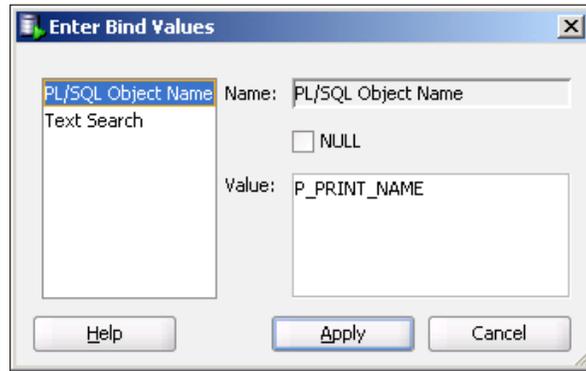
- When you click on any of the options for the first time, the following dialog box pops up. There you can select the connection, if you have multiple connections in your connect list:



3. Once the connection is selected from the drop-down list, another dialog box appears and prompts for user inputs:
 - For **Program Unit Arguments**, the dialog asks for **Package** or **Program Unit Name**.
 - For **Search Source Code**, the dialog box asks for **PL/SQL Object Name** or **Text Search**.
 - For **Unit Line Counts**, there is no dialog box, as it generates the line count report for all the schema objects:



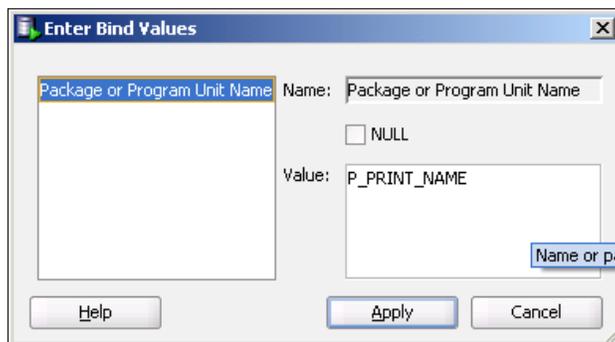
4. We can demonstrate **Search Source Code** by providing P_PRINT_NAME as input:



5. Click on **Apply** to generate the source code report:

Owner	PL/SQL Object Name	Type	Line	Text
ORADEV	P_PRINT_NAME	PROCEDURE	1	PROCEDURE p_print_name (p_name VARCHAR2)
ORADEV	P_PRINT_NAME	PROCEDURE	2	IS
ORADEV	P_PRINT_NAME	PROCEDURE	3	l_name VARCHAR2(255);
ORADEV	P_PRINT_NAME	PROCEDURE	4	BEGIN
ORADEV	P_PRINT_NAME	PROCEDURE	5	l_name := UPPER(p_name);
ORADEV	P_PRINT_NAME	PROCEDURE	6	DBMS_OUTPUT.PUT_LINE(l_name);
ORADEV	P_PRINT_NAME	PROCEDURE	7	END;

6. Demonstrate the generation of argument report from **Program Unit Arguments**:



- Click on the **Apply** button to generate the argument report for the given program unit:

Owner	Package	Program_Unit	Position	Argument	In_Out
ORADEV	(null)	P_PRINT_NAME	1	P_NAME	In

The DBMS_DESCRIBE package

The DBMS_DESCRIBE package is an Oracle built-in package which is used to gather information about the Oracle PL/SQL object—making it an essential Oracle data access component. It is owned by the SYS user, and all other users hitting the server, access its public synonym.

It contains only one subprogram that, is DESCRIBE_PROCEDURE.

The structure of the DESCRIBE_PROCEDURE subprogram is as follows:

```

DBMS_DESCRIBE.DESCRIBE_PROCEDURE (
  object_name          IN  VARCHAR2,
  reserved1            IN  VARCHAR2,
  reserved2            IN  VARCHAR2,
  overload              OUT NUMBER_TABLE,
  position              OUT NUMBER_TABLE,
  level                 OUT NUMBER_TABLE,
  argument_name         OUT VARCHAR2_TABLE,
  datatype              OUT NUMBER_TABLE,
  default_value         OUT NUMBER_TABLE,
  in_out                OUT NUMBER_TABLE,
  length                OUT NUMBER_TABLE,
  precision             OUT NUMBER_TABLE,
  scale                 OUT NUMBER_TABLE,
  radix                 OUT NUMBER_TABLE,
  spare                 OUT NUMBER_TABLE,
  include_string_constraints OUT BOOLEAN DEFAULT FALSE);

```

The DBMS_DESCRIBE procedure can extract the following information for a given procedure:

DBMS_DESCRIBE		
Overload feature	Level (For composite datatypes only)	Arguments -Name -Position -DataType and precision -Default Value (Default '0') -Parameter mode (IN=0, OUT=1, IN OUT=2)

Note that the procedure accepts three IN mode parameters, one OUT parameter, and 12 OUT parameters of associative array type.

Two reserved parameters are the Reserved parameters and must be kept NULL. The remaining parameters are the OUT parameters of the Associative array type whose definition is as follows:

```
TYPE VARCHAR2_TABLE IS TABLE OF VARCHAR2(30)
  INDEX BY BINARY_INTEGER;
TYPE NUMBER_TABLE IS TABLE OF NUMBER
  INDEX BY BINARY_INTEGER;
```

For the P_PRINT_NAME procedure, the DBMS_DESCRIBE package works as follows:

```
/*Connect to ORADEV user*/
Conn ORADEV/ORADEV
Connected.

/*Enable the serveroutput to display the error messages*/
SET SERVEROUTPUT ON
DECLARE

/*Declare the local variable of DBMS_DESCRIBE associative array type*/
v_overload DBMS_DESCRIBE.NUMBER_TABLE;
v_position DBMS_DESCRIBE.NUMBER_TABLE;
v_level DBMS_DESCRIBE.NUMBER_TABLE;
v_arg_name DBMS_DESCRIBE.VARCHAR2_TABLE;
```

```
v_datatype DBMS_DESCRIBE.NUMBER_TABLE;
v_def_value DBMS_DESCRIBE.NUMBER_TABLE;
v_in_out DBMS_DESCRIBE.NUMBER_TABLE;
v_length DBMS_DESCRIBE.NUMBER_TABLE;
v_precision DBMS_DESCRIBE.NUMBER_TABLE;
v_scale DBMS_DESCRIBE.NUMBER_TABLE;
v_radix DBMS_DESCRIBE.NUMBER_TABLE;
v_spare DBMS_DESCRIBE.NUMBER_TABLE;
BEGIN
/*Call the procedure DESCRIBE_PROCEDURE for P_PRINT_NAME*/
DBMS_DESCRIBE.DESCRIBE_PROCEDURE
(
  'P_PRINT_NAME',
  null, null,
  v_overload,
  v_position,
  v_level,
  v_arg_name,
  v_datatype,
  v_def_value,
  v_in_out,
  v_length,
  v_precision,
  v_scale,
  v_radix,
  v_spare,
  null
);
/*Iterate the argument array V_ARG_NAME to list the argument details
of the object*/
FOR i IN v_arg_name.FIRST .. v_arg_name.LAST
LOOP
/*Check if the position is zero or not*/
IF v_position(i) = 0 THEN
/*Zero position is reserved for RETURN types*/
DBMS_OUTPUT.PUT('This is the RETURN data for the function: ');
DBMS_OUTPUT.NEW_LINE;
ELSE
/*Print the argument name*/
DBMS_OUTPUT.PUT ('The argument name is: '||v_arg_name(i));
DBMS_OUTPUT.NEW_LINE;
END IF;
/*Display the position, type and mode of parameters*/
```

```
        DBMS_OUTPUT.PUT_LINE('The argument position is: ' || v_
position(i));
        DBMS_OUTPUT.NEW_LINE;
        DBMS_OUTPUT.PUT_LINE('The argument datatype is: ' || v_
datatype(i));
        DBMS_OUTPUT.NEW_LINE;
        DBMS_OUTPUT.PUT_LINE('The argument mode is: ' || v_in_out(i));
        DBMS_OUTPUT.NEW_LINE;
    END LOOP;
END;
/
```

```
The argument name is: P_NAME
The argument position is:1
The argument datatype is:1
The argument mode is:0
PL/SQL procedure successfully completed.
```

DBMS_UTILITY.FORMAT_CALL_STACK

We often encounter scenarios where subprogram calls have been extensively branched and nested among themselves. A subprogram can be called from multiple program units and it might be required to trace the complete invocation path.

The `FORMAT_CALL_STACK` function of the `DBMS_UTILITY` package is used to extract the current call stack as a formatted text string. The call stack contains the information about the sequential calls made from a program to another program. Every call, in the stack, is stored by the line number of the subprogram invocation.

Suppose a procedure P3 calls another procedure P2. P2 makes a call to another stored subprogram P1. Now, in P1, a call trace report can be embedded to see the call path:

```
/*Create the procedure P1*/
CREATE OR REPLACE PROCEDURE P1
IS
BEGIN
    dbms_output.put_line(substr(dbms_utility.format_call_stack, 1,
255));
END;
/
```

Procedure created.

```
/*Create the procedure P2*/
CREATE OR REPLACE PROCEDURE P2
```

```
IS
BEGIN
/*Call procedure P1*/
  P1;
END;
/

Procedure created.

/*Create the procedure P3*/
CREATE OR REPLACE PROCEDURE P3
IS
BEGIN
/*Call procedure P2*/
  P2;
END;
/

Procedure created.

/*Enable the serveroutput to display the error messages*/
SET SERVEROUTPUT ON

/*Start a PL/SQL block to invoke P3*/
BEGIN
/*Call P3*/
  P3;
END;
/

----- PL/SQL Call Stack -----
  object      line  object
  handle      number name
23D06844         4 procedure ORADEV.P1
23CEAD38         4 procedure ORADEV.P2
23EDCB38         4 procedure ORADEV.P3
23CF00CC         2 anonymous block
PL/SQL procedure successfully completed.
```

In the above output, the call stack shows the calls traversing from the anonymous block to procedure P1. Starting from the last, an anonymous block calls P3, which calls P2, and reaches P1. The call stack would appear different for different paths used to reach P1. If another procedure P4 calls P2, and hence P1, the call stack would show P4 in place of P3.

Tracking propagating exceptions in PL/SQL code

We are well versed in and aware of the propagation behavior of exceptions in PL/SQL. But locating the exact position from where the exception got raised has always been a cumbersome job for developers. After the failure of `SQLERRM` to truncate the error messages after 512 characters, `FORMAT_ERROR_STACK` is used to serve this purpose to some extent by presenting complete error messages up to 2000 characters, without truncation.

Oracle 10g Release 1 provides an error handling function under the `DBMS_UTILITY` package known as `FORMAT_ERROR_BACKTRACE`, to handle scenarios of exception propagation. The function produces a formatted string containing the stack of program unit information with line numbers. It helps developers to locate the exact program unit which has raised the exception.

Let us conduct a small case study where we will explicitly raise the exception in one program and try to access the same from another PL/SQL block.

The `P_TRACE` procedure declares a local exception and rises from the executable section:

```
/*Connect to ORADEV user*/
Conn ORADEV/ORADEV
Connected.

/*Create procedure P_TRACE*/
CREATE OR REPLACE PROCEDURE p_trace
IS

/*Declare a local user defined exception*/
  xlocal EXCEPTION;
BEGIN

/*Raise the local exception*/
  RAISE xlocal;
END;
/

Procedure created.
```

First, we will present the situation prior to the introduction of `FORMAT_ERROR_BACKTRACE` in Oracle 10g:

```
/*Call P_TRACE in a PL/SQL anonymous block*/
BEGIN
  P_TRACE;
EXCEPTION
WHEN OTHERS then
```

```

/*Display the error message using SQLERRM*/
  DBMS_OUTPUT.PUT_LINE(sqlerrm);
END;
/

User-Defined Exception
PL/SQL procedure successfully completed.

```

Note that the block output does not clearly specify the default program unit which caused the exception propagation. It is because `SQLERRM` logs only the last exception to occur.

Now, we will modify the same anonymous PL/SQL block and include `FORMAT_ERROR_BACKTRACE` to log the program unit which raises the exception:

```

/*Create the PL/SQL anonymous block*/
SQL> BEGIN
  P_TRACE;
  EXCEPTION
    WHEN OTHERS then

/*Print the error stack using FORMAT_ERROR_BACKTRACE*/
  DBMS_OUTPUT.PUT_LINE( DBMS_UTILITY.FORMAT_ERROR_BACKTRACE );
  END;
/

ORA-06512: at "ORADEV.P_TRACE", line 5
ORA-06512: at line 2

```

The output given by `FORMAT_ERROR_BACKTRACE` shows the name of the `P_TRACE` program unit along with the line number (5, in this case) which raises the exception. This error stack information expands as the nesting of calls increases. It helps to trace, debug, and log the correct information of the ongoing database activities and take appropriate action.

Determining identifier types and usages

All the local declarations of a program unit are categorized as **identifiers**. An identifier's declaration locates a memory on the server and keeps it busy until the program unit is executed or terminated. Redundant identifiers must be recognized within a program so as to restrict them from holding a chunk of memory for no operation.

Oracle provides a tool known as PL/Scope to monitor the activities of identifiers in a program. It is one of the new features in Oracle 11g.

The PL/Scope tool

The PL/Scope tool compiles and captures the information of the identifiers declared and used in a program. Once the feature is enabled, the language compiler filters out the identifier's information and stores it in a dictionary view called `USER_IDENTIFIERS`. An identifier is recognized by its name, type, and usage.

Let us examine some of the key features of the PL/Scope tool:

- Only unwrapped program units can use the PL/Scope tool.
- The feature can be enabled by setting a new initialization parameter called `PLSCOPE_SETTINGS`.
- The compiler stages the identifiers' information only in the `SYSAUX` tablespace. The feature remains deactivated if the `SYSAUX` tablespace is unavailable.
- The identifier information can be viewed in `[DBA | ALL | USER]_IDENTIFIERS`.
- The feature can be enabled for the whole database, for a session, or only for an object. It implies that a program can be compiled with different compilation parameters from the current session or database settings. The object level specification overrides the session or system level setting of the parameter.

PL/Scope brings great benefit to any large-size database application where developers frequently check the existence of an identifier so as to avoid redundancy. In addition, it can work as a PL/SQL IDE to build up a repository of all identifiers under multiple categories based on their type and usage.

The PL/Scope identifier collection

The PL/Scope feature can be enabled by setting a system parameter called `PLSCOPE_SETTINGS`. By default, the feature is disabled as the parameter value is `IDENTIFIERS:NONE`. The valid values for the parameters are `NONE` for disabled and `ALL` for enabled parameters.

A DBA can modify the value of the `PLSCOPE_SETTINGS` compilation parameter as `IDENTIFIERS:ALL` to enable the feature to collect the identifier information. Once the feature is activated at the required level, Oracle captures identifier information of all program units which are compiled henceforth.

Setting `PLSCOPE_SETTINGS` at system or session level:

```
ALTER [SYSTEM | SESSION]
SET PLSCOPE_SETTINGS = ['IDENTIFIERS:ALL' | 'IDENTIFIERS:NONE']
```

Setting `PLSCOPE_SETTINGS` at object level:

```
ALTER [PROGRAM NAME] COMPILE
PLSCOPE_SETTINGS = ['IDENTIFIERS:ALL' | 'IDENTIFIERS:NONE']
```

For illustration and demonstration purpose, we will keep the setting as `IDENTIFIERS:ALL`. A DBA performs it as shown in the following code snippet:

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

/*Modify the PLSCOPE_SETTINGS*/
ALTER SYSTEM SET PLSCOPE_SETTINGS = 'IDENTIFIERS:ALL';
Session altered.

