Database Programming with PL/SQL

15-1
Using PL/SQL Initialization Parameters
Objectives

This lesson covers the following objectives:

• Describe how `PLSQL_CODE_TYPE` can improve execution speed

• Describe how `PLSQL_OPTIMIZE_LEVEL` can improve execution speed

• Use `USER_PLSQL_OBJECT_SETTINGS` to see how a PL/SQL program was compiled
Purpose

• In many programming environments, fast program execution is imperative.

• In an earlier lesson, you learned how coding techniques such as the NOCOPY hint and Bulk Binding can improve the execution speed of PL/SQL programs.

• Setting PL/SQL initialization parameters can help to make your PL/SQL programs run even faster.
What are Initialization Parameters?

• *Initialization parameters* are used to change the way your database session interacts with the Oracle server.

• All initialization parameters have a name, a data type, and a default value.

• They can be used to adjust security, improve performance, and do many other things.

• Many of them have nothing to do with PL/SQL. In this lesson, you learn how to use two initialization parameters that change how your PL/SQL code is compiled.

• Do not confuse initialization parameters with the formal and actual parameters that we pass to subprograms.
Two PL/SQL Initialization Parameters

- The names of these initialization parameters are:
  - PLSQL_CODE_TYPE
  - PLSQL_OPTIMIZE_LEVEL

- PLSQL_CODE_TYPE is a VARCHAR2 with possible values INTERPRETED (the default value) and NATIVE.

- PLSQL_OPTIMIZE_LEVEL is a NUMBER with possible values 0, 1, 2 (the default), and 3.
Changing the Value of a Parameter

• You can change any initialization parameter’s value by executing an `ALTER SESSION` SQL statement:

```
ALTER SESSION SET PLSQL_CODE_TYPE = NATIVE;
CREATE OR REPLACE PROCEDURE run_faster_proc ...;
ALTER SESSION SET PLSQL_CODE_TYPE = INTERPRETED;
CREATE OR REPLACE PROCEDURE run_slower_proc ...;
```

• The new parameter value will be used until you log off, or until you change the value again.
Using PLSQL_CODE_TYPE

- If PLSQL_CODE_TYPE is set to INTERPRETED (the default), your source code is compiled to bytecode format.
- If the parameter value is changed to NATIVE, your source code will be compiled to native machine code format.
- You don’t need to know what these formats mean or how they work; the important thing is that native machine code PL/SQL executes faster than bytecode PL/SQL.
Using **PLSQL_CODE_TYPE**: Example

• To see the change in performance, we need some PL/SQL code that takes a long time to execute: 0.02 seconds doesn't seem much slower than 0.01!

• Let's compile a long-running procedure using **INTERPRETED** (notice how quickly it compiles):

```sql
CREATE OR REPLACE PROCEDURE longproc IS
  v_number PLS_INTEGER;
BEGIN
  FOR i IN 1..50000000 LOOP
    v_number := v_number * 2;
    v_number := v_number / 2;
  END LOOP;
END longproc;
```

**Procedure created.**

0.02 seconds
Using **PLSQL_CODE_TYPE**: Example

• The compile was quick, but see how long the procedure takes to run.

• Eleven seconds!

• That's much longer than most of the procedures and functions you have been writing.

```
BEGIN
  longproc;
END;
```

```
Statement processed.
11.34 seconds
```
Using **PLSQL_CODE_TYPE**: Example

• Let’s compile it again using **NATIVE**:

```sql
ALTER SESSION SET PLSQL_CODE_TYPE = NATIVE;

CREATE OR REPLACE PROCEDURE longproc IS
  v_number PLS_INTEGER;
BEGIN
  FOR i IN 1..50000000 LOOP
    v_number := v_number * 2;
    v_number := v_number / 2;
  END LOOP;
END longproc;
```

• Notice the procedure takes longer to compile than before (0.08 seconds compared to 0.02 seconds).
Using `PLSQL_CODE_TYPE`: Example

- But now let’s execute the `NATIVE` mode procedure:

```sql
BEGIN
  longproc;
END;
```

- The execution is about twice as fast in this case (5.7 seconds compared to 11.34 seconds).

