



A stylized world map in shades of gray, centered on the Atlantic Ocean, serving as a background for the text.

# Tushar Gadhia

---

Senior Product Manager  
Server Technologies Division  
Oracle Corporation



# Techniques for Developing Faster PL/SQL Applications

---



# Agenda

---

- **Reasons for Slow Execution Speed**
- **Identifying Problem Areas**
- **Correcting Performance Problems**
- **Future PL/SQL Performance Projects**
- **Conclusion**



## Introduction

PL/SQL is an easy-to-use, high-performance procedural language extension of SQL. Many Fortune 500 companies have successfully used the PL/SQL platform to build mission-critical applications. In one of the customer surveys done in 1998, it was voted as the most important and the most satisfying feature from Oracle. It was also voted as being very important for businesses in the next several years.

It is important to build applications that perform the necessary processing as fast and efficiently as possible.

As with any programming language, there are a wide variety of reasons why a PL/SQL application might not be as fast as one might hope. This presentation describes several reasons for poor execution speed, a basic but powerful tool to help identify problem areas, some corrective actions which might be appropriate, and some guidance regarding possible future PL/SQL projects which might influence application tuning and design.

---

# Reasons for Slow Execution Speed



# Reasons for Slow Execution Speed

---

- **Poorly written SQL**
- **Poor design**
- **Poor programming**
- **Ignorance of PL/SQL fundamentals**
- **Non-tuned shared pool memory**
- **Non-awareness of undocumented PL/SQL behavior**



## **Reasons for Slow Execution Speed**

It is important that your PL/SQL application does what it is expected to do. It is equally important to make sure that it achieves the performance targets for response time and throughput.

Slow execution speed is often the result of poorly designed or written SQL code, poor programming style, ignorance of PL/SQL fundamentals, shared pool memory that is not properly tuned, and a lack of understanding of undocumented PL/SQL behavior.

# Poorly Written SQL

- Reason for slow PL/SQL code is often badly written SQL
- MYTH: PL/SQL consumes a LOT of time!!
- Action:
  - Analyze SQL performance - determine execution plan:
    - EXPLAIN PLAN
    - TKPROF
    - Oracle Trace
  - Redesign SQL statements
  - Send proper Hints to the query optimizer



## Poorly Written SQL

In our experience, the most important reason for slow PL/SQL code is badly written SQL. PL/SQL programs tend to be relatively simple, with a significant amount of the complexity captured by the SQL statements; no amount of poorly written PL/SQL is going to have a noticeable impact on total performance if the program contains a significant number of properly tuned SQL statements.

A lot of the tuning can be focused on optimizing the way your PL/SQL applications manipulate data in the database through SQL statements. There are several steps you can take to improve the performance of the SQL code in your PL/SQL applications.

Once SQL statements have been identified as the bottleneck of your PL/SQL program, take steps to analyze SQL performance and pinpoint the problem areas. Determine the execution plan of SQL statements. Some of the methods available to do this are:

EXPLAIN PLAN statement

TKPROF utility

Oracle Trace facility

**Note:** For more information on these methods, refer to Oracle documentation.

Next, redesign the SQL statements that are the cause of slow execution. Sending proper hints to the query optimizer will also help in eliminating common problems such as unnecessary full table scans or lack of important indices.

# Poor Design/Programming

- **Poorly written algorithm**
  - Wrong choice of sort, search, or CPU-intensive function
- **Poor programming**
  - Declaring variables that are not used
  - Initializations/computations which could be pulled out of loops
  - Passing unneeded parameters to functions



## Poor Design/Programming

Poor design and/or programming is often the result of schedule crunches or other outside constraints. Under these circumstances, even the most experienced and knowledgeable programmers are often likely to write code that can result in poor performance.

### Poorly Written Algorithm

No matter how well-tuned a computer language might be for any given task, a slow or badly written algorithm will dominate the performance. A poorly chosen or implemented sort, search, or other common oft-repeated or CPU-intensive function can completely ruin an application's performance. Suppose that the most commonly used function in an application is a lookup function with hundreds of possible targets. If this lookup function can be written as a hash or a binary search, but is instead written as a linear search, the performance of the entire application will suffer greatly. This sort of problem is much more common than one might think.

### Poor Programming

The usual items which a language optimizer might solve can cause a significant performance problem in PL/SQL because there is not yet an optimizer phase in the PL/SQL compiler. Some examples of this sort of problem include declaration of variables which are not used, initializations or computations which could be pulled out of the loops, passing unneeded parameters to functions, and so on. Usually, an optimizer can be relied upon to clean up such problems, so sloppiness of this sort is often not a problem and thus can creep into an application. Until PL/SQL builds an optimizer phase into the compiler, this sort of sloppiness may cause significant problems.

# Ignorance of PL/SQL Fundamentals

- **Copy-in, Copy-out parameter passing**
  - Parameter passing and assignment semantics differ between PL/SQL and C
- **Numeric datatypes**
  - **PLS\_INTEGER** is the most efficient datatype for integer operations
    - It is implemented using fast native representations and operations
    - It requires less storage than **INTEGER** and **NUMBER** values



## Ignorance of PL/SQL Fundamentals

### Copy-In, Copy-Out Parameter Passing

Parameter passing and assignments which would result in a single pointer copy in C generally result in a deep copy of the entire data structure in PL/SQL. So, if your C program needed a deep copy and you ported it to PL/SQL without taking PL/SQL semantics into consideration, you might do an extra copy after PL/SQL has already done one for you. The fundamentals of a language can have a significant impact on overall program design.

### Numeric Datatypes

The **NUMBER** data type and several of its subtypes are 22 byte database-format numbers, designed with an emphasis on portability and arbitrary scale and precision rather than performance. Of all the numeric data types available in PL/SQL, only **PLS\_INTEGER** is implemented using fast native representations and operations. It requires less storage than **INTEGER** or **NUMBER** values. **PLS\_INTEGER** is, therefore, the most efficient datatype for integer operations. If the huge precision of the database number is not required, using **NUMBER** or **INTEGER** for integer operations is slower than using **PLS\_INTEGER**.

