Best Practices for PL/SQL

Objectives

At the end of this session, you will be able to:

- Know the Best Practices for PL/SQL Program
- Understand Modularity
- Learn Challenges of Local Module
- Learn Anchor Declarations of Variables
- Know the Use of Bind Variables
- Learn the Use of Cursors & Control Structures
- Use Bulk Collect Clause
- Use the FORALL Clause
- Know BEFORE VS AFTER TRIGGER
- Understand Optimizing SQL

Agenda

- Best Practices for PL/SQL Program
- Modularity
- Challenges of Local Module
- Learn Anchor Declarations of Variables
- Use of Bind Variables
- Cursors & Control Structures
- Bulk Collect Clause
- The FORALL Clause
- BEFORE VS AFTER TRIGGER
- Optimizing SQL

Best Practices for PL/SQL Programs

- A best practice workflow for single program construction
  - Focus on testing and improving the readability of one's code
- A best practice workflow for the application as a whole
  - Focus on how best to write (and not write) SQL

Modularity

- Improves software Reusability
- Hides the Complexity of execution of a specific operation behind a name
- We use top-down design to hide Complexity
- We use small and narrowly-focused packages
  - Encapsulate logically related types, items & subprograms in a named PL/SQL module
- Advantages of using Packages:
  - Easier Application Design
  - Overloading
  - Information Hiding
  - Better Performance

Modularity (Contd…)

Always:

- Reuse existing programs whenever possible
- Write tiny chunks (blocks) of code
- Build lots of Local or Nested Modules
  - Avoid Spaghetti code
  - Keep your executable sections small / tiny

Challenges of Local Module

- Requires Discipline
  - Always be on the lookout for opportunities to Refactor
  - Make conscious decisions about the scope of your programs
- Need to Read from the Bottom, Up
  - For a program with local modules, start at the Bottom (main executable section) and read Up
  - Takes some time getting used to
- Your IDE should reveal the Internal Program Structure
  - Toad's Program Navigator & SQL Navigator's Code Explorer are ideal facilitators of Top-down Design and Local Modularization

Challenges of Local Module (Contd…)

- Define Subprograms close to usage

- A subprogram can be defined at any of these levels:
  - Local - Within another Subprogram
  - Private - Defined in the Package Body
  - Public - Defined in the Package Specification

- The best rule to follow is:

  Define subprograms as close as possible to their usage.

  The shorter the distance from usage to definition, the easier it is to find, understand and maintain that code

Anchor Declarations of Variables

- 2 choices to declare a variable:
  - Hard-coding the data type
  - Anchoring the data type to another Structure

Hard-coded Declaration

```
ename  VARCHAR2(30);
totsales NUMBER (10,2);
```

Anchored Declarations

```
v_ename emp.ename%TYPE;
Totsales  pkg.sales_amt%TYPE;
```

- Prefer Anchored Declarations to Explicit data type references
  - %TYPE for Scalar Structures
  - %ROWTYPE for Composite Structures

Use of Bind Variables

- Avoids unnecessary Parsing
- A key to improved Code Performance
- PL/SQL code improves by several orders of magnitude, depending on the Type and Size of data

```
DECLARE
BEGIN
    FOR i IN 1 .. 1000 LOOP
       EXECUTE IMMED
IATE
                'SELECT item_id, qty
               FROM items
               WHERE qty = ' || i;
    END LOOP;
END;
```

```
DECLARE
BEGIN
    FOR i IN 1 .. 1000 LOOP
            EXECUTE IMMEDIATE
     'SELECT item_id, qty
     FROM items
            WHERE qty = :x' using i;
    END LOOP;
END;
```

Cursors & Control Structures

- Use of Explicit cursors for fetching data from a table
  - Faster
  - Exception Handling is easier
  - Greater programmatic control

- Use of Cursor FOR LOOP
  - No need to write OPEN, FETCH and CLOSE
  - Results in lesser Lines of Code & Errors

- An Exception is not raised if no rows are affected by an UPDATE or DELETE statement

  SQL%ROWCOUNT or SQL%FOUND is the only way to trap such kind of conditions

Cursors & Control Structures (Contd…)

- Select Your Control Structures carefully

Multiple Conditional Statements

```
IF quantity <= 5  THEN
  Shipping_method :=standard_shipping;
END IF;

IF quantity > 5  THEN
  Shipping_method :=
expedited_shipping;
END IF;
```

Replacing IF with ELSIF

```
IF quantity <= 5  THEN
  Shipping_method :=
standard_shipping;
ELSIF quantity > 5 THEN
Shipping_method :=
expedited_shipping;
END IF;
```