- `NATIVE` mode will always execute faster than `INTERPRETED` mode, and depending on the source code, it may execute much faster.
Using **PLSQL_CODE_TYPE**: A Second Example

Let’s compile and execute an even longer procedure that includes a SQL statement:

```sql
ALTER SESSION SET PLSQL_CODE_TYPE = INTERPRETED;

CREATE OR REPLACE PROCEDURE sqlproc IS
    v_count PLS_INTEGER;
BEGIN
    FOR i IN 1..500000 LOOP
        SELECT COUNT(*) INTO v_count FROM countries;
    END LOOP;
END sqlproc;
BEGIN
    sqlproc;
END;
```

**Statement processed.**

39.07 seconds
Using **PLSQL_CODE_TYPE**: A Second Example

- Now compile and execute it using **NATIVE**:

```sql
ALTER SESSION SET PLSQL_CODE_TYPE = NATIVE;

CREATE OR REPLACE PROCEDURE sqlproc IS
  v_count PLS_INTEGER;
BEGIN
  FOR i IN 1..500000 LOOP
    SELECT COUNT(*) INTO v_count FROM countries;
  END LOOP;
END sqlproc;
BEGIN
  sqlproc;
END;
```

- Not much faster this time, is it? Why not?
NATIVE Compilation and SQL Statements

• Compiling a PL/SQL program with `PLSQL_CODE_TYPE = NATIVE` creates native PL/SQL code, but *not* native SQL code (there’s no such thing!).

• So the PL/SQL Engine executes faster, but SQL statements execute at the same speed as before.

• And SQL statements usually take far longer to execute than PL/SQL statements, especially when the tables contain thousands of rows.

• To speed up SQL statements, you use other techniques, such as Bulk Binding and choosing the correct indexes for your tables.
Does your PL/SQL Program Contain Useless Code?

• Examine this code:

```sql
CREATE OR REPLACE PROCEDURE obviouslybadproc IS
  v_number PLS_INTEGER := 1;
BEGIN
  IF v_number = 1 THEN
    DBMS_OUTPUT.PUT_LINE('This will always be displayed');
  ELSE
    DBMS_OUTPUT.PUT_LINE('This will never be displayed');
  END IF;
END obviouslybadproc;
```

• Silly, isn’t it?
• Of course, you would never write useless lines of code that can never be executed, would you?
• Look at the next example:
Does your PL/SQL Program Contain Useless Code?

• Not quite so obvious now, is it?
• In large, complex PL/SQL programs, it is all too easy to write code that can never be executed, or exceptions that can never be raised.

```sql
CREATE OR REPLACE PROCEDURE notsoobviousproc IS
  v_number    PLS_INTEGER;
BEGIN
  FOR i IN REVERSE 1..50 LOOP
    v_number := 50 - i;
    IF MOD(i,v_number) > 25 THEN
      DBMS_OUTPUT.PUT_LINE('Could this ever be displayed?');
    ELSE
      DBMS_OUTPUT.PUT_LINE('This will be displayed');
    END IF;
  END LOOP;
END notsoobviousproc;
```
Does your PL/SQL Program Contain Useless Code?

- Unnecessary code can slow down both creating and executing the program.

```sql
CREATE OR REPLACE PROCEDURE notsoobviousproc IS
  v_number  PLS_INTEGER;
BEGIN
  FOR i IN REVERSE 1..50 LOOP
    v_number := 50 - i;
    IF MOD(i,v_number) > 25 THEN
      DBMS_OUTPUT.PUT_LINE('Could this ever be displayed?');
    ELSE
      DBMS_OUTPUT.PUT_LINE('This will be displayed');
    END IF;
  END LOOP;
END notsoobviousproc;
```
The `PLSQL_OPTIMIZE_LEVEL` Initialization Parameter

- `PLSQL_OPTIMIZE_LEVEL` can be used to control what the PL/SQL Compiler does with useless code, as well as giving other performance benefits.
- Its value must be an integer between 0 and 3, inclusive.
- The higher the value, the more effort the compiler makes to optimize the code for execution.
- The optimizing compiler is enabled to level 2 by default.
The **PLSQL_OPTIMIZE_LEVEL** Initialization Parameter

The effects are:

- **With** `PLSQL_OPTIMIZE_LEVEL = 0`, the compiled code will run more slowly, but it will work with older versions of the Oracle software.