# Ignorance of PL/SQL Fundamentals

- **Implicit conversion**
  - PL/SQL performs implicit conversion between structurally different datatypes
  - Common with NUMBER data types

```
DECLARE
    n NUMBER;
BEGIN
    n := n + 15;      -- converted
    n := n + 15.0;  -- not converted
    ...
END;
```



## Ignorance of PL/SQL Fundamentals (continued)

### Implicit Conversion

PL/SQL does implicit conversions between structurally different types at runtime. A common case where implicit conversions result in a performance penalty, but can be avoided, is with numeric types. For instance, assigning a PLS\_INTEGER variable to a NUMBER variable or vice-versa results in a conversion, since their representations are different. Such implicit conversions can happen during parameter passing as well.

In the above example, the integer literal 15 is represented internally as a signed 4-byte quantity, so PL/SQL must convert it to an Oracle number before the addition. However, the floating-point literal 15.0 is represented as a 22-byte Oracle number, so no conversion is necessary.

# Ignorance of PL/SQL Fundamentals

- **Constraint checks**
  - **NOT NULL constraint incurs overhead**

```
PROCEDURE calc_m IS
  m NUMBER NOT NULL:=0;
  a NUMBER;
  b NUMBER;
BEGIN
  ...
  m := a + b;
  ...
END;
```

```
PROCEDURE calc_m IS
  m NUMBER; --no constraint
  a NUMBER;
  b NUMBER;
BEGIN
  ...
  m := a + b;
  IF m IS NULL THEN
    -- raise error
  END IF;
END;
```

## Ignorance of PL/SQL Fundamentals (continued)

### Constraint Checks

Several of the PL/SQL numeric data types are constrained. PL/SQL allows any variable to be constrained to only NOT NULL values. Constraint checks, including the NOT NULL constraint, require an extra byte code instruction on every assignment. The NOT NULL constraint, therefore, does not come for free; it requires the PL/SQL compiler to ensure NOT NULLness after every operation. Constrained data types used in PL/SQL blocks which have no associated exception handler are excellent candidates for replacement by corresponding unconstrained types.

Consider the example on the left in the slide, which uses the NOT NULL constraint for *m*.

Because *m* is constrained by NOT NULL, the value of the expression *a + b* is assigned to a temporary variable, which is then tested for nullity. If the variable is not null, its value is assigned to *m*. Otherwise, an exception is raised. However, if *m* were not constrained, the value would be assigned to *m* directly.

A more efficient way to write the same example is shown on the right in the slide.

# Ignorance of PL/SQL Fundamentals

- **Conditional control statements**

- In logical expressions, PL/SQL stops evaluating the expression as soon as the result is determined

- **Scenario 1**

```
IF (sal < 1500) OR (comm IS NULL) THEN
    ...
END IF;
```

- **Scenario 2**

```
IF credit_ok(cust_id) AND (loan < 5000) THEN
    ...
END IF;
```



## Ignorance of PL/SQL Fundamentals (continued)

### Conditional Control Statements

When evaluating a logical expression, PL/SQL stops evaluating the expression as soon as the result can be determined. Performance can be improved by tuning conditional constructs carefully.

#### Example 1:

```
IF (sal < 1500) OR (comm IS NULL) THEN
    ...
END IF;
```

In this example, when the value of *sal* is less than 1,500 the left operand yields TRUE, so PL/SQL need not evaluate the right operand (because OR returns TRUE if either of its operands is true).

#### Example 2:

```
IF credit_ok(cust_id) AND (loan < 5000) THEN
    ...
END IF;
```

In this example, the Boolean function CREDIT\_OK is always called. However, if you switch the operands of AND as follows, the function is called only when the expression *loan < 5000* is true (because AND returns TRUE only if both its operands are true):

```
IF (loan < 5000) AND credit_ok(cust_id) THEN
    ...
END IF;
```

# Ignorance of PL/SQL Fundamentals

- **Behavior of NULLs**

- Comparisons involving NULLs always yield NULL

– A  
yi

– T

– In  
yi

```
...  
x := 5;  
y := NULL;  
...  
IF x != y THEN          -- yields NULL, not TRUE  
    sequence_of_statements; -- not executed  
END IF;
```

statements is not executed

- **Useful built-in functionality**

- Several highly-optimized built-in functions available in PL/SQL



## Ignorance of PL/SQL Fundamentals (continued)

### Behavior of NULLs

When working with NULLs, you can avoid some common mistakes by keeping in mind the rules above.

#### Example:

```
...  
x := 5;  
y := NULL;  
...  
IF x != y THEN          -- Yields NULL, not TRUE  
    sequence_of_statements; -- not executed  
END IF;
```

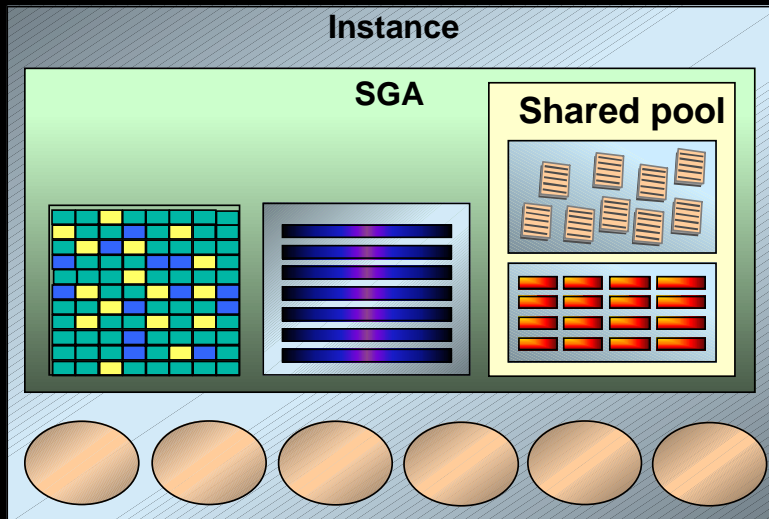
In this example, you might expect the sequence of statements to execute because  $x$  and  $y$  seem unequal. But, NULLs are indeterminate. Whether or not  $x$  is equal to  $y$  is unknown. Therefore, the IF condition yields NULL, and the sequence of statements is bypassed.

