Advanced Oracle PL/SQL Developer's Guide
Second Edition

Oracle Database is one of the most reliable relational databases used by a variety of professionals around the globe. Oracle Database developers, administrators, performance specialists, and architects extensively use the database features to design database backend. Oracle PL/SQL provides a rich platform for database developers to build scalable database applications.

Advanced Oracle PL/SQL Developer's Guide, Second Edition, is a handy technical reference that starts with a refresher on the PL/SQL basic concepts and an introduction to the underlying features of Oracle 12c. Each chapter, then, introduces advanced PL/SQL feature blended with the latest update from Oracle Database 12c.

By the end of this book, you will master the advanced features of PL/SQL programming and will be comfortable in applying many of those in database development.

Who this book is written for
This book is for Oracle professionals who handle database development and manage PL/SQL code. Readers are expected to have a basic knowledge of Oracle Database architecture and the PL/SQL programming fundamentals. Certification aspirants can use this book to prepare for 1Z0-146 examination to be an Oracle Certified Professional in Advanced PL/SQL.

What you will learn from this book
- Learn and understand the key SQL and PL/SQL features of Oracle Database 12c
- Understand the new Multitenant architecture and Database In-Memory option of Oracle Database 12c
- Find out more about the advanced concepts of the Oracle PL/SQL language such as external procedures, BULK COLLECT and FORALL, securing data using Virtual Private Databases (VPD), SecureFiles, and Result cache
- Check out Oracle Advanced Security "Defense-in-depth" offering from Oracle and learn how to mask your confidential data using Data Redaction
- Trace, analyze, profile, and debug PL/SQL code while developing database applications
- Discover the capabilities of Oracle SQL Developer 4.0, the primary IDE for not just the database developers, but also for administrators and architects.
- Get acquainted with the tuning practices to write optimal PL/SQL code and develop robust applications.

Saurabh K. Gupta

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 8 'Tuning the PL/SQL Code'
- A synopsis of the book’s content
About the Author

Saurabh K. Gupta is a seasoned database technologist with extensive experience in designing high performance and highly available database applications. His technology focus has been centered around Oracle Database architecture, Oracle Cloud platform, Database In-Memory, Database Consolidation, Multitenant, Exadata, Big Data, and Hadoop. He has authored the first edition of this book. He is an active speaker at technical conferences from Oracle Technology Network, IOUG Collaborate’15, AIOUG Sangam, and Tech Days. Connect with him on his twitter handle (or SAURABHKG) or through his technical blog www.sbhoracle.wordpress.com, with comments, suggestions, and feedback regarding this book.
How many of us would believe that PL/SQL was introduced as a scripting language for executing a bunch of SQL scripts? Well, that's true. With the growing need to build computational logic and condition-based constructs, and to manage exception rules within databases, Oracle Corporation first released PL/SQL along with Oracle Database Version 6.0 with a limited set of capabilities. Within its capacity, PL/SQL was capable of creating program units that could not be stored inside the database. Eventually, Oracle's release in the application line, SQL *Forms version V3.0, included the PL/SQL engine and allowed developers to implement the application logic through procedures. Back then, PL/SQL used to be part of the transaction processing option in Oracle 6 and the procedural option in Oracle 7. Since the time of its ingenuous beginning, PL/SQL has matured immensely as a standard feature of Oracle Database. It has been enthusiastically received by the developer community, and the credit goes to its support for advanced elements such as modular programming, encapsulation, support for objects and collections, program overloading, native and dynamic SQL, and exception handling.

PL/SQL is loosely derived from Ada (named after Ada Lovelace, an English mathematician who is regarded as the first computer programmer), a high-level programming language, which complies with the advanced programming elements. Building a database backend for an application demands the ability to design the database architecture, skills to code complex business logics, and expertise in administering and protecting the database environment. One of the principal reasons why PL/SQL is a key enabler in the development phase is its tight integration with Oracle's SQL language. In addition to this, it provides a rich platform for implementing the business logic in the Oracle Database layer and store them as procedures or functions for subsequent use. As a procedural language, PL/SQL provides a diverse range of datatypes, iterative and control constructs, conditional statements, and exception handlers.
Preface

In a standard software development space, an Oracle database developer is expected to get involved in schema design; code business logics on the server side by using functions, procedures, or packages; implement action rules by using triggers; and support client-side programs in setting up the application interface. While building the server-side code, developers should understand that their code contributes to the application’s performance and scalability. Language basics are expected to be resilient, but while building robust and secure applications using PL/SQL, developers must take advantage of best practices and try to use advanced language features. This book focuses on the advanced features of PL/SQL validated up to the latest Oracle Database 12c.

Learning by example has always been a well-attested approach for diving deep into a concept. This book will enable you to master the latest enhancements and new features of Oracle Database 12c. For efficient reading, you just have to be familiar with the PL/SQL fundamentals so that you can relate to the evolution of an advanced feature from its ever-expanding roots.

This book closely follows the outline of the Oracle University certification; that is, the Oracle Certified Advanced PL/SQL Developer Professional (1Z0-146) exam. One of the most sought after certifications in the developer community, the 1Z0-146 certification’s objectives are quite comprehensive, and touch upon the various progressive areas of PL/SQL. To name a few, PL/SQL code performance, maintenance, bulk processing techniques, PL/SQL collections, security implementation, and the handling of large objects. For certification aspirants, this book will serve as a one-stop exam guide. At many stages, this book goes beyond the certification objectives and attempts to build a deep understanding of the concepts. Therefore, mid-level database developers will find this book a handy language reference and would be keen to have it on their bookshelves.

My last work on the same subject will remain close to my heart, but this one is straight from my experience. I hope that this book will help you improve your PL/SQL development skills and gain confidence in using advanced features, along with meticulous familiarization of Oracle Database 12c.

"The only real security that a man can have in the world is a reserve of knowledge, experience and ability"

- Henry Ford
What this book covers

Chapter 1, *Overview of PL/SQL Programming Concepts*, provides an overview of PL/SQL fundamentals. It refreshes the basic concepts, such as PL/SQL language features, the anonymous block structure, exception handling, and stored subprograms.

Chapter 2, *Oracle 12c SQL and PL/SQL New Features*, talks about the new features of Oracle Database 12c. It starts with the idea of consolidation of databases on a cloud and how the Oracle 12c Multitenant architecture addresses the requirements. It consolidates the new features in Oracle 12c SQL and PL/SQL, and explains each of them with examples. It will help you to feel the essence of Oracle Database 12c and understand what the driving wheel of innovation is. A section on the Oracle Database 12c In-memory option will familiarize you with the breakthrough feature in the analytics and warehouse space.

Chapter 3, *Designing PL/SQL Code*, primarily focuses on the PL/SQL cursor’s design and handling. You will get to learn the basics of cursor design, cursor types and cursor variables, handling cursors in PL/SQL, and design guidelines. This chapter will also include the enhancements made by Oracle Database 12c with respect to cursors.

Chapter 4, *Using Collections*, introduces you to the world of collections; namely, associative arrays, nested tables, and varrays. Taking you all the way from their creation in SQL and PL/SQL to design considerations, this chapter makes you wise enough to choose the right collection type in a given situation. A section on Oracle Database 12c enhancements to collections introduces a very handy feature that will allow you to join a table and collection.

Chapter 5, *Using Advanced Interface Methods*, focuses on a powerful feature of PL/SQL: how to execute external procedures in PL/SQL. You will learn and understand the specifics of executing a C or Java program in PL/SQL as an external procedure through step-by-step demonstration. This chapter also mentions the Oracle Database 12c enhancement which allows you to secure external procedures through an additional safety net.

Chapter 6, *Virtual Private Database*, provides a detailed overview of the Oracle Database Security Defense-in-depth architecture and focuses on one of the developer-centric features, known as the Virtual Private Database. Oracle Database 12c security enhancements and a demonstration of data redaction will make you understand Oracle’s security offerings.

Chapter 7, *Oracle SecureFiles*, provides a thorough understanding of handling large objects in Oracle and focuses on storage optimizations made by SecureFiles. Introduced in Oracle 11g, SecureFiles is the new storage mechanism that scores high on its advanced features, such as compression, encryption, and deduplication. This chapter also helps you with the recommended migration methods from older LOBs to SecureFiles.
Chapter 8, *Tuning the PL/SQL Code*, introduces the best practices for tuning PL/SQL code. It starts with the PL/SQL optimizer and rolls through the benefits of native compilation, PL/SQL code writing skills, and code evaluation design. This chapter includes the changes in Oracle 12c with respect to large object handling.

Chapter 9, *Result Cache*, explains the result caching feature in Oracle Database. It is a powerful caching mechanism that enhances the performance of SQL queries and PL/SQL functions that are repeatedly executed on the server. This chapter also discusses the enhancements made to the feature in Oracle Database 12c.

Chapter 10, *Analyzing, Profiling, and Tracing PL/SQL Code*, details the techniques used to analyze, profile, and trace PL/SQL code. If you are troubleshooting PL/SQL code for performance, you must learn the profiling and tracing techniques. In an enterprise application environment, these practices are vital weapons in a developer’s arsenal.

Chapter 11, *Safeguarding PL/SQL Code against SQL injection*, describes ways to protect your PL/SQL from being attacked. A vulnerable piece of code is prone to malicious attacks and runs the risk of giving away sensitive information. Efficient code writing and proofing the code from external attacks can help to minimizing the attack surface area. In this chapter, you will learn the practices for safeguarding your code against external threats.

Chapter 12, *Working with Oracle SQL Developer*, describes the benefits of the Oracle SQL Developer for developers, database administrators, and architects. This chapter not only helps you get started with SQL Developer, but also helps you gain a better understanding of the new features of SQL Developer 4.0 and 4.1.
Tuning the PL/SQL Code

Tuning the PL/SQL Code

Database tuning refers to an exercise that aims to improve the performance of a database. The art of optimizing the database performance largely depends on how well one understands the database architecture, application design and server environment. The areas that can be potentially tuned are the application design, network, SQL queries, and PL/SQL code. This exercise starts with the detection and identification of a problem, followed by analysis and tuning recommendations. In this chapter, we will see tuning practices related to the Oracle PL/SQL code.

