Birla Institute of Technology and Science, Pilani.
Database Systems
Lab No #7

Today's Topics:

- Transactions
- PL/SQL

TRANSACTIONS

- Databases are all about sharing data, so it is common for multiple users to be accessing and even changing the same data at the same time. The simultaneous execution of operations is called concurrency. Sometimes concurrency can get us into trouble if our changes require multiple SQL statements. In general, if two or more users access the same data and one or more of the statements changes the data, we have a conflict. This is a classic problem in database systems; it is called the isolation or serializability problem. If the users perform multiple steps, conflicts can cause incorrect results to occur. To deal with this problem, databases allow the grouping of a sequence of SQL statements into an indivisible unit of work called a transaction. A transaction ends with either a commit or a rollback:
    - **Commit—**A commit permanently stores all of the changes performed by the transaction.
    - **Rollback—**A rollback removes all of the updates performed by the transaction, no matter how many rows have been changed. A rollback can be executed either by the DBMS to prevent incorrect actions or explicitly by the user.

- The DBMS provides the following guarantees for a transaction, called the ACID properties: Atomicity, consistency, Isolation, and durability. These properties will be covered in the course in detail.
- SQL starts a transaction automatically when a new statement is executed if there is no currently active transaction. This means that a new transaction begins automatically with the first statement after the end of the previous transaction or the beginning of the session.
- A user may explicitly start a transaction using the START TRANSACTION statement.

```
SET TRANSACTION NAME <string>; --in oracle
```

- Commit:

```
SET TRANSACTION NAME ‘t1’;
UPDATE vrs SET inventory = inventory + 10;
SELECT * FROM vrs;
COMMIT;
SELECT * FROM vrs;
```

- Rollback:

```
SET TRANSACTION NAME ‘t2’;
UPDATE vrs SET inventory = inventory -20;
SELECT * FROM vrs;
ROLLBACK;
```

```
SELECT * FROM vrs;
```

- Here note that without using start transaction, if you execute the update query alone, the values are automatically committed. There is no way to revert back.

SAVEPOINTS

- SQL allows you to create named placeholders, called savepoints, in the sequence of statements in a transaction. You can rollback to a savepoint instead of to the beginning of the transaction. Only the changes made after the savepoint are undone. To set a savepoint, use the SAVEPOINT command:

```
SAVEPOINT <savepoint name>
```

- If we create a savepoint, we can rollback to that savepoint with the following:

```
ROLLBACK TO SAVEPOINT <savepoint name>
```

- Executing ROLLBACK without designating a savepoint or executing a COMMIT deletes all savepoints back to the start of the transaction. A rollback to a particular savepoint deletes all intervening savepoints.

```
UPDATE vrs SET inventory = inventory + 25;
SELECT * FROM vrs;
SAVEPOINT spoint1;
UPDATE vrs SET inventory = inventory - 15;
SELECT * FROM vrs;
SAVEPOINT spoint2;
UPDATE vrs SET inventory = inventory + 30;
SELECT * FROM vrs;
SAVEPOINT spoint3;
ROLLBACK TO SAVEPOINT spoint1;
SELECT * FROM vrs;
```

EXERCISES ON EMPLOYEES DATABASE:

- Write a queries for each of the following based on employee database created in the previous labs. The scheme for employee database is shown below.

1. Delete the Administration department and all of its subdepartments without using transactions.
2. Using a transaction, delete the Administration department and all of its subdepartments.

PL/SQL

Overview of PL/SQL

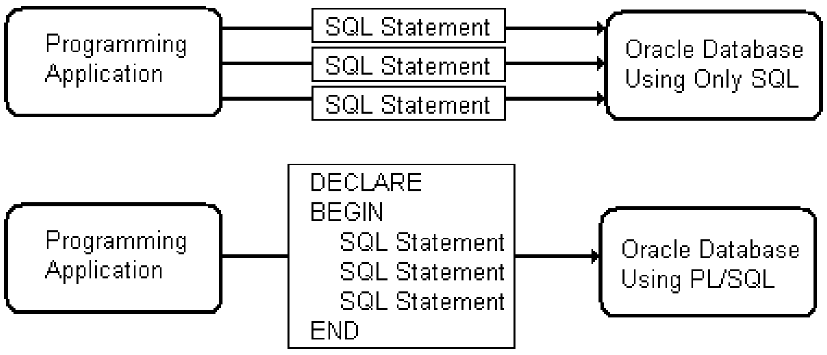
PL/SQL is a block-structured language. That is, the basic units (procedures, functions, and anonymous blocks) that make up a PL/SQL program are logical blocks, which can contain any number of nested sub-blocks. Typically, each logical block corresponds to a problem or sub problem to be solved. PL/SQL is not case-sensitive.

PL/SQL provides additional capabilities that SQL lacks:

- Secure code through encryption and by storing code on a server instead of a client computer.
- Handle exceptions that arise due to data entry errors or programming errors.
- Process record with iterative loop code that manipulates records one at a time.
- Work with variables, records, arrays, objects, and other common programming language constructs.

These are the advantages of PL/SQL.