### Useful Built-in Functionality

There are several powerful, highly-optimized built-in functions which should be used when possible rather than attempting to recreate them in PL/SQL: REPLACE, TRANSLATE, SUBSTR, INSTR, RPAD, LTRIM, and so on. Rather than simulating these built-in functions with hand-coded PL/SQL procedures, it is often much better to use the built-in function. Even in a case where the built-in function appears to be overkill, it is often better to use it than to code the subset of functionality explicitly.

# Non-tuned Shared Pool Memory

- Frequently used packages that get flushed out from memory often decrease performance



## Non-tuned Shared Pool Memory

When you reference a program element, such as a procedure or a package, its compiled version is loaded into the shared pool memory area, if it is not already present there. It remains there until the memory is needed by other resources and the package has not been used recently. If it gets flushed out from memory, the next time any object in the package is needed, the whole package has to be loaded in memory again, which involves time and maintenance to make space for it.

If the package is already present in the shared memory area, your code executes faster. It is, therefore, important to make sure that packages that are used very frequently are always present in memory. The larger the shared pool area, the more likely it is that the package remains in memory. However, if the shared pool area is too large, you waste memory. When tuning the shared pool, make sure it is large enough to hold all the frequently needed objects in your application.

## Pinning Packages

Sizing the shared pool properly is one of the ways of ensuring that frequently used objects are available in memory whenever needed, so that performance improves. Another way to improve performance is to pin frequently used packages in the shared pool.

When a package is pinned, it is not aged out with the normal least recently used (LRU) mechanism that Oracle otherwise uses to flush out a least recently used package. The package remains in memory no matter how full the shared pool gets or how frequently you access the package.

You pin packages with the help of the `DBMS_SHARED_POOL` package. For more information on this topic, refer to Oracle documentation.

# Undocumented PL/SQL Behavior

- **VARCHAR2 Datatype**
  - VARCHAR2 variables with constraints up to 2000 are stack allocated
  - VARCHAR2 variables with constraints above 2000 are heap allocated
  - Tune for speed ==> VARCHAR2(2000 or less)
  - Tune for memory ==> (VARCHAR2(2000+))
- Table of Record is slower for two-dimensional representation than Record of Table
- Recursive implementation of common algorithms is less efficient than iterative implementation



## Undocumented PL/SQL Behavior

### VARCHAR2 Datatype

As an optimization, the PL/SQL virtual machine stack allocates some variables based upon their declared size, occasionally wasting memory to avoid the overhead of a heap allocation. For example, VARCHAR2(2000) and below are stack allocated, where larger VARCHAR2 variables are heap allocated as required. If the variable is not used, wasting up to 2000 bytes of memory by optimistically stack allocating the memory could adversely affect performance. On the other hand, if the variable can never contain more than 1000 bytes, declaring it as VARCHAR2(1000) saves a call to heap allocate the memory which would be incurred had the variable been declared as VARCHAR2(30000). The larger than required constraint might be used to avoid changes in case the data size requirements increased in a future revision of the code, but that sort of forethought has a price in performance.

### Table of Records versus Record of Table

A table of records is a much slower representation for two-dimensional data than is record of table.

### Recursive versus Iterative Implementation

Recursive implementations of common algorithms tend to be less efficient than iterative implementations, because of some minor inefficiencies in the way function calls are implemented.

---

# Identifying Problem Areas



# Identifying Problem Areas

- **Application Analysis: Evaluate performance and identify areas that need improvement.**



- **PL/SQL Profiler (in Oracle8i):**
  - **Tool to localize and identify PL/SQL performance problems**
    - **Counts the number of times each line was executed**
    - **Determines how much time was spent on each line**
  - **Information is stored in database tables, and can be accessed at any desired granularity**



## Identifying Problem Areas

To improve the performance of any application, there are two steps involved:

1. Find out why there is a problem. This is the application analysis phase.
2. Fix the problem.

### Application Analysis

The reason the application analysis phase is required is that PL/SQL is being used to develop larger and larger applications. In addition, PL/SQL packages are increasingly being used as components. So, it is no longer easy to identify and isolate performance problems.

### Profiler

Oracle8i PL/SQL provides a new tool called the Profiler which can be used to determine the execution time profile (or runtime behavior) of applications. The Profiler can be used to figure out which part of a particular application is running slowly. Such a tool is crucial in identifying performance bottlenecks. It can help you focus your efforts on improving the performance of only the relevant PL/SQL components, or, even better, the particular program segments where a lot of execution time is being spent.

The Profiler provides functions for gathering “profile” statistics, such as the total number of times each line was executed; time spent executing each line; and minimum and maximum duration spent on execution of a given line of code. For example, developers can generate profiling information for all named library units used in a single session. This information is stored in database tables that can then be later queried.

# Working with the Profiler

- **The Profiler is implemented with DBMS\_PROFILE package. APIs to record information:**
  - **START\_PROFILER(run)**
  - **STOP\_PROFILER**
  - **FLUSH\_DATA**



## Working with the Profiler

Should simple inspection fail to provide sufficient fodder for performance improvement, use of the Profiler is the logical next step. Turn on the Profiler, run your application for long enough to obtain coverage of a good portion of your code, then flush the data to the database.

The Profiler is controlled by the DBMS\_PROFILE package. APIs available to record information are:

- **START\_PROFILER:** In order to get profiling information, you first need to start the Profiler by calling the DBMS\_PROFILER.START\_PROFILER procedure and associating a specific “run” comment with the profile run.
- **STOP\_PROFILER:** To stop the profiling, you must call the DBMS\_PROFILER.STOP\_PROFILER procedure.
- **FLUSH\_DATA:** You may call the DBMS\_PROFILER.FLUSH\_DATA procedure during the session to save incremental data and to free memory for allocated profiler data structures.