Programs written in Oracle PL/SQL can be tuned to optimize data-centric and CPU-intensive operations. In a PL/SQL program unit, statements like loops, dynamic SQL statements, routine calls, and the compilation mode can be tuned for better performance. This chapter will discuss the different code compilation techniques and their benefits. The chapter outline is as follows:

• The PL/SQL Compiler
  ° Subprogram inline

• Native compilation of the PL/SQL Code

• Tuning the PL/SQL Code
  ° Build secure applications using bind variables
  ° Call parameters by reference
  ° Avoid an implicit data type conversion
  ° Understand the **NOT NULL** constraint
  ° Select an appropriate numeric data type
  ° Bulk processing in PL/SQL
The PL/SQL Compiler
The PL/SQL compiler converts a piece of PL/SQL code that is written in user-readable language semantics to system code. At a higher level, the code compilation is a three-step process that includes code parsing, parameter binding, and translation into machine code (M-code). The compiler raises a syntax or compilation error if a PL/SQL construct or statement is incorrectly used. Once the PL/SQL program code is error-free and argument binding is done, the system code is generated and stored in the database.

Until Oracle 10g Release 1, the M-code was generated without any code optimization. Starting from Oracle Database 10g Release 2, M-code uses the PL/SQL optimizer to apply code formatting for better performance. The PL/SQL optimizer level is governed by a compilation parameter known as PLSQL_OPTIMIZE_LEVEL. The parameter value can be set at the system or session level as a number between 1 to 3. By default, the parameter value is 2.

The PL/SQL optimizer uses multiple techniques to optimize the PL/SQL code, for example, the removal of dead code from the program unit, or inlining the calls to subprograms. Let us first see how subprogram inlining works.

Subprogram inlining in PL/SQL
A PL/SQL program unit with a locally-declared and invoked subprogram can be optimized using the inlining method. While optimizing the program code, the PL/SQL optimizer replaces the subprogram invocation calls by the subprogram body itself. For the PL/SQL compiler to inline a local subprogram, the following criteria must be met:

- Set PLSQL_OPTIMIZE_LEVEL to 2 and use PRAGMA INLINE: The compiler inlines only those subprograms that are specified with PRAGMA INLINE.
- Set PLSQL_OPTIMIZE_LEVEL to 3: The compiler inlines all the subprogram calls. However, you can specify PRAGMA INLINE to prevent the inlining of a particular subprogram. To disable the inlining of a program unit, specify PRAGMA INLINE ([subprogram unit], 'NO').

PRAGMA INLINE directs the compiler to inline the subprogram calls that are just succeeding it. For example, in the following PL/SQL block, the subprogram call is inlined in Line 5 but not in Line 7:

```plsql
PROCEDURE P_SUM_NUM (P_A NUMBER, P_B NUMBER)        /* Line 1 */
...                                                 /* Line 2 */
BEGIN                                               /* Line 3 */
PRAGMA INLINE ('P_SUM_NUM', 'YES')                  /* Line 4 */
result_1 := P_SUM_NUM (10, 20);                     /* Line 5 */
```
At most of the scenarios, subprogram inlining improves the performance of a PL/SQL program. If the subprogram is a large unit, the cost of inlining would be more than the modular execution.

```plsql
result_2 := P_SUM_NUM (100, 200); /* Line 7 */
END; /* Line 8 */
/* Line 9 */
```

PRAGMA INLINE ([subprogram], 'NO') will always override PRAGMA INLINE ([subprogram], 'YES') in the same declaration.

It is recommended that you inline those utility subprograms that are frequently invoked in a PL/SQL block. The intra-unit inlining is traceable through session-level warnings. The session-level warnings can be turned on by setting the PLSQL_WARNINGS parameter to ENABLE:ALL.

**PRAGMA INLINE**

PRAGMA is a compiler directive that hints the compiler. Any error in the usage of PRAGMA results in a compilation error. Note that a PRAGMA, which accepts an argument, cannot accept a formal argument but always needs an actual argument.

PRAGMA INLINE was introduced in Oracle Database 11g to support subprogram inlining for better PL/SQL performance. When the compilation parameter PLSQL_OPTIMIZE_LEVEL is 2, you have to specify the subprogram to be inlined using PRAGMA INLINE. When PLSQL_OPTIMIZE_LEVEL is 3, the PL/SQL optimizer tries to inline most of the subprogram calls, without requiring any PRAGMA specification. At this stage, if you perceive an inlined subprogram to be irrelevant or undesirable, you can disable the inlining of that particular subprogram through PRAGMA INLINE.

PRAGMA INLINE impacts only when PLSQL_OPTIMIZE_LEVEL is either 2 or 3. When PLSQL_OPTIMIZE_LEVEL is 1, it has no effect.
Tuning the PL/SQL Code

**PLSQL_OPTIMIZE_LEVEL**

Oracle Database 10g introduced the `PLSQL_OPTIMIZE_LEVEL` initialization parameter to enable or disable PL/SQL optimization. If enabled, the optimizer deploys several optimization techniques in accordance with the level. The compilation settings of a PL/SQL program can be queried from the `USER_PLSQL_OBJECT_SETTINGS` dictionary view.

Until Oracle Database 10g, the parameter value could be either 0, 1, or 2. Starting from Oracle Database 11g, the new optimization level can be enabled by setting the parameter value to 3. Note that the default value of the parameter is 2. The parameter value can be modified using the `ALTER_SYSTEM` or `ALTER_SESSION` statements. Only the PL/SQL programs that are compiled after the parameter modification are impacted. You can also specify the `PLSQL_OPTIMIZE_LEVEL` value at the time of explicit compilation of a program unit. For example, a `P_OPT_LVL` procedure can be recompiled using the following statement:

```
ALTER PROCEDURE p_opt_lvl COMPILE PLSQL_OPTIMIZE_LEVEL=1
/
```

The following PL/SQL procedure demonstrates the subprogram inlining using `PRAGMA INLINE`. The procedure adds the series \((1^2) + (2^2) + (3^2) + \ldots + (N^2)\) up to \(N\) terms:

```sql
/*Create a procedure*/
CREATE OR REPLACE PROCEDURE P_SUM_SERIES(p_count NUMBER)
IS

  l_series NUMBER := 0;
  l_time NUMBER;

/*Declare a local subprogram which returns the double of a number*/
FUNCTION F_CROSS (p_num NUMBER, p_multiplier NUMBER) RETURN NUMBER
IS

  l_result NUMBER;

BEGIN
  l_result := p_num * p_multiplier;
  RETURN (l_result);
END F_CROSS;

BEGIN

/*Capture the start time*/
  l_time := DBMS_UTILITY.GET_TIME();
```

Chapter 8

/*Begin the loop for series calculation*/
FOR J IN 1..p_count
LOOP

    /*Set inlining for the local subprogram*/
    PRAGMA INLINE (F_CROSS, 'YES');
    l_series := l_series + F_CROSS(J,2);
END LOOP;

    /*Time consumed to calculate the result*/
    DBMS_OUTPUT.PUT_LINE('Execution time:'||TO_CHAR(DBMS_UTILITY.GET_TIME() - L_TIME));

END;
/

Case 1: When PLSQL_OPTIMIZE_LEVEL = 0
At level 0, the PL/SQL optimizer doesn't kick in, so no code optimization is enabled. The compiler maintains the code evaluation order, and performs object checks.

Enable PLSQL_WARNINGS to capture the inline operations:

    ALTER SESSION SET plsql_warnings = 'enable:all'
    /
    Session altered.

Let's recompile the P_SUM_SERIES procedure with the PLSQL_OPTIMIZE_LEVEL=0 setting:

    ALTER PROCEDURE P_SUM_SERIES COMPILE PLSQL_OPTIMIZE_LEVEL=0
    /
    SP2-0804: Procedure created with compilation warnings

show errors
Errors for PROCEDURE P_SUM_SERIES:

    LINE/COL ERROR
    -------------------------------
    1/1        PLW-05018: unit P_SUM_SERIES omitted optional AUTHID clause;
                 default value DEFINER used

--- [233] ---
Let's execute the procedure with a relatively large input value for a computation-intensive operation:

```
BEGIN
    p_sum_series (10000000);
END;
/
```

Execution time: 776

PL/SQL procedure successfully completed.

**Case 2: When PLSQL_OPTIMIZE_LEVEL = 1**

At level 1, the PL/SQL optimizer applies conventional optimization techniques to a PL/SQL program unit. It eliminates the dead code, and skips the redundant and unnecessary code in the program while generating the p-code instructions.

What is **dead code**? If a PL/SQL statement remains unchanged within an iterative or conditional construct, and doesn't contribute to the program logic, it is known as dead code. While translating the program into the system code instructions, the PL/SQL optimizer identifies the pieces of dead code and prevents their conversion.

Let's recompile the `P_SUM_SERIES` procedure using the `ALTER PROCEDURE` statement for `PLSQL_OPTIMIZE_LEVEL = 1` and check the warnings:

```
ALTER PROCEDURE P_SUM_SERIES COMPILE PLSQL_OPTIMIZE_LEVEL = 1
/
```

```
SP2-0804: Procedure created with compilation warnings
```

```
show errors
Errors for PROCEDURE P_SUM_SERIES:
```

```
LINE/COL ERROR
-----------------------------------------------
1/1   PLW-05018: unit P_SUM_SERIES omitted optional AUTHID clause;
      default value DEFINER used
```

You can also query the compilation warnings from the `USER_ERRORS` dictionary view.

Although `PRAGMA` was specified, the `F_CROSS` invocation was not inlined. Therefore, the subprogram inlining doesn't seem to work when `PLSQL_OPTIMIZE_LEVEL` is set to 1:

```
BEGIN
    P_SUM_SERIES (10000000);
END;
```
Case 3: When PLSQL_OPTIMIZE_LEVEL = 2

At level 2, the PL/SQL optimizer performs the intelligent code optimization through techniques like explicit subprogram inlining and code reorganization.

Recompile the P_SUM_SERIES procedure with PLSQL_OPTIMIZE_LEVEL=2 using the ALTER PROCEDURE statement:

```sql
ALTER PROCEDURE p_sum_series COMPILE PLSQL_OPTIMIZE_LEVEL=2
/
```

Let's retrieve the compilation warnings. Oracle includes a new set of compilation warnings to demonstrate the inlining flow in the program. The PL/SQL optimizer is instructed by PRAGMA to inline the subprogram in line 28. At line 8, the function was removed and its body was merged into the main program. Line 28 shows the inlining request and action:

```sql
SQL> show errors
Errors for PROCEDURE P_SUM_SERIES:

LINE/COL  ERROR
--------  -------------------------------------------------
1/1       PLW-05018: unit P_SUM_SERIES omitted optional AUTHID clause; default value DEFINER used
8/3       PLW-06006: uncalled procedure "F_CROSS" is removed.
28/7      PLW-06004: inlining of call of procedure 'F_CROSS' requested
28/7      PLW-06005: inlining of call of procedure 'F_CROSS' was done
```

Let's check how a level 2-optimized and compiled program performs:

```sql
BEGIN
    P_SUM_SERIES (10000000);
END;
/
```

Well done! It runs in 456ms, which is almost half of its baseline, that is, 2 times better performance.
Case 4: When PLSQL_OPTIMIZE_LEVEL = 3

Level 3 optimization focuses on the predictive and automatic inlining and code refactoring. Modify the compilation parameter in the session using the following statement:

```
ALTER SESSION SET plsql_optimize_level=3
/
```

Session altered.

