

Performance and Tuning



ORACLE

Copyright © 2004, Oracle. All rights reserved.

Objectives

After completing this lesson, you should be able to do the following:

- **Tune PL/SQL code**
- **Identify and tune memory issues**
- **Recognize network issues**
- **Perform native and interpreted compilation**

ORACLE

Copyright © 2004, Oracle. All rights reserved.

Objectives

In this lesson, the performance and tuning topics are divided into four main groups

- Tuning PL/SQL code
- Memory issues
- Network issues
- Native and interpreted compilation.

In the “Tuning PL/SQL Code” section, you learn why it is important to write smaller executable sections of code; when to use SQL or PL/SQL; how bulk binds can improve performance; how to use the FORALL syntax; how to rephrase conditional statements; about data types and constraint issues.

In the memory issues section, you learn about the shared pool and what you can do programmatically to tune it.

In the network issues section, you learn why it is important to group your OPEN-FOR statements when passing host cursor variables to PL/SQL; when it is appropriate to use client-side PL/SQL; how to avoid unnecessary reparsing; how to utilize array processing.

In the compilation section, you learn about native and interpreted compilation.

Tuning PL/SQL Code

You can tune your PL/SQL code by:

- **Writing smaller executable sections of code**
- **Comparing SQL with PL/SQL and where one is appropriate over the other**
- **Understanding how bulk binds can improve performance**
- **Using the `FORALL` support with bulk binding**
- **Handling and saving exceptions with the `SAVE EXCEPTIONS` syntax**
- **Rephrasing conditional statements**
- **Identifying data type and constraint issues**

ORACLE

Copyright © 2004, Oracle. All rights reserved.

Tuning PL/SQL Code

By tuning your PL/SQL code, you can tailor its performance to best meet your needs. In the following pages you learn about some of the main PL/SQL tuning issues that can improve the performance of your PL/SQL applications.

Modularizing Your Code

- **Limit the number of lines of code between a `BEGIN` and `END` to about a page or 60 lines of code.**
- **Use packaged programs to keep each executable section small.**
- **Use local procedures and functions to hide logic.**
- **Use a function interface to hide formulas and business rules.**



ORACLE

Copyright © 2004, Oracle. All rights reserved.

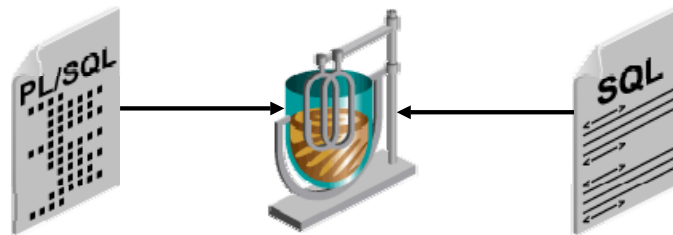
Write Smaller Executable Sections

By writing smaller sections of executable code, you can make the code easier to read, understand, and maintain. When developing an application, use a stepwise refinement. Make a general description of what you want your program to do, and then implement the details in subroutines. Using local modules and packaged programs can help in keeping each executable section small. This will make it easier for you to debug and refine your code.

Comparing SQL with PL/SQL

Each has its own benefits:

- **SQL:**
 - **Accesses data in the database**
 - **Treats data as sets**
- **PL/SQL:**
 - **Provides procedural capabilities**
 - **Has more flexibility built into the language**



ORACLE

Copyright © 2004, Oracle. All rights reserved.

SQL Versus PL/SQL

Both SQL and PL/SQL have their strengths. However, there are situations where one language is more appropriate to use than the other.

You use SQL to access data in the database with its powerful statements. SQL processes sets of data as groups rather than as individual units. The flow-control statements of most programming languages are absent in SQL, but present in PL/SQL. When using SQL in your PL/SQL applications, be sure not to repeat a SQL statement. Instead, encapsulate your SQL statements in a package and make calls to the package.

Using PL/SQL, you can take advantage of the PL/SQL-specific enhancements for SQL, such as autonomous transactions, fetching into cursor records, using a cursor FOR loop, using the RETURNING clause for information about modified rows, and using BULK COLLECT to improve the performance of multirow queries.

While there are advantages of using PL/SQL over SQL in several cases, use PL/SQL with caution, especially under the following circumstances:

- Performing high-volume inserts
- Using user-defined PL/SQL functions
- Using external procedure calls
- Using the utl_file package as an alternative to SQL*Plus in high-volume reporting

Comparing SQL with PL/SQL

- **Some simple set processing is markedly faster than the equivalent PL/SQL.**

```
BEGIN
  INSERT INTO inventories2
    SELECT product_id, warehouse_id
    FROM main_inventories;
END;
```

- **Avoid using procedural code when it may be better to use SQL.**

```
...FOR I IN 1..5600 LOOP
  counter := counter + 1;
  SELECT product_id, warehouse_id
    INTO v_p_id, v_wh_id
    FROM big_inventories WHERE v_p_id = counter;
  INSERT INTO inventories2 VALUES(v_p_id, v_wh_id);
END LOOP;...
```



ORACLE

Copyright © 2004, Oracle. All rights reserved.

SQL Versus PL/SQL (continued)

The SQL statement explained in the slide is a great deal faster than the equivalent PL/SQL loop. Take advantage of the simple set processing operations implicitly available in the SQL language, as it can run markedly faster than the equivalent PL/SQL loop. Avoid writing procedural code when SQL would work better.

However, there are occasions when you will get better performance from PL/SQL even when the process could be written in SQL. Correlated updates are slow. With correlated updates, a better method is to access only correct rows using PL/SQL. The following PL/SQL loop is faster than the equivalent correlated update SQL statement.

```
DECLARE
  CURSOR cv_raise IS
    SELECT deptno, increase
    FROM emp_raise;
BEGIN
  FOR dept IN cv_raise LOOP
    UPDATE big_emp
      SET sal = sal * dept.increase
      WHERE deptno = dept.deptno;
  END LOOP;
  ...

```

Comparing SQL with PL/SQL

- **Instead of:**

```
...  
INSERT INTO order_items  
  (order_id, line_item_id, product_id,  
   unit_price, quantity)  
VALUES (...
```

- **Create a stand-alone procedure:**

```
insert_order_item (  
  2458, 6, 3515, 2.00, 4);
```

- **Or, a packaged procedure:**

```
orderitems.ins (  
  2458, 6, 3515, 2.00, 4);
```

ORACLE

Copyright © 2004, Oracle. All rights reserved.

Encapsulate SQL Statements

From a design standpoint, do not embed your SQL statements directly within application code. It is better if you write procedures to perform your SQL statements.