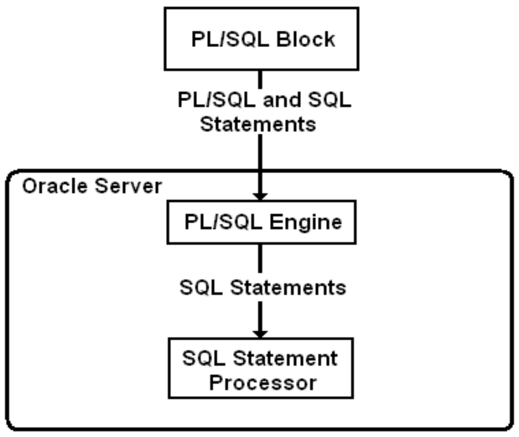
- **Better Application Performance:** PL/SQL provides the capability to define a "block" of programming statements, and can transmit this block to an Oracle database as a unit of work. When multiple SELECT statements are part of a PL/SQL block, there are still only two network trips. This improves system response time by decreasing the amount of network and performance overhead that is incurred with an approach where each SELECT statement processes individually.



- **Productivity, Portability, and Security:** PL/SQL stored procedures run on a server instead of a client computer. This means that the procedures can be secured from tampering by unscrupulous hackers by restricting access through the use of standard Oracle database security. Consider a typical business requirement to update a customer order. Instead of granting access to the customer order table, we can grant access to a procedure that has been coded to update the table.

PL/SQL BLOCK STRUCTURE

A PL/SQL blocks combine statements that represent a single logical task. When the PL/SQL engine is located on the server, the entire PL/SQL block is passed to the PL/SQL engine on the Oracle server. PL/SQL blocks can be divided into two groups: *named* and *anonymous*.



An anonymous block executes by first sending the block to the PL/SQL engine on the server where it is compiled. In contrast, a named PL/SQL block is compiled only at the time of its creation, or if it has been modified.

<u>Structure of a PLSQL program:</u>

```
DECLARE
    Declaration statements
BEGIN
    Executable statements
EXCEPTION
    Exception-handling statements
END;
```

/* Declare and Exception parts are optional. Every statement must be terminated by semicolon. Identifiers start with alphabets. Comments as shown. Reserved words cannot be used. Statements may span multiple lines*/

<u>To enable output to console window in Oracle SQL Developer</u>

From the menu go to: View > Dbms Output.
From the new window (popped-up in the log area) click on the plus sign to enable output.

anonymous block

```
DECLARE
     num_age     NUMBER(3) := 20;  -- assign value to variable
BEGIN
     num_age := 20;
     DBMS_OUTPUT.PUT_LINE('My age is: ' || TO_CHAR(num_age));
END;
```

Declarations:

Our program stores values in variables and constants. As the program executes, the values of variables can change, but the values of constants cannot.

Data Type	**Usage**	**Sample Declaration**
VARCHAR2	Variable-length character strings	FirstName VARCHAR2(30);
CHAR	Fixed-length character strings	StudentGender CHAR(1);
NUMBER	Floating, fixed-point, or integer numbers	Salary NUMBER(6);
PLS_INTEGER	Integers for indexing purpose.	StudentID PLS_INTEGER;
DATE	Dates	TodaysDate DATE;
BOOLEAN	TRUE / FALSE / NULL values	OrderFlag BOOLEAN;
LOB (It has four data types: BFILE, BLOB, CLOB and NCLOB)	Large Objects	Message CLOB;
%TYPE	Assumes the data type of the database field	CustAddress customer.cadd%TYPE;
%ROWTYPE	Assumes the data type of a database row	CustOrderRecord cust_order%ROWTYPE;

```
HIREDATE     DATE;
ROOTTWO        CONSTANT NUMBER: = 1.414
ACCT_ID      VARCHAR2 (5) NOT NULL: = 'PS001';
ingredients INGREDIENTS%ROWTYPE
foodgrp           ingredients.foodgroup%type
```

The first declaration names a variable of type DATE. The second declaration names a variable of CONSTANT type NUMBER and uses the assignment operator (:=) to assign an initial value of 1.414 to the variable. The third declaration names a variable of type VARCHAR2, specifies the NOT NULL constraint, and assigns an initial value of 'PS001' to the variable.

how to use data types

```
DECLARE
   wages           NUMBER;
   hours_worked    NUMBER := 40;
   hourly_salary   NUMBER := 22.50;
   bonus           NUMBER := 150;
   country         VARCHAR2(128);
   counter         NUMBER := 0;
   done            BOOLEAN;
   valid_id        BOOLEAN;
  TYPE myarr IS VARRAY(10) of NUMBER; --array
 TYPE commissions IS TABLE OF NUMBER INDEX BY VARCHAR2(10); --
associative array
  t_arr        myarr:= myarr();
  comm_tab       commissions;
BEGIN
```

```
  t_arr.extend; --append element
  t_arr(1):=1;
  t_arr.extend(2);
  t_arr(2):=1;
  t_arr(3):=1.9;
 wages := (hours_worked * hourly_salary) + bonus;
   country := 'France';
   done := (counter > 100);
   valid_id := TRUE;
   comm_tab('France') := 20000 * 0.15;
        DBMS_OUTPUT.PUT_LINE( to_char(wages) || ' ' || country || ' ' 
|| to_char(comm_tab('France')) );
END;
```

Error Handling: This section contains statements that are executed whenever a runtime error occurs within the block. When a runtime error occurs, program control is passed to the exception-handling section of the block. The runtime error is then evaluated, and a specific exception is raised or executed..