This profiling information is stored in database tables that can then be queried to obtain information on time spent in each PL/SQL line, module, and routine.

Third-party vendors can use the profiling API to build graphical, customizable tools. Developers can use Oracle8i’s sample (demo) text-based report writer to gather meaningful data about their applications. You can use the profiling API to analyze the performance of your PL/SQL applications and to locate bottlenecks. You can then use the profile information to appropriately tune your application.

# Identifying Problem Areas - Profiler

<u>Frequ</u> <u>-ency</u>	<u>Time per</u> <u>invocation</u>	<u>Total</u> <u>Time</u>
21,891	.00001187	.25996
4,181	.00001374	.05745
17,710	.00001090	.19310
6,765	.00001446	.09783
10,945	.00007887	.86328

```
function fib(n pls_integer)
  return pls_integer is
begin
  if n = 0 then
    return 0;
  elsif n = 1 then
    return 1;
  else
    return fib(n-1)+
           fib(n-2);
  end if;
end;
```

## Note:

- information recorded in database tables
- simple interface in Oracle8i



## Sample Profile Data

Here is an example that illustrate the usefulness of the profiler.

The numbers here correspond to the information collected by the profiler for this example program. The function fib computes the Fibonnaci numbers; Fib(0) is 0, Fib(1) is 1, Fib(2) is Fib(1)+Fib(0) = 1, and so on.

The profiler traces the execution of the program, computes the time spent at each line and also the time spent within each PL/SQL module and routine.

Profiling data provides questions and not answers... one such question could be: why does "return 1" statement take 3 times more time than "return 0" statement (on the average)?

# Profiling Data

- **Interpreting Profiling Data**
  - Critical code components
  - Critical data structures
- **Actions**
  - Tune SQL or non-SQL code?
  - Algorithmic changes required?
  - Select more appropriate data structure?
  - Utilize any of the new 8.x features?



## Profiling Data

### Interpreting Profiling Data

The next step is to analyze why the time is spent in certain code segments or in accessing certain data structures. Once again, the profiler only raises questions; it does not provide answers.

Find the problem areas by querying the resulting performance data. Concentrate on the packages and procedures which use a significant total amount of time, inspecting them for all of the problems described earlier in this presentation.

The Profiler can be used to identify the critical code components and data structures, but the users need to determine what actions to take.

### Actions

The result of the analysis can be used to rework the algorithms used in the application. For example, due to exponential growth in data, the linear search that was used in the previous version of the application might no longer be the best choice and we might need to change it to use a more appropriate search algorithm.

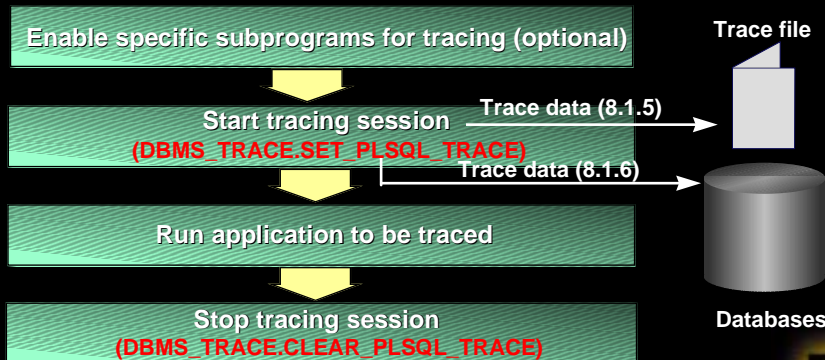
In addition, the analysis phase can point out the inefficiencies caused by the choice of inappropriate data structures and may indicate that some other data structures need to be chosen.

The above analysis must be considered together with the new features that are provided in the PL/SQL 8i. These features can be used to extract more performance from the applications. Several of these features are discussed later in this presentation.

**Note:** Even with the Profiler, one should probably focus on SQL statements, loops, recursive function invocations, assignments, and parameter passing as likely sources of difficulty.

# Tracing PL/SQL Execution

The **DBMS\_TRACE** package provides an API for tracing the execution of PL/SQL programs on the server



## Tracing PL/SQL Execution

In large and complex PL/SQL applications, it can sometimes get difficult to keep track of subprogram calls when a number of them call each other. By tracing your PL/SQL code you can get a clearer idea of the paths and order in which your programs execute.

While a facility to trace your SQL code has been around for a while, Oracle now provides an API for tracing the execution of PL/SQL programs on the server. You can use the Trace API, implemented on the server as the DBMS\_TRACE package, to trace the execution of programs by function or by exception.

### DBMS\_TRACE

DBMS\_TRACE provides subprograms to start and stop PL/SQL tracing in a session. The trace data gets collected as the program executes, and it is written out to the Oracle Server trace file (8.1.5) or database tables (8.1.6).

A typical trace session involves:

- Enabling specific subprograms for trace data collection (optional)

- Starting the PL/SQL tracing session  
(DBMS\_TRACE.SET\_PLSQL\_TRACE)

- Running the application which is to be traced

- Stopping the PL/SQL tracing session  
(DBMS\_TRACE.CLEAR\_PLSQL\_TRACE)

# Controlling the Trace

- In large applications, enable specific subprograms for trace
- Choose tracing levels:
  - Tracing calls:
    - Level 1: Trace all calls
    - Level 2: Trace calls to enabled subprograms only
  - Tracing exceptions:
    - Level 1: Trace all exceptions
    - Level 2: Trace exceptions raised in enabled subprograms only



## Controlling the Trace

Profiling large applications may produce a huge amounts of data which can be difficult to manage. before turning on the trace facility, you have the option to control the volume of data collected by enabling specific subprograms for trace data collection.

During the trace session, you can choose to trace all program calls and exceptions, or enabled program calls and exceptions raised in them. Choose the tracing levels shown above.