/*View the current setting of PLSCOPE_SETTINGS parameter*/
SELECT value
FROM v$parameter
WHERE name='plscope_settings'
/

VALUE
-----
IDENTIFIERS:ALL
```

Another important aspect of PL/Scope is that it is stored only in the `SYSAUX` tablespace. If the `SYSAUX` tablespace is unavailable, the feature remains in passive state. Though the server does not raise any error message while logging, a warning is raised:

```
/*Verify the PL/Scope occupancy in SYSAUX tablespace*/
SELECT occupant_desc, schema_name, space_usage_kbytes
FROM v$sysaux_occupants
WHERE occupant_name='PL/SCOPE'
/

OCCUPANT_DESC                SCHEMA_NAME    SPACE_USAGE_KBYTES
-----
PL/SQL Identifier Collection  SYS                2496
```

The PL/Scope report

The PL/Scope report can be generated from the [DBA | ALL | USER]_IDENTIFIERS dictionary view. [DBA | ALL | USER] provides different flavors to the view as per the invoker's role:

```
/*Display the USER_IDENTIFIERS structure*/
SQL> desc USER_IDENTIFIERS
```

Name	Null?	Type
-----		-----
NAME		VARCHAR2(30)
SIGNATURE		VARCHAR2(32)
TYPE		VARCHAR2(18)
OBJECT_NAME	NOT NULL	VARCHAR2(30)
OBJECT_TYPE		VARCHAR2(13)
USAGE		VARCHAR2(11)
USAGE_ID		NUMBER
LINE		NUMBER
COL		NUMBER
USAGE_CONTEXT_ID		NUMBER

Note the information captured for an identifier in the preceding USER_IDENTIFIERS description:

- SIGNATURE: The unique hash code of the identifier
- TYPE: The user defined type of the identifier (Variable, Cursor, Formal, and so on)
- OBJECT_NAME: The object name within which they are declared, assigned, or used
- OBJECT_TYPE: The type of the object using the identifier
- USAGE: The identifier action as CALL, ASSIGNMENT, DEFINITION, DECLARATION, or REFERENCE
- USAGE_ID: The unique key of the identifier usage
- LINE, COL: Exact location of the identifier in the program

Illustration

Let us now demonstrate the PL/Scope tool to capture identifier information of a program unit. The program unit is a function which declares a cursor and local variables to get the location of an input employee.

The current setting of PLScope_SETTINGS is IDENTIFIERS:ALL:

```

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

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

```

Generate the PL/Scope identifier report from the USER_IDENTIFIERS dictionary view:

```

/*Query the identifier information from the view*/
SELECT name, type, object_name, usage
FROM user_identifiers
WHERE object_name='F_GET_LOC'
/

```

NAME	TYPE	OBJECT_NAME	USAGE
L_LOC	VARIABLE	F_GET_LOC	REFERENCE
C_DEPT	CURSOR	F_GET_LOC	CALL
L_LOC	VARIABLE	F_GET_LOC	ASSIGNMENT
C_DEPT	CURSOR	F_GET_LOC	CALL
C_DEPT	CURSOR	F_GET_LOC	CALL
NUMBER	NUMBER DATATYPE	F_GET_LOC	REFERENCE
L_LOC	VARIABLE	F_GET_LOC	DECLARATION

P_EMPNO	FORMAL IN	F_GET_LOC	REFERENCE
C_DEPT	CURSOR	F_GET_LOC	DECLARATION
NUMBER	NUMBER DATATYPE	F_GET_LOC	REFERENCE
NUMBER	NUMBER DATATYPE	F_GET_LOC	REFERENCE
P_EMPNO	FORMAL IN	F_GET_LOC	DECLARATION
F_GET_LOC	FUNCTION	F_GET_LOC	DEFINITION
F_GET_LOC	FUNCTION	F_GET_LOC	DECLARATION

14 rows selected.

Assume that the database setting does not allow the capturing of identifiers' information—only this feature can be enabled for the selective program units. A program unit can be compiled as follows:

```
/*Alter the function with PLScope_SETTINGS at object level*/
SQL> ALTER FUNCTION F_GET_LOC COMPILE PLScope_
SETTINGS='IDENTIFIERS:ALL';
```

Function altered.

For better presentation purpose, you can generate the above output in report format:

```
/*Generate interactive report for identifiers*/
WITH v AS
(
  SELECT Line,
         Col,
         INITCAP(NAME) Name,
         LOWER(TYPE) Type,
         LOWER(USAGE) Usage,
         USAGE_ID, USAGE_CONTEXT_ID
  FROM USER_IDENTIFIERS
  WHERE Object_Name = 'F_GET_LOC'
  AND Object_Type = 'FUNCTION'
)
SELECT LINE, RPAD(LPAD(' ', 2*(Level-1)) ||Name, 20, '.')||' '||
RPAD(Type, 20)|| RPAD(Usage, 20)
IDENTIFIER_USAGE_CONTEXTS
FROM v
  START WITH USAGE_CONTEXT_ID = 0
  CONNECT BY PRIOR USAGE_ID = USAGE_CONTEXT_ID
  ORDER SIBLINGS BY Line, Col
/
```

Refer to the following screenshot for the output:

LINE	IDENTIFIER	USAGE	CONTEXTS
1	F_Get_Loc	function	declaration
1	F_Get_Loc	function	definition
1	P_Empno	formal in	declaration
1	Number	number datatype	reference
2	Uarchar2	character datatype	reference
5	C_Dept	cursor	declaration
9	P_Empno	formal in	reference
10	L_Loc	variable	declaration
10	Uarchar2	character datatype	reference
13	C_Dept	cursor	call
14	C_Dept	cursor	call
14	L_Loc	variable	assignment
15	C_Dept	cursor	call
17	L_Loc	variable	reference

14 rows selected.

Applications of the PL/Scope report

The PL/Scope identifier report can achieve the following objectives:

- It searches all the identifiers declared in a schema (USAGE = 'DECLARATION'). The SELECT query lists all identifiers declared in the schema as shown in the following screenshot:

```
/*List the identifiers declared in the current schema*/
SELECT NAME, SIGNATURE, TYPE
FROM USER_IDENTIFIERS
WHERE USAGE='DECLARATION'
ORDER BY OBJECT_TYPE, USAGE_ID
/
```

- It searches all identifiers of a specific type (BLOB, CURSOR, CONSTANT, and so on) used in a schema. The following SQL query lists all the cursors declared in the schema:

```
/*List the identifiers declared as CURSOR in the current schema*/
SELECT NAME, SIGNATURE, OBJECT_NAME, TYPE
FROM USER_IDENTIFIERS
WHERE USAGE='DECLARATION'
AND TYPE = 'CURSOR'
ORDER BY OBJECT_TYPE, USAGE_ID
/
```

- It searches all redundant identifiers within a program unit which are not referenced inside the executable section of the program:

```
/*List the redundant identifiers declared in the current schema*/
SELECT NAME, OBJECT_NAME, TYPE, SIGNATURE
FROM USER_IDENTIFIERS T
WHERE USAGE='DECLARATION'
AND NOT EXISTS (SELECT 1
                FROM USER_IDENTIFIERS
                WHERE SIGNATURE=T.SIGNATURE
                AND USAGE<>'DECLARATION')
/
```

- It determines the actions performed on an identifier in a program:

```
/*List the actions on a specific identifier in the schema*/
SELECT name, object_name, type, usage, line
FROM USER_IDENTIFIERS T
WHERE signature='C6DC4D2D5770696415F7EC524AFADAE4'
/
```

The DBMS_METADATA package

The DBMS_METADATA package was introduced in Oracle9i. It is a metadata API which is used to extract the definitions (DDL) of schema objects. The package was introduced to get rid of DDL exports, which used to produce poorly formatted DDL scripts. It is a powerful package which can generate DDL and retrieve relevant information associated with an object in XML (by default), or textual format. The package is owned by SYS while all other users work with its public synonym.

The package provides utilities to set the required formatting for the DDL, transforms, and parse items. Once the formatting settings start over, using transform handlers, the definition of an object can be retrieved as XML or text. It also provides the flexibility to execute DDL. Let us see some of the major features of DBMS_METADATA:

- Generate DDL through GET_DDL (GET_XML is its XML equivalent).
- Generate DDL for object dependencies through GET_DEPENDENT_DDL (GET_DEPENDENT_XML is its XML equivalent).
- Generate DDL for system grants on an object through GET_GRANTED_DDL (GET_GRANTED_XML is its XML equivalent).
- Manage and modify object definitions such as add column, drop column, rename table, manage partitions, indexes, and so on.

- Callable from the `SELECT` statements.
- `DBMS_METADATA` uses public synonyms of SYS-owned object and table types.
- Additional options have been added in the Oracle 10g release.
- Data pump (Oracle 10g) uses `DBMS_METADATA` to retrieve schema object DDLs.

DBMS_METADATA data types and subprograms

As referred to earlier, the `DBMS_METADATA` package uses the public synonyms of SYS-owned data structures. The following list shows SYS-owned object types:

- `SYS.KU$_PARSED_ITEM`: It is the object to capture the attributes of object metadata of a single object. The object structure looks as follows:

```
CREATE TYPE sys.ku$_parsed_item AS OBJECT
(
  item VARCHAR2(30),
  value VARCHAR2(4000),
  object_row NUMBER
)
```

`ITEM`, `VALUE` form the attribute name value pair for `OBJECT_ROW`.

- `SYS.KU$_PARSED_ITEMS`: It is a nested table of `SYS.KU$_PARSED_ITEM` to hold the object metadata attributes for multiple objects.
- `SYS.KU$_DDL`: It is an object type to capture the DDL of an object along with its parsed item information. The object type structure looks as follows:

```
CREATE TYPE sys.ku$_ddl AS OBJECT
(
  ddlText CLOB,
  parsedItem sys.ku$_parsed_items
)
```

The parsed object information is stored in `PARSEDITEM`.

- `SYS.KU$_DDL`: It is a nested table of `SYS.KU$_DDL` returned by the `FETCH_DDL` subprogram to hold the metadata of an object transformed into multiple DDL statements.
- `SYS.KU$_MULTI_DDL`: It is an object type to hold the DDL for an object in multiple transforms.
- `SYS.KU$_MULTI_DDLS`: It is a nested table of `SYS.KU$_MULTI_DDL` returned by the `CONVERT` subprogram.

- **SYS.KU\$_ERRORLINE:** It is an object type to capture the error information. The object type structure is as follows:

```
CREATE TYPE sys.ku$_ErrorLine IS OBJECT
(
  errorNumber NUMBER,
  errorText VARCHAR2(2000)
)
/
```

- **SYS.KU\$_ERRORLINES:** It is the nested table of the **SYS.KU\$_ERRORLINE** object type to hold the bulk error information during extraction of each DDL statement.
- **SYS.KU\$_SUBMITRESULT:** It is an object type to capture the complete error information incurred in a DDL statement. The object type structure is as follows:

```
CREATE TYPE sys.ku$_SubmitResult AS OBJECT
(
  ddl sys.ku$_ddl,
  errorLines sys.ku$_ErrorLines
)
/
```

- **SYS.KU\$_SUBMITRESULTS:** It is a nested table of the **SYS.KU\$_SUBMITRESULT** object type to hold multiple DDL statements and corresponding error information.

In the preceding list, **KU\$_PARSED_ITEM** and **KU\$_DDL** are the most frequently used object types of the package.

The following table lists the **DBMS_METADATA** subprograms (reference: Oracle documentation)

Subprogram	Remarks
ADD_TRANSFORM function	Specifies a transform that FETCH_[XML DDL CLOB] applies to the XML representation of the retrieved objects
CLOSE procedure	Invalidates the handle returned by OPEN and cleans up the associated state
CONVERT functions and procedures	Convert an XML document to DDL

Subprogram	Remarks
FETCH_[XML DDL CLOB] functions and procedures	Return metadata for objects meeting the criteria established by OPEN, SET_FILTER, SET_COUNT, ADD_TRANSFORM, and so on
GET_[XML DDL CLOB] functions	Fetch the metadata for a specified object as XML or DDL, using only a single call
GET_QUERY function	Returns the text of the queries that are used by FETCH_[XML DDL CLOB]
OPEN function	Specifies the type of object to be retrieved, the version of its metadata, and the object model
OPENW function	Opens a write context
PUT function	Submits an XML document to the database
SET_COUNT procedure	Specifies the maximum number of objects to be retrieved in a single FETCH_[XML DDL CLOB] call
SET_FILTER procedure	Specifies restrictions on the objects to be retrieved, for example, the object name or schema
SET_PARSE_ITEM procedure	Enables output parsing by specifying an object attribute to be parsed and returned
SET_TRANSFORM_PARAM and SET_REMAP_PARAM procedures	Specify parameters to the XSLT style sheets identified by transform_handle

Out of the preceding list, the subprograms can be segregated based on their work function and utilization:

Subprograms used to retrieve multiple objects from the database	Subprograms used to submit XML metadata to the database
ADD_TRANSFORM function	ADD_TRANSFORM function
CLOSE procedure 2	CLOSE procedure 2
FETCH_[XML DDL CLOB] functions and procedures	CONVERT functions and procedures
GET_QUERY function	OPENW function
GET_[XML DDL CLOB] functions	PUT function
OPEN function	SET_PARSE_ITEM procedure
SET_COUNT procedure	SET_TRANSFORM_PARAM and SET_REMAP_PARAM procedures

Subprograms used to retrieve multiple objects from the database

Subprograms used to submit XML metadata to the database

SET_FILTER procedure
SET_PARSE_ITEM procedure
SET_TRANSFORM_PARAM and
SET_REMAP_PARAM procedures

Parameter requirements

The parameter requirements for the DBMS_METADATA subprograms are as follows:

- Parameters are case sensitive
- Parameters cannot be passed by named notation, but by position only

The DBMS_METADATA transformation parameters and filters

As listed in the preceding API list, the SET_TRANSFORM_PARAM subprogram is used to format and control the DDL output. It is used for both retrieval and submission of metadata from or to the database. It is an overloaded procedure with the following syntax:

```
DBMS_METADATA.SET_TRANSFORM_PARAM
(
  transform_handle IN NUMBER,
  name IN VARCHAR2,
  value IN VARCHAR2,
  object_type IN VARCHAR2 DEFAULT NULL
);
DBMS_METADATA.SET_TRANSFORM_PARAM
(
  transform_handle IN NUMBER,
  name IN VARCHAR2,
  value IN BOOLEAN DEFAULT TRUE,
  object_type IN VARCHAR2 DEFAULT NULL
);
DBMS_METADATA.SET_TRANSFORM_PARAM
(
  transform_handle IN NUMBER,
  name IN VARCHAR2,
  value IN NUMBER,
  object_type IN VARCHAR2 DEFAULT NULL
);
```

From the preceding syntax:

- **TRANSFORM_HANDLE:** It is the handler, either from `ADD_TRANSFORM`, or a generic handler constant `SESSION_TRANSFORM`, to affect the whole session.
- **NAME:** It is the name of the parameter to be modified.
- **VALUE:** It is the transformed value.

Now, we will see some of the common sets of parameters applicable to all objects in a schema:

Parameter	Value	Meaning
PRETTY	TRUE FALSE (default value is TRUE)	If TRUE, produces properly indented output
SQLTERMINATOR	TRUE FALSE (default value is FALSE)	If TRUE, appends SQL terminator (; or /) after each DDL
DEFAULT	TRUE FALSE	If TRUE, resets all parameters to their default state
INHERIT	TRUE FALSE	If TRUE, inherits session level settings

For tables and views, the valid transform handlers are as follows:

Parameter	Value	Meaning
SEGMENT_ATTRIBUTES	TRUE FALSE (default value is TRUE)	If TRUE, includes segment, tablespace, logging and physical attributes
STORAGE	TRUE FALSE (default value is FALSE)	If TRUE, includes storage clause
TABLESPACE	TRUE FALSE	If TRUE, includes tablespace specification
CONSTRAINTS	TRUE FALSE	If TRUE, includes table constraints
REF_CONSTRAINTS	TRUE FALSE	If TRUE, includes referential constraints
CONSTRAINTS_AS_ALTER	TRUE FALSE	If TRUE, includes constraints in the ALTER TABLE statements
OID	TRUE FALSE	If TRUE, includes the object table OID
SIZE_BYTE_KEYWORD	TRUE FALSE	If TRUE, includes the BYTE keywords in string type column specifications
FORCE	TRUE FALSE	If TRUE, creates view with the FORCE option

Filters can be imposed on the working schema objects by using the `DBMS_METADATA.SET_FILTER` procedure. It takes the metadata handle, filter name, and its value as input. It can be used to set include and exclude filters:

```
PROCEDURE set_filter(  
  handle      IN NUMBER,  
  name        IN VARCHAR2,  
  value       IN VARCHAR2|BOOLEAN|NUMBER,  
  object_type_path VARCHAR2  
);
```

Some of the frequently used filters are schema, user, object dependencies, table data, tables, indexes, constraints, and so on. There are more than 70 filters available until Oracle 11g. It can be set as follows:

```
DBMS_METADATA.SET_FILTER(handle, 'SCHEMA', 'ORADEV');  
DBMS_METADATA.SET_FILTER(handle, 'NAME', 'DEPARTMENTS');
```

Working with `DBMS_METADATA`—illustrations

We will illustrate the usage of browsing APIs of `DBMS_METADATA`.

Case 1—retrieve the metadata of a single object

`DBMS_METADATA.GET_DDL` can be called from the `SELECT` query:

```
/*Connect as ORADEV*/  
Conn ORADEV/ORADEV  
Connected.  
  
/*Execute the DBMS_METADATA.GET_DDL to get the DDL for EMPLOYEES  
table*/  
SELECT dbms_metadata.get_ddl('TABLE', 'EMPLOYEES', 'ORADEV')  
FROM DUAL  
/  
EMPLOYEE_DDL  
CREATE TABLE "ORADEV"."EMPLOYEES"  
(  
  "EMPNO" NUMBER(4,0),  
  "ENAME" VARCHAR2(10) NOT NULL ENABLE,  
  "JOB" VARCHAR2(9),  
  "MGR" NUMBER(4,0),  
  "HIREDATE" DATE,  
  "SAL" NUMBER(7,2),  
  "COMM" NUMBER(7,2),  
  "DEPTNO" NUMBER(2,0),
```

```

PRIMARY KEY ("EMPNO")
USING INDEX PCTFREE 10 INITRANS 2 MAXTRANS 255 COMPUTE STATISTICS
(
  INITIAL 65536 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS 2147483645
  PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1 BUFFER_POOL DEFAULT
  FLASH_CACHE DEFAULT CELL_FLASH_CACHE DEFAULT
)
TABLESPACE "USERS" ENABLE
)
SEGMENT CREATION IMMEDIATE
PCTFREE 10 PCTUSED 40 INITRANS 1 MAXTRANS 255 NOCOMPRESS LOGGING
STORAGE(INITIAL 65536 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS
2147483645
 0 FREELISTS 1 FREELIST GROUPS 1 BUFFER_POOL DEFAULT FLASH_CACHE
DEFAULT
  CELL_FLASH_CACHE DEFAULT)
TABLESPACE "USERS"

```

The same DDL generation can also be achieved through a generic function. The function `F_DDL_TABLE` takes a table name as input and fetches its DDL script. It returns the DDL script as CLOB. Since the transform handler opens only for the `TABLE` of `ORADEV` schema, DDL scripts can only be used for tables:

```

/*Create function to get DDL of a given table*/
CREATE OR REPLACE FUNCTION get_ddl_table (p_table_name varchar2)
RETURN CLOB IS
  l_hdl NUMBER;
  l_th NUMBER;
  l_doc CLOB;
BEGIN
  /*specify the OBJECT TYPE*/
  l_hdl := DBMS_METADATA.OPEN('TABLE');

  /*use FILTERS to specify the objects desired*/
  DBMS_METADATA.SET_FILTER(l_hdl , 'SCHEMA', 'ORADEV');
  DBMS_METADATA.SET_FILTER(l_hdl , 'NAME', p_table_name);

  /*request to be TRANSFORMED into creation DDL*/
  l_th := DBMS_METADATA.ADD_TRANSFORM(l_hdl, 'DDL');

  /*FETCH the object*/
  l_doc := DBMS_METADATA.FETCH_CLOB(l_hdl);

  /*release resources*/
  DBMS_METADATA.CLOSE(l_hdl);

  RETURN l_doc;
END;
/
Function created.

```

The above DDL scripts generated from the query and function comprise the storage clause, tablespace, and segment information, thus making them bulky and large. These clauses can be skipped by setting the transform handlers for the session. Before extracting the DDL, we will set the STORAGE, SEGMENT_ATTRIBUTES, PRETTY, SQLTERMINATOR, and REF_CONSTRAINTS handlers for the session transform handler.