Pros

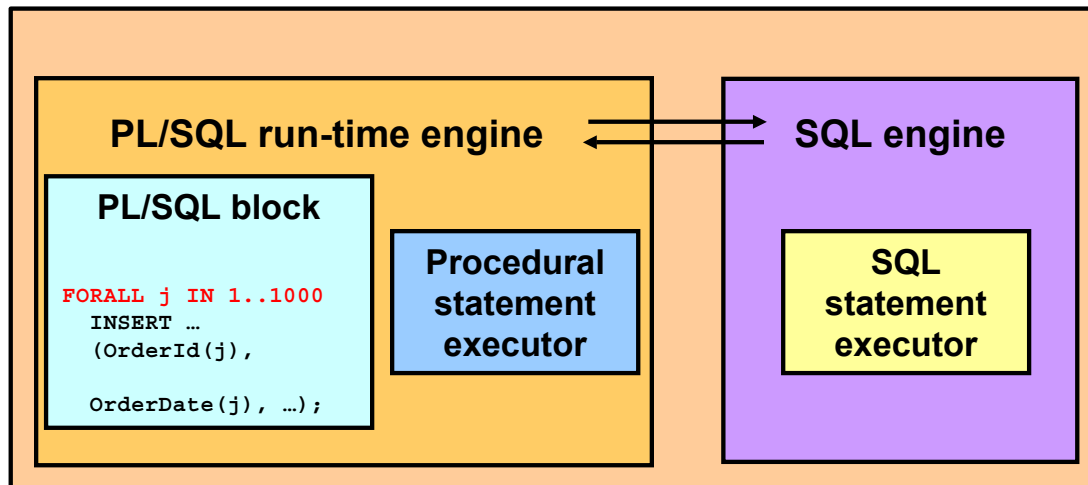
- If you design your application so that all programs that perform an insert on a specific table use the same INSERT statement, your application will run faster because of less parsing and reduced demands on the SGA memory.
- Your program will also handle DML errors consistently.

Cons

- You may need to write more procedural code.
- You may need to write several variations of update or insert procedures to handle the combinations of columns that you are updating or inserting into.

Using Bulk Binding

Use bulk binds to reduce context switches between the PL/SQL engine and the SQL engine.



ORACLE

Copyright © 2004, Oracle. All rights reserved.

Bulk Binding

With bulk binds, you can improve performance by decreasing the number of context switches between the SQL and PL/SQL engine. When a PL/SQL program executes, each time a SQL statement is encountered, there is a switch between the PL/SQL engine to the SQL engine. The more the number of switches, the lesser the efficiency.

Improved Performance

Bulk binding enables you to implement array fetching. With bulk binding, entire collections, and not just individual elements, are passed back and forth. Bulk binding can be used with nested tables, varrays, and associative arrays.

The more rows affected by a SQL statement, the greater is the performance gain with bulk binding.

Using Bulk Binding

Bind whole arrays of values all at once, rather than looping to perform fetch, insert, update, and delete on multiple rows.

- **Instead of:**

```
...
FOR i IN 1 .. 50000 LOOP
  INSERT INTO bulk_bind_example_tbl
    VALUES (...);
END LOOP; ...
```

- **Use:**

```
...
FORALL i IN 1 .. 50000
  INSERT INTO bulk_bind_example_tbl
    VALUES (...);
END; ...
```



Copyright © 2004, Oracle. All rights reserved.

Using Bulk Binding

In the first example shown, one row is inserted into the target table at a time. In the second example, the FOR loop is changed to a FORALL (which has an implicit loop) and all immediately subsequent DML statements are processed in bulk. The following are the entire code examples along with timing statistics for running each FOR loop example.

First, create the demonstration table:

```
CREATE TABLE bulk_bind_example_tbl (
  num_col NUMBER,
  date_col DATE,
  char_col VARCHAR2(40));
```

Second, set the SQL*Plus TIMING variable on. Setting this on enables you to see the approximate elapsed time of the last SQL statement:

```
SET TIMING ON
```

Third, run this block of code that includes a FOR loop to insert 50,000 rows:

```
DECLARE
  TYPE typ_numlist IS TABLE OF NUMBER;
  TYPE typ_datelist IS TABLE OF DATE;
  TYPE typ_charlist IS TABLE OF VARCHAR2(40)
    INDEX BY PLS_INTEGER;
  -- continued onto next page...
```

Using Bulk Binding (continued)

```

n typ_numlist := typ_numlist();
d typ_datelist := typ_datelist();
c typ_charlist;

BEGIN
  FOR i IN 1 .. 50000 LOOP
    n.extend;
    n(i) := i;
    d.extend;
    d(i) := sysdate + 1;
    c(i) := lpad(1, 40);
  END LOOP;
  FOR I in 1 .. 50000 LOOP
    INSERT INTO bulk_bind_example_tbl
      VALUES (n(i), d(i), c(i));
  END LOOP;
END;
/
PL/SQL procedure successfully completed.
Elapsed: 00:00:17.62

```

Last, run this block of code that includes a FORALL loop to insert 50,000 rows. Note the significant decrease in the timing when using the FORALL processing:

```

DECLARE
  TYPE typ_numlist IS TABLE OF NUMBER;
  TYPE typ_datelist IS TABLE OF DATE;
  TYPE typ_charlist IS TABLE OF VARCHAR2(40)
    INDEX BY PLS_INTEGER;

  n typ_numlist := typ_numlist();
  d typ_datelist := typ_datelist();
  c typ_charlist;

BEGIN
  FOR i IN 1 .. 50000 LOOP
    n.extend;
    n(i) := i;
    d.extend;
    d(i) := sysdate + 1;
    c(i) := lpad(1, 40);
  END LOOP;
  FORALL I in 1 .. 50000
    INSERT INTO bulk_bind_example_tbl
      VALUES (n(i), d(i), c(i));
END;
/

PL/SQL procedure successfully completed.
Elapsed: 00:00:02.08

```

Using Bulk Binding

Use BULK COLLECT to improve performance:

```
CREATE OR REPLACE PROCEDURE process_customers
(p_account_mgr customers.account_mgr_id%TYPE)
IS
  TYPE typ_numtab IS TABLE OF
    customers.customer_id%TYPE;
  TYPE typ_chartab IS TABLE OF
    customers.cust_last_name%TYPE;
  TYPE typ_emailtab IS TABLE OF
    customers.cust_email%TYPE;
  v_custnos      typ_numtab;
  v_last_names   typ_chartab;
  v_emails       typ_emailtab;
BEGIN
  SELECT customer_id, cust_last_name, cust_email
    BULK COLLECT INTO v_custnos, v_last_names, v_emails
  FROM customers
  WHERE account_mgr_id = p_account_mgr;
  ...
END process_customers;
```

ORACLE

Copyright © 2004, Oracle. All rights reserved.

Using BULK COLLECT

When you require a large number of rows to be returned from the database, you can use the BULK COLLECT option for queries. This option enables you to retrieve multiple rows of data in a single request. The retrieved data is then populated into a series of collection variables. This query will run significantly faster than if it were done without the BULK COLLECT.

You can use the BULK COLLECT option with explicit cursors too:

```
BEGIN
  OPEN cv_customers INTO customers_rec;
  FETCH cv_customers BULK COLLECT INTO
    v_custnos, v_last_name, v_mails;
  ...
```

Using Bulk Binding

Use the RETURNING clause to retrieve information about rows being modified:

```
DECLARE
  TYPE      typ_replist IS VARRAY(100) OF NUMBER;
  TYPE      typ_numlist IS TABLE OF
              orders.order_total%TYPE;
  repids    typ_replist :=
              typ_replist(153, 155, 156, 161);
  totlist    typ_numlist;
  c_big_total CONSTANT NUMBER := 60000;
BEGIN
  FORALL i IN repids.FIRST..repids.LAST
    UPDATE  orders
    SET      order_total = .95 * order_total
    WHERE    sales_rep_id = repids(i)
    AND      order_total > c_big_total
    RETURNING order_total BULK COLLECT INTO Totlist;
END;
```

ORACLE

Copyright © 2004, Oracle. All rights reserved.