**Note:** For more information on this topic, refer to the Oracle product documentation.

---

# Correcting Performance Problems



# Correcting Performance Problems

- Repair Basic Flaws
- Use new Oracle8i performance features:
  - **Bulk Binds** - faster PL/SQL and SQL interaction
  - **NOCOPY** parameter passing hint - faster parameter passing
  - **Native dynamic SQL** - faster execution of dynamic SQL



## Correcting Performance Problems

In general, correcting performance problems is a fairly simple matter, once they have been properly identified.

One option is to identify and correct the various basic flaws that have been discussed in this presentation earlier. Additionally, Oracle8i has introduced several new performance-boosting features that can help your PL/SQL application perform better. These features are:

- Bulk Binds
- NOCOPY parameter paassing hint
- Native dynamic SQL

# Repair Basic Flaws

- Tune SQL statements
- Avoid extra copies
- Do not declare variable you are not going to use
- Pull unnecessary code out of loops
- Do not pass unneeded parameters
- Remove dead code
- Apply sound design/programming techniques
- Use built-in functions whenever possible
- Select appropriate datatypes
- Avoid implicit conversion
- Avoid the NOT NULL constraint whenever possible
- Use PLS\_INTEGER for integer operations
- Tune shared pool memory

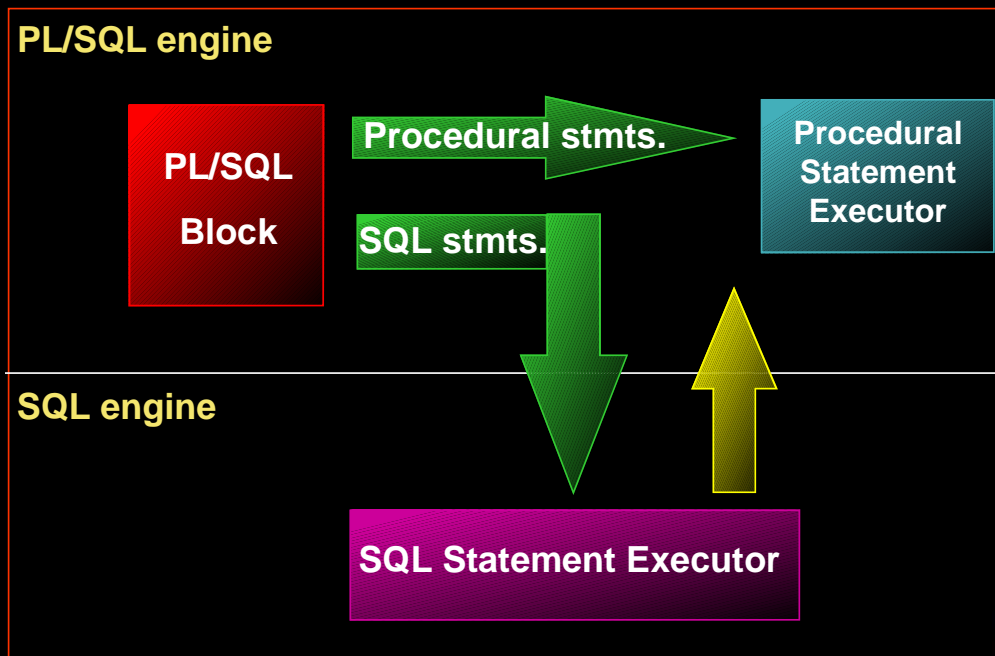


## Correcting Performance Problems

### Repair Basic Flaws

The above is a list of some of the main concepts that have been discussed earlier in this presentation. An awareness of these while developing PL/SQL applications will help you in creating robust, reliable, and efficient applications.

# Binding



## Binding

The Oracle server has three execution engines: one each for PL/SQL, SQL, and Java. The above diagram shows the PL/SQL and SQL engines. When running PL/SQL blocks and subprograms, the PL/SQL engine runs procedural statements but sends the SQL statements to the SQL engine, which parses and executes the SQL statement and, in some cases, returns data to the PL/SQL engine. During execution, every SQL statement causes a *context switch* between the two engines, which results in a performance penalty.

Performance can be improved substantially by minimizing the number of context switches required to run a particular block or subprogram. When a SQL statement runs inside a loop that uses collection elements as bind variables, the large number of context switches required by the block can cause poor performance. As you may already know, collections include nested tables, VARRAYs, index-by tables, and host arrays.

Binding is the assignment of values to PL/SQL variables in SQL statements.

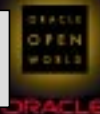
# Bulk Binding

- **Definition:**
  - Bind entire collection/array of values at once, rather than loop to perform fetch, insert, update, and delete on multiple rows
- **Keywords to support bulk binding:**
  - **FORALL** instructs PL/SQL engine to bulk-bind input collections before sending them to the SQL engine

```
FORALL index IN lower_bound..upper_bound  
    sql_statement;
```

- **BULK COLLECT** instructs SQL engine to bulk-bind output collections before returning them to the PL/SQL engine

```
... BULK COLLECT INTO  
    collection_name[,collection_name] ...
```



## Bulk Binding

Bulk binding is binding an entire collection at once rather than iteratively. Without bulk binding, the elements in a collection are sent to the SQL engine individually, whereas bulk binds pass the entire collection back and forth between the two engines.

### Improved Performance

Using bulk binding, you can improve performance by reducing the number of context switches required to run SQL statements that use collection elements. With bulk binding, entire collections, not just individual elements, are passed back and forth.

### Keywords to Support Bulk Binding

Oracle8i PL/SQL supports new keywords to support bulk binding. These are:

#### FORALL

The keyword FORALL instructs the PL/SQL engine to bulk-bind input collections before sending them to the SQL engine. Although the FORALL statement contains an iteration scheme, it is not a FOR loop.

#### BULK COLLECT

The keywords BULK COLLECT instruct the SQL engine to bulk-bind output collections before returning them to the PL/SQL engine. This allows you to bind locations into which SQL can return retrieved values in bulk. Thus you can use these keywords in the SELECT INTO, FETCH INTO, and RETURNING INTO clauses.