Case Statements

- A better alternative to Nested IF-ELSE statements
- Handy when number of conditions to check for is large
- Using CASE for mutually exclusive condition checking:

```
CASE quantity
        WHEN <=5 THEN shipping_method :=  standard_shipping;
        WHEN > 5 THEN shipping_method :=  expedited_shipping;
        ELSE shipping_method :=  0
END CASE;
```

Fetch into Cursor Records

```
name VARCHAR2 (30);
minbal NUMBER(10,2);
BEGIN
OPEN company_pkg.allrows;
FETCH company_pkg.allrows
INTO name, minbal;
IF name = 'ACME' THEN ...
CLOSE company_pkg.allrows;
```

Fetching into individual variables hard-codes number of items in select list.

```
rec
company_pkg.allrows%ROWTYPE;
BEGIN
OPEN company_pkg.allrows;
FETCH company_pkg.allrows INTO
rec;
IF rec.name = 'ACME' THEN ...
CLOSE company_pkg.allrows;
```

If the cursor select list changes, it doesn't necessarily affect your code

Memory Usage Analysis

- Complex data structures (collections, objects, records) can take up substantial amounts of memory
- Be aware of memory consumption
- Know how to analyze memory usage
- Adjust usage as needed
- Examine how you can do your own analysis

The BULK COLLECT Clause

- Use BULK COLLECT, % ATTRIBUTES wherever required
- BULK COLLECT loads multiple rows into the collections rather than one at a time
- Reduces strain on resources by reducing the pass to the database

```
CREATE OR REPLACE PROCEDURE
no_bulk_proc is
  CURSOR item_cur IS
     SELECT items.item_id, qty FROM items;

  item_rec item_cur%ROWTYPE;
BEGIN
    OPEN item_cur;
    LOOP
        FETCH item_cur INTO item_rec;
        EXIT WHEN item_cur%notfound;

dbms_output.put_line(item_rec.item_id);
        dbms_output.put_line(item_rec.qty);
    END LOOP;
END no_bulk_proc;
```

```
CREATE OR REPLACE PROCEDURE bulk_proc IS
  CURSOR item_cur IS
       SELECT item_id, qty FROM items;

  TYPE t_item IS TABLE OF ITEMS.item_id%TYPE
INDEX BY BINARY_INTEGER;
  TYPE t_qty IS TABLE OF ITEMS.qty%TYPE INDEX
BY BINARY_INTEGER;

  v_item t_item;
  v_qty t_qty;
BEGIN
      OPEN item_cur;
      FETCH item_cur bulk collect INTO v_item, v_qty
limit 100;

      FOR i IN v_item.first .. v_item.last LOOP
           dbms_output.put_line(v_item(i));
           dbms_output.put_line(v_qty(i));
      END LOOP;
        CLOSE item_cur;
END bulk_proc;
```

The FORALL Clause

- Used for bulk-bind input operations before sending them to SQL engine
- Used with inserts, updates & deletes
- Used for Moving data from collections to tables
- Instead of executing repetitive, individual DML statements, you can write your code like this:

```
PROCEDURE upd_for_dept (...)  IS
BEGIN
   FORALL indx IN list_of_emps.FIRST .. list_of_emps.LAST
      UPDATE employee
         SET salary = newsal_in
       WHERE employee_id = list_of_emps (indx);
END;
```

Triggers

BEFORE Vs AFTER Trigger:

- Never use BEFORE trigger for Validations
- Use BEFORE triggers ONLY to modify: NEW value
- AFTER row triggers are slightly more efficient than BEFORE row triggers
- With BEFORE row triggers, affected data blocks must be read (logical read, not physical read) once for the trigger and then again for the triggering statement
- With AFTER row triggers, the data blocks must be read only once for both the triggering statement and the trigger

Optimizing SQL in PL/SQL Programs

- SQL statements are among the most critical elements of our code base and should not be taken for granted
- SQL statements change constantly, forcing us to change code
- They cause most of the performance problems
- They cause many runtime errors in an application

