CSE2DBF – CSE4DBF

Stored Procedures and Stored Functions

Basic PL/SQL (Procedural Language extensions to SQL)

- Before we start looking at some examples on stored procedures, we will go through basic PL/SQL syntax in the following few slides.
- A stored procedure in Oracle basically contains a set of SQL statements grouped together with PL/SQL syntax.
- PL/SQL structure:

```
DECLARE
        [local_variable_declarations]

BEGIN
        program body;

END ;
/
```

Basic PL/SQL: Local Variable Declaration

TYPE

- In many cases, a PL/SQL variable will be used to manipulate data stored in a database table. In this case, the variable should have the same type as the table column.
- For example the **`first_name`** column of the **`STUDENT`** table has type **VARCHAR2(20)**. Based on this we can declare a variable as follows;

```
DECLARE
V_FirstName        VARCHAR2(20);
```

But what if the table definition of **`first_name`** changed? For example to **VARCHAR2(25)**! Therefore to declare a variable of the same data type of that the table definition

```
DECLARE
V_Firstname STUDENT.first_name%type

Syntax
<variable name><table name>.<column name>%type
```

Basic PL/SQL: Local Variable Declaration

- Local variable declarations can be divided into two types:
  - Simple Variable

```
temp_name      VARCHAR2(30);
temp_salary    staff.salary%type;
```

  - Cursor Variable

```
CURSOR <cursor name> IS
   SELECT …
   FROM …
   WHERE … ;
```

STUDENT

student_id	first_name	last_name	department
101	John	Smith	Computer Science
102	Robert	Hodges	Computer Science
103	Maria	Lopez	Mathematics
104	Harry	Burke	Physics
105	Annie	Nguyen	Computer Science

STUDENT

student_id	first_name	last_name	department
101	John	Smith	Computer Science
102	Robert	Hodges	Computer Science
103	Maria	Lopez	Mathematics
104	Harry	Burke	Physics
105	Annie	Nguyen	Computer Science

```
DECLARE
        [local_variable_declarations]

BEGIN
        program body;

END ;
/
```

Basic PL/SQL: Program Body

- In the program body section, several different structure can be used, including:

  - IF-THEN-ELSE
  - WHILE - LOOP
  - Cursor Fetching:

```
FOR <variable name> IN <cursor name> LOOP
    ……
END LOOP;
```

  - Insert statement
  - Update statement
  - Delete statement

Basic PL/SQL: Program Body

- Whenever you want to display an output to the screen for debugging purpose, you can insert the following line in your PL/SQL block:

```
DBMS_OUTPUT.PUT_LINE('Data inserted …');
```

- Make sure you turn on the server output in SQL Plus by typing:

```
SQL> SET SERVEROUTPUT ON;
```

Basic PL/SQL: Simple Variable Assignment

Example

```
DECLARE

  V_string1 VARCHAR2(10);
  V_string2 VARCHAR2(15);
  V_numeric NUMBER;

BEGIN

  V_string1:='Hello';
  V_string2:= V_string1;
  V_numeric:=12;

  DBMS_OUTPUT.PUT_LINE('My first PL/SQL'||V_string2);

END;
/
```

Basic PL/SQL: IF-THEN-ELSE example

```
DECLARE
  v_NumberSeats ROOMS.number_seats%TYPE;
  v_Comment VARCHAR2(35);

BEGIN
  -- Retrieve the number of seats in the room identified
  -- by ID 99999. Store the result in v_NumberSeats.

  SELECT number_seats
  INTO v_NumberSeats
  FROM ROOMS
  WHERE room_id = 99999;

  IF v_NumberSeats < 50 THEN
    v_Comment := 'Fairly small';
  ELSE
    IF v_NumberSeats < 100 THEN
       v_Comment := 'A little bigger';
    ELSE
       v_Comment := 'Lots of room';
    END IF;
  END IF;

  INSERT INTO TEMP_TABLE (char_col)
         VALUES (v_Comment);
END;
/
```

ROOMS

room_id	number_seats
10000	100
20000	60
99999	20

TEMP_TABLE

char_col

Basic PL/SQL: IF-THEN-ELSE (ctd.)