The RETURNING Clause

Often, applications need information about the row affected by a SQL operation, for example, to generate a report or take a subsequent action. Using the RETURNING clause, you can retrieve information about rows you have just modified with the INSERT, UPDATE, and DELETE statements. This can improve performance because it enables you to make changes, and at the same time, collect information of the data being changed. As a result, fewer network round trips, less server CPU time, fewer cursors, and less server memory are required. Without the RETURNING clause, you need two operations: one to make the change, and a second operation to retrieve information about the change.

In the example shown, the `order_total` information is retrieved from the `ORDERS` table and collected into the `totlist` collection. The `totlist` collection is returned in bulk to the PL/SQL engine.

If you did not use the RETURNING clause, you would need to perform two operations, one for the UPDATE, and another for the SELECT:

```
UPDATE  orders SET order_total = .95 * order_total
WHERE    sales_rep_id = p_id
AND      order_total > c_big_total;
```

```
SELECT order_total FROM  orders
WHERE    sales_rep_id = p_id AND order_total > c_big_total;
```

The RETURNING Clause (continued)

In the following example, you update the credit limit of a customer and at the same time retrieve the customer's new credit limit into a SQL*Plus environment variable:

```
CREATE OR REPLACE PROCEDURE change_credit
  (p_in_id   IN   customers.customer_id%TYPE,
   o_credit  OUT NUMBER)
IS
BEGIN
  UPDATE customers
  SET   credit_limit = credit_limit * 1.10
  WHERE customer_id = p_in_id
  RETURNING credit_limit INTO o_credit;
END change_credit;
/
VARIABLE g_credit NUMBER
EXECUTE change_credit(109, :g_credit)
PRINT g_credit
```

Using SAVE EXCEPTIONS

- You can use the **SAVE EXCEPTIONS** keywords in your **FORALL** statements:

```
FORALL index IN lower_bound..upper_bound
SAVE EXCEPTIONS
{insert_stmt | update_stmt | delete_stmt}
```

- Exceptions raised during execution are saved in the **%BULK_EXCEPTIONS** cursor attribute.
- The attribute is a collection of records with two fields:

Field	Definition
ERROR_INDEX	Holds the iteration of the FORALL statement where the exception was raised
ERROR_CODE	Holds the corresponding Oracle error code

- Note that the values always refer to the most recently executed **FORALL** statement.

ORACLE

Copyright © 2004, Oracle. All rights reserved.

Handling FORALL Exceptions

To handle exceptions encountered during a **BULK BIND** operation, you can add the keyword **SAVE EXCEPTIONS** to your **FORALL** statement. Without it, if any one row fails during the **FORALL** loop, the loop execution is terminated. **SAVE_EXCEPTIONS** allows the loop to continue processing and is required if you want the loop to continue.

All exceptions raised during the execution are saved in the cursor attribute **%BULK_EXCEPTIONS**, which stores a collection of records. This cursor attribute is available only from the exception handler.

Each record has two fields. The first field, **%BULK_EXCEPTIONS(i).ERROR_INDEX**, holds the “iteration” of the **FORALL** statement during which the exception was raised. The second field, **BULK_EXCEPTIONS(i).ERROR_CODE**, holds the corresponding Oracle error code.

The values stored by **%BULK_EXCEPTIONS** always refer to the most recently executed **FORALL** statement. The number of exceptions is saved in the count attribute of **%BULK_EXCEPTIONS**, that is, **%BULK_EXCEPTIONS.COUNT**. Its subscripts range from 1 to **COUNT**. If you omit the keywords **SAVE EXCEPTIONS**, execution of the **FORALL** statement stops when an exception is raised. In that case, **SQL%BULK_EXCEPTIONS.COUNT** returns 1, and **SQL%BULK_EXCEPTIONS** contains just one record. If no exception is raised during the execution, **SQL%BULK_EXCEPTIONS.COUNT** returns 0.

Handling FORALL Exceptions

```
DECLARE
  TYPE NumList IS TABLE OF NUMBER;
  num_tab      NumList :=
                NumList(100,0,110,300,0,199,200,0,400);
  bulk_errors  EXCEPTION;
  PRAGMA      EXCEPTION_INIT (bulk_errors, -24381 );
BEGIN
  FORALL i IN num_tab.FIRST..num_tab.LAST
    SAVE EXCEPTIONS
    DELETE FROM orders WHERE order_total < 500000/num_tab(i);
  EXCEPTION WHEN bulk_errors THEN
    DBMS_OUTPUT.PUT_LINE('Number of errors is: '
                        || SQL%BULK_EXCEPTIONS.COUNT);
  FOR j IN 1..SQL%BULK_EXCEPTIONS.COUNT
  LOOP
    DBMS_OUTPUT.PUT_LINE (
      TO_CHAR(SQL%BULK_EXCEPTIONS(j).error_index) ||
      ' / ' ||
      SQLERRM(-SQL%BULK_EXCEPTIONS(j).error_code) );
  END LOOP;
END;
/
```

ORACLE

Copyright © 2004, Oracle. All rights reserved.

Example

In this example, the `EXCEPTION_INIT` pragma defines an exception named `BULK_ERRORS` and associates the name to the `ORA-24381` code, which is an "Error in Array DML". The PL/SQL block raises the predefined exception `ZERO_DIVIDE` when `i` equals 2, 5, 8. After the bulk-bind is completed, `SQL%BULK_EXCEPTIONS.COUNT` returns 3 because of trying to divide by zero three times. To get the Oracle error message (which includes the code), we pass `SQL%BULK_EXCEPTIONS(i).ERROR_CODE` to the error-reporting function `SQLERRM`.

Here is the output:

```
Number of errors is: 5
Number of errors is: 3
2 / ORA-01476: divisor is equal to zero
5 / ORA-01476: divisor is equal to zero
8 / ORA-01476: divisor is equal to zero
```

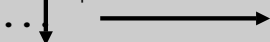
PL/SQL procedure successfully completed.

Rephrasing Conditional Control Statements

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

- **Scenario 1:**

```
IF TRUE | FALSE OR (v_sales_rep_id IS NULL) THEN
  ..
  ..
  ..
END IF;
```



- **Scenario 2:**

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

ORACLE

Copyright © 2004, Oracle. All rights reserved.

Rephrase Conditional Control Statements

In logical expressions, improve performance by tuning conditional constructs carefully.

When evaluating a logical expression, PL/SQL stops evaluating the expression as soon as the result can be determined. For example, in the first scenario in the slide, which involves an OR expression, when the value of the left operand yields TRUE, PL/SQL need not evaluate the right operand (because OR returns TRUE if either of its operands is true).