- This is similar to creating a document using Microsoft Word 2007, but saving it in Word 97-2003 format.

- **With** `PLSQL_OPTIMIZE_LEVEL = 1`, the compiler will remove unnecessary code and exceptions from the executable code, such as the useless code in the two examples on previous slides.
The `PLSQL_OPTIMIZE_LEVEL` Initialization Parameter

- The order of the source code is not typically changed.
- With `PLSQL_OPTIMIZE_LEVEL = 2`, (the default), the compiler will remove useless code as before, but will also sometimes move code to a different place if it will execute faster there.
- For example, if a frequently-called procedure in a large package is coded near the end of the package body, the compiler will move it nearer to the beginning.
- Your source code is never changed, only the compiled code.
The **PLSQL_OPTIMIZE_LEVEL** Initialization Parameter

- **PLSQL_OPTIMIZE_LEVEL = 3** gives all the benefits of values 1 and 2, plus subprogram inlining.
- This means that the compiled code of another called subprogram is copied into the calling subprogram, so that only one compiled unit of code is executed.
- The source code itself is not changed, it is only the executable code that is optimized.
PLSQL_OPTIMIZE_LEVEL: An Example

• The compiled code of \textsc{callingproc} now contains the code of both subprograms, as if it had been written as part of \textsc{callingproc} instead of as a separate subprogram.

• \textsc{calledproc} also still exists as a separate subprogram and can still be called from other places.

\begin{verbatim}
CREATE OR REPLACE PROCEDURE calledproc IS BEGIN...END calledproc;

ALTER SESSION SET PLSQL_OPTIMIZE_LEVEL = 3;

CREATE OR REPLACE PROCEDURE callingproc IS BEGIN
    ...
calledproc;
    ...
END;
\end{verbatim}
Using

**USER_PLSQL_OBJECT_SETTINGS**

You can see how your PL/SQL programs were compiled by querying the **USER_PLSQL_OBJECT_SETTINGS** Data Dictionary view:

```sql
SELECT name, type, plsql_code_type AS CODE_TYPE, plsql_optimize_level AS OPT_LVL
FROM USER_PLSQL_OBJECT_SETTINGS WHERE name = 'TESTPROC';
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>CODE_TYPE</th>
<th>OPT_LVL</th>
</tr>
</thead>
<tbody>
<tr>
<td>TESTPROC</td>
<td>PROCEDURE</td>
<td>INTERPRETED</td>
<td>2</td>
</tr>
</tbody>
</table>
Using
USER_PLSQL_OBJECT_SETTINGS

```
ALTER SESSION SET PLSQL_OPTIMIZE_LEVEL = 1;

CREATE OR REPLACE PROCEDURE testproc ...END testproc;

-- or ALTER PROCEDURE testproc COMPILE;

SELECT name, type, plsql_code_type AS CODE_TYPE,
     plsql_optimize_level AS OPT_LVL
FROM USER_PLSQL_OBJECT_SETTINGS WHERE name = 'TESTPROC';
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>CODE_TYPE</th>
<th>OPT_LVL</th>
</tr>
</thead>
<tbody>
<tr>
<td>TESTPROC</td>
<td>PROCEDURE</td>
<td>INTERPRETED</td>
<td>1</td>
</tr>
</tbody>
</table>
Terminology

Key terms used in this lesson included:

- `PLSQL_CODE_TYPE`
- `PLSQL_OPTIMIZE_LEVEL`
- PL/SQL Initialization Parameter
- `USER_PLSQL_OBJECT_SETTINGS`
Summary

In this lesson, you should have learned how to:

• Describe how `PLSQL_CODE_TYPE` can improve execution speed

• Describe how `PLSQL_OPTIMIZE_LEVEL` can improve execution speed

• Use `USER_PLSQL_OBJECT_SETTINGS` to see how a PL/SQL program was compiled