To see the implicit inlining of the subprogram, let's recreate the `P_SUM_SERIES` procedure without the `PRAGMA` specification:

```sql
/*Create a procedure*/
CREATE OR REPLACE PROCEDURE P_SUM_SERIES(p_count NUMBER) IS
  l_series NUMBER := 0;
  l_time NUMBER;

/*Declare a local subprogram to return the double of a number*/
FUNCTION F_CROSS (p_num NUMBER) RETURN NUMBER IS
  BEGIN
    RETURN (p_num * 2);
  END F_CROSS;

BEGIN

  /*Capture the start time*/
  l_time := DBMS_UTILITY.GET_TIME();

  /*Begin the loop for series calculation*/
  FOR J IN 1..p_count
    LOOP

    /*Set inlining for the local subprogram*/
    l_series := l_series + F_CROSS (J);

  END LOOP;

  /*Time consumed to calculate the result*/
  DBMS_OUTPUT.PUT_LINE('Execution time:'||TO_CHAR(DBMS_UTILITY.GET_TIME() - L_TIME));

END;
```

SP2-0805: Procedure altered with compilation warnings
Let's check the compilation warnings:

show errors
Errors for PROCEDURE P_SUM_SERIES:

LINE/COL ERROR
-------- --------------------------------------------------
1/1      PLW-05018: unit P_SUM_SERIES omitted optional AUTHID clause; default value DEFINER used
8/3      PLW-06006: uncalled procedure "F_CROSS" is removed.
25/7     PLW-06005: inlining of call of procedure 'F_CROSS' was done

Note the warnings at line 8 and line 24. No PRAGMA specified, but the PL/SQL optimizer inlines the subprogram. Let's see how the procedure execution is impacted by level 3 optimization:

BEGIN
  P_SUM_SERIES (10000000);
END;
/

Execution time:420
PL/SQL procedure successfully completed.

The execution time is almost equal to the level 2 optimization, because the technique to optimize the code was similar in both the cases.

The consolidation of the execution numbers is shown in the following table:

<table>
<thead>
<tr>
<th>PLSQL_OPTIMIZE_LEVEL</th>
<th>Inlining Requested</th>
<th>Inlining Done</th>
<th>Execution time</th>
<th>Performance factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLSQL_OPTIMIZE_LEVEL = 0</td>
<td>Yes</td>
<td>No</td>
<td>776</td>
<td>1</td>
</tr>
<tr>
<td>PLSQL_OPTIMIZE_LEVEL = 1</td>
<td>Yes</td>
<td>No</td>
<td>710</td>
<td>1.1x</td>
</tr>
<tr>
<td>PLSQL_OPTIMIZE_LEVEL = 2</td>
<td>Yes</td>
<td>Yes</td>
<td>456</td>
<td>1.7x</td>
</tr>
<tr>
<td>PLSQL_OPTIMIZE_LEVEL = 3</td>
<td>No</td>
<td>Yes</td>
<td>420</td>
<td>1.8x</td>
</tr>
</tbody>
</table>

It is recommended that you have PLSQL_OPTIMIZE_LEVEL as 2 for the database development and production environments. It ideally optimizes the PL/SQL code and enables the database developers to control the subprogram inlining.
Native and interpreted compilation techniques

Until Oracle 9i, all PL/SQL program units were compiled in interpreted mode. Starting with Oracle 9i, PL/SQL programs can be compiled either in interpreted mode or native mode. Let's quickly understand what are interpreted and native compilation modes.

The PL/SQL compiler generates machine code for the program units. Machine code is a set of instructions stored in the database dictionaries that runs against the PL/SQL virtual machine (PVM). At the time of the program invocation, the M-code is scanned by one of the subroutines in the PVM. The scanning process involves the identification of the operation code and operands and routing the call to the appropriate subroutine. The scanning of the machine code instructions consumes systems resources, which may impact the runtime performance of the code. This is how interpreted compilation works. In the case of native compilation, a shareable dynamic linked library (DLL) is generated instead of a machine code. At runtime, the DLL is directly invoked and it invokes the subroutine along with the required set of arguments. With DLL, you don't have to go through the scanning procedure at runtime, which contributes to better code performance.

Until Oracle Database 9i and 10g, the challenge with native compilation was to generate a platform-specific shareable and nonportable library. It was made possible by the C compiler and linker commands located in $ORACLE_HOME/plsql/spnc_commands, which would compile and link the C translation of the machine code to a platform-specific library. In Oracle Database 9i, the libraries were stored on a file system, while in Oracle Database 10g, the libraries were stored in a catalog (database dictionary). A copy of the shared libraries was stored on a file system for backup and operating-system operations. The file system location used to be specified in PLSQL_NATIVE_LIBRARY_DIR and PLSQL_NATIVE_LIBRARY_SUBDIR_COUNT. The shared libraries were automatically extracted, restored, and deleted from the file system. If any of the file system libraries were lost or dropped accidentally, they were re-extracted and restored in the file system after the database was restarted.

The shared libraries generated during native compilation are stored in the NCOMP_DLL$ dictionary.

Native compilation is supported with the Real Application Cluster (RAC) database option.
This approach worked well until the production DBAs showed reluctance in having the C compiler on production environments and procuring an additional C compiler license.

**Oracle Database 11g Real Native Compilation**

Oracle Database 11g introduced the real native compilation technique to address the below:

- Remove the C compiler and linker dependencies to generate platform specific libraries
- Improve code compilation performance
- No compromise with the runtime performance

Starting from Oracle Database 11g, the Oracle Database can generate platform-specific machine code from PL/SQL code and store it in the database catalog or an internal dictionary without the intervention of a third party C compiler. The native code is then loaded directly from the catalog without staging it on a filesystem. Oracle Database 11g real native compilation improves the compilation performance by a degree of magnitude.

The following list summarizes the salient features of real native compilation process:

- There is no dependency on a third party C Compiler.
- The native code is stored in the `SYSTEM` tablespace.
- There is no DBA task to set the filesystem initialization parameter.
- The compilation method is controlled by an initialization parameter: `PLSQL_CODE_TYPE [native/interpreted]`.
- Native compilation is supported for all types of PL/SQL program units.
- In `INTERPRETED` mode, the PL/SQL code is compiled to the equivalent machine code. The machine code instructions are scanned during the runtime and routed to the appropriate subroutine on the PL/SQL Virtual Machine.
- In `NATIVE` mode, the PL/SQL code is compiled to machine code, which is directly translated to the platform-specific. At runtime, the machine code is executed natively by the PL/SQL Virtual Machine.
- The real native compilation mode can be set at system level, session level, and object level. A natively compiled program unit can call an interpreted program and vice versa.
Selecting the appropriate compilation mode

During the database development phase, the PL/SQL program units tend to get recompiled frequently. At this stage, you would want to have the faster compilation of programs. Therefore, you can choose to compile the program units in interpreted mode.

Once the database is deployed for production, and has fewer chances of frequent recompilation, you can choose native compilation. PL/SQL native compilation enhances the runtime performance of the program units. Computation-intensive programs are expected to benefit most from the native compilation method. The performance gains will be less if the program is frequently switching the context between the SQL and PL/SQL engines.

Oracle allows having a mix of program units with different compilation modes. However, keep in mind that if a natively compiled unit invokes an interpreted unit, the program execution performance will be impacted.

Setting the compilation mode

The compilation method is set using the PLSQL_CODE_TYPE parameter. The admissible values for the parameter are INTERPRETED and NATIVE. The default value of the parameter is INTERPRETED. The parameter can be set using the ALTER SYSTEM or ALTER SESSION statement.

At the SYSTEM level:

ALTER SYSTEM SET PLSQL_CODE_TYPE = [NATIVE | INTERPRETED]

At the SESSION level:

ALTER SESSION SET PLSQL_CODE_TYPE = [NATIVE | INTERPRETED]

Alternatively, you can also compile a particular PL/SQL program unit in the native or interpreted method by recompiling an object using the ALTER <object type> statement. For example, the following script will recompile a procedure with a new value for PLSQL_CODE_TYPE, but reusing the existing compilation parameters:

ALTER PROCEDURE <procedure name> COMPILE PLSQL_CODE_TYPE=NATIVE REUSE SETTINGS
/

The PL/SQL program units retain their compilation mode setting unless they are recompiled in a different compilation mode.
Querying the compilation settings

The compilation mode of an object can be queried from the data dictionary view: [USER | DBA | ALL]_PLSQL_OBJECT_SETTINGS. This dictionary view contains the compilation settings for standalone program units, package specification, package body, and packaged subprograms:

/*Describe the structure of USER_PLSQL_OBJECT_SETTINGS*/

SQL> DESC USER_PLSQL_OBJECT_SETTINGS

Name                Null?       Type
------------------- ----------- ----------------
NAME                 NOT NULL    VARCHAR2(30)
TYPE                NOT NULL    VARCHAR2(12)
PLSQL_OPTIMIZE_LEVEL   NULL       NUMBER
PLSQL_CODE_TYPE       NULL       VARCHAR2(4000)
PLSQL_DEBUG           NULL       VARCHAR2(4000)
PLSQL_WARNINGS        NULL       VARCHAR2(4000)
NLS_LENGTH_SEMANTICS  NULL       VARCHAR2(4000)
PLSQL_CCFLAGS         NULL       VARCHAR2(4000)
PLSCOPE_SETTINGS      NULL       VARCHAR2(4000)
ORIGIN_CON_ID         NULL       NUMBER

Compiling a program unit for native or interpreted compilation

In this section, let's check how a PL/SQL program unit can be compiled in different compilation modes at the object level.