Now, consider the second scenario in the slide, which involves an AND expression. 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 `v_order_total < 5000` is true (because AND returns TRUE only if both its operands are true):

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



Rephrasing Conditional Control Statements

If your business logic results in one condition being true, use the **ELSIF** syntax for mutually exclusive clauses:

```
IF v_acct_mgr = 145 THEN
    process_acct_145;
END IF;
IF v_acct_mgr = 147 THEN
    process_acct_147;
END IF;
IF v_acct_mgr = 148 THEN
    process_acct_148;
END IF;
IF v_acct_mgr = 149 THEN
    process_acct_149;
END IF;
```



```
IF v_acct_mgr = 145
THEN
    process_acct_145;
ELSIF v_acct_mgr = 147
THEN
    process_acct_147;
ELSIF v_acct_mgr = 148
THEN
    process_acct_148;
ELSIF v_acct_mgr = 149
THEN
    process_acct_149;
END IF;
```



ORACLE

Copyright © 2004, Oracle. All rights reserved.

Mutually Exclusive Conditions

If you have a situation where you are checking a list of choices for a mutually exclusive result, use the **ELSIF** syntax, as it offers the most efficient implementation. With **ELSIF**, after a branch evaluates to **TRUE**, the other branches are not executed.

In the example shown on the right, every **IF** statement is executed. In the example on the left, after a branch is found to be true, the rest of the branch conditions are not evaluated.

Sometimes you do not need an **IF** statement. For example, the following code can be rewritten without an **IF** statement:

```
IF date_ordered < sysdate + 7 THEN
    late_order := TRUE;
ELSE
    late_order := FALSE;
END IF;

--rewritten without an IF statement:
late_order := date_ordered < sysdate + 7;
```

Avoiding Implicit Data Type Conversion

- **PL/SQL performs implicit conversions between structurally different data types.**
- **Example: When assigning a PLS_INTEGER variable to a NUMBER variable**

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

numbers

strings

dates

ORACLE

Copyright © 2004, Oracle. All rights reserved.

Avoid Implicit Data Type Conversion

PL/SQL automatically performs implicit conversions between structurally different types at run time for you. By avoiding implicit conversions, you can improve the performance of your code. The major problems with implicit data type conversion are:

- It is non-intuitive and can result in unexpected results.
- You have no control over the implicit conversion.

In the example shown, assigning a PLS_INTEGER variable to a NUMBER variable or vice versa results in a conversion, because their representations are different. Such implicit conversions can happen during parameter passing as well. 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.

To avoid implicit data type conversion, you can use the built-in functions:

- TO_DATE
- TO_NUMBER
- TO_CHAR
- CAST

Using PLS_INTEGER Data Type for Integers

Use PLS_INTEGER when dealing with integer data:

- **It is an efficient data type for integer variables.**
- **It requires less storage than INTEGER or NUMBER.**
- **Its operations use machine arithmetic, which is faster than library arithmetic.**



ORACLE

Copyright © 2004, Oracle. All rights reserved.

Use PLS_INTEGER for All Integer Operations

When you need to declare an integer variable, use the PLS_INTEGER data type, which is the most efficient numeric type. That is because PLS_INTEGER values require less storage than INTEGER or NUMBER values, which are represented internally as 22-byte Oracle numbers. Also, PLS_INTEGER operations use machine arithmetic, so they are faster than BINARY_INTEGER, INTEGER, or NUMBER operations, which use library arithmetic.

Furthermore, INTEGER, NATURAL, NATURALN, POSITIVE, POSITIVEN, and SIGNTYPE are constrained subtypes. Their variables require precision checking at run time that can affect the performance.

The Oracle Database 10g data types BINARY_FLOAT and BINARY_DOUBLE are also faster than the NUMBER data type.

Understanding the NOT NULL Constraint

```
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;
```

ORACLE

Copyright © 2004, Oracle. All rights reserved.

The NOT NULL Constraint

In PL/SQL, using the NOT NULL constraint incurs a small performance cost. Therefore, use it with care. Consider the example on the left in the slide that 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.

Note that the subtypes NATURALN and POSTIVEN are defined as NOT NULL subtypes of NATURAL and POSITIVE. Using them incurs the same performance cost as seen above.

Using the NOT NULL Constraint

Slower

No extra coding is needed.

When an error is implicitly raised, the value of *m* is preserved.

Not Using the Constraint

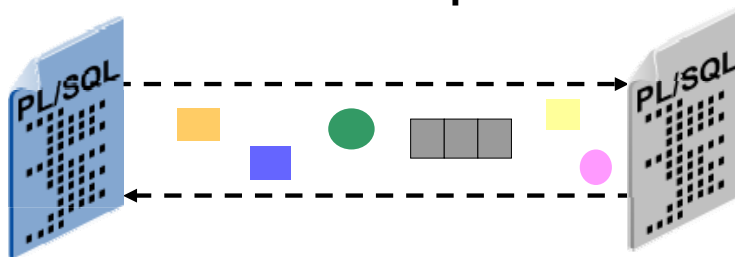
Faster

Requires extra coding that is error prone

When an error is explicitly raised, the old value of *m* is lost.

Passing Data Between PL/SQL Programs

- The flexibility built into PL/SQL enables you to pass:
 - Simple scalar variables
 - Complex data structures
- You can use the `NOCOPY` hint to improve performance with `IN OUT` parameters.



ORACLE

Copyright © 2004, Oracle. All rights reserved.

Passing Data Between PL/SQL Programs

You can pass simple scalar data or complex data structures between PL/SQL programs.

When passing collections as parameters, you may encounter a slight decrease in performance as compared with passing scalar data, but the performance is still comparable. However, when passing `IN OUT` parameters that are complex (such as collections) to a procedure, you will experience significantly more overhead because a copy of the parameter value before the routine is executed is stored. The stored value must be kept in case an exception occurs. You can use the `NOCOPY` compiler hint to improve performance in this situation. `NOCOPY` instructs the compiler to not make a backup copy of the parameter that is being passed. Be careful when using the `NOCOPY` compiler hint because should your program encounter an exception, your results are not predictable.

Passing Data Between PL/SQL Programs

Passing records as parameters to encapsulate data, as well as, write and maintain less code:

```
DECLARE
  TYPE CustRec IS RECORD (
    customer_id      customers.customer_id%TYPE,
    cust_last_name   VARCHAR2(20),
    cust_email       VARCHAR2(30),
    credit_limit      NUMBER(9,2));
  ...
  PROCEDURE raise_credit (cust_info CustRec);
```

ORACLE

Copyright © 2004, Oracle. All rights reserved.

Passing Records as Arguments

You can declare user-defined records as formal parameters of procedures and functions as shown above. By using records to pass values, you are encapsulating the data being passed, and it requires less coding than defining, assigning, and manipulating each record field individually.