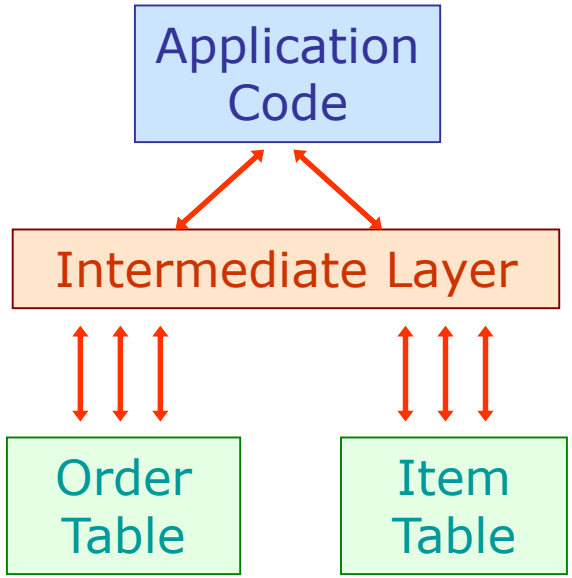
Writing SQL – Set the Rule

- Don't repeat the same logical SQL statement
  - Repetition makes it almost impossible to maintain & optimize
- Use PL/SQL only when you need it
  - "Straight" SQL is almost always faster and easier
- Don't write any SQL statements in the application code (user-facing) layer
  - Both on the front end and the back end
  - That is, no hard-coded SQL in Java or .Net code

Writing SQL – Systemize & Automate

- Think of SQL as a service that is provided to you, not something you write
  - Or if you write it, you put it somewhere so that it can be easily found, reused and maintained

- This service has programs defined in the Data Access Layer
  - Known as table APIs or data Encapsulation, these programs have intelligence about business transactions & underlying tables



Optimizing SQL in PL/SQL Programs (Contd…)

Tips to enhance performance in SQL scripts:

- Never do a calculation on an indexed column

  (Example, WHERE salary*5 > :myvalue)

- Use the UNION statement instead of OR conditions

- Avoid the use of NOT IN or HAVING in the WHERE clause. Instead, use the NOT EXISTS clause

- Specify numeric values in numeric form & character values in character form

Optimizing SQL in PL/SQL Programs (Contd…)

- Avoid specifying NULL in an indexed column
- Avoid the LIKE parameter if = will suffice. Using any Oracle function will invalidate the index, causing a full-table scan
- Never mix data types in Oracle queries, as it will invalidate the index. If the column is numeric, remember not to use quotes (e.g., salary = 50000). For char index columns, always use single quotes
- Oracle's rule-based optimizer looks at the order of table names in the FROM clause to determine the driving table
- Make sure that the last table specified in the FROM clause is the one that returns least rows
- In other words, specify multiple tables with the largest result set table specified first in the FROM clause

Optimizing SQL in PL/SQL Programs (Contd…)

- Avoid using sub-queries when a JOIN will do the job
- Use the DECODE function to minimize the number of times a table has to be selected
- WHERE EXISTS sub-queries are better than JOIN if the number of records in driver query are less
- WHERE EXISTS can be better than JOIN when driving from parent records and we want to make sure that at least one child exists

Optimizing SQL in PL/SQL Programs (Contd…)

- Try to group multiple sub queries into one
- Beware of Implicit data type conversions occurring on indexed columns
- NEVER use SELECT * FROM in application code
- Avoid using the DISTINCT clause as it always requires a sort operation
- Excessive use of DISTINCT clause may point to an underlying data model problem (e.g. a missing table)

Error Management

- Application code must raise, handle, log and communicate errors in a consistent, robust manner
- Error Management - Sets the Rules
- Application developers do not write their own exception handling and raising code
  - Instead, they call programs to do the work for them
- Don't hard-code -20NNN error codes and their related messages
  - And if you use the -20NNN codes, maintain a repository of the numbers you have used
- Give names to any system exceptions you need to handle. Avoid codes like this:

```
WHEN OTHERS THEN
        IF      SQLCODE = -24381 THEN...
        ELSIF  SQLCODE = -1855   THEN...
```

Best practices for Error Management

- Use package-based utilities to raise, handle and log exceptions
- Compensate for PL/SQL weaknesses
- Some general guidelines:
  - Avoid hard-coding error numbers and messages
  - Build and use reusable components for raising, handling and logging errors
- Application-level code should not contain:
  - RAISE_APPLICATION_ERROR: Don't leave it to the developer to decide how to raise
  - PRAGMA EXCEPTION_INIT: Avoid duplication of error definitions

Best Practices to Write SQL and PL/SQL

- CATG, BT16 team has written a basic document which describes the best coding practices to write SQL & PL/SQL

Summary

In this session, we have covered:

- Best practices for PL/SQL Program
- Modularity
- Challenges of Local Module
- Learn Anchor Declarations of Variables
- Use of Bind Variables
- Cursors & Control Structures
- Bulk Collect Clause
- The FORALL Clause
- BEFORE Vs AFTER Trigger
- Optimizing SQL

Thank You