NOTES:

Before you can compile the previous PL/SQL statements, you need to make sure that you have the `ROOMS` table, and the `TEMP_TABLE` in your database with all the necessary attributes. Make sure you also have a `room_id=99999` in your database (check the correspondence number_seats of this particular room).

On successful compilation, Oracle will display

```
'PL/SQL procedure successfully completed'
```

Check your `TEMP_TABLE` and see whether it has the correct data in it.

Basic PL/SQL: Loops

Simple Loops

```
DECLARE
      v_Counter NUMBER :=1;

    BEGIN
      LOOP
        -- Insert a row into TEMP_TABLE with the
        -- current value of the loop counter.

        INSERT INTO TEMP_TABLE
        VALUES (v_Counter, 'Loop index');

        v_Counter := v_Counter + 1;

        -- Exit condition - when the loop counter > 50 we will
        -- break out of the loop.

        IF v_Counter > 50 THEN
          EXIT;
        END IF;
      END LOOP;
END;
/
```

TEMP_TABLE

counter	comment

TEMP_TABLE

counter	comment
1	Loop Index
…	…
50	Loop Index

Basic PL/SQL: Loops

While Loops

```
DECLARE
  v_Counter NUMBER:=1;

BEGIN

  WHILE v_Counter <= 50 LOOP
    INSERT INTO TEMP_TABLE2
    VALUES (v_Counter, 'Loop index');
    v_Counter := v_Counter + 1;

  END LOOP;

END;
/
```

Basic PL/SQL: Loops

While Loops

```
DECLARE
  v_Counter NUMBER:=1;
  v_Counter2 NUMBER :=100;

BEGIN

  WHILE v_Counter <= 50 LOOP
    INSERT INTO TEMP_TABLE2
    VALUES (v_Counter, 'Loop index');
    v_Counter := v_Counter + 1;

        WHILE v_Counter2 >= 50 LOOP
           DBMS_OUTPUT.PUT_LINE (v_Counter2);
           v_Counter2 := v_Counter2 - 1;
        END LOOP;

  END LOOP;

END;
/
```

Stored Procedures

- A ***stored procedure*** is a self-contained schema object (database's metadata) that logically groups a set of SQL statements written in respective database and trigger language (eg. PL/SQL® in ORACLE™, PL/pgSQL in Postgres) to perform a *specific task*.

- The concept of *procedure* in PL/SQL is the same as that in other high level programming languages, when it is called, it may accept parameters, perform some operations, and return to the caller.

- To make your PL/SQL programs persistent (i.e. stored in the database), you need to create a stored procedure and encapsulate your PL/SQL statements into the stored procedure.

Stored Procedures: Syntax

```
CREATE [OR REPLACE] PROCEDURE <procedure name>
[(parameter [{IN | OUT | IN OUT}] type,.....,
 parameter [{IN | OUT | IN OUT}] type)] AS

        [local_variable_declarations]

BEGIN
        procedure_body;

END <procedure name>;
/
```

Stored Procedures: Syntax

Mode {IN | OUT | IN OUT} description

IN

The value of the actual parameter is passed into the procedure when the procedure is invoked. Inside the procedure, the formal parameter is considered **read-only**; it cannot be changed. Then the procedure finishes and control returns to the calling environment, the actual parameter is not changed.

OUT

Any value the actual parameter has when the procedure is called is ignored. Inside the procedure, the formal parameter is considered **write-only**; it can only be assigned to and cannot be read from. When the procedure finishes and control returns to the calling environment, the contents of the formal parameter are assigned to the actual parameter.