1. Show the compilation mode of the session:

/*Connect as sysdba*/
connect sys/oracle as sysdba

/*Display current setting of parameter PLSQL_CODE_TYPE*/
SELECT name, value
FROM v$parameter
WHERE name = 'plsql_code_type'
/

NAME         VALUE
------------- ------------------
plsql_code_type            INTERPRETED
Tuning the PL/SQL Code

2. Create a standalone function, which currently gets compiled in the INTERPRETED mode:

```sql
/*Create a function*/
CREATE OR REPLACE FUNCTION f_get_caps (p_name VARCHAR2)
RETURN VARCHAR2
IS
BEGIN
    RETURN UPPER (p_name);
END;
/
Function created.
```

3. Verify the compilation mode of the F_GET_CAPS function:

```sql
/*Query the compilation settings for the function F_GET_CAPS*/
SELECT name, type, plsql_code_type, plsql_optimize_level OPTIMIZE
FROM USER_PLSQL_OBJECT_SETTINGS
WHERE name = 'F_GET_CAPS'
/
NAME       TYPE         PLSQL_CODE_TYPE   OPTIMIZE
---------- ------------ ----------------- ----------
F_GET_CAPS FUNCTION     INTERPRETED       2
```

4. Recompile the function using native mode:

```sql
/*Explicitly compile the function*/
ALTER FUNCTION f_get_caps COMPILE PLSQL_CODE_TYPE=NATIVE
/
Function altered.
```

5. Confirm the change in the compilation mode of the F_GET_CAPS function:

```sql
/*Query the compilation settings for the function F_GET_CAPS*/
SELECT name, type, plsql_code_type, plsql_optimize_level OPTIMIZE
FROM USER_PLSQL_OBJECT_SETTINGS
WHERE name = 'F_GET_CAPS'
/
NAME       TYPE         PLSQL_CODE_TYPE OPTIMIZE
---------- ------------ --------------- ---------
F_GET_CAPS FUNCTION     NATIVE          2
```
6. You can also verify the generation of a shared library for the `F_GET_CAPS` function by querying the `NCOMP_DLL$` dictionary view. The library is stored as a BLOB in the `NCOMP_DLL$` view:

```sql
connect sys/system as sysdba
SELECT object_name, dllname
FROM ncomp_dll$, dba_objects
WHERE obj#=object_id
AND owner='SCOTT'
/
```

<table>
<thead>
<tr>
<th>OBJECT_NAME</th>
<th>DLLNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>F_GET_CAPS</td>
<td>465F4745545F434150535F5F53434F54545F5F465F</td>
</tr>
<tr>
<td></td>
<td>5F3934303837</td>
</tr>
</tbody>
</table>

The library is automatically deleted from the dictionary if the object is recompiled with a different compilation mode.

### Recompiling a database for a PL/SQL native or interpreted compilation

You can compile all the PL/SQL program units in a database from interpreted to native compilation mode and vice versa. Oracle enables the bulk compilation of the PL/SQL program units through inbuilt scripts: `dbmsupnv.sql` and `dbmsupgin.sql`. The `dbmsupgnv.sql` script compiles all the program units using native compilation mode while `dbmsupgin.sql` compiles all the program units in interpreted mode.

The following points should be considered during the planning stage:

- Choose the right compilation mode—the selection of the appropriate compilation mode is the key to ensure better code performance.
- `SYSDBA` or a user with `DBA` privileges can run the recompilation procedure. If the user is protected through the Oracle Database Vault option, the user must be granted the `DV_PATCH_ADMIN` role.
- Object type specifications cannot be compiled in native compilation mode.
- Skip the PL/SQL package specifications without an executable section from native compilation.
- You cannot exclude any PL/SQL program unit while compiling an entire database for interpreted compilation.
The following steps show how to compile a database for native or interpreted compilation mode:

1. Shutdown the database—you must shutdown the Oracle Database and the TNS listener. Ensure that all connections to the database from the application tier are terminated.

2. Set `PLSQL_CODE_TYPE` as `NATIVE` or `INTERPRETED` in the parameter file. If you are using a server parameter file (spfile), you can modify the parameter value before a database is shutdown, or after the database is started. The `dbmsupgnv.sql` script also sets `PLSQL_CODE_TYPE` before compiling the PL/SQL program units:
   
   /*Alter the system to set the new compilation mode*/
   SQL> ALTER SYSTEM SET PLSQL_CODE_TYPE=NATIVE SCOPE=SPFILE
          /
   System altered.

3. Verify and set `PLSQL_OPTIMIZE_LEVEL` as 2 (or higher).

4. Start the database in the upgrade mode:
   
   SQL> connect sys/oracle as sysdba

   Connected to an idle instance.

   /*Startup in upgrade mode*/
   SQL> startup upgrade
   ORACLE instance started.
   ...
   ...
   Database mounted.
   Database opened.

   If you are working in an Oracle Database 12c Multitenant architecture, you have to open the pluggable database in the UPGRADE and RESTRICTED mode:

   ALTER PLUGGABLE DATABASE <pdb name> OPEN UPGRADE RESTRICTED
   /

5. Connected as the `SYS` user, execute the `dbmsupgnv.sql` script:
   
   /*Execute the recompilation script*/
   SQL> @ORACLE_HOME/rdbms/admin/dbmsupgnv.sql
   ...
   ...
   #################################################################
   ######
dbmsupgnv.sql completed successfully. All PL/SQL procedures, functions, type bodies, triggers, and type bodies objects in the database have been invalidated and their settings set to native.

Shut down and restart the database in normal mode and run utlrp.sql to recompile invalid objects.

The dbmsupgnv.sql and dbmsupgin.sql scripts create a PL/SQL package sys.dbmsncdb with two subprograms: SETUP_FOR_NATIVE_COMPILE and SETUP_FOR_INTERPRETED_COMPILE. The dbmsupgnv.sql involves SETUP_FOR_NATIVE_COMPILE with a user supplied input. The input should be TRUE to compile all functions, procedures, package bodies, triggers, and type bodies. If the input is FALSE, package and object type specifications are also included in the process. If the PL/SQL package specifications contain an executable section, you should provide the input as FALSE.

The script modifies the compilation of all the PL/SQL programs to native but invalidates them at the same time.

6. Shut down and restart the database in the NORMAL mode:

```sql
/*Shutdown and startup the database*/
SQL> shutdown immediate
Database closed.
Database dismounted.
ORACLE instance shut down.

SQL> startup
ORACLE instance started.
...  
...  
Database mounted.
Database opened.
```
7. Execute the utlrp.sql script to recompile the invalidated objects. Although the objects that have been invalidated get automatically recompiled whenever they are subsequently invoked, it is recommended that you compile and revalidate the objects to reduce recompilation latencies.

It is recommended that you run utlrp.sql in a restricted session. Any connection request from the application services or a database session may lead to deadlocks.

SQL> ALTER SYSTEM ENABLE RESTRICTED SESSION
/
System altered.

SQL> @ORACLE_HOME\rdbms\admin\utlrp.sql

The script recompiles all the program units that have a default compilation mode. You can rerun the script any number of times to recompile the invalidated objects. The recompilation operation can be optimized through parallelization wherein the degree of parallelism is selected based on CPU_COUNT and PARALLEL_THREADS_PER_CPU.

For troubleshooting and diagnosis, you can use the following queries:

° Query the invalid objects:

```sql
SELECT o.OWNER,
       o.OBJECT_NAME,
       o.OBJECT_TYPE
FROM DBA_OBJECTS o, DBA_PLSQL_OBJECT_SETTINGS s
WHERE o.OBJECT_NAME = s.NAME
AND o.STATUS='INVALID'
/
```

° Query the count of the PL/SQL programs in each compilation mode:

```sql
SELECT TYPE,
       PLSQL_CODE_TYPE,
       COUNT(*)
FROM DBA_PLSQL_OBJECT_SETTINGS
WHERE PLSQL_CODE_TYPE IS NOT NULL
GROUP BY TYPE, PLSQL_CODE_TYPE
ORDER BY TYPE, PLSQL_CODE_TYPE
/
```
The PL/SQL objects with NULL plsql_code_type are Oracle's internal objects.

° Query the number of the PL/SQL objects that are compiled so far using utlrp.sql:

```
SELECT COUNT(*) FROM UTL_RECOMP_COMPILED
```

8. Disable the restricted session:

```
SQL> ALTER SYSTEM DISABLE RESTRICTED SESSION
```

System altered.

If you want to rollback the compilation mode to interpreted, you can follow the same steps, and replace dbmsupgnv.sql with the dbmsupgin.sql script.

**Tuning PL/SQL code**

We have just discussed two key features in the performance management space. The PL/SQL code optimization and compilation can significantly improve runtime performance. Apart from the code optimization techniques, there are several efficient coding practices that may impact overall PL/SQL performance. PL/SQL performance management is often regarded as less tedious, when compared to the database tuning exercise because the area of operation is a program unit and doesn't require the instance to be restarted. In this section, we will discuss some of the selective but key features to improve PL/SQL coding practices for better performance.

**Build secure applications using bind variables**

It is highly encouraged to use bind variables in SQL statements while building robust database applications. Bind variables reduce the parsing overhead by enabling an SQL statement to be executed using its already parsed image stored in the shared SQL area of the database memory.

Every SQL statement executed by the SQL engine is parsed, optimized, and executed. Only after the SQL statement is executed is the result fetched and displayed to the user. The SQL execution steps are briefly described, as follows:

- Generate a hash value for the SQL statement.
- Look for a matching hash value of cached cursors in the shared pool.
Tuning the PL/SQL Code

- If the match is not found, Oracle continues with the statement parsing and optimization stages. This is hard parsing. It is a CPU-intensive operation and involves contention for latches in the shared SQL area.

- If the match is found, reuse the cursor and reduce parsing to a privilege check. This is a soft parse. Thereafter, without needing further optimization, the explain plan is retrieved from the library cache, and the SQL query is executed.

Oracle Database 12c introduced adaptive query optimization, which allows the optimizer to re-optimize the existing explain plans upon subsequent executions of an SQL query.

- The result set is fetched.

In an enterprise application, many `SELECT` statements or transactional statements (`INSERT`, `UPDATE`, or `DELETE`) are similar in structure but they run with different predicate values. The hashes for two of the same SQL statements with different values will never be the same, thus resulting in a hard parse. A hard parse, being a non-scalable expensive operation, degrades the application performance. The only way to get rid of the problem is to encourage soft parsing by maximizing the use of cached cursors and associated explain plans.

The use of bind variables promotes soft parsing. A bind variable is a substitution variable that acts as a placeholder for the literal values. Bind variables enable an SQL statement to appear the same even after multiple runs with different input values. This reduces hard parsing and the existing plans in the library cache can be used to optimize the current SQL statements.

Avoid using hard-coded literals in SQL queries and PL/SQL programs.

For example, the following `SELECT` query will have the same hash value for the different values of employee ids:

```sql
/*Select EMP table with a bind variable*/
SELECT ename,
       deptno,
       sal
FROM emp
WHERE empno = :empid
/
```
Oracle PL/SQL supports the use of bind variables. All references to a block variable or program argument are treated as a bind variable. In the case of a dynamic SQL, either using `DBMS_SQL` or `EXECUTE IMMEDIATE`, you must use bind variables. Bind variables help dynamic SQL in two ways. First, it improves the code performance. Secondly, it reduces the risk of SQL injection by covering the vulnerable areas. Let's conduct a small illustration to see the benefits of bind variables in PL/SQL.