When you call a function that returns a record, use the notation:

`function_name(parameters).field_name`

For example, the following call to the `NTH_HIGHEST_ORD_TOTAL` function references the field `ORDER_TOTAL` in the `ORD_INFO` record:

```
DECLARE
  TYPE OrdRec IS RECORD (
    v_order_id      NUMBER(6),
    v_order_total   REAL);
  v_middle_total   REAL;
  FUNCTION nth_highest_total (n INTEGER) RETURN OrdRec IS
    order_info OrdRec;
  BEGIN
    ...
    RETURN order_info; -- return record
  END;
  BEGIN
    -- call function
    v_middle_total := nth_highest_total(10).v_order_total;
```

Passing Data Between PL/SQL Programs

Use collections as arguments:

```
PACKAGE cust_actions IS
  TYPE NameTabTyp IS TABLE OF
customer.cust_last_name%TYPE
  INDEX BY PLS_INTEGER;
  TYPE CreditTabTyp IS TABLE OF
customers.credit_limit%TYPE
  INDEX BY PLS_INTEGER;
  ...
  PROCEDURE credit_batch( name_tab IN NameTabTyp ,
                           credit_tab IN CreditTabTyp,
                           ... );
  PROCEDURE log_names ( name_tab IN NameTabTyp );
END cust_actions;
```

ORACLE

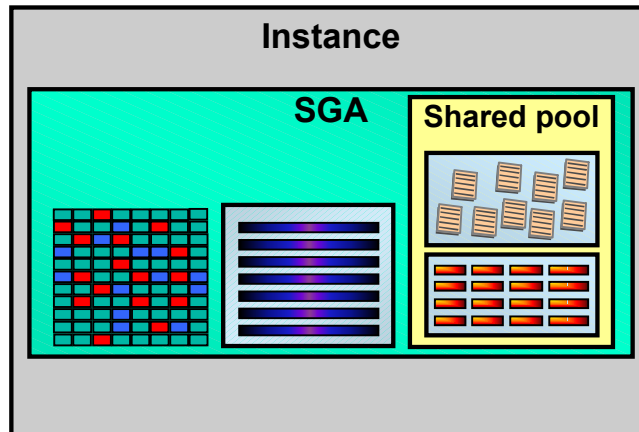
Copyright © 2004, Oracle. All rights reserved.

Passing Collections as Arguments

You can declare collections as formal parameters of procedures and functions. In the example in the slide, associative arrays are declared as the formal parameters of two packaged procedures. If you were to use scalar variables to pass the data, you would need to code and maintain many more declarations.

Identifying and Tuning Memory Issues

Tuning the shared pool:



ORACLE

Copyright © 2004, Oracle. All rights reserved.

Tuning the Size of the Shared Pool of the SGA

When you invoke 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.

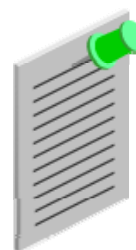
If the package is already present in the shared memory area, your code executes faster. It is, therefore, important to make sure that packages that are used very frequently are always present in memory. The larger the shared pool area, the more likely it is that the package remains in memory. However, 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.

Note: Tuning the shared pool is usually a DBA's responsibility.

Pinning Objects

Pinning:

- Is used so that objects avoid the Oracle least recently used (LRU) mechanism and do not get flushed out of memory
- Is applied with the help of the `sys.dbms_shared_pool` package:
 - `sys.dbms_shared_pool.keep`
 - `sys.dbms_shared_pool.unkeep`
 - `sys.dbms_shared_pool.sizes`



ORACLE

Copyright © 2004, Oracle. All rights reserved.

What Is Pinning a Package?

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 the Oracle server 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 `sys.dbms_shared_pool` package. This package contains three procedures:

Procedure	Description
<code>keep</code>	Use this procedure to pin objects to the shared pool.
<code>unkeep</code>	Use this procedure to age out an object that you have requested to be kept in the shared pool.
<code>sizes</code>	Use this procedure to dump the contents of the shared pool to the <code>DBMS_OUTPUT</code> buffer. It can show the objects in the shared pool that are larger than the specified size, in kilobytes.

Pinning Objects

Syntax:

```
SYS.DBMS_SHARED_POOL.KEEP(object_name, flag)
```

```
SYS.DBMS_SHARED_POOL.UNKEEP(object_name, flag)
```

Example:

```
...  
BEGIN  
  SYS.DBMS_SHARED_POOL.KEEP ('OE.OVER_PACK', 'P');  
  ...  
  SYS.DBMS_SHARED_POOL.UNKEEP ('OE.OVER_PACK', 'P');  
  ...  
END;  
...
```

ORACLE

Copyright © 2004, Oracle. All rights reserved.

Using sys.dbms_shared_pool

You can pin and unpin packages, procedures, functions, types, triggers, and sequences. This may be useful for certain semi-frequently used large objects (larger than 20 KB), because when large objects are brought into the shared pool, a larger number of other objects (much more than the size of the object being brought in) may need to be aged out in order to create a contiguous area large enough. Pinning occurs when the `sys.dbms_shared_pool.keep` procedure is invoked.

To create `DBMS_SHARED_POOL`, run the `DBMSPOOL.SQL` script. The `PRVTPPOOL.PLB` script is automatically executed after `DBMSPOOL.SQL` runs.

Using `sys.dbms_shared_pool` (continued)

Syntax Definitions

where: *object_name* Name of the object to keep
 flag (Optional) If this is not specified, then the package assumes that the first parameter is the name of a package/procedure/function and resolves the name.

 ' P ' or ' p ' indicates a package/procedure/function. This is the default.

 ' T ' or ' t ' indicates a type.

 ' R ' or ' r ' indicates a trigger.

Pinning Objects

- **Pin objects only when necessary.**
- **The `keep` procedure first queues an object for pinning before loading it.**
- **Pin all objects soon after instance startup to ensure contiguous blocks of memory.**

ORACLE

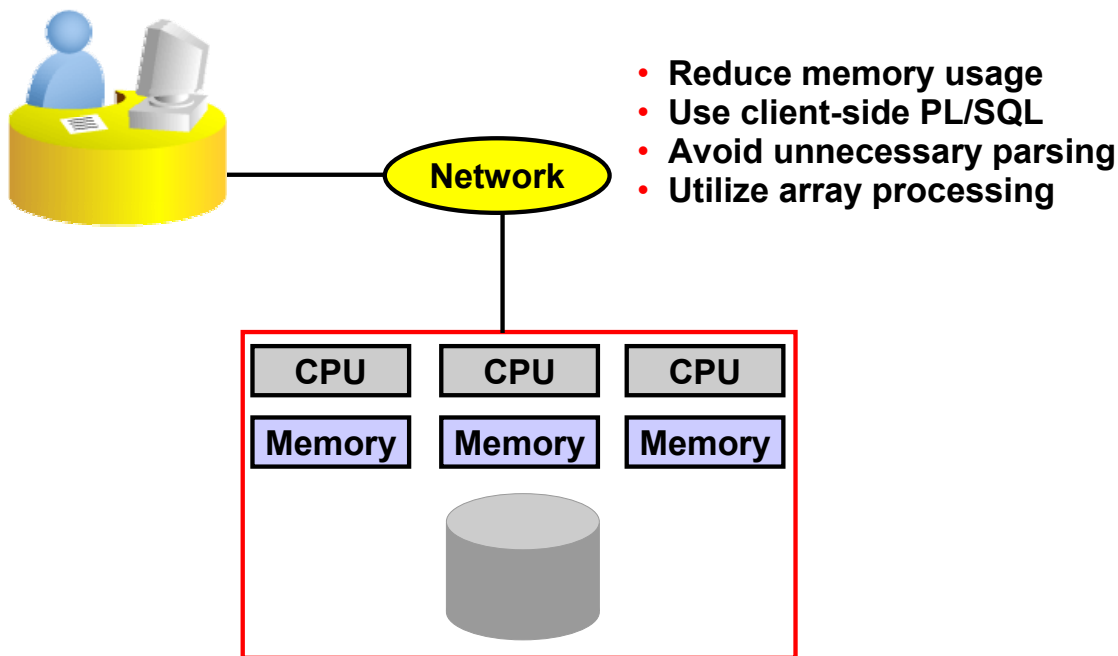
Copyright © 2004, Oracle. All rights reserved.