# Bulk Binds - Examples

```
FORALL j IN 1..1000
  INSERT INTO Orders VALUES
    (orderId(j),orderDate(j),...);
```

```
SELECT orderDate
BULK COLLECT INTO oldDates
WHERE MONTHS_BETWEEN(sysdate,orderDate) > 9;
```

```
DELETE FROM Orders
WHERE MONTHS_BETWEEN(sysdate,orderDate) > 9
RETURNING orderId
BULK COLLECT INTO deletedIds;
```



ORACLE

## Bulk Binds - Examples

The slide contains three examples:

- The first example shows how to use FORALL with an INSERT statement.
- The second example shows how to use BULK COLLECT INTO with a SELECT statement.
- The third example shows how to use BULK COLLECT INTO with a DELETE statement.

# Bulk Binds

## FORALL ... BULK COLLECT INTO Example

```
FORALL j IN 1..1000
  DELETE Orders
  WHERE orderDate = oldDates(j) -- bulk
     AND orderLoc  = oldLoc      -- scalar
  RETURNING orderId
  BULK COLLECT INTO deletedIds;
```



### Bulk Binds - FORALL .... BULK COLLECT INTO Example

In the example above, the FORALL statement causes the DELETE statement to be executed once for each different element of the oldDates table. Many rows can be deleted for each instance of the WHERE clause; all the orderIds returned by different executions of the DELETE statement are appended to (or bulk collected into) the deletedIds table.

# When to Use Bulk Binds

---

- **SQL statements inside PL/SQL loops**
- **Collection elements as bind variables**
- **Four or more rows processed together**



## When to Use Bulk Binds

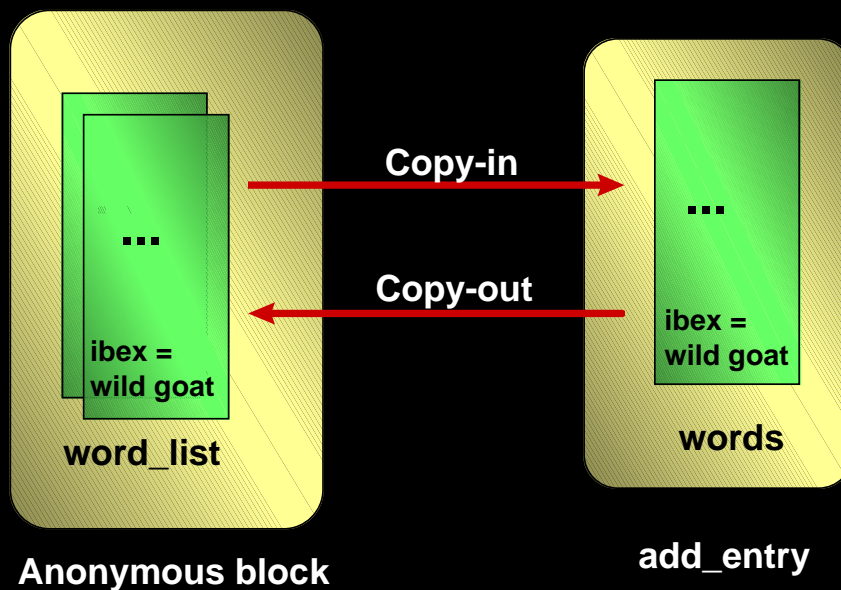
Consider using bulk binds when you have:

- SQL statements inside PL/SQL loops
- A need for using collection elements as bind variables
- A need for processing four or more rows together in an iteration. The more rows affected by a SQL statement, the greater the gains.

Internal Oracle benchmark tests using bulk binds show performance improvements of up to 30%.

# NOCOPY Parameter Passing Hint

## Problem in pre- 8i Releases



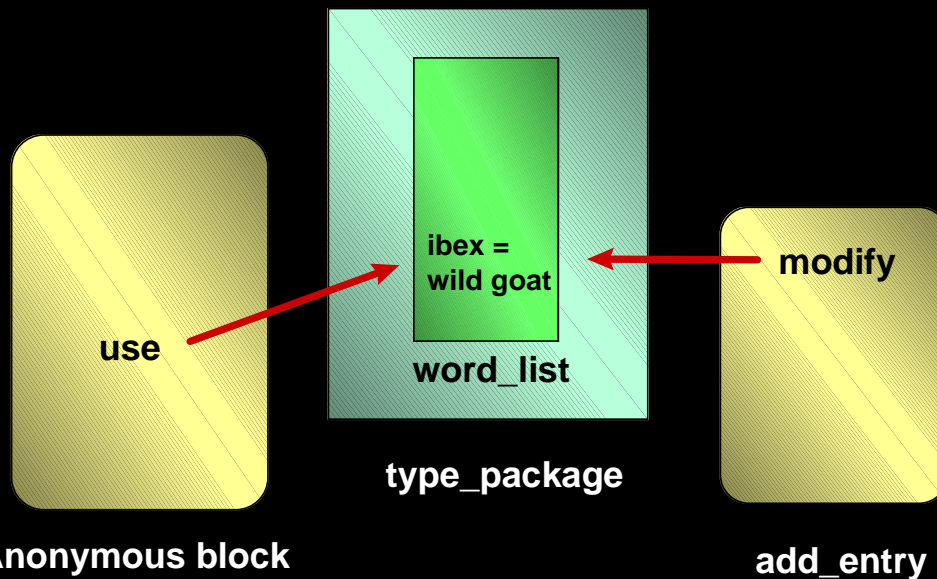
## NOCOPY Parameter Passing Hint (Problem in Pre-8i Releases)

As we know, the IN OUT parameters in PL/SQL are passed using copy-in and copy-out semantics. The example demonstrates how the copy-in, copy-out semantics works. Let's assume that word\_list corresponds to a dictionary and is a huge collection of (word,meaning) pairs.