IN OUT

This mode is a combination of IN and OUT. The value of the actual parameter is passed into the procedure when the procedure is invoked. Inside the procedure, the formal parameter can be read from and written to. When the procedure finishes and control returns to the calling environment, the contents of the formal parameter are assigned to the actual parameter

Stored Procedures: Example 1

```
CREATE OR REPLACE PROCEDURE ModeTest (
  p_InParameter    IN NUMBER,
  p_OutParameter   OUT NUMBER,
  p_InOutParameter IN OUT NUMBER) AS

  v_LocalVariable  NUMBER;
BEGIN
  -- Assign p_InParameter to v_LocalVariable. This is legal, since we are reading from an IN
  -- parameter and not writing   to it.
  v_LocalVariable := p_InParameter;  -- Legal

  -- Assign 7 to p_InParameter. This is ILLEGAL, since we are writing to an IN parameter.
  p_InParameter := 7;  -- Illegal

  -- Assign 7 to p_OutParameter. This is legal, since we are writing to an OUT parameter and
  -- not reading from it.
  p_OutParameter := 7;  -- Legal

  -- Assign p_OutParameter to v_LocalVariable. This is
  -- ILLEGAL, since we are reading from an OUT parameter.
  v_LocalVariable := p_outParameter;  -- Illegal

  -- Assign p_InOutParameter to v_LocalVariable. This is legal,
  -- since we are reading from an IN OUT parameter. */
  v_LocalVariable := p_InOutParameter;  -- Legal

  -- Assign 7 to p_InOutParameter. This is legal, since we are writing to an IN OUT parameter.
  p_InOutParameter := 7;  -- Legal
END ModeTest;
```

Stored Procedures: Example 2

```
CREATE OR REPLACE PROCEDURE AddNewStudent (
  p_ID          STUDENTS.id%TYPE,
  p_FirstName   STUDENTS.first_name%TYPE,
  p_LastName    STUDENTS.last_name%TYPE,
  p_Major       STUDENTS.major%TYPE) AS

BEGIN
  -- Insert a new row in the students table. Use
  -- 0 for total_current_credits.

      INSERT INTO STUDENTS (ID, first_name, last_name,
                            major, total_current_credits)
      VALUES (p_ID, p_FirstName, p_LastName, p_Major, 0);

END AddNewStudent;
/
```

STUDENTS

id	first_name	last_name	major	total_current_credits

Stored Procedures: Example 3 - exception

```
CREATE OR REPLACE PROCEDURE credit_account (acct NUMBER, credit NUMBER) AS

/* This procedure accepts two arguments: an account number and an amount of
money to credit to the specified account. If the specified account does not
exist, a new account is created. */

        old_balance NUMBER;
        new_balance NUMBER;

BEGIN
        SELECT balance INTO old_balance
        FROM ACCOUNTS
        WHERE acct_id = acct;

        new_balance := old_balance + credit;

        UPDATE ACCOUNTS SET balance = new_balance
        WHERE acct_id = acct;

        EXCEPTION
                WHEN NO_DATA_FOUND THEN
                        INSERT INTO ACCOUNTS (acct_id, balance)
                        VALUES (acct, credit);
END credit_account;
/
```

ACCOUNTS

acct_id	acct_name	balance
1001	Rick G	200
1002	Daryl D	500
1003	Carol P	1000

```
execute credit_account ('1001', 500);

execute credit_account ('1004', 500);
```

Stored Procedures: Cursors

- In order to process a SQL statement, Oracle will allocate an area of memory known as the ***context area***. The context area contains information necessary to complete the processing, including the number of rows processed by the statement, a pointer to the parsed representation of the statement, and in the case of a query, the *active* ***set***, which is the set of rows returned by the query.

- A ***cursor*** is a handle, or pointer, to the context area. Through the cursor, a PL/SQL program can control the context area and what happens to it as the statement is processed. The following PL/SQL block illustrates a cursor fetch loop, in which multiple rows of data are returned from a query.

Stored Procedures: Cursors

EMPLOYEE

eName	eSalary
Donnie	2000
Robert	3000
Julio	4000

SYNTAX: Declaring a CURSOR

```
CURSOR <cursor_name> IS
     SELECT statement;
```