Guidelines for Pinning Objects

- Pin objects only when necessary. Otherwise, you may end up setting aside too much memory, which can have a negative impact on performance.
- The `keep` procedure does not immediately load a package into the shared pool; it queues the package for pinning. The package is loaded into the shared pool only when the package is first referenced, either to execute a module or to use one of its declared objects, such as a global variable or a cursor.
- Pin all your objects in the shared pool as soon after instance startup as possible, so that contiguous blocks of memory can be set aside for large objects.

Note: You can create a trigger that fires when the database is opened (`STARTUP`). Using this trigger is a good way to pin packages at the very beginning.

Identifying Network Issues



ORACLE

Copyright © 2004, Oracle. All rights reserved.

Guidelines for Reducing Network Traffic

Reducing network traffic is one of the key components of tuning because network issues impact performance. When your code is passed to the database, a significant amount of time is spent in the network. The following are some guidelines for reducing network traffic to improve performance:

- When passing host cursor variables to PL/SQL, you can reduce network traffic by grouping OPEN-FOR statements. For example, the following PL/SQL block opens five cursor variables in a single round trip:

```
/* anonymous PL/SQL block in host environment */
BEGIN
  OPEN :cust_cv      FOR SELECT * FROM customers;
  OPEN :order_cv     FOR SELECT * FROM orders;
  OPEN :ord_item_cv  FOR SELECT * FROM order_items;
  OPEN :wh_cv        FOR SELECT * FROM warehouses;
END;
```
- When you pass host cursor variables to a PL/SQL block for opening, the query work areas to which they point remain accessible after the block completes so that your OCI or Pro*C program can use these work areas for ordinary cursor operations.
- When finished, close the cursors.

Identifying Network Issues

- **Group OPEN-FOR statements when passing host cursor variables to PL/SQL.**
- **Use client-side PL/SQL when appropriate.**
- **Avoid unnecessary reparsing.**
- **Utilize array processing.**
- **Use table functions to improve performance.**
- **Use the RETURNING clause when appropriate.**



ORACLE

Copyright © 2004, Oracle. All rights reserved.

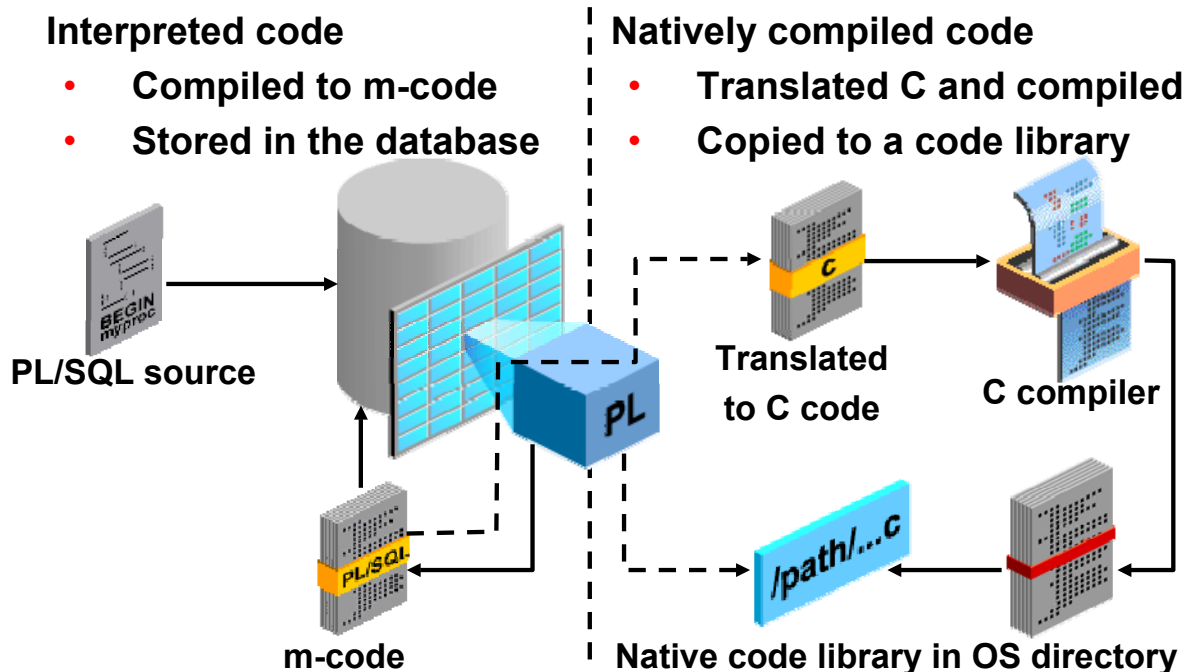
Guidelines for Reducing Network Traffic (continued)

- If your application is written using development tools that have a PL/SQL engine in the client tool, as in the Oracle Developer tools, and the code is not SQL intensive, reduce the load on the server by doing more of your work in the client and let the client-side PL/SQL engine handle your PL/SQL code.
- When a PL/SQL block is sent from the client to the server, the client can keep a reference to the parsed statement. This reference is the statement handle when using OCI, or the cursor cache entry when using precompilers. If your application is likely to issue the same code more than once, it needs to parse it only the first time. For all subsequent executions, the original parsed statement can be used, possibly with different values for the bind variables. This technique is more appropriate with OCI and precompilers because they give you more control over cursor processing.
In PL/SQL, this technique can be used with the `dbms_sql` package, in which the interface is similar to OCI. After a statement is parsed with `dbms_sql.parse`, it can be executed multiple times.

Guidelines for Reducing Network Traffic (continued)

- OCI and precompilers have the ability to send and retrieve data using host arrays. With this technique, large amounts of data can travel over the network as one unit rather than taking several trips. While PL/SQL does not directly use this array interface, if you are using PL/SQL from OCI or precompilers, take advantage of this interface.
- Use the RETURNING clause.
- By using table functions, rows of the result set can be returned a few at a time, reducing the memory overhead for producing large result sets within a function.

Native and Interpreted Compilation



ORACLE

Copyright © 2004, Oracle. All rights reserved.

Native and Interpreted Compilation

On the left of the vertical dotted line, a program unit processed as interpreted PL/SQL is compiled into machine-readable code (m-code), which is stored in the database and interpreted at run time.

On the right of the vertical dotted line, the PL/SQL source is subjected to native compilation, where the PL/SQL statements are compiled to m-code that is translated into C code. The m-code is not retained. The C code is compiled with the usual C compiler and linked to the Oracle process using native machine code library. The code library is stored in the database but copied to a specified directory path in the operating system, from which it is loaded at run time. Native code bypasses the typical run-time interpretation of code.

Note: Native compilation cannot do much to speed up SQL statements called from PL/SQL, but it is most effective for computation-intensive PL/SQL procedures that do not spend most of their time executing SQL.