```
/*Connect to ORADEV*/
Conn ORADEV/ORADEV
Connected.

/*Set transform handler for STORAGE*/
EXEC DBMS_METADATA.SET_TRANSFORM_PARAM (DBMS_METADATA.SESSION_
TRANSFORM, 'STORAGE', false);

/*Set transform handler for SEGMENT_ATTRIBUTES*/
EXEC DBMS_METADATA.SET_TRANSFORM_PARAM (DBMS_METADATA.SESSION_
TRANSFORM, 'SEGMENT_ATTRIBUTES', false);

/*Set transform handler for PRETTY*/
EXEC DBMS_METADATA.SET_TRANSFORM_PARAM
(DBMS_METADATA.SESSION_TRANSFORM, 'PRETTY', true);

/*Set transform handler for SQLTERMINATOR*/
EXEC DBMS_METADATA.SET_TRANSFORM_PARAM( DBMS_METADATA.SESSION_TRANSFOR
M, 'SQLTERMINATOR', true);

/*Set transform handler for REF_CONSTRAINTS*/
EXEC DBMS_METADATA.SET_TRANSFORM_PARAM(
DBMS_METADATA.SESSION_TRANSFORM, 'REF_CONSTRAINTS', false);

/*Set transform handler for TABLESPACE*/
EXEC DBMS_METADATA.SET_TRANSFORM_PARAM( DBMS_METADATA.SESSION_
TRANSFORM, 'TABLESPACE', false);

/*Set transform handler for SIZE_BYTE_KEYWORD*/
EXEC DBMS_METADATA.SET_TRANSFORM_PARAM(
DBMS_METADATA.SESSION_TRANSFORM, 'SIZE_BYTE_KEYWORD', false);
```

Now, extracting the DDL script in a SELECT statement:

```
/*Generate the DDL for EMPLOYEES table*/
SELECT dbms_metadata.get_ddl('TABLE', 'EMPLOYEES', 'ORADEV') EMPLOYEE_
DDL
FROM dual;

EMPLOYEE_DDL
-----
CREATE TABLE "ORADEV"."EMPLOYEES"
(
  "EMPNO" NUMBER(4,0),
  "ENAME" VARCHAR2(10) NOT NULL ENABLE,
  "JOB" VARCHAR2(9),
```

```

    "MGR" NUMBER(4,0),
    "HIREDATE" DATE,
    "SAL" NUMBER(7,2),
    "COMM" NUMBER(7,2),
    "DEPTNO" NUMBER(2,0),
    PRIMARY KEY ("EMPNO") ENABLE
) ;

```

Case 2—retrieve the object dependencies on the F_GET_LOC function

Refer to the following code snippet:

```

/*Retrieve object dependency for F_GET_LOC function*/
SELECT dbms_metadata.get_dependent_ddl
       ('OBJECT_GRANT','F_GET_LOC','ORADEV') OBJ_GRANTS
FROM DUAL;

OBJ_GRANTS
-----
GRANT EXECUTE ON "ORADEV"."F_GET_LOC" TO "NANCY";

```

The output of the SELECT query shows that the ORADEV user has granted the EXECUTE privilege on the function F_GET_LOC to the NANCY user.

Case 3—retrieve system grants on the ORADEV schema

Refer to the following code snippet:

```

/*Retrieve system grants for the ORADEV user*/
SELECT dbms_metadata.get_granted_ddl
       ('SYSTEM_GRANT','ORADEV') SYS_GRANTS
FROM dual;

SYS_GRANTS
-----
GRANT DEBUG ANY PROCEDURE TO "ORADEV";
GRANT DEBUG CONNECT SESSION TO "ORADEV";
GRANT CREATE ANY CONTEXT TO "ORADEV";
GRANT CREATE LIBRARY TO "ORADEV";
GRANT UNLIMITED TABLESPACE TO "ORADEV";
GRANT CREATE SESSION TO "ORADEV";

```

Case 4—retrieve objects of function type in the ORADEV schema

It follows the same approach from Case 1 for tables owned by the ORADEV user. Here, we will create a generic function to retrieve the DDL of a given function:

```
/*Create the function to generate DDL of a given function*/
CREATE OR REPLACE FUNCTION F_GET_FUN_DDL (P_NAME VARCHAR2)
RETURN CLOB IS
    l_hdl NUMBER;
    l_th NUMBER;
    l_doc CLOB;
BEGIN
/*Open the transform handler*/
    l_hdl := DBMS_METADATA.OPEN('FUNCTION');
/*Set filter for the schema and the function name*/
    DBMS_METADATA.SET_FILTER(l_hdl, 'SCHEMA', 'ORADEV');
    DBMS_METADATA.SET_FILTER(l_hdl, 'NAME', P_NAME);
/*Generate the DDL and fetch in a local CLOB variable*/
    l_th := DBMS_METADATA.ADD_TRANSFORM(l_hdl, 'DDL');
    l_doc := DBMS_METADATA.FETCH_CLOB(l_hdl);
    DBMS_METADATA.CLOSE(l_hdl);
/*Return the DDL*/
    RETURN l_doc;
END;
/
Function created
```

Testing the above function for the function F_GET_LOC:

```
/*Declare an environment variable*/
VARIABLE M_FUN_DDL clob;
/*Execute the function*/
EXEC :M_FUN_DDL := F_GET_FUN_DDL ('F_GET_LOC');
/*Print the variable*/
PRINT M_FUN_DDL
CREATE OR REPLACE FUNCTION "ORADEV"."F_GET_LOC" (P_EMPNO NUMBER)
RETURN NUMBER
IS
CURSOR C_DEPT IS
    SELECT d.loc
    FROM employees e, departments d
    WHERE e.deptno = d.deptno
    AND e.empno = P_EMPNO;
    l_loc NUMBER;
    l_red number;
```

```

    L_BLUE NUMBER;
BEGIN
    OPEN C_DEPT;
    FETCH C_DEPT INTO l_loc;
    CLOSE C_DEPT;
    RETURN l_loc;
END;
```

Similarly, DDL scripts of all functions in a schema can be retrieved by holding all the functions in a cursor and iterating it to call the `F_GET_FUN_DDL` function.

Summary

In this chapter, we understood the usage of Oracle supplied packages and dictionary views to find the coding information. We got introduced to a new feature in Oracle 11g, the PL/Scope tool, and learned how to determine the usage of an identifier in the PL/SQL program. At the end of the chapter, we covered the `DBMS_METADATA` package and demonstrated the extraction of a schema object definition as XML or DDL using the package.

In the next chapter, we will overview the strategies of tracing and profiling in PL/SQL.

Practice exercise

- Which of the following dictionary views is used to get information about the subprogram arguments?
 - `ALL_OBJECTS`
 - `ALL_ARGUMENTS`
 - `ALL_DEPENDENCIES`
 - `ALL_PROGRAMS`
- The tablespace information on a database server:

```

SELECT tablespace_name
FROM DBA_TABLESPACES
/

TABLESPACE_NAME
-----
SYSTEM
UNDOTBS1
TEMP
USERS
EXAMPLE
```

You execute the following command in the session:

```
SQL> ALTER SESSION SET PLScope_SETTINGS = 'IDENTIFIERS:ALL';  
Session altered.
```

Identify the correct statements:

- a. The identifier information would be captured by PL/Scope for the program created or compiled in the session.
 - b. The identifier information would not be captured by PL/Scope as IDENTIFIERS:ALL can be enabled only at the SYSTEM level.
 - c. The identifier information would be captured by PL/Scope only for the programs which are created in the session.
 - d. The identifier information would not be captured by PL/Scope since the SYSAUX tablespace is not available.
3. The parameters specified in DBMS_METADATA are case sensitive:
 - a. True
 - b. False
 4. DBMS_UTILITY.FORMAT_CALL_STACK accomplishes which of the following objectives?
 - a. Captures exceptions in a PL/SQL block.
 - b. Prepares the stack of sequential calls.
 - c. Prepares the stack of execution actions.
 - d. Prepares the stack of block profiler.
 5. Choose the accomplishments of the DBMS_METADATA package.
 - a. Generates a report of invalidated objects in a schema.
 - b. Generates DDL for a given or all object(s) in a schema.
 - c. Generates an object to table dependency report in a schema
 - d. Generates a report of object statistics in a schema
 6. The PL/Scope tool can store the identifier data only in the USERS tablespace.
 - a. True
 - b. False
 7. Which of the following are valid parameter values of SET_TRANSFORM_PARAM for tables?
 - a. STORAGE
 - b. FORCE
 - c. PRETTY
 - d. INHERIT

11

Profiling and Tracing PL/SQL Code

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

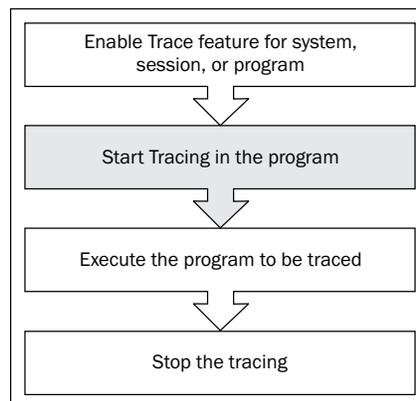
Tracing the PL/SQL programs

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:



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.

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:

DBMS_TRACE constant	Default	Remarks
TRACE_ALL_CALLS	1	Traces all calls
TRACE_ENABLED_CALLS	2	Traces calls which are enabled for tracing
TRACE_ALL_EXCEPTIONS	4	Traces all exceptions
TRACE_ENABLED_EXCEPTIONS	8	Traces exceptions which are enabled for tracing
TRACE_ALL_SQL	32	Traces all SQL statements
TRACE_ENABLED_SQL	64	Traces SQL statements which are enabled for tracing
TRACE_ALL_LINES	128	Traces each line
TRACE_ENABLED_LINES	256	Traces lines which are enabled for tracing
TRACE_PAUSE	4096	Pauses tracing (controls tracing process)
TRACE_RESUME	8192	Resume tracing (controls tracing process)
TRACE_STOP	16384	Stops tracing (controls tracing process)
TRACE_LIMIT	16	Limits the trace information (controls tracing process)
TRACE_MINOR_VERSION	0	Administer tracing process
TRACE_MAJOR_VERSION	1	Administer tracing process
NO_TRACE_ADMINISTRATIVE	32768	Prevents tracing of administrative events such as: <ul style="list-style-type: none">• PL/SQL Trace Tool started• Trace flags changed• PL/SQL Virtual Machine started• PL/SQL Virtual Machine stopped
NO_TRACE_HANDLED_EXCEPTIONS	65536	Prevents tracing of handled exceptions

The subprograms contained in the DBMS_TRACE package are as follows:

DBMS_TRACE subprogram	Remarks
CLEAR_PLSQL_TRACE procedure	Stops trace data dumping in session
GET_PLSQL_TRACE_LEVEL function	Gets the trace level
GET_PLSQL_TRACE_RUNNUMBER function	Gets the current sequence of execution of trace

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]
```

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)

RUN_END	DATE
RUN_FLAGS	VARCHAR2 (2047)
RELATED_RUN	NUMBER
RUN_SYSTEM_INFO	VARCHAR2 (2047)
SPARE1	VARCHAR2 (256)

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:

```
/*Describe the PLSQL_TRACE_EVENTS table structure*/
SQL> desc plsqli_trace_events
```

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

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  
/
```

Synonym created.

```
/*Create synonym for PLSQL_TRACE_RUNNUMBER sequence*/
CREATE PUBLIC SYNONYM plsqli_trace_runnumber FOR plsqli_trace_
runnumber
/
```

Synonym created.

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

Grant succeeded.

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

Grant succeeded.

```
/*Grant privileges on the PLSQL_TRACE_RUNNUMBER*/
GRANT select ON plsqli_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
```

```
/*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 ...)

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
/
```

```
          RUNID RUN_OWNER                RUN_DATE
-----
          1 ORADEV                29-JAN-12
```

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,
```

```

        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:

RUNID	EVENT_COMMENT	EVENT_UNIT	EVENT_UNIT	EVENT_UNIT_KIND	EVENT_LINE
1	PL/SQL Trace Tool started				
1	Trace flags changed				
1	Return from procedure call	SYS	DBMS_TRACE	PACKAGE BODY	21
1	Return from procedure call	SYS	DBMS_TRACE	PACKAGE BODY	76
1	Return from procedure call	SYS	DBMS_TRACE	PACKAGE BODY	81
1	PL/SQL Virtual Machine stopped				
1	PL/SQL Virtual Machine started		<anonymous>	ANONYMOUS BLOCK	0
1	Procedure Call		<anonymous>	ANONYMOUS BLOCK	1
1	Procedure Call	ORADEU	F_GET_LOC	FUNCTION	13
1	Return from procedure call	ORADEU	F_GET_LOC	FUNCTION	9
1	Return from procedure call	ORADEU	F_GET_LOC	FUNCTION	18
1	PL/SQL Virtual Machine stopped				
1	PL/SQL Virtual Machine started		<anonymous>	ANONYMOUS BLOCK	0
1	Procedure Call		<anonymous>	ANONYMOUS BLOCK	3
1	Procedure Call	SYS	DBMS_TRACE	PACKAGE BODY	94
1	Procedure Call	SYS	DBMS_TRACE	PACKAGE BODY	72
1	Procedure Call	SYS	DBMS_TRACE	PACKAGE BODY	66
1	Return from procedure call	SYS	DBMS_TRACE	PACKAGE BODY	12
1	Return from procedure call	SYS	DBMS_TRACE	PACKAGE BODY	67
1	Procedure Call	SYS	DBMS_TRACE	PACKAGE BODY	75
1	PL/SQL trace stopped				

21 rows selected.

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.

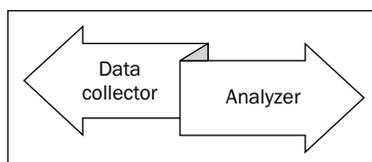
Oracle hierarchical profiler—the DBMS_HPROF package

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.

The following table shows the DBMS_HPROF subprograms:

Subprogram	Description
ANALYZE function	Analyzes the raw profiler output and produces hierarchical profiler information in database tables
START_PROFILING procedure	Starts hierarchical profiler data collection in the user's session
STOP_PROFILING procedure	Stops profiler data collection in the user's session

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:

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

```
/*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*/
```

```
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);

/*Execure 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.".".".__plsqli_vm"
P#X 7
P#C PLSQL.".".".__anonymous_block"
```

```

P#X 695
P#C PLSQL."ORADEV"."F_GET_LOC"::8."F_GET_LOC"#762ba075453b8b0d #1
P#X 6
P#C PLSQL."ORADEV"."F_GET_LOC"::8."F_GET_LOC.C_
DEPT"#980980e97e42f8ec #5
P#X 15
P#C SQL."ORADEV"."F_GET_LOC"::8."__static_sql_exec_line6" #6
P#X 67083
...

```

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_lineline#` indicates dynamic SQL statement execution at line#
 - `__pkg_init` indicates PL/SQL package initialization
 - `__plsqli_vm` indicates PL/SQL virtual machine call
 - `__sql_fetch_lineline#` indicates fetch operation at line#
 - `__static_sql_exec_lineline#` 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.

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

```
/*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:

```
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;
/
```

7. Query the profiling log tables

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

```

      RUNID TOTAL_ELAPSED_TIME RUN_COMMENT
-----
          1          106407 Analyzing the execution of F_GET_LOC

```

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:

RUNID	OWNER	MODULE	TYPE	FUNCTION	NAMESPAC	FUNCTION_ELAPSED_TIME	CALLS
1				__anonymous_block	PLSQL	772	2
1				__plsql_on	PLSQL	23	2
1	ORADEU	F_GET_LOC	FUNCTION	F_GET_LOC	PLSQL	45	1
1	ORADEU	F_GET_LOC	FUNCTION	F_GET_LOC.C_DEPT	PLSQL	21	1
1	SYS	DBMS_HPROP	PACKAGE BODY	STOP_PROFILING	PLSQL	0	1
1	ORADEU	F_GET_LOC	FUNCTION	__sql_fetch_line14	SQL	38463	1
1	ORADEU	F_GET_LOC	FUNCTION	__static_sql_exec_line6	SQL	67683	1

7 rows selected.