This imposes huge CPU and memory overhead because the parameters need to be copied-in when the routine is called and then copied-out when the routine returns. The overhead is HUGE when the parameters involved are large data structures, such as large strings or collections.

# NOCOPY Parameter Passing Hint

## Workaround in pre- 8i Releases



### NOCOPY Parameter Passing Hint (Workaround in Pre-8i Releases)

This known performance problem forced the users to use package variables to “pass” arguments around.

Though this gets one around the performance problem imposed by by-value semantics, it has some obvious problems - the workaround increases the maintenance cost by making the programs non-modular, and results in higher maintenance costs.

# NOCOPY Parameter Passing Hint

- **Description:**
  - Ability to pass OUT and IN OUT parameters without the overhead of copies between the actuals and formals
- **Benefit:**
  - Huge performance benefits (30% to 200% faster); especially useful for passing large structures
- **Syntax:**

```
<parameter-specification> :=  
  <parameter-name> IN OUT NOCOPY <type-name>
```



## NOCOPY Parameter Passing Hint

As you already know, PL/SQL supports three parameter passing modes - IN, OUT, and IN OUT.

The IN parameter is passed by reference. That is, a pointer to the IN actual parameter is passed to the corresponding formal parameter. So, both parameters reference the same memory location, which holds the value of the actual parameter. However, the OUT and IN OUT parameters are passed by value.

When an application passes parameters using the OUT and IN OUT modes, the database copies the parameter values to protect the original values in case the function or procedure raises exceptions. Passing large structures, such as strings and collections, places a burden on memory and CPU resources. One alternative is to use package-global variables, rather than parameters, but this results in non-modular programs.

Oracle8i supports a new NOCOPY modifier for OUT and IN OUT parameters, which is a hint to the compiler to indicate that the compiler should attempt to pass IN OUT and OUT parameters by reference when possible.

Because the actual parameters are passed by reference rather than copied into the memory for the corresponding formal parameters, NOCOPY provides significant benefits, especially when passing large structures as parameters. Oracle internal benchmark testing showed improvements of 30 to 200 % improvements for medium-to-large PL/SQL tables passed as parameters.

Note that the NOCOPY modifier is neither required nor allowed with IN parameters; IN parameters are always passed by reference.

The syntax for this is very simple and is shown on the slide.

# NOCOPY Parameter Passing Hint

- **Semantics:**
  - **NOCOPY is a hint, not a directive; compiler attempts to pass by reference but does not promise to do so**
    - **When the call is a remote-procedure call**
    - **When the actual parameter being passed is an expression value**
    - **When implicit conversion is involved while binding an actual to a formal parameter**



## NOCOPY Parameter Passing Hint

### Semantics

Since NOCOPY is a hint and not a directive, Oracle does not guarantee that a parameter will be passed by reference, although an attempt is made to pass the actual by reference when possible. Here are few example cases where the parameter will NOT be passed by reference:

- when the call is a remote-procedure call
- when the actual parameter being passed is an expression value
- when there is an implicit conversion involved when binding an actual parameter to a formal

Because pass by reference is not guaranteed to happen, there are some semantics that are not guaranteed to happen:

- If you had true pass-by-reference and you pass the same actual parameter across different formal parameters, you would expect aliasing to happen. In the case of NOCOPY, there is no guarantee that aliasing will ALWAYS happen under these conditions. So you must not rely on aliasing behavior.
- If you had COPY IN / COPY OUT, you would expect that uncommitted changes would be rolled back if you had an unhandled exception. In the case of true pass-by-reference, you would expect that uncommitted changes would NOT be rolled back if you had an unhandled exception. However, in the case of NOCOPY, since it is not guaranteed to be pass-by-reference, there is no guarantee that uncommitted changes will NOT be rolled back upon an unhandled exception. So you must not rely on rollback-on-exception semantics when using NOCOPY.

# Native Dynamic SQL

## Dynamic SQL in general

- **Description:**
  - Ability to dynamically build and submit SQL statements from PL/SQL
- **Benefits:**
  - Used in applications (such as database query tools, interactive database applications, reusable application code) that allow users to choose query search criteria or optimizer hints at run time
- **Pre-Oracle8i PL/SQL solution:**
  - Use DBMS\_SQL



## Dynamic SQL

Dynamic SQL refers to the ability to build and submit SQL statements (including DML, DDL, transaction control, session control, and anonymous block statements) at run time.

Dynamic SQL is useful when not all of the necessary information is available at compilation time - for example, when using a database query tool that allows you to choose query search criteria or optimizer hints at run time. Dynamic SQL lets you do things like create a procedure that operates on a table whose name is unknown until run time, or accept user input that defines the SQL statement to execute.

Prior to Oracle8i, PL/SQL developers could include dynamic SQL in applications by using the Oracle-supplied DBMS\_SQL package. However, performing simple operations using DBMS\_SQL involves a fair amount of coding. In addition, because DBMS\_SQL is based on a procedural API, it incurs high procedure-call and data-copy overhead.

# Native Dynamic SQL

- **Benefits of native dynamic SQL over DBMS\_SQL:**
  - Highly intuitive syntax
  - A lot less code
  - Much easier to write & maintain code
  - Much faster (30% to 400%)



## Native Dynamic SQL

Oracle8i PL/SQL supports dynamic SQL natively in the language.

In contrast with DBMS\_SQL, the highly intuitive syntax for native dynamic SQL makes it much easier to write and maintain your code. In addition, native dynamic SQL code is considerably more compact. Furthermore, native dynamic SQL bundles the statement preparation, binding, and execution steps into a single operation, thereby improving performance. Internal Oracle benchmark tests using native dynamic SQL show a 30 to 400 percent performance improvement over DBMS\_SQL.