The following PL/SQL block finds the count of distinct object type accessible by the `SCOTT` user. It executes the `SELECT` query using `EXECUTE IMMEDIATE`. In order to get the accurate results, we will flush the shared pool:

```sql
connect sys/oracle as sysdba
ALTER SYSTEM FLUSH SHARED_POOL
/
connect scott/tiger
SET SERVEROUT ON

/*Start the PL/SQL block*/
DECLARE
    /*Local block variables */
    l_count NUMBER;
    l_stmt VARCHAR2(4000);
    clock_in NUMBER;
    clock_out NUMBER;

    TYPE v_obj_type is TABLE OF varchar2(100);
    l_obj_type v_obj_type := v_obj_type ('TYPE', 'PACKAGE', 'PROCEDURE',
                                          'TABLE', 'SEQUENCE', 'OPERATOR', 'SYNONYM');
BEGIN
    /*Capture the start time */
    clock_in := DBMS_UTILITY.GET_TIME ();

    /*FOR loop to iterate the collections */
    FOR I IN l_obj_type.first..l_obj_type.last LOOP
        l_stmt := 'SELECT count(*)
                  FROM all_objects
                  WHERE object_type = ''||l_obj_type(i)||''';
        EXECUTE IMMEDIATE l_stmt INTO l_count;
    END LOOP;

    /*Capture the end time */
```
Tuning the PL/SQL Code

clock_out := DBMS_UTILITY.GET_TIME ();

DBMS_OUTPUT.PUT_LINE ('Execution time without bind variables:' || TO_CHAR(clock_out-clock_in));

END;
/

Execution time without bind variables:948

PL/SQL procedure successfully completed.

Now, we'll rewrite the above PL/SQL block using bind variables:

connect sys/oracle as sysdba
ALTER SYSTEM FLUSH SHARED_POOL
/
connect scott/tiger
SET SERVEROUT ON

/*Start the PL/SQL block */
declare

/*Local block variables */
l_count NUMBER;
l_stmt VARCHAR2(4000);
clock_in NUMBER;
clock_out NUMBER;

 TYPE v_obj_type is TABLE OF varchar2(100);
l_obj_type v_obj_type := v_obj_type ('TYPE', 'PACKAGE', 'PROCEDURE','TABLE', 'SEQUENCE', 'OPERATOR', 'SYNONYM');
BEGIN

/*Capture the start time */
clock_in := DBMS_UTILITY.GET_TIME ();

/*FOR loop to iterate the collection */
FOR I IN l_obj_type.first..l_obj_type.last LOOP

/*Build the SELECT statement by using bind variable*/
l_stmt := 'SELECT count(*)
FROM all_objects
WHERE object_type = :p1';
/*Use dynamic SQL to execute the SELECT statement*/
EXECUTE IMMEDIATE l_stmt INTO l_count USING l_obj_type(i);
END LOOP;

clock_out := DBMS_UTILITY.GET_TIME ();

DBMS_OUTPUT.PUT_LINE ('Execution time with bind variables:'||TO_CHAR(clock_out-clock_in));

END;
/

Execution time with bind variables:121

PL/SQL procedure successfully completed.

The block with the bind variables gets executed at least 8 times faster than the one which uses literals in the SQL query inside the PL/SQL block. The reason for the performance gain is the soft parsing of the SELECT statement.

In the case of legacy applications or access protected applications, you might face difficulties in modifying the code to include the bind variables. Oracle makes this daunting task easier by controlling the cursor sharing behavior through a switch.

You can set CURSOR_SHARING parameter to EXACT or FORCE to share the cursors across the sessions in a database instance. If the parameter is set to EXACT, only the SQL statements that have exactly the same structure and parameters will be shared. If the parameter is set to FORCE, then Oracle attempts to substitute all the literals in a query with system-generated bind variables. By default, the parameter value is set to EXACT. Cursor sharing greatly improves the performance. At the same time, forced cursor sharing involves extra effort in searching for the same cursor in the shared pool.

The SIMILAR value of CURSOR_SHARING has been deprecated in Oracle Database 12c.

Call parameters by reference

A PL/SQL program invocation can pass a parameter either by its value or by reference. Arguments in the IN mode are passed by reference (the default) while the arguments in the OUT and IN OUT modes are passed by value. In the case of pass by value method, the parameter value before the invocation has to be stored in a temporary variable. This temporary variable is a memory variable and is used to assign the value back to the parameter, if the subprogram call ends in an unhandled exception. If the PL/SQL program accepts composite data types and complex variables in the OUT and IN OUT modes, copying and maintaining the temporary variable can cause a performance overhead, thereby slowing down the program execution.
Tuning the PL/SQL Code

Oracle recommends that you specify the `NOCOPY` hint for the pass by value parameters. A parameter with the pass mode as `OUT NOCOPY` or `IN OUT NOCOPY` is passed by reference, thus avoiding the overhead of maintaining a temporary variable. For simple data values, which are generally small, the gain will be negligible.

**Avoiding an implicit data type conversion**

Oracle converts the data types to a compatible type during the operations like assignment, predicates with value comparison, or passing parameters to subprograms. This is an implicit activity that follows a data type precedence to convert the data types. However, it may cause an overhead when executing a critical program. The following are the best practices that can reduce the implicit data type conversion in the programs:

- Understand that SQL data types and PL/SQL data types have different internal representations. You should explicitly convert the SQL type variables to PL/SQL types and then use those in an expression. For example, `PLS_INTEGER` is a PL/SQL data type that uses machine arithmetic to speed up compute intensive operations. If `NUMBER` type variables are used in an expression, Oracle uses library arithmetic, and therefore, there are no performance gains.

- A variable getting assigned to a table column should be of the same data type to avoid an implicit data type conversion. As a best practice, declare the variables with the `%TYPE` attribute. Similarly, a record can be declared with the `%ROWTYPE` attribute. Such variable declarations always remain synchronized with the database columns, improve the program's scalability, and reduce human errors. Consider the following declarations:

  ```plsql
  empname emp.ename%type
  type emp_rec is record of emp%rowtype;
  CURSOR c IS
  SELECT prod_id, prod_code, prod_name
  FROM products;
  prod_rec c%rowtype;
  ```

- Use the SQL conversion function to convert the to-be assigned variable to the target variable's data type. Some of the conversion functions are `TO_CHAR`, `TO_NUMBER`, `TO_DATE`, `TO_TIMESTAMP`, `CAST`, and `ASCIISTR`. 
• Pass a variable of correct data type while invoking the PL/SQL program units with parameters. You can also include overloaded subprograms in a PL/SQL package that accepts the parameters of the different data types. You can invoke the packaged subprogram that best matches the available set of variables. For example, the following package has two overloaded functions. You can invoke either of the two depending on the type of variables available:

```sql
CREATE OR REPLACE PACKAGE pkg_sum AS
    FUNCTION p_sum_series (p_term PLS_INTEGER, p_factor PLS_INTEGER) RETURN PLS_INTEGER;
    FUNCTION p_sum_series (p_term NUMBER, p_factor NUMBER) RETURN NUMBER;
END;
/
```

Understanding the NOT NULL constraint

A block variable can be declared `NOT NULL` at the time of the declaration. Oracle performs the nullability test every time a value is assigned to the `NOT NULL` variable. In large PL/SQL program units, the nullability check can be an overhead and impact the PL/SQL block’s runtime performance. Instead, it is advised to have an explicit nullability check for the variable in the executable section of the block.

The following PL/SQL program compares the performance of a `NOT NULL` variable and nullable local variable:

```sql
/*Enable the SERVEROUTPUT to display block results*/
SET SERVEROUTPUT ON

/*Start the PL/SQL block*/
DECLARE
    l_nn_num  NUMBER NOT NULL := 0;
    l_num     NUMBER := 0;
    clock_in  NUMBER;
    clock_out NUMBER;
BEGIN

    /*Capture the start time*/
    clock_in := DBMS_UTILITY.GET_TIME();

    /*Start the loop*/
    FOR I IN 1..1000000
        LOOP
            l_term := 1;
            l_factor := 2;
            l_result := l_term;
            FOR J IN 1..l_factor
                l_result := l_result + l_term;
            END LOOP;
            l_result := 0;
        END LOOP;

    /*Capture the end time*/
    clock_out := DBMS_UTILITY.GET_TIME();

    /*Display the result*/
    DBMS_OUTPUT.PUT_LINE('Time taken: ' || (clock_out - clock_in) || ' seconds');
END;
/
```
Tuning the PL/SQL Code

```plsql
l_nn_num := l_nn_num + i;
END LOOP;

clock_out := DBMS_UTILITY.GET_TIME();

/*Compute the time difference and display*/
DBMS_OUTPUT.PUT_LINE('Time for NOT NULL:'||TO_CHAR(clock_out-clock_in));

/*Capture the start time*/
clock_in := DBMS_UTILITY.GET_TIME();

/*Start the loop*/
FOR I IN 1..1000000
  LOOP
    l_num := l_num + i;
  END LOOP;

clock_out := DBMS_UTILITY.GET_TIME();

/*Compute the time difference and display*/
DBMS_OUTPUT.PUT_LINE('Time for NULL:'||TO_CHAR(clock_out-clock_in));
END;
/

Time for NOT NULL:111
Time for NULL:76

PL/SQL procedure successfully completed.
```

In the preceding PL/SQL block, the nullable variable outperforms the NOT NULL variable by one and a half times. The reason for this performance gain is the reduction in the number of nullability checks.