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_mf.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

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 microsecs (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 form. 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 Elapsed Time \(microsecs\)](#)

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 Descendants Elapsed Time \(microsecs\)](#)
- [Function Elapsed Time \(microsecs\) Data sorted by Total Function Call Count](#)
- [Function Elapsed Time \(microsecs\) Data sorted by Mean Subtree Elapsed Time \(microsecs\)](#)
- [Function Elapsed Time \(microsecs\) Data sorted by Mean Function Elapsed Time \(microsecs\)](#)
- [Function Elapsed Time \(microsecs\) Data sorted by Mean Descendants Elapsed Time \(microsecs\)](#)
- [Module Elapsed Time \(microsecs\) Data sorted by Total Function Elapsed Time \(microsecs\)](#)
- [Module Elapsed Time \(microsecs\) Data sorted by Module Name](#)
- [Module Elapsed Time \(microsecs\) Data sorted by Total Function Call Count](#)
- [Namespace Elapsed Time \(microsecs\) Data sorted by Total Function Elapsed Time \(microsecs\)](#)
- [Namespace Elapsed Time \(microsecs\) Data sorted by Namespace](#)
- [Namespace Elapsed Time \(microsecs\) Data sorted by Total Function Call Count](#)
- [Parents and Children Elapsed Time \(microsecs\) Data](#)

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:

Function Elapsed Time (microsecs) Data sorted by Total Subtree Elapsed Time (microsecs)

106407 microsecs (elapsed time) & 9 function calls

Subtree	Ind%	Function	Ind%	Descendants	Ind%	Calls	Ind%	Function Name
106407	100%	23	0.0%	106384	100%	2	22.2%	plsql_vm
106384	100%	772	0.7%	105612	99.3%	2	22.2%	anonymous_block
105612	99.3%	45	0.0%	105567	99.2%	1	11.1%	ORADEV_F_GET_LOC_F_GET_LOC (line 1)
67104	63.1%	21	0.0%	67083	63.0%	1	11.1%	ORADEV_F_GET_LOC_F_GET_LOC_C_DEPT (line 5)
67083	63.0%	67083	63.0%	0	0.0%	1	11.1%	ORADEV_F_GET_LOC__static_sql_exec_line6 (line 6)
38463	36.1%	38463	36.1%	0	0.0%	1	11.1%	ORADEV_F_GET_LOC__sql_fetch_line14 (line 14)
0	0.0%	0	0.0%	0	0.0%	1	11.1%	SYS.REPR_HPROF_STOP_PROFILING (line 59)

- 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:

Subtree	Ind%	Function	Ind%	Cum%	Descendants	Ind%	Calls	Ind%	Function Name
67083	63.0%	67083	63.0%	63.0%	0	0.0%	1	11.1%	OPARV.F_GET_LOC_static_sql_exec_line6 (line 6)
38463	36.1%	38463	36.1%	99.2%	0	0.0%	1	11.1%	OPARV.F_GET_LOC_sql_fetch_line14 (line 14)
106384	100%	772	0.7%	100%	105612	99.3%	2	22.2%	_anonymous_block
105612	99.3%	45	0.0%	100%	105567	99.2%	1	11.1%	OPARV.F_GET_LOC.F_GET_LOC (line 1)
106407	100%	23	0.0%	100%	106384	100%	2	22.2%	_plsql_wm
67104	63.1%	21	0.0%	100%	67083	63.0%	1	11.1%	OPARV.F_GET_LOC.F_GET_LOC.C_DEPT (line 8)
0	0.0%	0	0.0%	100%	0	0.0%	1	11.1%	SYS.DUMP_HPROF_STOP_PROFILING (line 52)

- 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:

Function	Ind%	Calls	Ind%	Namespace
861	0.8%	7	77.8%	PLSQL
105546	99.2%	2	22.2%	SQL

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

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.

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

```
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

12

Safeguarding PL/SQL Code against SQL Injection Attacks

Oracle database is, undoubtedly, the uncrowned monarch of "Information Business" across the globe. Though it has narrowed the gap between the expectations and the potential, the question, "Is my information secure?" still hovers the DBMS philosophies. We often discuss the vectors of language strength, performance, storage, and data security. But code vulnerability and security share equal stake in data security. Nevertheless, the strength of SQL and PL/SQL is unquestionable, but vulnerable code writing might motivate a hacker to smuggle through the code and perform vicious manipulations in the data.

In this chapter, we will expand our bandwidth to understand PL/SQL code security. We will understand how "loose code writing" can encompass the code base injection and hence, the data. We will cover the following topics:

- SQL injection
 - Introduction and understanding
- Immunizing SQL injection attacks
 - Reducing the attack surface
 - Avoiding dynamic SQL
 - Using Bind argument
 - Sanitizing inputs with `DBMS_ASSERT`
- Testing the code against the SQL injection flaws

SQL injection—an introduction

SQL injection is a database intrusion that occurs when an unauthorized "malicious" user hacks the PL/SQL code and draws unintended access to the database. Once the code has been cracked, the malicious user can pull out confidential information from the database. There can be many more hazardous consequences of code injection.

In 1998, Rain Forest Puppy (RFP) was the first to identify the "technology vulnerabilities" in his paper "NT Web Technology Vulnerabilities" for "Phrack 54". Later, the injective techniques were studied by many technology experts and evangelists to chalk out the best practices of code writing to dilute such acts. Till date, many application exploitation cases have been registered on account of code injection. For reference, check out <http://www.computerworld.com.au/index.php/id;683627551>. The applications working with personal information or financial data are more prone to injective attacks.

SQL injection—an overview

In the past, the reason for SQL injection was the vulnerability in the middleware—the layer which lies between the data and the client. Unfortunately, it victimizes most of the applications. The middleware layer acts as a communicating interface between the data and the client. The fact that the inputs received from the client acts as the hacker's weapon in major cases is undeniable. However, the code base can sustain such attacks and immunize any chances of SQL injection. Certain penetrable areas have been identified in PL/SQL code developments which can be improvised to safeguard the application against the smuggling attacks. The code sections which are best candidates of attack are:

- PL/SQL subprogram executed with owner's execution rights
- Dynamic SQL using direct inputs in the programs
- Non-sanitized inputs from the client

Hackers can employ a variety of techniques to hit upon penetrable code in an application. A serious injective attack can lead to the leakage of confidential information, unethical data manipulations, or even alteration in user access and the database state.

Let us check out a simple example of code injection.

The salary of an employee is highly confidential in an organization. Suppose a company's finance team uses a P_DISPLAY_SAL procedure to display the salary of all employees as a specific designation.

```
/*Connect to ORADEV user*/
Conn ORADEV/ORADEV
Connected.

/*Enable the SERVEROUTPUT to display the messages*/
SET SERVEROUTPUT ON

/*Create a procedure*/
CREATE OR REPLACE PROCEDURE P_GET_EMP_SAL (P_JOB VARCHAR2)
IS

/*Declare a ref cursor and local variables*/
TYPE C IS REF CURSOR;
CUR_EMP C;
L_ENAME VARCHAR2(100);
L_SAL NUMBER;
L_STMT VARCHAR2(4000);

BEGIN

/*Open the ref cursor for a Dynamic SELECT statement*/
L_STMT := 'SELECT ename, sal
          FROM employees
          WHERE JOB = ''' || P_JOB || '''';
OPEN CUR_EMP FOR L_STMT;
LOOP

/*Fetch the result set and print the result set*/
FETCH CUR_EMP INTO L_ENAME, L_SAL;
EXIT WHEN CUR_EMP%NOTFOUND;
DBMS_OUTPUT.PUT_LINE(RPAD(L_ENAME,6,' ') || '--' || L_SAL);
END LOOP;

CLOSE CUR_EMP;
END;
/

Procedure created.
```

The finance team uses the preceding procedure honestly to display the salaries of all salesmen as follows:

```
/*Testing the procedure for SALESMAN*/  
EXEC P_GET_EMP_SAL ('SALESMAN')  
  
ALLEN --2000  
WARD  --1650  
MARTIN--1650  
TURNER--1900
```

PL/SQL procedure successfully completed.

Now, we will attack the code with a malicious input and pull out the salary details of all the employees:

```
/*Testing the procedure with malicious input*/  
EXEC P_GET_EMP_SAL ('XXX' OR '1'='1')  
  
SMITH --9200  
ALLEN --2000  
WARD  --1650  
JONES --3375  
MARTIN--1650  
BLAKE --3250  
CLARK --2850  
SCOTT --3400  
KING  --5400  
TURNER--1900  
ADAMS --1500  
JAMES --1350  
FORD  --3400  
MILLER--1700
```

PL/SQL procedure successfully completed.

Observe the impact of SQL injection. A hacker can get access to the salary information of all the employees in the organization. An unauthenticated string input is concatenated with a "Always True" condition for unintended execution of a dynamic SQL. Applications, where client input is required to invoke an Oracle subprogram, run a high risk of code injection.

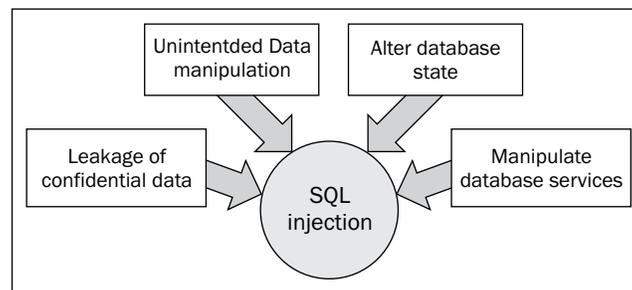
Types of SQL injection attacks

Based on the injection attack type and its impact analysis, the SQL injection attack can be classified into two categories:

- **First-order attack:** When the code attack is done from the client inputs to alter the objective of the invoked Oracle subprogram, the degree of attack is "one". As an impact of the attack, the data may lose its confidentiality due to unauthorized access by the hacker.
- **Second-order attack:** When the attack performs a different activity from the ongoing system activity, the attack comes under second-order attack.

The categories in the preceding list are just a broad categorization of the hacker's activities. Apart from these categories, the hackers can employ different techniques "blindly", resulting in the malfunctioning of the application.

The following diagram branches the impacts of SQL injection:



Preventing SQL injection attacks

SQL injection is a malicious practice, not a bug. Applications cannot be completely shielded against SQL injection but can be immunized against such acts. The code base development should take care to adopt the best practices which can evade the possibilities of code attack.

Let us briefly cover some of the precautionary measures which minimize injection attacks:

- **Avoid dynamic SQL:** Using dynamic SQL with "built up" inputs, they easily fell prey to injection attacks. Embedding dynamic SQLs within the programs must be avoided where static SQLs can be substituted. Static SQL must be used if all the query identifiers are known at the time of the code development. Otherwise, the dynamic SQL must use sanitized inputs or bind arguments to discourage the hackers from breaking through the code.

- **Monitor user privileges to reduce the attack surface:** A user must enjoy only the access for which he is authorized as per his role. Irrelevant and excess privileges must be revoked to reduce the access perimeter of a user.
In addition, the PL/SQL subprograms must be invoked by the invoker's rights and not the owner's or the creator's rights.
- **Use bind arguments:** The dynamic SQLs seeking inputs must make use of the bind arguments. It is a highly recommended programming tip to reduce the injective attacks on the code. It reduces the possibility of breaking through the code by providing concatenated inputs. Besides shielding against the code attack, bind arguments also improve code performance. It is because the usage of bind variables in a query avoids hard parsing and it pushes the Oracle server to reuse the execution plans for the SQL queries.
- **Sanitize client inputs with DBMS_ASSERT:** The inputs from the client must be verified before using them in the program logic. The `DBMS_ASSERT` package provides niche subprograms to validate the inputs from the application layer.

Immunizing SQL injection attacks

We will discuss the ways to immunize code against SQL injection in detail. Besides the ways which are listed above, we will discuss some additional tricks too, to reduce SQL injection attacks.

Reducing the attack's surface

Reducing the attack's surface is one of the preventive measures that are proactively used to fight the SQL injection attacks. It aims to minimize the area of operation and visibility of the hackers by controlling the privileges and execution rights of a user on the accessible subprograms. The technique is helpful when a user plays a defined role in an application but is still bestowed with a lot more irrelevant privileges from the admin. The attack perimeter can be reduced by:

- Controlling the user privileges
- Creating the program units with invoker's rights

Controlling user privileges

The DBA must keep a hawk eye on the roles of the users in the application to prevent any malicious motivation. The availability of additional spare privileges might end up in misuse and, hence, might threaten the database security. The DBA must revoke irrelevant privileges from the user. For example, a user, UREP, plays the role of a report generator in a team. As per his role, he must have only the `SELECT` privilege on the tables; he should not have rights to perform any transaction. The DBA must revoke the DML privileges:

```
SQL> REVOKE INSERT, UPDATE, DELETE ON EMPLOYEES FROM UREP  
/
```

```
Revoke succeeded.
```

Besides controlling the user privileges, the client-based application must intelligently handle the exposing of the database APIs and the required inputs. The end user interfaces must use only driving APIs which require user input. The user inputs should be treated with their actual data types instead of type casts.

Invoker's and definer's rights

As per the default behavior of Oracle, a subprogram is executed by its owner's or definer's rights.

Suppose a user A created a procedure P to insert sales data in the `SALES` table. The user A grants the `EXECUTE` privilege on procedure P to another user B, who has no such privilege to insert data into the `SALES` table. The user B executes the procedure P. Will it be executed successfully? Of course, it will execute because the user B executes the procedure P with its owner's rights which have the privilege to create sales data. This implies that the definer's rights not only offer subprogram execution privileges but also share privileges on the objects which are referenced inside the subprogram body.

This default behavior can become chaotic, if wrongly used. An attacker can get unauthorized access to an API, which can be used vindictively. In such cases, the subprogram invokers' rights must override the subprogram definer's rights. The `AUTHID CURRENT_USER` clause is used to override the invokers' rights over the definer's rights.

We will conduct a small case study to understand the fact that a user must invoke a non-owned subprogram at the cost of his owned rights.

We create a procedure to modify the default tablespace of a user in SYS. Note that only DBA has the privilege to modify the tablespace of a user. The following program has been created for demonstration purpose only:

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

/*Enable the SERVEROUTPUT to display the messages*/
SET SERVEROUTPUT ON

/*Create the procedure to alter the tablespace*/
CREATE OR REPLACE PROCEDURE p_mod_tablespace
(P_USERNAME VARCHAR2 DEFAULT NULL,
 P_TABLESPACE VARCHAR2 DEFAULT NULL)
IS
  V_STMT VARCHAR2(500);
BEGIN

/*Dynamically alter the user to modify default tablespace*/
  V_STMT:='ALTER USER '||p_username ||
          ' default tablespace '|| P_TABLESPACE;

/*Execute the dynamic statement*/
  EXECUTE IMMEDIATE v_stmt;
END p_mod_tablespace;
/
```

Procedure created.

For demonstration purpose, the DBA grants the EXECUTE privilege on the procedure to the ORADEV user.

```
/*Grant execute on the procedure to ORADEV*/
SQL> GRANT execute ON p_mod_tablespace TO ORADEV
/
```

Grant succeeded.

Verify that the ORADEV user doesn't has sufficient privilege to modify the tablespace of a user:

```
/*Connect to ORADEV user*/
SQL> CONN ORADEV/ORADEV
Connected.
```

```

/*Verify the privileges of ORADEV user*/
SQL> ALTER USER nancy DEFAULT TABLESPACE system
/
ALTER USER NANCY DEFAULT TABLESPACE USERS
*
ERROR at line 1:
ORA-01031: insufficient privileges

```

The ORADEV user executes the P_MOD_TABLESPACE procedure for the preceding operation:

```

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

/*Enable the SERVEROUTPUT to display the messages*/
SET SERVEROUTPUT ON

/*Execute the procedure to modify the default tablespace of user
NANCY*/
SQL> EXEC SYS.P_MOD_TABLESPACE ('NANCY','SYSTEM');

PL/SQL procedure successfully completed.

```

The procedure executed successfully because it uses the execution privileges of SYS, not of ORADEV. The ORADEV user enjoys only the invocation privilege.

Verify the change in the tablespace for the NANCY user

```

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

/*Check the default tablespace of NANCY*/
SELECT username, default_tablespace
FROM dba_users
WHERE username='NANCY'
/

```

USERNAME	DEFAULT_TABLESPACE
NANCY	SYSTEM

Notice that a user can perform unauthorized activities as he executes the subprogram with the definer's rights. The DBA must realize the unintentional attacks resulting in the modification of important information.

Let us recreate the P_MOD_TABLESPACE procedure with the AUTHID CURRENT_USER option and repeat the steps:

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

/*Enable the SERVEROUTPUT to display the messages*/
SET SERVEROUTPUT ON

/*Create the procedure to alter the tablespace*/
CREATE OR REPLACE PROCEDURE p_mod_tablespace
(P_USERNAME VARCHAR2 DEFAULT NULL,
 P_TABLESPACE VARCHAR2 DEFAULT NULL)
/*Specify the AUTHID CURRENT_USER clause*/
AUTHID CURRENT_USER
IS
  V_STMT VARCHAR2(500);
BEGIN
/*Dynamically alter the user to modify default tablespace*/
  V_STMT:='ALTER USER '||p_username ||
          ' default tablespace '|| P_TABLESPACE;

/*Execute the dynamic statement*/
  EXECUTE IMMEDIATE v_stmt;
END p_mod_tablespace;
/

Procedure created.
```

Now, the ORADEV user reconnects and invokes the procedure to revert back the tablespace changes in the last activity:

```
/*Connect to ORADEV user*/
Conn ORADEV/ORADEV
Connected.

/*Execute the procedure to modify tablespace of NANCY back to USERS*/
SQL> EXEC SYS.P_MOD_TABLESPACE ('NANCY','USERS');
BEGIN SYS.P_MOD_TABLESPACE ('NANCY','USERS'); END;

*
ERROR at line 1:
ORA-01031: insufficient privileges
ORA-06512: at "SYS.P_MOD_TABLESPACE", line 9
ORA-06512: at line 1
```

Now, the procedure execution fails as it is executed with the privileges of ORADEV and not of SYS. As ORADEV is a normal user, he cannot update the tablespace information for a user. Thus, the AUTHID CURRENT_USER clause can be used to minimize the chances of misusing the privileges.

Avoiding dynamic SQL

Dynamic SQL is the most vulnerable point identifiable in a PL/SQL program. A dynamically built up SELECT statement, which uses the parameter accepted by the subprogram, is an open invitation to attackers. In these scenarios, developers must predict and discover the scalability of the SQL query. If the query identifiers such as selected columns and table name are known at the runtime, static SQL must be encouraged. Dynamic SQL must come into the picture only when the complete SQL query has to be built up during runtime or dynamic DDL statements.

Static SQL statements in the PL/SQL program run rare threat of injection unless the attacker achieves code writing access. They are performance efficient also as they reduce the time consumed in identifier substitution and query building.

Let us observe the above recommendations in the following illustration.

The following P_SHOW_DEPT procedure accepts an employee ID and displays the corresponding department number:

```
/*Connect to ORADEV user*/
Conn ORADEV/ORADEV
Connected.