You can natively compile both the supplied Oracle packages and your own PL/SQL code. Compiling all PL/SQL code in the database means that you see the speedup in your own code and all the built-in PL/SQL packages. If you decide that you will have significant performance gains in database operations using PL/SQL native compilation, Oracle recommends that you compile the whole database using the `NATIVE` setting.

Native and Interpreted Compilation (continued)

Features and Benefits of Native Compilation

The PL/SQL native compilation process makes use of a `makefile`, called `spnc_makefile.mk`, located in the `$ORACLE_HOME/plsql` directory. The `makefile` is processed by the Make utility that invokes the C compiler, which is the linker on the supported operating system, to compile and link the resulting C code into shared libraries. The shared libraries are stored inside the database and are copied to the file system. At run time, the shared libraries are loaded and run when the PL/SQL subprogram is invoked.

In accordance with Optimal Flexible Architecture (OFA) recommendations, the shared libraries should be stored near the data files. C code runs faster than PL/SQL, but it takes longer to compile than m-code. PL/SQL native compilation provides the greatest performance gains for computation-intensive procedural operations.

Examples of such operations are data warehouse applications and applications with extensive server-side transformations of data for display. In such cases, expect speed increases of up to 30%.

Limitations of Native Compilation

As stated, the key benefit of natively compiled code is faster execution, particularly for computationally intensive PL/SQL code, as much as 30% more. Consider that:

- Debugging tools for PL/SQL do not handle procedures compiled for native execution. Therefore, use interpreted compilation in development environments, and natively compile the code in a production environment.
- The compilation time increases when using native compilation, because of the requirement to translate the PL/SQL statement to its C equivalent and execute the Make utility to invoke the C compiler and linker for generating the resulting compiled code library.
- If many procedures and packages (more than 5,000) are compiled for native execution, a large number of shared objects in a single directory may affect performance. The operating system directory limitations can be managed by automatically distributing libraries across several subdirectories. To do this, perform the following tasks before natively compiling the PL/SQL code:
 - Set the `PLSQL_NATIVE_LIBRARY_SUBDIR_COUNT` database initialization parameter to a large value, such as 1,000, before creating the database or compiling the PL/SQL packages or procedures.
 - Create `PLSQL_NATIVE_LIBRARY_SUBDIR_COUNT` subdirectories in the path specified in the `PLSQL_NATIVE_LIBRARY_DIR` initialization parameter.

Switching Between Native and Interpreted Compilation

- **Setting native compilation**

- **For the system:**

```
ALTER SYSTEM SET plsql_compiler_flags='NATIVE';
```

- **For the session:**

```
ALTER SESSION SET plsql_compiler_flags='NATIVE';
```

- **Setting interpreted compilation**

- **For the system level:**

```
ALTER SYSTEM
    SET plsql_compiler_flags='INTERPRETED';
```

- **For the session:**

```
ALTER SESSION
    SET plsql_compiler_flags='INTERPRETED';
```



Copyright © 2004, Oracle. All rights reserved.

Switching Between Native and Interpreted Compilation

The `PLSQL_COMPILER_FLAGS` parameter determines whether PL/SQL code is natively compiled or interpreted, and determines whether debug information is included. The default setting is `INTERPRETED, NON_DEBUG`. To enable PL/SQL native compilation, you must set the value of `PLSQL_COMPILER_FLAGS` to `NATIVE`.

If you compile the whole database as `NATIVE`, then Oracle recommends that you set `PLSQL_COMPILER_FLAGS` at the system level.

To set compilation type at the system level (usually done by a DBA), execute the following statements:

```
ALTER SYSTEM SET plsql_compiler_flags='NATIVE';
ALTER SYSTEM SET plsql_compiler_flags='INTERPRETED';
```

To set compilation type at the session level, execute one of the following statements:

```
ALTER SESSION SET plsql_compiler_flags='NATIVE';
ALTER SESSION SET plsql_compiler_flags='INTERPRETED';
```

Switching Between Native and Interpreted Compilation (continued)

Parameters Influencing Compilation

In all circumstances, whether you intend to compile a database as `NATIVE` or you intend to compile individual PL/SQL units at the session level, you must set all required parameters.

The system parameters are set in the `initSID.ora` file by using the `SPFILE` mechanism.

Two parameters that are set as system-level parameters are the following:

- The `PLSQL_NATIVE_LIBRARY_DIR` value, which specifies the full path and directory name used to store the shared libraries that contain natively compiled PL/SQL code
- The `PLSQL_NATIVE_LIBRARY_SUBDIR_COUNT` value, which specifies the number of subdirectories in the directory specified by the `PLSQL_NATIVE_LIBRARY_DIR` parameter. Use a script to create directories with consistent names (for example, `d0`, `d1`, `d2`, and so on), and then the libraries are automatically distributed among these subdirectories by the PL/SQL compiler.

By default, PL/SQL program units are kept in one directory.

The `PLSQL_COMPILER_FLAGS` parameter can be set to a value of `NATIVE` or `INTERPRETED`, either as a database initialization for a systemwide default or for each session using an `ALTER SESSION` statement.

Summary

In this lesson, you should have learned how to:

- **Tune your PL/SQL application. Tuning involves:**
 - Using the `RETURNING` clause and bulk binds when appropriate
 - Rephrasing conditional statements
 - Identifying data type and constraint issues
 - Understanding when to use SQL and PL/SQL
- **Tune the shared pool by using the Oracle-supplied package `dbms_shared_pool`**
- **Identify network issues that impact processing**
- **Use native compilation for faster PL/SQL execution**

ORACLE

Copyright © 2004, Oracle. All rights reserved.

Summary

There are several methods that help you tune your PL/SQL application.

When tuning PL/SQL code, consider using the `RETURNING` clause and/or bulk binds to improve processing. Be aware of conditional statements with an `OR` clause. Place the fastest processing condition first. There are several data type and constraint issues that can help in tuning an application.

You can use the Oracle-supplied package `dbms_shared_pool` to pin frequently used packages, procedures, and functions to the shared pool.

You can reduce network traffic by:

- Reducing memory usage
- Using client-side PL/SQL
- Avoiding unnecessary parsing
- Utilizing array processing

By using native compilation, you can benefit from performance gains for computation-intensive procedural operations.

Practice Overview

This practice covers the following topics:

- **Pinning a package**
- **Tuning PL/SQL code to improve performance**
- **Coding with bulk binds to improve performance**

ORACLE

Copyright © 2004, Oracle. All rights reserved.

Practice Overview

In this practice, you will tune some of the code you have created for the OE application.

- Use `dbms_shared_pool` to pin a package in memory
- Break a previously built subroutine in smaller executable sections
- Pass collections into subroutines
- Add error handling for `BULK INSERT`

For detailed instructions about performing this practice, see Appendix A, “Practice Solutions.”

Practice 7

1. In this exercise, you will pin the fine-grained access package created in Lesson 6.
Note: If you have not completed practice 6, run the following files in the \$HOME/soln folder:

```
sol_06_02.sql
sol_06_03.sql
sol_06_04.sql
sol_06_05.sql
```

Using the DBMS_SHARED_POOL.KEEP procedure, pin your SALES_ORDERS_PKG.