Example:

```
CREATE OR REPLACE PROCEDURE salcheck(MinSalary number) AS

   CURSOR executive IS
      SELECT eName, eSalary
      FROM EMPLOYEE
      WHERE eSalary > MinSalary;

 BEGIN
   FOR v_cursrec IN executive LOOP

      dbms_output.put_line
         (v_cursrec.eName||' '|| v_cursrec.eSalary);
   END LOOP;

END salcheck;
/
```

execute salCheck (2000);

executive

eName	eSalary
Robert	3000
Julio	4000

Stored Procedures: Cursors

TEMP_TABLE

desc_col

Insert into TEMP_TABLE, students name and the total credits only for students who have completed more than 300 credit points.

```
CREATE OR REPLACE PROCEDURE CheckStudentCompletion AS
  CURSOR c_student IS
      SELECT id, last_name, total_current_credits
      FROM STUDENTS
      where total_current_credit > 300;

BEGIN

 FOR v_StudentRecord IN c_student LOOP
        INSERT INTO TEMP_TABLE (desc_col)
        VALUES (v_StudentRecord.id ||      '  '
                ||v_StudentRecord.last_name
                ||' final semester student!');
END LOOP;

END CheckStudentCompletion;
/
```

STUDENTS

id	last_name	total_current_credits
100	Doe	200
200	Wood	320
300	Nguyen	320
400	Perez	360

EXCEPTION: TOO_MANY_ROWS

```
CREATE OR REPLACE PROCEDURE TestException AS
  v_LastName   STUDENTS.last_name%TYPE,

BEGIN

 Select last_name
 into v_LastName
 from students;

EXCEPTION
   WHEN TOO_MANY_ROWS THEN
        ;


END CheckStudentCompletion;
/
```

STUDENTS

id	last_name	total_current_credits
100	Doe	200
200	Wood	320
300	Nguyen	320
400	Perez	360

Stored Procedures: Execute & Drop

- To display the results of the previous stored procedure we can do the following:

```
EXECUTE procedure name [param1,param2..];
```

- To delete the stored procedure we can do the following:

```
DROP PROCEDURE procedure name;
```

Stored Function: Syntax

```
CREATE [OR REPLACE] FUNCTION <function name>
[(parameter [{IN | OUT | IN OUT}] type,....,
 parameter [{IN | OUT | IN OUT}] type)]
 RETURN <return type> IS
       [local_variable_declarations]

BEGIN

              function_body;

END <function name>;
/
```

Stored Function

- *A function is very similar to a procedure, however, a procedure call is a PL/SQL statement by itself, while a function call is called as **part of an expression**. The RETURN statement is used to return control to the calling environment with a value.*

```
CREATE OR REPLACE FUNCTION ClassInfo (
  /* Returns 'Full' if the class is completely full,
     'Some Room' if the class is over 80% full,
     'More Room' if the class is over 60% full,
     'Lots of Room' if the class is less than 60% full, and
     'Empty' if there are no students registered. */
  p_Department classes.department%TYPE,
  p_Course     classes.course%TYPE)
  RETURN VARCHAR2 IS

  v_CurrentStudents NUMBER;
  v_MaxStudents     NUMBER;
  v_PercentFull     NUMBER;

BEGIN
        << Function Body is in the NEXT SLIDE >>
END ClassInfo;
/
```

Stored Function (ctd.)

```
BEGIN
  -- Get the current and maximum students for the requested
  -- course.
  SELECT current_students, max_students
    INTO v_CurrentStudents, v_MaxStudents
    FROM CLASSES
    WHERE department = p_Department
    AND course = p_Course;

  -- Calculate the current percentage.
  v_PercentFull := (v_CurrentStudents / v_MaxStudents) * 100;

IF v_PercentFull = 100 THEN
    RETURN 'Full';
  ELSIF v_PercentFull > 80 THEN
    RETURN 'Some Room';
  ELSIF v_PercentFull > 60 THEN
    RETURN 'More Room';
  ELSIF v_PercentFull > 0 THEN
    RETURN 'Lots of Room';
  ELSE
    RETURN 'Empty';
  END IF;
END ClassInfo;
/
```

CLASSES

Department	Course	Current_Students	Max_ Students
CSCE	BIT	180	200
CSCE	BCS	50	50
Maths	BSc	40	60
Physics	BSc	0	20