**Selection of an appropriate numeric data type**

The `PLS_INTEGER` data type is a part of the `NUMBER` data type family. Most of the number or integer data types are generic ones and the reason behind this is to support code portability. However, `PLS_INTEGER` is specifically designed for performance. It was introduced in Oracle Database Release 7 to speed up the computation-intensive operations through native machine arithmetic instead of library arithmetic.
The following PL/SQL block compares the performance of the PLS_INTEGER and NUMBER variables:

```plsql
/*Enable the SERVEROUTPUT to display block results*/
SET SERVEROUTPUT ON

/*Start the PL/SQL block*/
DECLARE
  l_pls_int PLS_INTEGER := 1;
  l_num NUMBER := 1;
  l_factor PLS_INTEGER := 2;
  clock_in NUMBER;
  clock_out NUMBER;
BEGIN
  /*Capture the start time*/
  clock_in := DBMS_UTILITY.GET_TIME();

  /*Begin the loop to perform a mathematical calculation*/
  FOR I IN 1..10000000
    LOOP
      /*The mathematical operation increments a variable by one*/
      l_num := l_num + l_factor;
    END LOOP;
  clock_out := DBMS_UTILITY.GET_TIME();

  /*Display the execution time consumed*/
  DBMS_OUTPUT.PUT_LINE('Time by NUMBER:'||TO_CHAR(clock_out-clock_in));

  /*Capture the start time*/
  clock_in := DBMS_UTILITY.GET_TIME();

  /*Begin the loop to perform a mathematical calculation*/
  FOR J IN 1..10000000
    LOOP
      /*The mathematical operation increments a variable by one*/
      l_pls_int := l_pls_int + l_factor;
    END LOOP;
  clock_out := DBMS_UTILITY.GET_TIME();

  /*Display the time consumed*/
  DBMS_OUTPUT.PUT_LINE('Time by PLS_INTEGER:'||TO_CHAR(clock_out-clock_in));
```

---

[255]
The performance of the PLS_INTEGER variable is at least three times better than a NUMBER variable. An arithmetic expression having a mix of the PLS_INTEGER and NUMBER type variables will use library arithmetic and no performance gains will be obtained.

PLS_INTEGER is a 32-bit data type that can store values in the range of $-2,147,483,648$ to $2,147,483,647$. It accepts integer values only. If a floating value is assigned to PLS_INTEGER, it is rounded to the nearest integer. If the PLS_INTEGER precision range is violated, Oracle raises an ORA-01426: numeric overflow exception. To resolve such scenarios, Oracle 11g introduced a new subtype of PLS_INTEGER, which is known as SIMPLE_INTEGER. The SIMPLE_INTEGER data type has the same range as that of PLS_INTEGER, but in the case of numeric overflows, it is automatically set to $-2,147,483,648$ instead of raising an exception. As the overflow check is suppressed for SIMPLE_INTEGER, it is faster than PLS_INTEGER.

SIMPLE_INTEGER is a NOT NULL data type; therefore, all local variables must be initialized or defaulted to a definitive value.

The following PL/SQL anonymous block declares a SIMPLE_INTEGER variable and defaults it to $2,147,483,646$, that is, the last value in the precision range. In the program body, the variable is incremented by 1. Let's check what happens:

```sql
/*Enable the SERVEROUTPUT to display block results*/
SET SERVEROUTPUT ON

/*Start the PL/SQL block*/
DECLARE
    l_simple SIMPLE_INTEGER:= 2147483646;
BEGIN
    /*Increment the variable by 1*/
    l_simple:= l_simple +1;
    DBMS_OUTPUT.PUT_LINE('After 1st increment:'|| l_simple);

    /*Re-Increment the variable by 1*/
    l_simple:= l_simple +1;
    DBMS_OUTPUT.PUT_LINE('After 2nd increment:'|| l_simple);
EXCEPTION
```
WHEN OTHERS THEN
  DBMS_OUTPUT.PUT_LINE('Numeric Overflow exception occurred');
END;
/

After 1st increment: 2147483647
After 2nd increment: -2147483648

PL/SQL procedure successfully completed.

Bulk processing in PL/SQL

Bulk data processing is one of the key features in performance management. Bulk SQL is a feature that reduces context switches between the SQL and PL/SQL processing engines. One of the biggest benefits of PL/SQL development is its seamless integration with the SQL language. The SQL statements can directly appear in the PL/SQL program body. In a PL/SQL unit with the SQL statements and PL/SQL constructs, the PL/SQL engine and SQL engine communicate through a context switch. The PL/SQL engine switches the program context to the SQL engine to execute the SQL statements and get the result. The frequent back-and-forth context switching between the processing engines causes a severe overhead and slows down the program execution. The row-by-row processing of records is quite slow compared to bulk processing.

Oracle implements bulk operations through the BULK COLLECT clause and the FORALL statement. The BULK COLLECT clause retrieves multiple records in a single fetch while the FORALL statement can process multiple DML statements in groups rather than row by row. In many implementations, the performance gains by the use of bulk processing features have been phenomenal.

BULK COLLECT

You can use BULK COLLECT in:
  - The SELECT...INTO statement
  - The RETURNING INTO clause with the UPDATE or DELETE statement
  - The FETCH INTO clause with an explicit cursor

The bulk collect feature depends largely on collections and can be used with all the three forms of collections, that is, associative arrays, nested tables, and varrays. Multiple records can be fetched into a collection in a single fetch. This reduces the number of the context switches between the two processing engines. During the fetch operation, the collection variables get densely populated. In the case of no rows being fetched, the collections are left empty resulting in an empty collection.
Bulk operations are a CPU-intensive operation. They cannot be parallelized.

Until Oracle Database 9i, \texttt{BULK COLLECT} could be used only with static SQL statements, but starting with Oracle Database 10g, it can be used in dynamic SQL statements too.

The \texttt{CREATE TABLE} script creates the test table with the sample data from the \texttt{ALL_OBJECTS} dictionary view to be used in this illustration. The \texttt{ALL_OBJECTS} dictionary view contains details of the schema objects. During our illustration, we will work with the object id, object type, and object name columns of the table:

\begin{verbatim}
CREATE TABLE local_objects AS 
SELECT * FROM all_objects 
/
/*Query the record count*/
SELECT COUNT(*)
FROM local_objects 
/
COUNT(*)
---------
73673
\end{verbatim}

The following PL/SQL block opens a cursor, fetches row by row, and counts the number of procedures that can be accessed by the \texttt{SCOTT} user:

\begin{verbatim}
SET SERVEROUTPUT ON
/*Start the PL/SQL block*/
DECLARE

/*Local PL/SQL variables*/
obj_id   local_objects.object_id%TYPE;
obj_type local_objects.object_type%TYPE;
obj_name local_objects.object_name%TYPE;

counter   NUMBER;
clock_in   NUMBER;
clock_out  NUMBER;

/*Cursor to fetch the object details*/
CURSOR c IS
SELECT object_id, object_type, object_name
\end{verbatim}
FROM local_objects;
BEGIN

/*Capture the start time*/
clock_in := DBMS_UTILITY.GET_TIME();
OPEN c;

LOOP
    FETCH c INTO obj_id, obj_type, obj_name;
    EXIT WHEN c%NOTFOUND;
    /*Count the number of procedures in the test table*/
    IF obj_type = 'PROCEDURE' THEN
        counter := counter+1;
    END IF;

END LOOP;
CLOSE c;

/*Capture the end time*/
clock_out := DBMS_UTILITY.GET_TIME();
DBMS_OUTPUT.PUT_LINE ('Time taken in row fetch:'||to_char (clock_out-clock_in));

END;
/

Time taken in row fetch:369

PL/SQL procedure successfully completed.

The row-by-row fetch took 369 hsec to get executed. There were 73673 context switches made between the PL/SQL and SQL engines. Now, let's apply the bulk fetch techniques to the preceding block, and check the performance gains:

SET SERVEROUTPUT ON
/*Start the PL/SQL block*/
DECLARE

/*Declare the local record and table collection*/
TYPE obj_rec IS RECORD
    (obj_id local_objects.object_id%TYPE,
     obj_type local_objects.object_TYPE%TYPE,
     obj_name local_objects.object_name%TYPE);

    TYPE obj_tab IS TABLE OF obj_rec;
    t_all_objs obj_tab;
Tuning the PL/SQL Code

```
BEGIN

/*Capture the start time*/
clock_in := DBMS_UTILITY.GET_TIME();

/*Select query to bulk fetch multi-record set in nested table collection*/
SELECT object_id, object_TYPE, object_name
BULK COLLECT INTO t_all_objs
FROM local_objects;

/*Loop through the collection to count number of procedures*/
FOR I IN t_all_objs.FIRST..t_all_objs.LAST
LOOP
  IF (t_all_objs(i).obj_type = 'PROCEDURE') THEN
    counter := counter+1;
  END IF;
END LOOP;

/*Capture the end time*/
clock_out := DBMS_UTILITY.GET_TIME();
DBMS_OUTPUT.PUT_LINE ('Time taken in bulk fetch:'||to_char (clock_out-clock_in));
END;
/

Time taken in bulk fetch:35

PL/SQL procedure successfully completed.
```

Well, the bulk SQL is around 10 times faster than the usual fetch operation. The reason is that there was only one context switch made between the SQL and PL/SQL engines. Note that the nested table collection variable is not required to be initialized for bulk operations.

The BULK COLLECT operation does not raise the NO_DATA_FOUND exception.
Bulk processing is a CPU-intensive operation, and fetching into a collection consumes your session memory. If the bulk collect fetches a large number of rows and the session memory is not large enough, you might experience a hung session or an abnormal termination. To resolve such cases, Oracle provides the LIMIT clause to control the number of records retrieved in a single fetch operation. The following PL/SQL block uses the LIMIT clause to control the number of records fetched during the FETCH operation:

```plsql
SET SERVEROUTPUT ON
/*Start the PL/SQL block */
DECLARE

/*Local block variables - object type record and nested table*/
TYPE obj_rec IS RECORD
(obj_id local_objects.object_id%TYPE,
 obj_type local_objects.object_TYPE%TYPE,
 obj_name local_objects.object_name%TYPE);

TYPE obj_tab is table of obj_rec;
t_all_objs obj_tab;

counter NUMBER;
p_rec_limit NUMBER := 100;
clock_in NUMBER;
clock_out NUMBER;

/*Cursor to fetch object details from LOCAL_OBJECTS*/
CURSOR C IS
SELECT object_id, object_TYPE, object_name
FROM local_objects;
BEGIN

/*Capture the start time*/
clock_in := DBMS_UTILITY.GET_TIME();
/*Open the cursor*/
OPEN c;
LOOP

/*Bulk fetch the records by specifying the limit*/
FETCH c BULK COLLECT INTO t_all_objs LIMIT p_rec_limit;
EXIT WHEN c%NOTFOUND;
/*FOR loop to count the procedures */
FOR I IN t_all_objs.FIRST..t_all_objs.LAST
LOOP
```

[261]
Tuning the PL/SQL Code

```sql
IF (t_all_objs(i).obj_type = 'PROCEDURE') THEN
    counter := counter+1;
END IF;
END LOOP;
t_all_objs.DELETE;
END LOOP;
CLOSE c;

clock_out := DBMS_UTILITY.GET_TIME();

DBMS_OUTPUT.PUT_LINE ('Time taken in controlled fetch:'||to_char(clock_out-clock_in));

END;
/

Time taken in controlled fetch:94

PL/SQL procedure successfully completed.

A controlled bulk operation, limited to 100 records per bulk fetch, took 94 hsec to get executed, which is still 4 times better than the row-by-row fetch operation. This time the program makes 737 context switches between the PL/SQL and SQL engines.

FORALL

More than iterative, FORALL is a declarative statement. The DML statements specified in FORALL are generated once, but they send in bulk to the SQL engine for processing, thus making only a single context switch. There can be only one DML statement in FORALL that can be INSERT, UPDATE, or DELETE, or the FORALL statement can be used, as per the following syntax:

```sql
FORALL index IN
[
    lower_bound...upper_bound |
    INDICES OF indexing_collection |
    VALUES OF indexing_collection
]
[SAVE EXCEPTIONS]
[DML statement]
```

If the DML statements contain bind variables, the bulk SQL binds all the statements in a single step. This is known as bulk binding.
Let's set up the environment for our performance test. The performance test will compare the execution time of a FOR loop versus a FORALL statement:

```sql
/*Create table T1 with basic object columns*/
CREATE TABLE T1
AS
SELECT object_id, object_type, object_name
FROM all_objects
WHERE 1=2
/

/*Create table T2 with basic object columns*/
CREATE TABLE T2
AS
SELECT object_id, object_type, object_name
FROM all_objects
WHERE 1=2
/
```

The following PL/SQL block bulk collects the data from the LOCAL_OBJECTS table, creates an interim collection (associative array) for only the synonym information, and inserts it into two different tables. For verification, we will query the records inserted in both the tables:

```sql
SET SERVEROUTPUT ON
/*Start the PL/SQL block*/
DECLARE

/*Local block variables - object type record and nested table collection*/
TYPE obj_rec IS RECORD
(obj_id local_objects.object_id%TYPE,
obj_type local_objects.object_TYPE%TYPE,
obj_name local_objects.object_name%TYPE);

TYPE obj_tab IS TABLE OF obj_rec;
TYPE syn_tab IS TABLE OF obj_rec index by pls_integer;
t_all_objs obj_tab;
t_all_syn syn_tab;

counter NUMBER := 1;
clock_in NUMBER;
clock_out NUMBER;
BEGIN
```
Tuning the PL/SQL Code

/*BULK COLLECT the data from LOCAL_OBJECTS in nested table
collection*/
SELECT object_id, object_TYPE, object_name
BULK COLLECT INTO t_all_objs
FROM local_objects;
FOR I IN t_all_objs.FIRST..t_all_objs.LAST
LOOP
  /*Capture all synonyms in another collection*/
  IF (t_all_objs(i).obj_type = 'SYNONYM') THEN
    t_all_syn(t_all_syn.count+1) := t_all_objs(i);
  END IF;
END LOOP;

clock_in := DBMS_UTILITY.GET_TIME();

/*FOR loop to insert the data in table T1*/
FOR I IN t_all_syn.first..t_all_syn.last
LOOP
  INSERT INTO t1 VALUES t_all_syn(i);
END LOOP;

clock_out := DBMS_UTILITY.GET_TIME();

DBMS_OUTPUT.PUT_LINE ('Time taken by FOR loop:'||to_char (clock_out-clock_in));

clock_in := DBMS_UTILITY.GET_TIME();

/*FORALL statement to insert the data in table T2*/
FORALL I IN t_all_syn.first..t_all_syn.last
  INSERT INTO t2 VALUES t_all_syn(i);

clock_out := DBMS_UTILITY.GET_TIME();

DBMS_OUTPUT.PUT_LINE ('Time taken by FORALL:'||to_char (clock_out-clock_in));
END;
/

Time taken by FOR loop:1637
Time taken by FORALL:29

PL/SQL procedure successfully completed.
This is phenomenal! The FORALL statement inserted more than 37000 records in less than a second, whereas the FOR LOOP statement took 16 sec to insert the same number of records. The reason is again the context switch between the processing engines; one switch against 37000 context switches.

In the case of a sparse collection, use INDICES OF or VALUES OF in the FORALL statement. It will use only those indexes of the collection that hold a value. The following PL/SQL block uses the FORALL statement to insert a sparse collection into the T2 table. To sparse the interim collection, the synonyms starting with SYS% are deleted from it:

```sql
/*Truncate table T2 for the current test*/
TRUNCATE TABLE t2
/
SET SERVEROUTPUT ON

/*Start the PL/SQL block*/
DECLARE

/*Local block variables - object type record and nested table collection*/
TYPE obj_rec IS RECORD
  (obj_id local_objects.object_id%TYPE,
   obj_type local_objects.object_TYPE%TYPE,
   obj_name local_objects.object_name%TYPE);

TYPE obj_tab IS TABLE OF obj_rec;
TYPE syn_tab IS TABLE OF obj_rec index by pls_integer;

t_all_objs obj_tab;
```
Tuning the PL/SQL Code

t_all_syn syn_tab;
counter NUMBER := 1;
clock_in NUMBER;
clock_out NUMBER;
BEGIN

/*BULK COLLECT the data from LOCAL_OBJECTS in nested table collection*/
SELECT object_id, object_TYPE, object_name
BULK COLLECT INTO t_all_objs
FROM local_objects;

/*FOR loop to collect all synonyms in nested table collection*/
FOR I IN t_all_objs.FIRST..t_all_objs.LAST
LOOP
  IF (t_all_objs(i).obj_type = 'SYNONYM') THEN
    t_all_syn(t_all_syn.count+1) := t_all_objs(i);
  END IF;
END LOOP;

/*Delete all the synonyms whose names starts with SYS%*/
FOR I in 1..t_all_syn.count
LOOP
  IF t_all_syn(i).obj_name like 'SYS%' THEN
    t_all_syn.delete (i);
  END if;
END LOOP;

clock_in := DBMS_UTILITY.GET_TIME();

/*Insert the sparse collection using INDICES of clause*/
FORALL I IN INDICES OF t_all_syn
  INSERT INTO t2 VALUES t_all_syn(i);

clock_out := DBMS_UTILITY.GET_TIME();

DBMS_OUTPUT.PUT_LINE ('Time taken by FORALL:' ||to_char (clock_out-clock_in));
END;
/
Time taken by FORALL:30

PL/SQL procedure successfully completed.

/*Query the record count in table T2*/
SQL> select count(*) from t2;

COUNT(*)
----------
37025

There were six synonyms starting with SYS% that were not inserted in this run.
If you hadn't used the INDICES OF clause, the block would have terminated with the

FORALL and exception handling

FORALL is specifically designed for bulk transactions. Any faulty transaction in the
bulk operation may cause the entire bulk operation to abort with an exception,
thereby rolling back all the changes made by the earlier transactions of the same
DML statement. Oracle gracefully deals with such scenarios by saving the erroneous
record and exception information separately and by continuing the DML execution.
There are two ways to handle an exception in the FORALL statement:

1. Abort the FORALL execution but commit the changes made by the earlier
transactions: Traditional exception handling but with a COMMIT in the
exception handler. For example, the following exception handle will commit
the transactions that have been executed already:

   EXCEPTION
   WHEN OTHERS THEN
       DBMS_OUTPUT.PUT_LINE (SQLERRM);
       COMMIT;
       RAISE;

2. Continue the FORALL execution and save the failed transactions: Use SAVE
exceptions with the FORALL statement. The feature is known as bulk
exception handling. With SAVE EXCEPTION, Oracle stores the faulty DML
details in the bulk exception logger known as SQL%BULK_EXCEPTIONS. After
the FORALL execution is over, database administrators can look into the
exception log and troubleshoot the defective records. The defective records
are skipped and logged under the SQL%BULK_EXCEPTIONS pseudocolumn.
For example, if FORALL generated 5000 update statements, out of which 13
were failed transactions, 4987 records will still be updated. The 13 defective
transactions will be logged in the SQL%BULK_EXCEPTIONS array structure
with the cursor index.
Tuning the PL/SQL Code

The \%BULK_EXCEPTIONS attribute maintains two fields—\%ERROR_INDEX and \%ERROR_CODE. \%ERROR_INDEX stores the defect record index where the exception was raised, while \%ERROR_CODE records the exception message. \%BULK_EXCEPTIONS.COUNT stores the count of exceptions raised during the execution of the \texttt{FORALL} statement. Note that the standard error code captured by the \%ERROR_CODE attribute is not prefixed with the negative (-) sign. Therefore, in order to fetch its equivalent error message, pass the error code multiplied by -1 to the \texttt{SQLERRM} function.

In the last illustration, we explicitly removed the values starting with \texttt{SYS\%}. For the current demonstration, we will add a check constraint on the \texttt{OBJECT\_NAME} column of the \texttt{T2} table. The following script adds the check constraint:

```sql
/*Truncate table T2 for current test*/
TRUNCATE TABLE t2
/

/*Add check constraint on OBJECT\_NAME*/
ALTER TABLE t2
  ADD CONSTRAINT t2_obj_name CHECK (object_name NOT LIKE 'SYS\%')
/
```

Now, we will run the following PL/SQL block to insert the dense collection into the \texttt{T2} table. As \texttt{SAVE EXCEPTIONS} has been specified along with the \texttt{FORALL} statement, if there are any exceptions while loading, they will be saved in the bulk exception log:

```sql
SET SERVEROUTPUT ON

/*Start the PL/SQL block*/
DECLARE

/*Local block variables - object type record and nested table collection*/
  TYPE obj_rec IS RECORD
   (obj_id local_objects.object_id%TYPE,
    obj_type local_objects.object_TYPE%TYPE,
    obj_name local_objects.object_name%TYPE);
  TYPE obj_tab IS TABLE OF obj_rec;
  TYPE syn_tab IS TABLE OF obj_rec index by pls_integer;
  t_all_objs obj_tab;
  t_all_syn syn_tab ;

  counter NUMBER := 1;
  clock_in NUMBER;
  clock_out NUMBER;
```

---

[268]
/*Declare user defined exception for error number -24381*/
bulk_errors EXCEPTION;
PRAGMA EXCEPTION_INIT(bulk_errors, -24381);
BEGIN

/*Bulk collect object details in nested table collection variable*/
SELECT object_id, object_TYPE, object_name
BULK COLLECT INTO t_all_objs
FROM local_objects;

/*Filter all synonyms and capture in another collection*/
FOR I IN t_all_objs.FIRST..t_all_objs.LAST
LOOP
    IF (t_all_objs(i).obj_type = 'SYNONYM') THEN
        t_all_syn(t_all_syn.count+1) := t_all_objs(i);
    END IF;
END LOOP;

/*FORALL statement to insert the records in table T2*/
FORALL I IN 1..t_all_syn.count

/*Save faulty records in bulk collection cursor attributes*/
SAVE EXCEPTIONS
    insert into t2 values t_all_syn(i);
EXCEPTION

/*Exception handler to query the failed transactions*/
WHEN BULK_ERRORS THEN
    FOR J IN 1..SQL%BULK_EXCEPTIONS.COUNT
    LOOP
        DBMS_OUTPUT.PUT_LINE('----------------------------------------');
        DBMS_OUTPUT.PUT_LINE('Error in INSERT:'||SQL%BULK_EXCEPTIONS(J).ERROR_INDEX);
        DBMS_OUTPUT.PUT_LINE('Error Message is: '||sqlerrm('-'||SQL%BULK_EXCEPTIONS(J).ERROR_CODE));
    END LOOP;
END;
/

/*Declare user defined exception for error number -24381*/
bulk_errors EXCEPTION;
PRAGMA EXCEPTION_INIT(bulk_errors, -24381);
BEGIN

/*Bulk collect object details in nested table collection variable*/
SELECT object_id, object_TYPE, object_name
BULK COLLECT INTO t_all_objs
FROM local_objects;

/*Filter all synonyms and capture in another collection*/
FOR I IN t_all_objs.FIRST..t_all_objs.LAST
LOOP
    IF (t_all_objs(i).obj_type = 'SYNONYM') THEN
        t_all_syn(t_all_syn.count+1) := t_all_objs(i);
    END IF;
END LOOP;

/*FORALL statement to insert the records in table T2*/
FORALL I IN 1..t_all_syn.count

/*Save faulty records in bulk collection cursor attributes*/
SAVE EXCEPTIONS
    insert into t2 values t_all_syn(i);
EXCEPTION

/*Exception handler to query the failed transactions*/
WHEN BULK_ERRORS THEN
    FOR J IN 1..SQL%BULK_EXCEPTIONS.COUNT
    LOOP
        DBMS_OUTPUT.PUT_LINE('----------------------------------------');
        DBMS_OUTPUT.PUT_LINE('Error in INSERT:'||SQL%BULK_EXCEPTIONS(J).ERROR_INDEX);
        DBMS_OUTPUT.PUT_LINE('Error Message is: '||sqlerrm('-'||SQL%BULK_EXCEPTIONS(J).ERROR_CODE));
    END LOOP;
END;
/

/*Declare user defined exception for error number -24381*/
bulk_errors EXCEPTION;
PRAGMA EXCEPTION_INIT(bulk_errors, -24381);
BEGIN

/*Bulk collect object details in nested table collection variable*/
SELECT object_id, object_TYPE, object_name
BULK COLLECT INTO t_all_objs
FROM local_objects;

/*Filter all synonyms and capture in another collection*/
FOR I IN t_all_objs.FIRST..t_all_objs.LAST
LOOP
    IF (t_all_objs(i).obj_type = 'SYNONYM') THEN
        t_all_syn(t_all_syn.count+1) := t_all_objs(i);
    END IF;
END LOOP;

/*FORALL statement to insert the records in table T2*/
FORALL I IN 1..t_all_syn.count

/*Save faulty records in bulk collection cursor attributes*/
SAVE EXCEPTIONS
    insert into t2 values t_all_syn(i);
EXCEPTION

/*Exception handler to query the failed transactions*/
WHEN BULK_ERRORS THEN
    FOR J IN 1..SQL%BULK_EXCEPTIONS.COUNT
    LOOP
        DBMS_OUTPUT.PUT_LINE('----------------------------------------');
        DBMS_OUTPUT.PUT_LINE('Error in INSERT:'||SQL%BULK_EXCEPTIONS(J).ERROR_INDEX);
        DBMS_OUTPUT.PUT_LINE('Error Message is: '||sqlerrm('-'||SQL%BULK_EXCEPTIONS(J).ERROR_CODE));
    END LOOP;
END;
/
Error in INSERT:3
Error Message is: ORA-02290: check constraint (.) violated

Error in INSERT:2089
Error Message is: ORA-02290: check constraint (.) violated

Error in INSERT:2091
Error Message is: ORA-02290: check constraint (.) violated

Error in INSERT:2093
Error Message is: ORA-02290: check constraint (.) violated

Error in INSERT:2094
Error Message is: ORA-02290: check constraint (.) violated

Error in INSERT:3545
Error Message is: ORA-02290: check constraint (.) violated

PL/SQL procedure successfully completed.

 SQL> select count(*) from t2;

 COUNT(*)
 ----------
   37025

The transactions that failed to get executed due to the check constraint violation were saved in the BULK_EXCEPTIONS cursor attribute. Besides these six records, all the other data was inserted, and this can be confirmed by a SELECT query on the table.

Summary

In this chapter, we have discussed the features that can tune your PL/SQL code to run faster. We have also seen how code compilation and optimization can speed up the performance of the PL/SQL programs. In the later half, we discussed the PL/SQL tuning features in detail with the help of demonstrations.

In the next chapter, we will focus on one of the major new features introduced in Oracle Database 11g. The feature is known as result caching that is primarily built to accelerate the performance of queries that are repeatedly executed. We will discuss the different flavors of result caching in the next chapter.
Practice exercise

- Identify the nature of the program that is best suited for the interpreted mode of compilation.
  1. The program unit contains multiple SQL statements.
  2. The program unit has just been developed and is in the debug stage.
  3. The program unit uses collections and bulk bind statements.
  4. The program unit is in the production phase.

- Choose the correct statements about the real native compilation mode in Oracle 11g.
  1. The compilation method uses the C compiler to convert the program into an equivalent C code.
  2. The compilation method mounts the shared libraries through the PLSQL_NATIVE_LIBRARY_DIR and PLSQL_NATIVE_LIBRARY_SUBDIR_COUNT parameters.
  3. The compilation does not use the C compiler but converts the program unit directly to the M code.
  4. The real native compilation is supported for RAC environments and participates in the backup recovery processes.

- Determine the behavior of the PLSQL_OPTIMIZE_LEVEL optimizer when it has been set to 3.
  1. The optimizer would inline the programs that are necessary.
  2. The optimizer would inline all the programs irrespective of the gains.
  3. The optimizer would inline only those subprograms which have PRAGMA INLINE.
  4. The setting has no effect on the inlining of the subprograms.

- Choose the correct statements about the compilation setting in Oracle.
  1. From Oracle 11g, the default value of PLSQL_CODE_TYPE is NATIVE.
  2. An object can be recompiled in a compilation mode that is different from the current database setting.
  3. During the database upgrade, PLSQL_CODE_TYPE must be modified in the pfile instance.
  4. In real native compilation, the libraries generated are stored in a secured file system.
Tuning the PL/SQL Code

- Identify the tuning tips in the following PL/SQL block:

```sql
DECLARE
  CURSOR C IS
    SELECT ENAME, SAL, COMM
    FROM EMPLOYEES;
  L_COMM NUMBER;
BEGIN
  FOR I IN C
  LOOP
    L_COMM := I.SAL + ((I.COMM/100) * (I.SAL * 12));
    DBMS_OUTPUT.PUT_LINE (I.ENAME||' earns '||L_COMM||' as commission');
  END LOOP;
END;
/
```

1. Use `BULK COLLECT` to select the employee data.
2. Declare `L_COMM` as `NOT NULL`.
3. Use `PLS_INTEGER` for `L_COMM`.
4. No tuning is required.

- Which of these statements are true about inlining in PL/SQL subprograms?
  1. The optimizer can inline only the standalone stored functions.
  2. The optimizer can inline only the locally declared subprograms.
  3. Inlining is always useful in the performance irrespective of the size of the subprogram.
  4. The optimizer cannot identify any subprogram for inlining when the optimizer level is set at 0.

- Examine the following code and determine the output:

```sql
DECLAR
  FUNCTION F_ADD (P_NUM NUMBER) RETURN NUMBER IS
BEGIN
  RETURN P_NUM + 10;
END;
BEGIN
 FOR I IN 122..382 LOOP
 PRAGMA INLINE (F_ADD,'YES');
```

1. Use `BULK COLLECT` to select the employee data.
2. Declare `L_COMM` as `NOT NULL`.
3. Use `PLS_INTEGER` for `L_COMM`.
4. No tuning is required.

- Which of these statements are true about inlining in PL/SQL subprograms?
  1. The optimizer can inline only the standalone stored functions.
  2. The optimizer can inline only the locally declared subprograms.
  3. Inlining is always useful in the performance irrespective of the size of the subprogram.
  4. The optimizer cannot identify any subprogram for inlining when the optimizer level is set at 0.

- Examine the following code and determine the output:

```sql
DECLAR
  FUNCTION F_ADD (P_NUM NUMBER) RETURN NUMBER IS
BEGIN
  RETURN P_NUM + 10;
END;
BEGIN
 FOR I IN 122..382 LOOP
 PRAGMA INLINE (F_ADD,'YES');
```
L_SUM := L_SUM + F_ADD (I);
END LOOP;
END;
/

1. **PLSQL_OPTIMIZE_LEVEL** is set to 2.
2. The **F_ADD** local function would not be called inline unless **PLSQL_OPTIMIZE_LEVEL** is set to 3.
3. The **F_ADD** local function may be called inline because **PLSQL_OPTIMIZE_LEVEL** is set to 2.
4. The **F_ADD** local function would be called inline because **PRAGMA INLINE** marks it for inline.
5. Inlining cannot be done for locally declared subprograms.

- The libraries generated from real native compilation are stored in the **SYSAUX** tablespace.
  1. True
  2. False

- Suggest the tuning considerations in the following PL/SQL block:
  DECLARE
  L_SUM NATURALN := 0;
  L_ID VARCHAR2(10);
BEGIN
  L_ID := 256;
  L_SUM := L_ID * 1.5;
END;

  1. The datatype of **L_SUM** can be changed to **NATURAL** and nullability can be verified in the executable section.
  2. **L_SUM** must not be initialized with zero.
  3. The 1.5 multiple must be assigned to a variable.
  4. **L_ID** must be of an appropriate datatype such as **NUMBER** or **PLS_INTEGER**.
Tuning the PL/SQL Code

- Identify the correct statements about PRAGMA INLINE.
  1. It is the fifth pragma in Oracle besides AUTONOMOUS_TRANSACTION, EXCEPTION_INIT, RESTRICT_REFERENCE, and SERIALLY_REUSABLE.
  2. It does not work for overloaded functions.
  3. It does not work for PLSQL_OPTIMIZE_LEVEL = 1.
  4. PRAGMA INLINE (<Function name>, 'YES') is meaningless at PLSQL_OPTIMIZE_LEVEL = 3, because the optimizer inlines all the subprograms.
Where to buy this book


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

Click here for ordering and shipping details.