Execute the DBMS_SHARED_POOL.SIZES procedure to see the objects in the shared pool that are larger than 500 kilobytes.

2. Open the lab_07_02.sql file and examine the package (the package body is shown below):

```
CREATE OR REPLACE PACKAGE BODY credit_card_pkg
IS

    PROCEDURE update_card_info
        (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no
        VARCHAR2)
    IS
        v_card_info typ_cr_card_nst;
        i INTEGER;
    BEGIN
        SELECT credit_cards
            INTO v_card_info
            FROM customers
            WHERE customer_id = p_cust_id;
        IF v_card_info.EXISTS(1) THEN -- cards exist, add more
            i := v_card_info.LAST;
            v_card_info.EXTEND(1);
            v_card_info(i+1) := typ_cr_card(p_card_type,
            p_card_no);
            UPDATE customers
                SET credit_cards = v_card_info
                WHERE customer_id = p_cust_id;
        ELSE -- no cards for this customer yet, construct one
            UPDATE customers
                SET credit_cards = typ_cr_card_nst
                    (typ_cr_card(p_card_type, p_card_no))
                WHERE customer_id = p_cust_id;
        END IF;
    END update_card_info;

-- continued on next page
```

Practice 7 (continued)

```
-- continued from previous page.
PROCEDURE display_card_info
  (p_cust_id NUMBER)
IS
  v_card_info typ_cr_card_nst;
  i INTEGER;
BEGIN
  SELECT credit_cards
    INTO v_card_info
    FROM customers
    WHERE customer_id = p_cust_id;
  IF v_card_info.EXISTS(1) THEN
    FOR idx IN v_card_info.FIRST..v_card_info.LAST LOOP
      DBMS_OUTPUT.PUT('Card Type: ' ||
v_card_info(idx).card_type || ' ');
      DBMS_OUTPUT.PUT_LINE('/ Card No: ' ||
v_card_info(idx).card_num );
    END LOOP;
  ELSE
    DBMS_OUTPUT.PUT_LINE('Customer has no credit cards. ');
  END IF;
END display_card_info;
END credit_card_pkg; -- package body
/
```

This code needs to be improved. The following issues exist in the code:

- The local variables use the INTEGER data type.
- The same SELECT statement is run in the two procedures.
- The same IF v_card_info.EXISTS(1) THEN statement is in the two procedures.

Practice 7 (continued)

3. To improve the code, make the following modifications:

Change the local INTEGER variables to use a more efficient data type.

Move the duplicated code into a function. The package specification for the modification is:

```
CREATE OR REPLACE PACKAGE credit_card_pkg
IS
    FUNCTION cust_card_info
        (p_cust_id NUMBER, p_card_info IN OUT typ_cr_card_nst )
        RETURN BOOLEAN;
    PROCEDURE update_card_info
        (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no
        VARCHAR2);
    PROCEDURE display_card_info
        (p_cust_id NUMBER);
END credit_card_pkg; -- package spec
/
```

Have the function return TRUE if the customer has credit cards. The function should return FALSE if the customer does not have credit cards. Pass into the function an uninitialized nested table. The function places the credit card information into this uninitialized parameter.

4. Test your modified code with the following data:

```
EXECUTE credit_card_pkg.update_card_info -
        (120, 'AM EX', 55555555555)
PL/SQL procedure successfully completed.
```

```
EXECUTE credit_card_pkg.display_card_info(120)
Card Type: Visa / Card No: 11111111
Card Type: MC / Card No: 2323232323
Card Type: DC / Card No: 4444444
Card Type: AM EX / Card No: 55555555555
```

```
PL/SQL procedure successfully completed.
```

```
-- Note: If you did not complete Practice 3, your results
-- will be:
```

```
EXECUTE credit_card_pkg.display_card_info(120)
Card Type: AM EX / Card No: 55555555555
```

```
PL/SQL procedure successfully completed.
```


Practice 7 (continued)

5. Open file lab_07_05a.sql. It contains the modified code from the previous question #3.

You need to modify the UPDATE_CARD_INFO procedure to return information (using the RETURNING clause) about the credit cards being updated. Assume that this information will be used by another application developer in your team, who is writing a graphical reporting utility on customer credit cards, after a customer's credit card information is changed.

Modify the code to use the RETURNING clause to find information about the row affected by the UPDATE statements.

You can test your modified code with the following procedure (contained in lab_07_05b.sql):

```
CREATE OR REPLACE PROCEDURE test_credit_update_info
(p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no NUMBER)
IS
    v_card_info typ_cr_card_nst;
BEGIN
    credit_card_pkg.update_card_info
        (p_cust_id, p_card_type, p_card_no, v_card_info);
END test_credit_update_info;
/
```

Test your code with the following statements set in boldface:

```
EXECUTE test_credit_update_info(125, 'AM EX', 123456789)
PL/SQL procedure successfully completed.
```

```
SELECT credit_cards FROM customers WHERE customer_id = 125;
CREDIT_CARDS(CARD_TYPE, CARD_NUM)
-----
TYP_CR_CARD_NST(TYP_CR_CARD('AM EX', 123456789))
```

Practice 7 (continued)

6. In this exercise, you will test exception handling with the `SAVE EXCEPTIONS` clause. Run the `lab_07_06a.sql` file to create a test table:

```
CREATE TABLE card_table
(accepted_cards VARCHAR2(50) NOT NULL);
```

Open the `lab_07_06b.sql` file and run the contents:

```
DECLARE
    type typ_cards is table of VARCHAR2(50);
    v_cards typ_cards := typ_cards
    ( 'Citigroup Visa', 'Nationscard MasterCard',
      'Federal American Express', 'Citizens Visa',
      'International Discoverer', 'United Diners Club' );
BEGIN
    v_cards.Delete(3);
    v_cards.DELETE(6);
    FORALL j IN v_cards.first..v_cards.last
        SAVE EXCEPTIONS
        EXECUTE IMMEDIATE
        'insert into card_table (accepted_cards) values (
        :the_card) '
        USING v_cards(j);
/
```

Note the output: _____

Practice 7 (continued)

6. (continued)

Open the lab_07_06c.sql file and run the contents:

```
DECLARE
type typ_cards is table of VARCHAR2(50);
v_cards typ_cards := typ_cards
( 'Citigroup Visa', 'Nationscard MasterCard',
  'Federal American Express', 'Citizens Visa',
  'International Discoverer', 'United Diners Club' );
bulk_errors EXCEPTION;
PRAGMA exception_init (bulk_errors, -24381 );
BEGIN
v_cards.Delete(3);
v_cards.DELETE(6);
FORALL j IN v_cards.first..v_cards.last
  SAVE EXCEPTIONS
  EXECUTE IMMEDIATE
    'insert into card_table (accepted_cards) values (
    :the_card)'
    USING v_cards(j);
EXCEPTION
WHEN bulk_errors THEN
  FOR j IN 1..sql%bulk_exceptions.count
  LOOP
    Dbms_Output.Put_Line (
      TO_CHAR( sql%bulk_exceptions(j).error_index ) || ':
      ' || SQLERRM(-sql%bulk_exceptions(j).error_code) );
  END LOOP;
END;
/
```

Note the output:_____

Why is the output different?