/*Enable the SERVEROUTPUT to display the messages*/
SET SERVEROUTPUT ON

/*Create the procedure*/
CREATE OR REPLACE PROCEDURE P_SHOW_DEPT
(P_ENAME VARCHAR2)
IS
    CUR SYS_REFCURSOR;
    l_ename VARCHAR2(100);
    l_deptno NUMBER;
BEGIN

/*Open ref cursor for a dynamic query using the input parameter*/
    OPEN CUR FOR 'SELECT ename, deptno
                FROM employees
                WHERE ename = ' || P_ENAME;
```

```
LOOP

/*Fetch and display the results*/
  FETCH CUR INTO l_ename, l_deptno;
  EXIT WHEN cur%notfound;
      DBMS_OUTPUT.PUT_LINE(RPAD(l_ename,6,' ') ||'--' || l_deptno);
  END LOOP;
END;
/
```

Procedure created.

```
/*Testing the procedure*/
SQL> EXEC p_show_dept (''KING'');
KING  --10
```

PL/SQL procedure successfully completed.

We will demonstrate how the objective of the procedure got changed due to the malicious inputs. The procedure was used to display the departments of employees but out of surprise, it can list the employees' salaries, too. The procedure input substituting an operand in the WHERE clause predicate is a clear threat to the data.

```
/*Invoking the procedure for a malicious input*/
EXEC P_SHOW_DEPT ('null UNION SELECT ENAME, SAL FROM EMPLOYEES');
```

```
ADAMS  --1500
ALLEN  --2000
BLAKE  --3250
CLARK  --2850
FORD   --3400
JAMES  --1350
JONES  --3375
KING   --5400
MARTIN--1650
MILLER--1700
SCOTT  --3400
SMITH  --9200
turner--1900
WARD   --1650
```

PL/SQL procedure successfully completed.

The preceding case demonstrates the vulnerability in dynamic SQLs. Leakage of confidential data!

Now as the query identifiers are already known in this case, we can replace the dynamic SQL by a static SQL. The P_SHOW_DEPT procedure can be rewritten as follows:

```

/*Create the procedure*/
CREATE OR REPLACE PROCEDURE P_SHOW_DEPT
(P_ENAME VARCHAR2)
IS
  CUR SYS_REFCURSOR;
  l_ename VARCHAR2(100);
  l_deptno NUMBER;
BEGIN
  /*Open ref cursor for a static query using the input parameter*/
  OPEN CUR FOR SELECT ename, deptno
                FROM employees
                WHERE ename = P_ENAME;

  LOOP
  /*Fetch and display the results*/
  FETCH CUR INTO l_ename, l_deptno;
  EXIT WHEN cur%notfound;
  DBMS_OUTPUT.PUT_LINE(RPAD(l_ename,6,' ') || '--' || l_deptno);
  END LOOP;
END;
/

Procedure created.

/*Testing the procedure*/
SQL> EXEC p_show_dept ('KING');
KING  --10

PL/SQL procedure successfully completed.

```

In the preceding code demonstration, notice that the query framing in a PL/SQL block minimizes the possibility of injection. In a dynamically framed SQL statement as a string, incoming parameters leave an open loop hole to the string where malicious inputs can flow in to deform the query. But static SQL queries use the input variables directly in the query predicates, which lowers down the probability of undesired deformation of the query. Now when we attempt to inject the procedure call again with the same input, the static SQL doesn't return any result, thus guarding the procedure against injective attacks.

```

/*Invoking the procedure for a malicious input*/
EXEC P_SHOW_DEPT ('null UNION SELECT ENAME, SAL FROM EMPLOYEES');

PL/SQL procedure successfully completed.

```

Another recommendation to immunize against attacks in dynamic SQLs is the usage of bind arguments. Dynamic SQLs must be implemented with bind arguments. We will see how to work with bind arguments in the next section.

Bind arguments

Bind arguments act as the placeholder in the dynamic SQL query. They can be substituted with actual arguments during query building at runtime. They minimize code injection attacks and yield good performance, too.

Dynamic SQL using concatenated inputs can substitute the concatenated parameter with a placeholder in the dynamic SQL or dynamic PL/SQL. At runtime, the placeholder can be replaced with an actual argument through the `USING` clause in the same positional order. Bind variables can successfully substitute the placeholders for the value operands in the `WHERE` clause.

Let us check the usage of bind arguments in our earlier example. We will recreate the `P_SHOW_DEPT` procedure using dynamic SQL and bind arguments. We will replace the parameter substitution in the dynamic SQL with a placeholder or bind variable (`:bind`).

```
/*Connect to ORADEV user*/
Conn ORADEV/ORADEV
Connected.

/*Enable the SERVEROUTPUT to display the messages*/
SET SERVEROUTPUT ON

/*Create the procedure*/
CREATE OR REPLACE PROCEDURE P_SHOW_DEPT
(P_ENAME VARCHAR2)
IS
    CUR SYS_REFCURSOR;
    l_ename VARCHAR2(100);
    l_deptno NUMBER;
BEGIN

/*Open ref cursor for a dynamic query using a bind variable*/
    OPEN CUR FOR 'SELECT ename, deptno
                FROM employees
                WHERE ename = :bind' USING P_ENAME;
    LOOP
```

```

/*Fetch and display the results*/
  FETCH CUR INTO l_ename, l_deptno;
  EXIT WHEN cur%notfound;
      DBMS_OUTPUT.PUT_LINE(l_ename || '--' || l_deptno);
  END LOOP;
END;
/

```

Procedure created.

```

/*Testing the procedure*/
SQL> EXEC p_show_dept ('KING');
KING--10

```

PL/SQL procedure successfully completed.

Now we will test the procedure against the malicious input:

```

/*Invoking the procedure for a malicious input*/
SQL> EXEC P_SHOW_DEPT ('null UNION SELECT ENAME, SAL FROM EMPLOYEES');

```

PL/SQL procedure successfully completed.

Once again, no result is returned. It is because the placeholder in the dynamic SQL substitutes a single string value. Here, a single string value is treated as `null UNION SELECT ENAME, SAL FROM EMPLOYEES`, which doesn't exist in the database.

Bind variables can be used as a placeholder in the dynamic SQL to substitute all types of inputs. It can be query identifiers such as columns or table names, keywords, and operands (like the one we just saw above). They are the preferred choice when the dynamic query uses the `IN` list or the `LIKE` operator.

As a limitation, bind arguments cannot substitute Oracle identifiers and keywords. Bind variables cannot be used in DDL statements and, also, they cannot substitute identifiers or keywords in a dynamic `SELECT` query.

Sanitizing inputs using `DBMS_ASSERT`

The inputs flowing from the client are another threat to code attacks. The string inputs to the dynamic SQL which do not use bind variables must be properly verified for purity and sanity before using them in the dynamic build of a SQL query. Frankly, it is the responsibility of both the client layer and middleware layer to authenticate the inputs. The client can programmatically perform the basic validation. A second layer check must be set up at the database side to sanitize the inputs supplied from the client.

Input sanitization becomes a mandatory activity when the dynamic SQL requires the substitution of Oracle identifiers.

The DBMS_ASSERT package

Oracle 10g release 2 introduced the DBMS_ASSERT package to sanitize the user inputs. The inputs from the application layer can be supplied to the DBMS_ASSERT subprograms and verified before they are employed in the program.

The DBMS_ASSERT package is owned by SYS and contains seven subprograms. These subprograms are listed in the following table:

Subprograms	Description
ENQUOTE_LITERAL function	Encloses a string literal within single quotes
ENQUOTE_NAME function	Encloses the input string in double quotes
NOOP functions	Overloaded function returns the value without any checking; does no operation
QUALIFIED_SQL_NAME function	Verifies that the input string is a qualified SQL name
SCHEMA_NAME function	Verifies that the input string is an existing schema name
SIMPLE_SQL_NAME function	Verifies that the input string is a simple SQL name
SQL_OBJECT_NAME function	Verifies that the input parameter string is a qualified SQL identifier of an existing SQL object

The most significant feature of DBMS_ASSERT is that most of its subprograms return the same input parameter as the output, after checking its properties. If the input fails expected "property", the VALUE_ERROR exception is raised.

Let us check the working of the subprograms.

The ENQUOTE_LITERAL subprogram can be used to sanitize the string inputs by enclosing them in single quotes. This function eliminates the possibility of leaking information by cladding an additional query using the UNION set operator:

```
/*Demonstrate the use of ENQUOTE_LITERAL*/
SELECT DBMS_ASSERT.ENQUOTE_LITERAL('KING')
FROM DUAL
/

DBMS_ASSERT.ENQUOTE_LITERAL('KING')
-----
'KING'
```

The `ENQUOTE_NAME` can be used to enclose Oracle identifiers in double quotes and verify quoted identifiers.

```

/*Demonstrate the use of ENQUOTE_NAME*/
SELECT DBMS_ASSERT.ENQUOTE_NAME('KING')
FROM DUAL
/

DBMS_ASSERT.ENQUOTE_NAME('KING')
-----
"KING"

```

The `SCHEMA_NAME` function validates the current schema name. This eliminates the possibility of accessing other schema objects:

```

/*Demonstrate the use of SCHEMA_NAME*/
SELECT DBMS_ASSERT.SCHEMA_NAME('PLSQL')
from dual
/

SELECT DBMS_ASSERT.SCHEMA_NAME('PLSQL')
*
ERROR at line 1:
ORA-44001: invalid schema
ORA-06512: at "SYS.DBMS_ASSERT", line 243

SELECT DBMS_ASSERT.SCHEMA_NAME('ORADEV')
from dual
/

DBMS_ASSERT.SCHEMA_NAME('ORADEV')
-----
ORADEV

```

Other subprograms — `SIMPLE_SQL_NAME` and `SQL_OBJECT_NAME` — are also of great relevance to validate the schema object names.

Identifier formatting and verification process

Oracle identifiers can be used in multiple contexts with different behaviors. This study is important to ensure the correct usage of an appropriate subprogram from a `DBMS_ASSERT` subprogram list. An identifier can be a quoted identifier, unquoted identifier, and a literal. All three contexts are entirely different from each other. Based on the context of the identifier in the scenario, the verification algorithm must be selected for sanitization.

We will check out for the identifier contexts. The different contexts are listed as follows:

- **Unquoted identifier:** This identifier obeys the naming convention of Oracle – it must begin with a letter followed by numbers or a set of defined special characters (`_`).

```
/*Use (employees) as unquoted identifier*/  
SELECT * FROM employees  
/
```

In the preceding query, the `employees` table acts as an unquoted identifier.

- **Quoted identifier:** It is enclosed with double quotes and follows no naming convention. It can start with a number (optionally) and can include any sort of characters.

```
/*Use (employees) as quoted identifier*/  
SELECT * FROM "employees"  
/
```

In the preceding query, the quoted identifier `"employees"` is different from the unquoted identifier `employees`. Quoted identifiers can be used as a method of code attack.

- **Literal:** It can be any fixed value used in the SQL query.

```
/*Demonstrate a literal*/  
SELECT * FROM employees WHERE ename = 'KING'  
/
```

In the preceding query, `'KING'` is a literal.

```
/*Use (employees) as a literal*/  
SELECT * FROM user_tables WHERE table_name='EMPLOYEEES'  
/
```

Note that the `EMPLOYEEES` table (identifier) acts as a literal in the preceding query.

Appropriate usage of verification algorithm is necessary to ensure the sanity of the identifier. We saw that an identifier can be any of the following:

- **Basic:** This identifier is always an unquoted identifier. As the Basic identifier is built up of a defined set of characters, it requires less formatting and is deemed to be sanitized.
- **Simple:** This identifier may or may not be a quoted identifier. The `SIMPLE_SQL_NAME` function can be used to verify its purity and sanity. It checks only the admissible character sets and not the length of the identifier.

If the identifier is unquoted, the function checks for the naming convention as applied to Basic identifier (allowed character set is A-Z, a-z, 0-9, \$, #, and _).

If the identifier is quoted, it can include any character set within the double quotes.

Check out the following illustration:

```
/*Demonstrate verification algorithm for Simple identifier using
quoted identifier*/
SQL> select DBMS_ASSERT.SIMPLE_SQL_NAME('"1select"')
       from dual
       /

DBMS_ASSERT.SIMPLE_SQL_NAME('"1SELECT"')
-----
"1select"

/*Demonstrate verification algorithm for Simple identifier using
unquoted identifier*/
SQL> select DBMS_ASSERT.SIMPLE_SQL_NAME('1select')
       from dual
       /
select DBMS_ASSERT.SIMPLE_SQL_NAME('1select')
       *
ERROR at line 1:
ORA-44003: invalid SQL name
ORA-06512: at "SYS.DBMS_ASSERT", line 146
```

- **Qualified:** This identifier is mainly used for the sanity check of database links, but behaves in a similar way to simple SQL names in most of cases. It can include more than one simple SQL name as a schema name, object and a DB link, too. They can follow any one of the following syntax:

```
<local qualified name> ::= <simple name> {'.' <simple name>}
<database link name> ::= <local qualified name> ['@' <connection
string>]
<connection string> ::= <simple name>
<qualified name> ::= <local qualified name> ['@' <database link
name>]
```

Check out the difference between SIMPLE_SQL_NAME and QUALIFIED_SQL_NAME in the following illustration:

```
/*Demonstrate verification algorithm for Qualified SQL identifier
using SIMPLE_SQL_NAME*/
SQL> select dbms_assert.simple_sql_name('schema.obj@dblink')
```

```
        from dual
      /
select dbms_assert.simple_sql_name('schema.obj@dblink') from dual
      *
ERROR at line 1:
ORA-44003: invalid SQL name
ORA-06512: at "SYS.DBMS_ASSERT", line 146

/*Demonstrate verification algorithm for Qualified SQL identifier
using QUALIFIED_SQL_NAME*/
SQL> select dbms_assert.qualified_sql_name('schema.obj@dblink')
      from dual
      /

DBMS_ASSERT.QUALIFIED_SQL_NAME('SCHEMA.OBJ@DBLINK')
-----
schema.obj@dblink
```

The behavior of quoted and unquoted qualified SQL names remains same as we discussed earlier.

DBMS_ASSERT—usage guidelines

The best practices to use the DBMS_ASSERT validation package are as follows:

- Unnecessary uppercase conversion of identifiers must be avoided.
 - **Case 1:** The following statement is not the correct usage of basic unquoted identifiers:

```
BAD_USAGE := sys.dbms_assert.SCHEMA_NAME(UPPER(MY_SCHEMA));
```

As the SCHEMA_NAME function is case sensitive, the quoted inputs must be provided to check their sanity. Explicit transformation of the identifiers must be avoided to ensure the accuracy of the result. If the input schema name is not a valid schema, Oracle raises a ORA-44001: invalid schema exception.

- **Case 2:** The following statement is a better practice to use the unquoted identifier:
 - **Case 3:** The best way to avoid any possibility of bad input is demonstrated in the following statement. The schema name has been unquoted by setting off the ENQUOTE property:

```
BEST_USAGE := sys.dbms_assert.ENQUOTE_NAME(
  sys.dbms_assert.SCHEMA_NAME(MY_SCHEMA), FALSE);
```

- Escape quotation marks when using `ENQUOTE_NAME`—avoid unnecessary quoting of identifiers when using `ENQUOTE_NAME`. Similarly, for `ENQUOTE_LITERAL`, single quotes in the input must be prevented. Note that `ENQUOTE_NAME` must be used with quoted identifiers only.
- `NULL` results from `DBMS_ASSERT` must be ignored. The subprograms `SIMPLE_SQL_NAME`, `QUALIFIED_SQL_NAME`, `SCHEMA_NAME`, and `SQL_OBJECT_NAME` sanitize the identifiers and the results returned are same as the input. Other subprograms such as `NOOP`, `ENQUOTE_NAME` and `ENQUOTE_LITERAL` can accept `NULL` values.
- Length validation check must be enforced in addition to the `DBMS_ASSERT` verification algorithms.
- Protect all injection prone parameters and code paths.
- The `DBMS_ASSERT` also exists as a public synonym; it is recommended to make all references to its subprograms by prefixing `SYS` which is the owning schema.
- Make use of `DBMS_ASSERT` specific exceptions to identify the actual exception raised by the bad inputs. The exceptions `ORA44001` to `ORA44004` are the `DBMS_ASSERT` exceptions:
 - `ORA44001` stands for `sys.dbms_assert.INVALID_SCHEMA_NAME`
 - `ORA44003` stands for `sys.dbms_assert.INVALID_SQL_NAME`
 - `ORA44002` stands for `sys.dbms_assert.INVALID_OBJECT_NAME`
 - `ORA44004` stands for `sys.dbms_assert.QUALIFIED_SQL_NAME`

DBMS_ASSERT—limitations

The `DBMS_ASSERT` package has certain limitations, as follows:

- No validation for TNS connection strings
- No validation for buffer overflow attacks
- It only checks the value property of an input value, it doesn't parse property of a value as a database identifier
- No validation for unprivileged access of objects

Testing the code for SQL injection flaws

Until now, we discussed the symptoms and remedies of SQL injection. We demonstrated the programming recommendations to mitigate the effects of code injections and smuggles. Assuring code quality and testing play a crucial role in taking preventive measures against hackers. Code testing resources must adopt a concrete strategy to discover and hit upon the code vulnerabilities before it invites an attacker to exploit the database. Now, we will discuss some of the testing considerations to test the code for SQL injection flaws.

Test strategy

A logical and effective test strategy must be employed to discover injection flaws. Of course, there is no magic practice to ooze out all flaws in the code.

The usual code reviews are part of static testing while testing the programs with sample data and inputs come under dynamic testing. These days, static testing has been absorbed into the development stage where developers, their peers and seniors review the code. Major syntactical errors, logical issues, code practices, and injection bugs can be traced at this level. The Dry Run concept can even check multiple scenarios and ensure bug-free application submission to the quality assurance team.

Reviewing the code

As a reviewer of the code, the first and foremost step is to check the attack surface. The code reviewer must measure the exposure of database programs in the client. In addition, he must check the privilege set available with the database users. Once these steps are passed with satisfaction, he must get into the code to search for key vulnerable areas.

In PL/SQL-based applications, always be careful to look for:

- Dynamic SQL build ups:
 - EXECUTE IMMEDIATE
 - REF CURSOR queries
 - DBMS_SQL
 - DBMS_SYS_SQL
- Check for appropriate usage of bind arguments
- Parameter input sanitization

Similarly, in Java or C client architecture, the reviewer must look for dynamic callable statement preparation.

Static code analysis

SQL injection attacks are mostly due to coding unawareness and dynamic SQLs. Therefore, static code analyzers cannot easily trace the application vulnerability. From the Oracle documentation, the term "Static Code Analysis" can be defined as follows:

Static code analysis is the analysis of computer software that is performed without executing programs built from that software. In most cases, the analysis is performed on some version of the source code and in other cases, some form of the object code. The term is usually applied to the analysis performed by an automated tool.

It is advisable that such analysis tools should not be considered as the testing benchmark and confirmatory tools. Instead, they can be used for white box testing where the application is tested for smooth logical flow and program executions for different nature of input data.

Fuzz tools

Fuzz testing is similar to doing "Bungee jumping" with the code. It is rough testing, which is not based out on any logic, but meant to measure the security and scalability of the application. It measures the sustainable degree up of an application to the bad and malicious inputs. Without any preconception of the system or program behavior, it uses raw inputs to check the program semantics. The environment for fuzz testing tools can be made explicitly by modifying the context values and manipulating the test data.

The bugs reported in fuzz testing may not always be real threats to the application, but they can be a crucial clue to the vulnerability and injective attacks.

Generating test cases

The last and final call is the preparation of test cases. Though it is kept aside very often during the application development, test cases are a crucial stage to measure the strengths, robustness, and client input validation. Remember the following points:

- Each input from the client must be individually tested. All the remaining parameters should be kept unchanged while generating a test case for varied behavior of client input.
- The best way to test SQL injection is to provide junk data, concatenated string inputs, and many more.
- Test with varied nature of input; try with object names, identifiers, dummy names to arrive at a positive conclusion.

Summary

In the chapter, we learned a malicious hacking concept – SQL Injection. We understood the causes of code attack and its impact on the database. We covered the techniques to safeguard an application against the injective attacks through demonstrations and illustrations. At the end of the chapter, we discussed some of the testing considerations to hit the vulnerable areas in the code.

Practice exercise

1. Which method would you employ to immunize the PL/SQL code against SQL Injection attacks?
 - a. Replace Dynamic SQLs with Static SQLs.
 - b. Replace concatenated inputs in Dynamic SQL with binds arguments.
 - c. Declare the PL/SQL program to be executed by its invoker's rights.
 - d. Removing string type parameters from the procedure.
2. Use static SQL to avoid SQL injection when all Oracle identifiers are known at the time of code execution.
 - a. True
 - b. False
3. Choose the impact of SQL injection attacks:
 - a. Malicious string inputs can extract confidential information.
 - b. Unauthorized access can drop a database.
 - c. It can insert ORDER data in EMPLOYEES table.
 - d. A procedure executed with owners' (SYS) rights can change the password of a user.
4. Pick the correct strategies to fight against of SQL injection
 - a. Sanitize the malicious inputs from the application layer with DBMS_ASSERT.
 - b. Remove string concatenated inputs from the Oracle subprogram.
 - c. Dynamic SQL should be removed from the stage.
 - d. Execute a PL/SQL program with its creator's rights.

-
5. Statistical Code analysis provides an efficient technique to trace application vulnerability by using ideal and expected parameter values.
 - a. True
 - b. False
 6. Fuzz tool technique is a harsh and rigorous format of testing which uses raw inputs and checks a program's sanctity.
 - a. True
 - b. False
 7. Choose the accomplishments achieved by the DBMS_ASSERT package to prevent SQL injection?
 - a. Enclose a given string in single quotes.
 - b. Enclose a given string in double quotes.
 - c. Verify a schema object name.
 - d. Verify a SQL simple and qualified SQL identifier.
 8. Identify the nature of a table name in the following SELECT statement