Stored Function (ctd.) – using CASE

```
BEGIN
  -- Get the current and maximum students for the requested
  -- course.
  SELECT current_students, max_students
    INTO v_CurrentStudents, v_MaxStudents
    FROM CLASSES
    WHERE department = p_Department
    AND course = p_Course;

  -- Calculate the current percentage.
  v_PercentFull := (v_CurrentStudents / v_MaxStudents) * 100;

CASE
WHEN v_PercentFull = 100 THEN
    RETURN 'Full';
WHEN v_PercentFull> 80 THEN
    RETURN 'Some Room';
WHEN v_PercentFull > 60 THEN
    RETURN 'More Room';
WHEN v_PercentFull > 0 THEN
    RETURN 'Lots of Room';
ELSE
    RETURN 'Empty';
END CASE;
END ClassInfo;
/
```

CLASSES

Department	Course	Current_Students	Max_ Students
CSCE	BIT	180	200
CSCE	BCS	50	50
Maths	BSc	40	60
Physics	BSc	0	20

Stored Function (ctd.)

To execute the previous function, we need to use an SQL statement:

```
SELECT department, course, classinfo(department, course)
FROM classes;
```

```
CASE
WHEN v_PercentFull = 100 THEN
    RETURN 'Full';
WHEN v_PercentFull> 80 THEN
    RETURN 'Some Room';
WHEN v_PercentFull > 60 THEN
    RETURN 'More Room';
WHEN v_PercentFull > 0 THEN
    RETURN 'Lots of Room';
ELSE
    RETURN 'Empty';
END CASE;
```

Department	Course	Classinfo(department, course)
CSCE	BIT	Some Room
CSCE	BCS	Full
Maths	BSc	More Room
Physics	BSc	Empty

CLASSES

Department	Course	Current_Students	Max_ Students
CSCE	BIT	180	200
CSCE	BCS	50	50
Maths	BSc	40	60
Physics	BSc	0	20

Calling a stored function from a procedure

```
CREATE OR REPLACE PROCEDURE RecordAlmostFullClasses AS
  CURSOR c_Classes IS
      SELECT department, course
      FROM CLASSES
      where ClassInfo(department, course))='Some Room'

BEGIN
  FOR v_ClassRecord IN c_Classes LOOP
    -- Record all classes which don't have very much room left
    -- in TEMP_TABLE. ClassInfo is a function given as an example
    -- in the previous slide.

       INSERT INTO temp_table (char_col) VALUES
       (v_ClassRecord.department || ' ' || v_ClassRecord.course ||
        ' is almost full!');
    END IF;
  END LOOP;
END RecordAlmostFullClasses;
/
```

TEMP_TABLE

char_col

CLASSES

Department	Course	Current_Students	Max_ Students
CSCE	BIT	180	200
CSCE	BCS	50	50
Maths	BSc	40	60
Physics	BSc	0	20

Stored Procedures and Functions

- To list all procedures in the user schema

```
SELECT name
FROM user_objects
WHERE OBJECT_TYPE = PROCEDURE;
```

- To list the code of a procedure in the user schema

```
SELECT *
FROM user_source
WHERE NAME='AddNewStudent';
```

- To list the error(s) of a procedure in the user schema

```
SELECT *
FROM user_errors
WHERE NAME='AddNewStudent';
```

Stored Procedures

- You should design stored procedures so that they have the following properties:
  - Define procedures to complete a single, focused task.
  - DO NOT define procedures that duplicate the functionality already provided by the database language.

Stored Procedures

- Advantages of using stored procedures
  - Security - Stored procedures can help enforce data security. You can restrict the database operations that users can perform by allowing them to access data only through procedures and functions.
  - Performance - A stored procedure can improve database performance because it reduces the amount of information that must be sent over network compared to issuing individual SQL statements. The information is send only once and thereafter invoked when it is used.
  - Memory allocation - Shared memory - only a single copy of the stored procedures needs to be loaded into memory for execution by multiple users.

Stored Procedures

- Advantages of using stored procedures (ctd.)
  - Modular Design - Stored procedures can be shared by applications that access the same database, eliminating duplicate code, coding errors and reducing the size of applications. This will increase the overall productivity.
  - Streamlined Maintenance - When a procedure is updated, the changes are automatically reflected in all applications that use it without the need to re-compile and re-link them. They are compiled and optimised only once for each client.
  - Integrity - Developing all applications around a common group of procedures helps to reduce coding errors and provide consistency of data access across all applications.

Next Lecture

Triggers