<u>Exceptions :</u>

Exceptions are of two types : user defined and predefined.

```
declare
creditlimit number(20);
withdrawal_amount number(20);
over_withdrawal exception;  --user defined exception
begin
if  creditlimit < withdrawal_amount then
      raise over_withdrawal;
            end  if;
exception
when  over_withdrawal then
DBMS_OUTPUT.PUT_LINE (‘balance insufficient’);
end;
```

Important predefined exceptions are NO_DATA_FOUND (when the query did not return any row) and TOO_MANY_ROWS (when the query returned more than one row). we don’t have to declare these exceptions . If we want to catch other exceptions (other predefined exceptions we don’t know but we want a default action for) then we can use the default exception handler OTHERS.

```
EXCEPTION
WHEN OTHERS THEN
ROLLBACK;
END;
```

Control Constructs

```
IF <condition> THEN <actions> [ ELSEIF <condition> THEN <actions>][ELSE
<actions>] ENDIF;
```

```
LOOP <sequence of statements>
IF <condition> THEN EXIT;
ENDIF;
END LOOP;
```

Important: there is no default exiting of loop so we **must** specify an explicit exit condition.

```
WHILE <condition is true> LOOP
               < sequence of statement >
END LOOP;
```

```
 FOR <counter> IN <lower bound> …<higher bound> LOOP
               <sequence of statement> END LOOP
```

example of if

```
DECLARE
    v_PurchaseAmount NUMBER(9,2) := 1001;
    v_DiscountAmount NUMBER(9,2);
BEGIN
    IF NOT (v_PurchaseAmount <= 1000) THEN
        v_DiscountAmount := v_PurchaseAmount * 0.05;
    END IF;
    DBMS_OUTPUT.PUT_LINE('Discount: ' || TO_CHAR(v_DiscountAmount));
END;
```

example of if else

```
DECLARE
    v_CustomerStatus CHAR(3) := '&CustomerStatus';
    v_PurchaseAmount NUMBER(9,2) := '&PurchaseAmount';
    v_DiscountAmount NUMBER(9,2);
BEGIN
    IF v_CustomerStatus = 'AAA' THEN
        IF v_PurchaseAmount > 1000 then
            v_DiscountAmount := v_PurchaseAmount * 0.05;
        ELSE
            v_DiscountAmount := v_PurchaseAmount * 0.02;
        END IF;
    ELSE
        v_DiscountAmount := 0;
    END IF;
    DBMS_OUTPUT.PUT_LINE('Discount: ' || TO_CHAR(v_DiscountAmount));
END;
/
```

Example of LOOP:

```
DECLARE
     v_Balance  NUMBER(9,2) := 100;
BEGIN
    LOOP
        v_Balance := v_Balance - 15;
        IF v_Balance <= 0 THEN
            EXIT;
        END IF;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('You may have paid too much.');
    DBMS_OUTPUT.PUT_LINE('Ending balance: ' || TO_CHAR(v_Balance));
END;
```

Example of WHILE LOOP:

```
DECLARE
            v_Counter NUMBER := 1;
BEGIN
     WHILE v_Counter < 5 LOOP
                DBMS_OUTPUT.PUT_LINE('Count = ' || TO_CHAR(v_Counter));
                 v_Counter := v_Counter + 1;
     END LOOP;
END;
```

Example of FOR LOOP:

```
DECLARE
     v_Rate NUMBER(5,4) := 0.06/12;
     v_Periods NUMBER := 12;
     v_Balance NUMBER(9,2) := 0;
BEGIN
    FOR i IN 1..v_Periods LOOP   -- loop number of periods
         v_Balance := v_Balance + 50;
         v_Balance := v_Balance + (v_Balance * v_Rate);
          DBMS_OUTPUT.PUT_LINE('Balance for Period ' || TO_CHAR(i) || '
' || TO_CHAR(v_Balance));
     END LOOP;
END;
```

EXERCISES:

I. Write a PL/SQL code to insert data into the temp table. If the entered number is greater than 25 ignore it. If the entered number is less than 25 then insert one row for each number starting from one. Insert (1) in (Num_store), (2) in (Num_store) & so on for all the numbers except 5, 10, 15 & 20. As soon as the number becomes five enter (5, FIVE) in (Num_store, Char_store). If it becomes 10 enter (10, Ten) in (Num_store, Char_store). If it becomes 15 enter (15, FIFTEEN, SYSDATE) in (Num_store, Char_store, Date_store) & when it becomes 20 enter (20, TWENTY) in (Num_store, Char_store).

Table: Temp

Field name	Width
Num_store	number(2)
Char_store	Varchar2(15)
Date_store	Date

---------&----------