```
SELECT TOTAL
FROM "ORDERS"
WHERE ORD_ID = P_ORDID
/
```

 - a. Unquoted identifier
 - b. Quoted identifier
 - c. Literal
 - d. Placeholder
 9. Which of the following DBMS_ASSERT subprogram modifies the input value?
 - a. SIMPLE_SQL_NAME
 - b. ENQUOTE_LITERAL
 - c. QUALIFIED_SQL_NAME
 - d. NOOP

10. The code reviews must identify certain vulnerable key areas for SQL Injection. Select the correct ones from the following list:
 - a. DBMS_SQL
 - b. BULK COLLECT
 - c. EXECUTE IMMEDIATE
 - d. REF CURSOR

11. The AUTHID CURRENT_USER clause achieves which of the following purposes?
 - a. Code executes with invoker's rights.
 - b. Code executes with current logged in user.
 - c. Eliminates SQL injection vulnerability.
 - d. Code executes with the creator's rights.

Answers to Practice Questions

Chapter 1, Overview of PL/SQL Programming Concepts

Question No.	Answer	Explanation
1	c	Currently, SQL Developer doesn't provide backup and recovery features. However, it can be done using a regular database export from SQL Developer.
2	a, c, and d	A function can be called from a SQL expression only if it doesn't hinder the database state and purity level.
3	a	The ALL_DEPENDENCIES dictionary view has been filtered by REFERENCED_TYPE and REFERENCED_OWNER for SYS owned tables and views.
4	c	The local variables are local to the block only. They cannot be referred outside their native PL/SQL block.
5	a, b, and d	An exception variable cannot be simply declared and used with RAISE_APPLICATION_ERROR. It has to be mapped to a self defined error number using PRAGMA EXCEPTION_INIT, and then raised through RAISE_APPLICATION_ERROR with an exception message.

Question No.	Answer	Explanation
6	a and b	<p>A function must return a value using the RETURN statement while a procedure might return a value through the OUT parameters.</p> <p>Standalone subprograms (functions and procedures) cannot be overloaded. Only the subprograms declared in a package can be overloaded.</p> <p>Both procedures and functions can accept parameters in either of the two modes – pass by value and pass by reference.</p>
7	c	<p>For implicit cursors, the %FOUND attribute is set to TRUE, if the SQL statement fetches a minimum of one record.</p>

Chapter 2, Designing PL/SQL Code

Question No.	Answer	Explanation
1	c	<p>All the cursor attributes, except %ISOPEN, must be accessed within the cursor execution cycle. Once the cursor is closed, the cursor work area is flushed.</p>
2	b and c	<p>The use of cursor FOR loops prevents erratic coding. Fetching the cursor data into a record reduces the overhead of declaring block variables.</p>
3	b	<p>Implicit cursor attributes hold the value of the last executed SQL query. Therefore, it must be referenced just after the SQL query</p>
4	a and b	<p>Cursor variables can point to several cursor objects (cursor work area) in shared memory. Ref cursor types can be declared in a package specification.</p>
5	a	<p>A strong ref cursor must mandatorily have the RETURN type specification. The RETURN type can be a table record structure or a user-defined type.</p>

Question No.	Answer	Explanation
6	a and b	Cursor variables can dynamically point to different work areas and, hence, different result sets. The biggest advantage of cursor variables is their ability to share the pointer variable amongst the client environments and other subprograms.
7	b	As the OPEN stage of cursor variables has to be explicitly specified, it cannot be opened with the cursor FOR loops.
8	a and d	A subtype inherits the complete table record structure and the NULL property of its columns.

Chapter 3, Using Collections

Question No.	Answer	Explanation
1	a and c	Associative arrays can have negative integer subscripts, positive integer subscripts and string subscripts. As associative arrays are treated as local arrays, initialization is not required for them.
2	c	Nested tables are an unbounded collection which can grow dynamically.
3	a	Varrays are always dense collections. Sparse varray doesn't exist.
4	c and d	BOOLEAN and NUMBER are not suitable index types for an associative array.
5	a and c	Yes, varray limit can be increased during runtime using the ALTER TABLE statement. If all the cells of a varray are populated with elements, LAST is equal to COUNT. This also holds true when the varray is empty.

Question No.	Answer	Explanation
6	b	The first DBMS_OUTPUT prints the first element from the default constructor. Once it is reassigned with a value in the executable section, the default values are overwritten.
7	a and c	Varrays are bounded collections which can accommodate data maximum up to the specified limit.
8	a, b, and c	EXISTS doesn't raises any exception. DELETE cannot be used with varrays.

Chapter 4, Using Advanced Interface Methods

Question No.	Answer	Explanation
1	a and c	The extproc process is a session specific process started by the Oracle listener and loads the shared library.
2	b	External procedure support was introduced in Oracle 8.
3	a	The PL/SQL wrapper containing the call specification is dependent on the database library object.
4	b	The TNS service ORACL_ CONNECTION_ DATA connects to the listener with the SID_ NAME in the CONNECT_ DATA parameter.
5	b	The loadjava utility loads the Java class to a specified user in a database.
6	a	Java programs are directly supported by Oracle and do not use the extproc process.
7	c	The external function name is case sensitive. The PL/SQL wrapper must use it exactly in same case as specified in the external program.

Chapter 5, Implementing VPD with Fine Grained Access Control

Question No.	Answer	Explanation
1	c	The policy function returns the predicate as <Column>=<Value>.
2	b	The security policy can be associated to one and only one schema object.
3	b and c	The DBMS_RLS package is a SYS owned built in package whose public synonym is shared amongst the users. It is a useful package to work with policies and policy groups.
4	b	There is no such privilege as CREATE_CONTEXT. It should be the CREATE ANY_CONTEXT privilege. All context metadata is in either ALL_CONTEXTS or DBA_CONTEXTS.
5	a	Context creation followed by the creation of its trusted package. A policy function is created for the predicate and the security is attached for the protection.
6	b	Policy groups are created by the collection of policies under them.
7	b and d	Default policy type is Dynamic. Shared static policy is the one where a static policy can be shared by multiple database objects. In such cases, the column appearing in the predicate must exist in all the tables.
8	c and d	Either a DBA or a user with the DBA role can create and drop an application context. A DBA can modify certain USERENV attributes, but not all.
9	a	The predicate information returned by the policy function is retained in SGA until the query is reparsed. During the parse stage, the current context information is matched with the latest. If the value has been changed, Oracle synchs the context values; otherwise the old value is retained.
10	a	The applicable policy tries to access the F_JOB_POLICY policy function which doesn't exist in the ORADEV schema.

Chapter 6, Working with Large Objects

Question No.	Answer	Explanation
1	a	LOBs can appear as a database column or a user defined object type attribute.
2	b	LOB type parameters can exist.
3	a	LOB data greater than 4 K is stored out of line with the current row. Mandatorily, it's a different LOB segment which may or may not be in the same tablespace.
4	b	The BLOB column must be initialized with <code>EMPTY_BLOB()</code>
5	c	The constructor methods <code>EMPTY_CLOB()</code> and <code>EMPTY_BLOB()</code> are used to initialize <code>NULL</code> and <code>NOT NULL</code> LOB types.
6	b and c	<code>FILEOPEN</code> works only with BFILEs.
7	b	Temporary LOBs are session specific.
8	c and d	BFILE is a read-only type. The files accessed through the BFILE locator open in read-only mode. They cannot be manipulated in any way during the BFILE access.
9	b and d	Temporary LOB is always an internal LOB which is used for manipulative actions in the LOB columns within a block.
10	c	The user must have read/write privilege on the directory to access the files contained in it.
11	b	A <code>LONG</code> column can be migrated to a LOB column using the following syntax: <pre>ALTER TABLE [<schema>.]<table_name> MODIFY (<long_column_name> { CLOB BLOB NCLOB } [DEFAULT <default_value>]) [LOB_storage_clause]</pre> Note that a <code>LONG</code> column can be migrated to <code>CLOB</code> or <code>NCLOB</code> while a <code>LONG RAW</code> column can be modified to <code>BLOB</code> only.

Question No.	Answer	Explanation
12	b	The BFILENAME function is used to return the LOB locator of a file which is located externally. It can be used for internal LOBs as well as external LOBs.
13	b and d	SecureFile is a new feature in Oracle 11g to store large objects with enhanced security, storage, and performance. Older LOBs may still exist as BasicFiles and can be migrated to SecureFiles.
14	a and c	The CREATE TABLE script executes successfully. The table and LOB are created in the default user tablespace. Oracle implicitly generates the LOB segment and LOB index. However, the segments are not created until the data has been inserted in the table.

Chapter 7, Using SecureFile LOBs

Question No.	Answer	Explanation
1	a, c, and d	SecureFiles can reside only on ASSM tablespaces. They are free from LOB index contention and high water mark contention. Up to 4 MB, SecureFile LOB data can be cached under the WGC component of the buffer cache.
2	a and d	Being part of the advanced compression, compression in SecureFile doesn't affect performance. It is intelligently handled by the LOB manager to perceive the impact of compression on LOB data.
3	b	Compression, deduplication, and encryption are mutually exclusive features of a SecureFile.
4	b	Table compression has nothing to do with SecureFile compression.

Question No.	Answer	Explanation
5	a and b	KEEP_DUPLICATES is the default option. The feature doesn't affect performance as the secure hash matching is a transparent process in the server.
6	c and d	The encryption keys are stored in the wallet directory. An encrypted SecureFile column cannot be modified for the encryption algorithm.
7	b	Online redefinition works with materialized views to get the latest snapshot of the source table, so that the ongoing data changes are not lost during redefinition processes.

Chapter 8, Compiling and Tuning to Improve Performance

Question No.	Answer	Explanation
1	a and b	Interpreted compilation mode is preferred when a program unit is in development stage and involves SQL statement processing.
2	c and d	The Real Native compilation method removes the dependency on C compiler to generate DLL for the program unit. Instead, the native DLLs are stored in the database dictionary itself. As the DLLs are stored in the dictionary, they can be a part of the normal backup and recovery.
3	b	The PLSQL_OPTIMIZE_LEVEL value set to 3 strictly inlines all the subprograms at high priority.

Question No.	Answer	Explanation
4	b and c	The <code>PLSQL_CODE_TYPE</code> value specified during recompilation of a program overrides the current system or session settings. Its default value is <code>INTERPRETED</code> and it must be updated in the <code>spfile</code> instance after the database upgrade process. In real native mode, the libraries are stored in the <code>SYSTEM</code> tablespace.
5	a and c	Usage of <code>BULK COLLECT</code> can pull multi-row data from the database in a single attempt, thus reducing context switches. <code>PLS_INTEGER</code> is a preferred data type in arithmetic calculations.
6	d	Inlining of subprograms is only supported at optimization level greater than one.
7	b and c	The <code>F_ADD</code> local function would be called inline because its call has been marked inline using <code>PRAGMA INLINE</code> . It might also be considered for inlining as the <code>PLSQL_OPTIMIZE_LEVEL</code> value is 2.
8	b	The DLLs generated from the Real Native compilation are stored in the <code>SYSTEM</code> tablespace.
9	a and d	The <code>NOT NULL</code> data types add overhead to check every assignment for nullity. <code>L_SUM</code> must be declared as <code>L_SUM NATURAL;</code>
10	a, c, and d	Usage of appropriate data types to avoid implicit typecasting improves performance. <code>L_ID</code> must be declared as <code>NUMBER</code> . <code>PRAGMA INLINE</code> works for <code>PLSQL_OPTIMIZE_LEVEL</code> values 2 and 3. At level 1, the Oracle optimizer doesn't consider any subprogram for inlining. At level 3, all subprograms are strictly called for inline. However, inlining of a subprogram can be set off by specifying <code>PRAGMA INLINE (<Function name>, 'NO')</code> .

Chapter 9, Caching to Improve Performance

Question No.	Answer	Explanation
1	a	The database server would cache the query results only when the user explicitly allocates the cache memory at the server and the caching feature is enabled. In the given scenario, caching is disabled as the value of the <code>RESULT_CACHE_MAX_SIZE</code> parameter is 0.
2	b and d	In automatic result caching, the <code>RESULT_CACHE</code> hint is ineffective as the server implicitly caches results of all SQL queries.
3	a	When the dependent table data gets updated, all the cached results get invalidated.
4	b	The cached results are stored at the server and are sharable across the sessions of the user.
5	b	PL/SQL result cache feature is operative only upon the functions which are declared as standalone or local to a stored subprogram or within a package.
6	a	The <code>RELIES_ON</code> clause has been deprecated in Oracle 11g R2.
7	b and c	The server doesn't cache the results of the queries which use sequence or any pseudo column (here <code>SYSDATE</code>).
8	b	PL/SQL function result cache works on server-side memory infrastructure which is the same for both SQL and PL/SQL. Only the results of functions can be cached at the server. The function must not be a pipelined one or the one declared with invoker's rights. It should accept parameters in the pass by reference mode of primitive data types only.
9	a and b	The valid values are <code>PUBLISHED</code> , <code>NEW</code> , <code>INVALID</code> , <code>BYPASS</code> , and <code>EXPIRED</code> .
10	a, b, c, and d	The <code>V\$RESULT_CACHE_STATISTICS</code> dynamic performance view stores the latest cache memory statistics.

Chapter 10, Analyzing PL/SQL Code

Question No.	Answer	Explanation
1	b	The ALL_ARGUMENTS dictionary view captures the information about the subprogram arguments.
2	d	Server places the identifier information in the SYSAUX tablespace.
3	a	The parameter values used in the subprograms of the DBMS_METADATA package are case sensitive.
4	b	The FORMAT_CALL_STACK forms the stack of all program units traversed by the server.
5	b and c	The DBMS_METADATA package can generate reports for table grants, object dependencies and DDL of given objects in a schema.
6	b	The PL/Scope tool can store the identifier data in the SYSAUX tablespace only.
7	a and b	For tables and views, the DDL script can be extracted without a storage clause by setting the STORAGE parameter to FALSE. Similarly, views scripts can be made force free by setting the FORCE parameter to FALSE.

Chapter 11, Profiling and Tracing PL/SQL Code

Question No.	Answer	Explanation
1	b	The Analyzer component interprets the raw profiler data and loads into the database tables.
2	b	The plshprof is a command-line utility to generate HTML reports from raw profiler output.
3	d	The PLSQL_DEBUG parameter has been deprecated starting from Oracle 11g.

Question No.	Answer	Explanation
4	c	The \$ORACLE_HOME\rdbms\admin\tracetab.sql script creates the trace log tables – PLSQL_TRACE_RUNS and PLSQL_TRACE_EVENTS.
5	c and d	The trace control levels cannot be used in combination with the other trace levels.
6	a and b	The plshprof utility is a command-line tool to convert raw profiler data into HTML reports.
7	c	The Analyzer component can trace multiple subprograms profiled into one trace file.
8	a	The max_depth parameter can be specified to limit the recursive levels in START_PROFILING.

Chapter 12, Safeguarding PL/SQL Code against SQL Injection Attacks

Question No.	Answer	Explanation
1	a, b, and c	Dynamic SQL is more prone to injective attacks. Static SQL must be preferred in major cases. In other cases, dynamic SQL must use bind variables.
2	a	If the SQL query identifiers are fixed for all the executions of a subprogram, static SQL can be used in the program.
3	a and d	SQL injection can lead to the leakage of confidential information and perform unauthorized activities.
4	a	The inputs from the application layer must be verified for purity before using in the application.
5	b	Statistical code analysis is used only for logical flow of the code but doesn't provide confirmation on the code vulnerability.

Question No.	Answer	Explanation
6	a	Fuzzing is a rough testing method to measure the resistivity and scalability of the program, which can discover the vulnerable areas of the code.
7	c and d	The <code>DBMS_ASSERT.SQL_OBJECT_NAME</code> subprogram validates the object contained in the current schema. The <code>SIMPLE_SQL_NAME</code> and <code>QUALIFIED_SQL_NAME</code> functions are used to verify the sanity of the SQL names.
8	b	The quoted identifier is used in queries enclosed within double quotes. Its meaning in the context is entirely different from the unquoted identifier.
9	b	<code>ENQUOTE_LITERAL</code> encloses a given string with single quotes.
10	a, c, and d	The Oracle keywords which implement dynamic SQL in the code are the most vulnerable areas in a PL/SQL code.
11	a and c	<code>AUTHID CURRENT_USER</code> eliminates the chances of SQL injection by executing a PL/SQL program with the rights of its invokers and not of the creator.

Index

Symbols

`%BULK_EXCEPTIONS` 59
`%BULK_EXCEPTIONS` attribute 252
`%BULK_ROWCOUNT` 59
`%FOUND` 36, 60
`%ISOPEN` 36, 60
`%NOTFOUND` 36, 60
`%ROWCOUNT` 36, 60
`[DBA | ALL | USER]_ARGUMENTS` 301, 302, 303
`[DBA | ALL | USER]_DEPENDENCIES` 308, 309
`[DBA | ALL | USER]_OBJECTS` 304, 305
`[DBA | ALL | USER]_PROCEDURES` 307
`[DBA | ALL | USER]_SOURCE` 306

A

ACID (Atomicity, Consistency, Isolation, and Durability) 171
ADD_TRANSFORM function 329
advanced features, SecureFiles
 compression 215, 216
 deduplication 214, 215
 enabling 214-216
 encryption 216-219
ALTER SESSION command 178
ALTER SYSTEM command 218
ALTER [SYSTEM | SESSION]
 command 274
AMM 271

ANALYZE function 352
AND logical operator
 using, for conditional control statements
 rephrasing 254
APPEND procedure 181
application life cycle development
 code writing 299
 tuning 299
architectural enhancements, SecureFiles
 CHUNK size 208
 high water mark contention 209
 inode and space management 208
 prefetching 209
 transformation management 208
 WGC 208
architecture, external routines
 components 122
 extproc process 122
 Shared library of external routine 123
ASSM 208
associative arrays
 about 83-87
 data type 85
 Inferred data 85
 PL/SQL scalar data type 85
 structure 85
 User-defined type 85
AUTHID CURRENT_USER option 374
Automatic Memory Management. *See*
 AMM
automatic result cache 274, 279, 280
Automatic Segment Space Management. *See*
 ASSM

B

BasicFiles

migrating, to SecureFiles 220

BasicFiles to SecureFiles migration

Online Redefinition method 220-224

Partition method 220

BFILE 173

BFILE, DBMS_LOB data types 180

BFILENAME function 179

Binary Large Object. *See* BLOB

BLOB 172

BLOB, DBMS_LOB data types 180

bulk binding

implementing 248, 249

SAVE_EXCEPTIONS, using 252, 253

using 249-251

BULK COLLECT

about 249

facts 250

C

cache 197, 270

cache grid 273

cache group 273

callback 123

callout 123

call specification 134

Character Large Object. *See* CLOB

CLOB 172

CLOSE procedure 181

code testing, for SQL injection flaws

code, reviewing 386

Fuzz testing 387

static code analysis 387

test case, generating 387

test strategy 386

coding information

[DBA | ALL | USER]_ARGUMENTS 301

[DBA | ALL | USER]_DEPENDENCIES

308, 309

[DBA | ALL | USER]_OBJECTS 304, 305

[DBA | ALL | USER]_PROCEDURES 307

[DBA | ALL | USER]_SOURCE 306

dictionary views 300

finding, SQL developer used 310-319

tracking 299-308

collection

about 81, 103

associative arrays, using 84

categorizing 83

characteristics 103, 104

type structure 82

initializing 115, 116

nested tables, using 84

non-persistent category 83

overview 82

persistent category 83

type, selecting 84

varrays, using 84

collection elements

manipulating 113, 115

collection methods, PL/SQL

about 105

COUNT function 106

DELETE function 112, 113

EXISTS function 105, 106

EXTEND function 109, 110

FIRST function 108

Last function 108

LIMIT function 107

NEXT function 109

PRIOR function 109

TRIM function 111

COLUMN_VALUE attribute 96

COMPARE function 181

compilation mode

choosing 230

interpreted compilation mode, choosing 230

native compilation mode, choosing 231

setting 231

setting, at database level 231

setting, at session level 231

settings, querying 232

compression feature 216

COMPRESS keyword 216

conditional control statements

rephrasing 254

rephrasing, AND logical operator used 254

rephrasing, OR logical operator used 254

CONNECT_DATA parameter 127, 133

CONSTRAINTS_AS_ALTER
 parameter 331
CONSTRAINTS parameter 331
context area 35, 55
CONVERTTOBLOB procedure 181
CONVERTTOCLOB procedure 181
COPY procedure 181
COUNT function 106
CREATETEMPORARY procedure 181
cross method
 package for Java class method, creating 140
Ctrl + Enter (F9) 19
cursor attributes
 about 59
 %FOUND 36, 60
 %ISOPEN 36, 60
 %NOTFOUND 36, 60
 %ROWCOUNT 36, 60
cursor design
 considerations 57, 58
 cursor variables, using 58
cursor execution cycle 56
cursor execution cycle, stages
 BIND 57
 CLOSE 57
 EXECUTE 57
 FETCH 57
 OPEN 56
 PARSE 56
cursor expressions 56
cursors
 cursor attributes 36, 37
 execution cycle 35
 FOR loop 38
 overview 35
cursor structures
 about 55
 cursor design considerations 57, 58
 cursor design, guidelines 58, 59
 cursor execution cycle 56
 cursor expressions 56
 cursor variables 56
 Dynamic SQL 56
 explicit cursors 56
 implicit cursors 56

cursor variables
 about 56, 66-68
 as arguments 71, 72
 processing 70, 71
 ref cursor, types 69
 restrictions 73

D

database configuration, for server-side
 result cache
 RESULT_CACHE_MAX_SIZE parameter 274
 RESULT_CACHE_MODE parameter 273
 RESULT_CACHE_REMOTE_EXPIRATION parameter 274
database dependency
 about 48
 direct 49