# Native Dynamic SQL

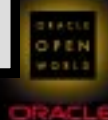
## EXECUTE IMMEDIATE statement

- **Description:**

Prepare, execute, deallocate dynamic SQL stmt

- **Syntax:**

```
EXECUTE IMMEDIATE <dynamic-string>
  [INTO {<list-of-define-variables> |
        <record-variable>}]
  [USING <list-of-bind-arguments>];
where
<list-of-define-variables> :=
  <define-variable> [,<define-variable>];
<list-of-bind-arguments> :=
  [IN | OUT | IN OUT] <bind-argument>
  [, [IN | OUT | IN OUT] <bind-argument>]...;
```



## Native Dynamic SQL (EXECUTE IMMEDIATE statement)

The EXECUTE IMMEDIATE statement in PL/SQL supports native dynamic SQL. It lets you prepare, execute, and deallocate a SQL statement dynamically at run time.

The syntax for this is fairly simple, as shown on this slide. Note the dynamic string in the syntax. You can fetch this string into a list of define variables or into a record variable using the optional INTO clause. You can bind arguments through the optional USING clause.

One limitation in 8.1.5 is that you cannot specify a dynamic number of bind variables in the USING clause. There is currently no way to dynamically describe the statement to find out how many binds there are, and what their exact types are, etc. For some limited scenarios, however, there is a simple workaround if you know what the type of the bind arguments are. The workaround involves using an IF statement to conditionally create your dynamic SQL statement or query and then using another IF statement later to conditionally execute the SQL statement with a separate list of bind arguments in the USING clause for each condition. This workaround is useful when you do not have a large variation on the list of bind variables.

# Native Dynamic SQL - Example

```
PROCEDURE insert_into_table (  
    table_name  varchar2,  
    deptnumber  number,  
    deptname    varchar2,  
    location    varchar2)  
IS  
    stmt_str    varchar2(200);  
BEGIN  
    stmt_str := 'insert into ' || table_name ||  
               ' values (:deptno,:dname,:loc)';  
    EXECUTE IMMEDIATE stmt_str  
        USING deptnumber, deptname, location;  
END;
```



## Native Dynamic SQL (Example)

In this example, the INSERT statement is built at run time in the “stmt\_str” string variable using values passed in as arguments to the “insert\_into\_table” procedure. The SQL statement held in “stmt\_str” is then executed via the EXECUTE IMMEDIATE statement. The bind variables “:deptno”, “:dname”, and “:loc” are bound to the arguments which, in this case, are the parameters “deptnumber”, “deptname”, and “location”.

# Native Dynamic SQL

## EXECUTE IMMEDIATE statement vs. DBMS\_SQL

```
PROCEDURE insert_into_table (table_name varchar2,
                             deptnumber number,
                             deptname varchar2,
                             location varchar2)
IS
  stmt_str varchar2(200);
  cur_hdl integer;
  rows_processed binary_integer;
BEGIN
  stmt_str := 'insert into ' || table_name ||
              ' values (:deptno, :dname, :loc)';
  cur_hdl := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(cur_hdl, stmt_str, dbms_sql.native);
  DBMS_SQL.BIND_VARIABLE(cur_hdl, ':deptno', deptnumber);
  DBMS_SQL.BIND_VARIABLE(cur_hdl, ':dname', deptname);
  DBMS_SQL.BIND_VARIABLE(cur_hdl, ':loc', location);
  rows_processed := DBMS_SQL.EXECUTE(cur_hdl);
  DBMS_SQL.CLOSE_CURSOR(cur_hdl);

  EXECUTE IMMEDIATE stmt_str USING deptnumber, deptname, location;
END;
```

### Native Dynamic SQL (Example of Native Dynamic SQL vs. DBMS\_SQL)

The example on the previous slide is shown above in conjunction with code that was written to perform the same operation using DBMS\_SQL. The struck out lines are the lines you would have needed to use DBMS\_SQL. This example clearly demonstrates how much easier it is to write and read native dynamic SQL code compared with calls to DBMS\_SQL.

# Transparent Performance Improvements

- **Optimization of package STANDARD builtins**
- **Faster anonymous block execution**
- **Faster RPC parameter passing**
- **Scalability (more users) improvements**
- **Caching of DLLs for improved external procedure performance**



## Transparent Performance Improvements

While the preceding pages discuss ways to improve performance, it is important to note that the PL/SQL interpreter has also been tuned in many ways to improve performance transparently. This means that if you simply upgrade your database to Oracle8i, your PL/SQL applications will automatically run faster without your having to change a single line of code.

For example, Oracle has optimized the built-in functions in package STANDARD for faster performance. In Oracle internal benchmark testing, calls made to built-ins (TO\_CHAR, TO\_DATE, and SUBSTR, for example) operated 10 to 30 percent faster in Oracle8i than they did in Oracle8 or Oracle7.

---

# Future PL/SQL Performance Projects



# Future PL/SQL Performance Projects

- **PL/SQL language optimizer**
- **Native compilation of PL/SQL byte code**
- **Native floating point datatype**
- **INDEX-BY tables indexed by VARCHAR2**
- **More transparent improvements**



## Future PL/SQL Performance Projects

Several new projects are currently under consideration for a future release of PL/SQL.

The optimizer should significantly reduce the need for explicit fixes of the basic code problems mentioned above. We also expect significant transparent performance benefits from the associated rewrite of our code generator. Native compilation should compliment the optimizer nicely, avoiding the cost of our byte code interpreter, but taking advantage of the more efficient generated code. A native floating point data type provides an much faster alternative to the slow but precise NUMBER type, much as PLS\_INTEGER does for INTEGER. Indexing tables with VARCHAR2 rather than BINARY\_INTEGER offers several potential performance benefits.

---

# Summary



# Summary

---

## Reference Books:

PL/SQL User's Guide and Reference, Release 8.1.5

Oracle8i Concepts, Release 8.1.5

Oracle8i SQL Reference, Release 8.1.5

Oracle8i Application Developer's Guide - Fundamentals, Release 8.1.5

Oracle8i Supplied Packages Reference, Release 8.1.5

## Contact Info:

- Tushar Gadhia: [tgadhia@us.oracle.com](mailto:tgadhia@us.oracle.com)
- Chris Racicot: [cracicot@us.oracle.com](mailto:cracicot@us.oracle.com)