 direct dependency, displaying 49
 enhancement 50
 indirect 49
 indirect dependency, displaying 49
 issues 50
 metadata 50
data caching 269
data definitions (DDL) 149
data type 74
DBMS_ASSERT package
 about 380, 381
 ENQUOTE_LITERAL function 380
 ENQUOTE_NAME function 380
 identifier contexts 382
 identifier, formatting 381
 limitations 385
 NOOP functions 380
 QUALIFIED_SQL_NAME function 380
 SCHEMA_NAME function 380
 SIMPLE_SQL_NAME function 380
 SQL_OBJECT_NAME function 380
 usage guidelines 384, 385
DBMS_ASSERT validation
 best practices 384
DBMS_DESCRIBE package 313
DBMS_HPROF package
 subprograms 352
 using 351

DBMS_LOB constants

CALL 180
DBMS_LOB constants 180
DBMS_LOB option types 180
DBMS_LOB option values 180
FILE_READONLY 180
LOBMAXSIZE 180
LOB_READONLY 180
LOB_READWRITE 180
SESSION 180

DBMS_LOB data types

BFILE 180
BLOB 180
CLOB 180
INTEGER 180
RAW 180
VARCHAR2 180

DBMS_LOB option types

OPT_COMPRESS 180
OPT_DEDUPLICATE 180
OPT_ENCRYPT 180

DBMS_LOB option values

COMPRESS_OFF 180
COMPRESS_ON 180
DEDUPLICATE_OFF 180
DEDUPLICATE_ON 180
ENCRYPT_OFF 180
ENCRYPT_ON 180

DBMS_LOB package

DBMS_LOB constants 180
DBMS_LOB data types 180
DBMS_LOB subprograms 181, 182
overview 179
rules, for BFILEs 183
rules, for internal LOBs 183
security model 179

DBMS_LOB subprograms

APPEND procedure 181
CLOSE procedure 181
COMPARE function 181
CONVERTTOBLOB procedure 181
CONVERTTOCLOB procedure 181
COPY procedure 181
CREATETEMPORARY procedure 181
ERASE procedure 181
FILECLOSEALL procedure 181
FILECLOSE procedure 181

FILEEXISTS function 181
FILEGETNAME procedure 181
FILEISOPEN function 181
FILEOPEN procedure 181
FRAGMENT_DELETE procedure 181
FRAGMENT_INSERT procedure 181
FRAGMENT_MOVE procedure 181
FRAGMENT_REPLACE procedure 181
FREETEMPORARY procedure 181
GETCHUNKSIZE function 182
GETLENGTH function 182
GETOPTIONS function 182
GET_STORAGE_LIMIT function 182
INSTR function 182
ISOPEN functions 182
ISTEMPORARY functions 182
LOADBLOBFROMFILE procedure 182
LOADCLOBFROMFILE procedure 182
LOADFROMFILE procedure 182
OPEN procedures 182
READ procedures 182
SETOPTIONS procedures 182
SUBSTR functions 182
TRIM procedures 182
WRITEAPPEND procedures 182
WRITE procedures 182

DBMS_METADATA

function type objects, retrieving in
ORADEV schema 336, 337
object dependencies on F_GET_LOC
function, retrieving 335
single object metadata, retrieving 332-334
system grants on ORADEV schema
retrieving 335

DBMS_METADATA package

about 326, 327
data types 327
parameter requirements 330
subprograms 328-330
SYS-owned object types 327, 328
transformation parameters 330, 331
transform handlers 331

DBMS_RESULT_CACHE package

about 273, 275
public constants 276
subprograms 276

DBMS_TRACE constant
 NO_TRACE_ADMINISTRATIVE 342
 NO_TRACE_HANDLED_EXCEPTIONS 342
 TRACE_ALL_CALLS 342
 TRACE_ALL_EXCEPTIONS 342
 TRACE_ALL_LINES 342
 TRACE_ALL_SQL 342
 TRACE_ENABLED_CALLS 342
 TRACE_ENABLED_EXCEPTIONS 342
 TRACE_ENABLED_LINES 342
 TRACE_ENABLED_SQL 342
 TRACE_LIMIT 342
 TRACE_MAJOR_VERSION 342
 TRACE_MINOR_VERSION 342
 TRACE_PAUSE 342
 TRACE_RESUME 342
 TRACE_STOP 342

DBMS_TRACE package
 about 341
 installing 341
 subprograms 341, 363

DBMS_TRACE subprogram
 CLEAR_PLSQL_TRACE procedure 342
 COMMENT_PLSQL_TRACE procedure 343
 GET_PLSQL_TRACE_LEVEL function 342
 GET_PLSQL_TRACE_RUNNUMBER function 342
 INTERNAL_VERSION_CHECK function 343
 LIMIT_PLSQL_TRACE procedure 343
 PAUSE_PLSQL_TRACE procedure 343
 PLSQL_TRACE_VERSION procedure 343
 RESUME_PLSQL_TRACE procedure 343
 SET_PLSQL_TRACE procedure 343

db_securefile parameter 212

Debug option 22

DEDUPLICATE keyword 215

deduplication feature 215

DEFAULT parameter 331

DELETE function 112, 113

dense collection 88

development steps, VPD implementation
 application context, creating 149
 context key, creating 149
 context key, setting 150
 Policy function, creating 150

directories, LOB data types 174

DIRECTORY parameter 218

DML operations, performing on nested table
 column, selecting 92
 data, inserting 91
 data, updating 92, 93

DML operations, performing on varray
 column, selecting 101, 102
 data, inserting 101
 varray instance, updating 102

DROP command 99

duration 197

Dynamic SQL 56

E

EMPTY_BLOB() function 178, 185

EMPTY_CLOB() function 178

encryption algorithms, SecureFiles
 3DES168 216
 AES128 216
 AES192 216
 AES256 216

ENCRYPT keyword 217

ENQUOTE_LITERAL function 380

ENQUOTE_NAME function 380

ERASE procedure 181

error_number parameter 43

ETL 351

exception
 about 39
 system-defined exception 39
 user-defined exceptions 41

exception handling, in PL/SQL
 about 39
 exception propagation 46-48
 system-defined exceptions 39
 user-defined exceptions 41

EXCEPTION variable 45

EXISTS function 105, 106

explicit cursors 35, 56, 62

EXTEND function 109, 110

EXTEND method 110

external C programs
 executing, from PL/SQL 131

- executing, through C program 131, 132
- external LOB** 171
- external procedures, benefits** 130
 - about 130
 - enhanced performance 130
 - Integration of strengths 130
 - logical extensibility 130
 - Reusability of client logic 130
- external program, publishing** 134
- external routines**
 - about 121, 122
 - architecture 122
 - diagrammatic representation 124
 - external procedures, benefits 130
 - extproc process 124
 - Oracle Net Configuration 125
- Extraction, Transformation, and Loading.** *See* ETL

F

- F_COMP_INT** function 234
- F_COMPUTE** function 139
- F_COMPUTE_SUM** function 139
- FGAC**
 - about 146
 - overview 146
 - working 147, 148
- FGD** 50
- F_GET_DOUBLE** function 32
- F_GET_FUN_DDL** function 337
- F_GET_LOC** function 354
- F_GET_NAME** function 301
- F_GET_SAL** function 291
- FILECLOSEALL** procedure 181
- FILECLOSE** procedure 181
- FILEEXISTS** function 181
- FILEGETNAME** procedure 181
- FILEISOPEN** function 181
- FILEOPEN** procedure 181
- Fine Grained Access Control** 145.
 - See* FGAC
- Fine Grained Dependency.** *See* FGD
- FIRST** function 108
- FORALL** function 238
- FORALL** loop 249
- FORCE** parameter 331

- FORMAT_CALL_STACK** function 316
- forward declaration** 248
- F_PRINT_NAME** function 303
- FRAGMENT_DELETE** procedure 181
- FRAGMENT_INSERT** procedure 181
- FRAGMENT_MOVE** procedure 181
- FRAGMENT_REPLACE** procedure 181
- FREETEMPORARY** procedure 181
- frst-order attack** 369
- functions**
 - about 29
 - characteristics 29
 - execution methods 31
 - SQL expressions calling functions, rules 32
 - standalone user-defined function 32
 - syntax 30
- fuzz testing** 387

G

- GETCHUNKSIZE** function 182
- GETLENGTH** function 182
- GETOPTIONS** function 182
- GET_QUERY** function 329
- GET_STORAGE_LIMIT** function 182
- GET_[XML | DDL | CLOB]** functions 329
- GRANT** command 174
- Graphical User Interface.** *See* GUI
- GUI** 13

H

- hierarchical profiler**
 - Analyzer 351
 - Data collector 351
 - DBMS_HPROF package, using 351
 - efficiencies 351
 - PL/SQL program profiling
 - demonstration 352-357
 - profiler information, viewing 352
- hierarchical profiling** 351

I

- identifier contexts, DBMS_ASSERT** package
 - basic 382
 - literal 382
 - qualified 383, 384

- quoted identifier 382
- simple 382
- unquoted identifier 382
- identifiers**
 - about 319
 - activities identification, PL/Scope tool used 319
- IF THEN ELSE expression** 254
- IMBD** 273
- implicit cursors** 35, 56, 60, 62
- INHERIT parameter** 331
- In Memory Database.** *See* IMBD
- INSTR function** 182
- INTEGER, DBMS_LOB data types** 180
- internal LOB** 171
 - BLOB type 172
 - CLOB type 172
 - NCLOB type 172
- interpreted compilation**
 - about 228
 - program unit, comparing 233-235
- intra unit inlining**
 - enabling 255
- ISOPEN functions** 182
- ISTEMPORARY functions** 182

J

- Java pool** 136
- Java programs**
 - executing, from PL/SQL 136
 - Java class method, calling 137
 - Java class, uploading into database 137
 - package for Java class method, creating 140
- Java programs, executing from PL/SQL**
 - about 136
 - Java class method, calling 137
 - Java class, uploading into database 137
 - loadjava utility 137-139
 - packages, creating 140
- Java Virtual Machine.** *See* JVM
- JVM** 136

L

- LAST function** 108
- LENGTH function** 194
- LIMIT function** 107
- LIMIT method** 107
- LISTENER.ora** 126
- Literal** 382
- LOADBLOBFROMFILE procedure** 182
- LOADCLOBFROMFILE procedure** 182
- LOADFROMFILE procedure** 182
- loadjava utility** 137, 139
- LOB**
 - closing 193
 - column states 193
 - migrating, from LONG data types 194-196
 - opening 193
 - restrictions 194
 - row, locking 193
- LOB column**
 - accessing 193, 194
 - data, inserting into 185
- LOB column states**
 - Empty 193
 - NULL 193
 - Populated 193
- LOB data**
 - deleting 192
 - modifying 190-192
 - selecting 189, 190
- LOB data type columns**
 - initializing 184
- LOB data types**
 - about 170, 172
 - BFILE 173
 - BFILEs, managing 178, 179
 - BFILEs, securing 178, 179
 - classification chart 171
 - columns, creating in table 175-177
 - creating 173
 - creating, syntax 175, 176
 - DBMS_LOB package 179
 - directories 173
 - external LOB 171
 - internal LOB 171
 - internal LOBs, managing 178
 - LOB locators 172
 - LOB value 172
 - managing 177-179
 - populating, external file used 185-189
 - temporary LOBs 173
- lob_loc** 197

LONG data types
limitations 170
migrating to LOB 194
LONG RAW data types
limitations 170

M

machine code (bytecode) 228
manual result cache 274, 277

N

native compilation. *See* NCOMP
NCOMP
about 228, 235
program unit, comparing 233-235
nested table
about 84-89
creating, as database object 90, 91
DML operations 91
features 94-97
in PL/SQL 93, 94
using 84
versus associative arrays 104
versus varray 105
NOCACHE mode 210
non-persistent collection
associative array 83
NOOP functions 380
NOT NULL constraint 241-243

O

OCI client results cache 273
OID parameter 331
OLTP 209
Online Redefinition method
about 221
LOB columns, migrating 222, 223
pre-requisites 221
Online Transaction Processing. *See* OLTP
OPEN function 329
OPEN procedures 182
OPENW function 329
Oracle 11g
about 170
memory infrastructure diagram 271

Oracle Common Language Runtime.
See ORACLRL

Oracle initialization parameter
enabling 256
PLSQL_OPTIMIZE_LEVELs 256-260

Oracle Net Configuration
about 125
LISTENER.ora 126-129
TNSNAMES.ora 125, 126
verifying 129, 130

Oracle-supplied packages
DBMS_ALERT 51
DBMS_HTTP 51
DBMS_LOCK 51
DBMS_OUTPUT 51
DBMS_SCHEDULER 51
DBMS_SESSION 51
packages 51
packages, categorizing 51, 52
reviewing 51
UTL_FILE 51
UTL_MAIL 51

Oracle Technology Network. *See* OTN

ORACLRL 123

OR logical operator
using, for conditional control statements
rephrasing 254

OTN 15

P

packages
about 33
advantage 33
components, package body 34
components, package specification 33, 34
creating, syntax 34

parent table 89

partition method
about 220
new SecureFile partition 220
partitioned table, creating 220

performance optimization 269

persistent collection
nested table 84
Varray (variable-size array) 84

PGA 36

- PL/Scope tool**
 - about 320
 - identifier collection 320, 321
 - identifier information, capturing 322-325
 - key features 320
 - report 322
 - report, objectives 325, 326
- plshprof utility**
 - about 357, 359
 - sample reports 359, 360
- PLS_INTEGER data type**
 - using, for arithmetic operations 243, 244
- PL/SQL**
 - about 10
 - accomplishments 10
 - collection methods 105
 - comparing, with SQL 239
 - exception handling 39
 - external C programs, executing 131
 - external C programs, executing through
 - external procedure 131-135
 - Java class method, calling 137
 - Java class, uploading into database 137
 - Java programs, executing 136
 - nested table 93, 94
 - PLS_INTEGER data type, using 243, 244
 - program 11
 - varray 99, 100
- PL/SQL block**
 - declaration 11
 - execution 11
 - header 11
 - structure 11
 - structure diagram 12
- PL/SQL code**
 - bulk binding, using 248-251
 - implicit data type conversion,
 - avoiding 239, 241
 - modularizing 246-248
 - NOT NULL constraint 241-243
 - profiling 339
 - SIMPLE_INTEGER data type,
 - using 245, 246
 - SQL, comparing with PL/SQL 239
 - tracing 339
 - tuning, areas 238
- PL/SQL code, designing**
 - cursor structures 55
 - cursor variables 66
 - subtypes 74
- PL/SQL code security 365**
- PLSQL_CODE_TYPE parameter 231**
- PLSQL_DEBUG parameter 343, 344**
- PL/SQL development environments**
 - SQL Developer 13
 - SQL*Plus 24
- PL/SQL function result cache**
 - argument and return type limitations 294
 - function structural limitations 294
 - limitations 294
- PL/SQL native compilation**
 - database, compiling 235, 236, 237
- PLSQL_OPTIMIZE_LEVEL=0 256, 257**
- PLSQL_OPTIMIZE_LEVEL=1 258, 259**
- PLSQL_OPTIMIZE_LEVEL=2 259, 260**
- PLSQL_OPTIMIZE_LEVEL=3 260-262**
- PL/SQL packages. *See* packages**
- PL/SQL program**
 - anonymous 11
 - named 11
 - nested 11
 - plshprof utility 357
 - profiling 350
 - tracing 340
- PL/SQL program, profiling**
 - hierarchical profiling 351
- PL/SQL program, tracing**
 - DBMS_TRACE package 341
 - methods 340
 - PLSQL_DEBUG parameter 343
- PL/SQL trace information**
 - viewing 344-347
- PL/SQL tracing**
 - demonstrating 347-350
- PLSQL_WARNINGS parameter 255, 262**
- policy types**
 - CONTEXT_SENSITIVE 154
 - DYNAMIC (Default) 154
 - SHARED_CONTEXT_SENSITIVE 154
 - SHARED_STATIC 154
 - STATIC 154

policy utilities

- drop 164
- refresh 164

Pragma 41

PRAGMA EXCEPTION_INIT 41

PRAGMA INLINE 262-264

PRETTY parameter 331

PRIOR function 109

Procedural Language-Structured Query Language. *See* **PL/SQL**

procedure

- about 27
- characteristics 27
- executing 28
- IN OUT parameters 27
- IN parameters 27
- OUT parameters 27
- syntax 28

Process Global Area. *See* **PGA**

profiling 339

Program Unit Arguments option 310

Q

QUALIFIED_SQL_NAME function 380

Quoted identifier 382

R

RAISE_APPLICATION_ERROR

procedure 43-46

RAW, DBMS_LOB data types 180

READ procedures 182

real native compilation

- about 229
- accomplishments 229

REF_CONSTRAINTS parameter 331

ref cursor

- about 69
- strong type 69
- SYS_REFCURSOR 69
- weak type 69

Remote-Ops (RO) 126

result cache

- about 270, 271
- implementing, in PL/SQL 288
- implementing, in SQL 277
- limitation 287

 OCI client 273

 server-side result cache 271

result cache implementation, in PL/SQL

- about 288
- cross-session availability 292
- invalidating 292, 293
- limitations 294
- RESULT_CACHE clause 288-291, 292

result cache implementation, in SQL

- automatic result cache 279, 280
- manual result cache 277, 279
- memory report, displaying 286
- read consistency 287
- result cache metadata 281
- SQL result cache, invalidating 284, 285

RESULT_CACHE_MAX_SIZE parameter 274

result cache metadata

- about 281
- dependent objects 283
- memory statistics 283, 284

RESULT_CACHE_MODE

parameter 273, 278

RESULT_CACHE_REMOTE_EXPIRATION

parameter 274

RLS 147

Row Level Security. *See* **RLS**

S

SCHEMA_NAME function 380, 381

second-order attack 369

SecureFile performance graph 209

SECUREFILE property 223

SecureFiles

- about 206
- advanced features, enabling 214
- architectural enhancements 207
- encryption algorithms 216
- feature 207
- LOB features 210
- metadata 213
- migrating, from BasicFiles 220
- working with 211-214

SEGMENT_ATTRIBUTES parameter 331

SERVEROUTPUT parameter 12

server-side result cache
 about 271
 database, configuring 273-275
 DBMS_RESULT_CACHE package 276, 277
 PL/SQL function result cache 272
 SQL query result cache 272

session cursors 56

SETOPTIONS procedures 182

SGA 83, 151

SIMPLE_INTEGER data type
 using 245, 246

SIMPLE_SQL_NAME function 380

SIZE_BYTE_KEYWORD parameter 331

SQL
 about 9
 comparing, with PL/SQL 239

SQL Developer
 about 13, 310
 accomplishments 13
 anonymous PL/SQL block, creating 21
 anonymous PL/SQL block, executing 21
 connection, creating 15
 DBMS_DESCRIBE package 313
 features 14
 history 15
 Oracle SQL Developer, start page 14
 PL/SQL code, debugging 21, 22
 scripts, editing 23
 scripts, saving 23
 SQL script, calling from 19, 20
 SQL statement, executing 18, 19
 SQL Worksheet 16, 18
 using, for coding information
 search 310-319

SQLERRM function 252

SQL injection
 about 366
 attack immunization 370
 attacks, preventing 369, 370
 attack types 369
 attack types, first-order attack 369
 attack types, second-order attack 369
 code injection, example 366, 368
 flaws, testing 386
 impacts 369
 overview 366

SQL injection attack immunization
 about 370
 attack surface, reducing 370
 bind arguments 378, 379
 definer's right 371-375
 dynamic SQL, avoiding 375-377
 input sanitization, DBMS_ASSERT
 used 379
 owner's right 371-375
 user privileges, controlling 371

SQL injection flaws
 code, testing 386

SQL_OBJECT_NAME function 380

SQL*Plus
 about 24, 25
 anonymous PL/SQL block, executing 26
 evolution cycle 24
 SQL statement, executing 26

SQLTERMINATOR parameter 331

START_PROFILING procedure 352

static code analysis 387

STOP_PROFILING procedure 352

STORAGE parameter 331

Structured Query Language. *See* SQL

subprograms, DBMS_RLS package
 ADD_GROUPED_POLICY 155
 ADD_POLICY 155
 ADD_POLICY_CONTEXT 155
 CREATE_POLICY_GROUP 155
 DELETE_POLICY_GROUP 155
 DISABLE_GROUPED_POLICY 155
 DROP_GROUPED_POLICY 155
 DROP_POLICY 155
 DROP_POLICY_CONTEXT 155
 ENABLE_GROUPED_POLICY 155
 ENABLE_POLICY 155
 REFRESH_POLICY 155

SUBSTR functions 182

subtype
 about 75
 benefits 77
 classifying 75
 evolution 75
 Oracle predefined subtypes 75
 type compatibility 77
 user-defined subtypes 76

Sum method
 package for Java class method, creating 140

SYS_CONTEXT function 150, 162

SYS.KU\$_DDL 327

SYS.KU\$_DDL\$ 327

SYS.KU\$_ERRORLINE 328

SYS.KU\$_ERRORLINES 328

SYS.KU\$_MULTI_DDL 327

SYS.KU\$_MULTI_DDL\$ 327

SYS.KU\$_PARSED_ITEM 327

SYS.KU\$_PARSED_ITEMS 327

SYS.KU\$_SUBMITRESULT 328

SYS.KU\$_SUBMITRESULTS 328

SYS_REFCURSOR 69

system-defined exception
 about 39
 list 40

System Global Area. *See* SGA

T

TABLESPACE parameter 331

TDE 210, 216

temporary LOBs
 creating 198, 199
 managing 197
 operations 196
 releasing 198, 199
 using 196
 validating 198, 199

tkprof utility 340

TNS 125

TRACE parameter 356

tracing
 about 339
 DBMS_APPLICATION_INFO method 340
 DBMS_SESSION and DBMS_MONITOR
 method 340
 DBMS_TRACE method 340
 PL/SQL programs 340

Transparent Data Encryption. *See* TDE

Transparent Network Substrate. *See* TNS

trcscss utility 340

TRIM function 111

TRIM procedures 182

U

UGA 83, 150

Unquoted identifier 382

user-defined exceptions
 about 41
 declaring, in PL/SQL block 41, 42
 declaring, ways 41
 RAISE_APPLICATION_ERROR procedure
 43-46

User Global Area. *See* UGA

USING clause 378

V

VARCHAR2, DBMS_LOB data types 180

varray
 about 84, 98, 99
 as database collection type 100
 DML operations 100
 in PL/SQL 99, 100
 using 84
 versus nested table 105

VARRAY.DELETE method 113

Virtual Private Database. *See* VPD

VPD
 about 145, 146
 features 148, 149
 implementing, development steps 149
 policy metadata 163, 164
 RLS, working 147
 working 147, 148

VPD implementation
 application context 150
 application context, using 159-162
 demonstrating 156-162
 development, steps 149
 policy association, DBMS_RLS
 package used 155
 policy function 153, 154
 simple security policy, using 157

W

WGC 208

WRITEAPPEND procedures 182

WRITE procedures 182



Thank you for buying Oracle Advanced PL/SQL Developer Professional Guide

About Packt Publishing

Packt, pronounced 'packed', published its first book "Mastering phpMyAdmin for Effective MySQL Management" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: www.packtpub.com.

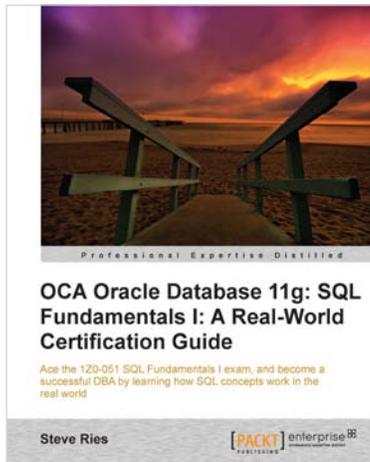
About Packt Enterprise

In 2010, Packt launched two new brands, Packt Enterprise and Packt Open Source, in order to continue its focus on specialization. This book is part of the Packt Enterprise brand, home to books published on enterprise software – software created by major vendors, including (but not limited to) IBM, Microsoft and Oracle, often for use in other corporations. Its titles will offer information relevant to a range of users of this software, including administrators, developers, architects, and end users.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

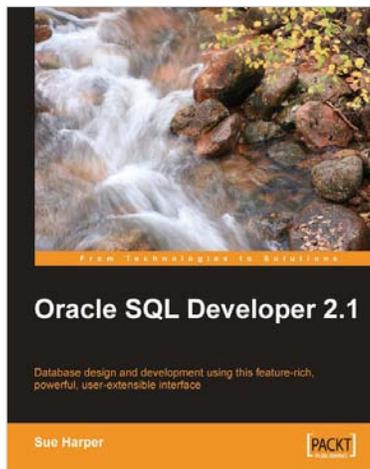


OCA Oracle Database 11g: SQL Fundamentals I: A Real World Certification Guide (1Z0-051)

ISBN: 978-1-84968-364-7 Paperback: 460 pages

Ace the 1Z0-051 SQL Fundamentals I exam, and become a successful DBA by learning how SQL concepts work in the real world

1. Successfully clear the first stepping stone towards attaining the Oracle Certified Associate Certification on Oracle Database 11g
2. This book uses a real world example-driven approach that is easy to understand and makes engaging
3. Complete coverage of the prescribed syllabus



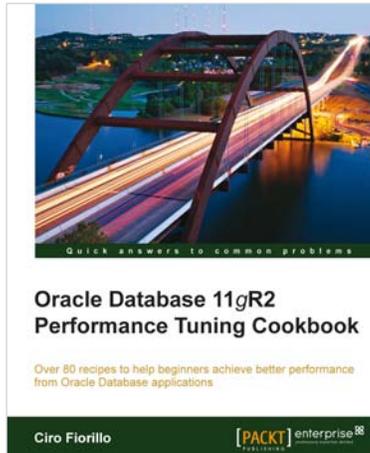
Oracle SQL Developer 2.1

ISBN: 978-1-847196-26-2 Paperback: 496 pages

Database design and development using this feature-rich, powerful, user-extensible interface

1. Install, configure, customize, and manage your SQL Developer environment
2. Includes the latest features to enhance productivity and simplify database development
3. Covers reporting, testing, and debugging concepts
4. Meet the new powerful Data Modeling tool - Oracle SQL Developer Data Modeler

Please check www.PacktPub.com for information on our titles

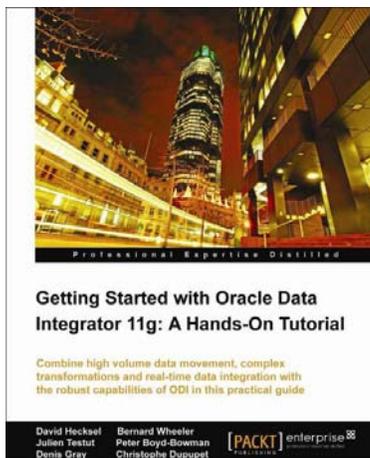


Oracle Database 11gR2 Performance Tuning Cookbook

ISBN: 978-1-84968-260-2 Paperback: 542 pages

Over 80 recipes to help beginners achieve better performance from Oracle Database applications

1. Learn the right techniques to achieve best performance from the Oracle Database
2. Avoid common myths and pitfalls that slow down the database
3. Diagnose problems when they arise and employ tricks to prevent them
4. Explore various aspects that affect performance, from application design to system tuning



Getting Started with Oracle Data Integrator 11g: A Hands-On Tutorial

ISBN: 978-1-84968-068-4 Paperback: 450 pages

Combine high volume data movement, complex transformations and real-time data integration with the robust capabilities of ODI in this practical guide

1. Discover the comprehensive and sophisticated orchestration of data integration tasks made possible with ODI, including monitoring and error-management
2. Get to grips with the product architecture and building data integration processes with technologies including Oracle, Microsoft SQL Server and XML files
3. A comprehensive tutorial packed with tips, images and best practices

Please check www.PacktPub.com for information on our titles