Unit 3 Performing SQL Operations from PL/SQL

3.1 Overview of SQL Support in PL/SQL

By extending SQL, PL/SQL offers a unique combination of power and ease of use. You can manipulate Oracle data flexibly and safely because PL/SQL fully supports all SQL data manipulation statements, transaction control statements, functions, and operators. PL/SQL also conforms to the current ANSI/ISO SQL standard.

- **Data Manipulation**

To manipulate Oracle data you can include DML operations, such as INSERT, UPDATE, and DELETE statements, directly in PL/SQL programs, without any special notation. You can also include the SQL COMMIT statement directly in a PL/SQL program.

- **Transaction Control**

Oracle is transaction oriented; that is, Oracle uses transactions to ensure data integrity. A transaction is a series of SQL data manipulation statements that does a logical unit of work

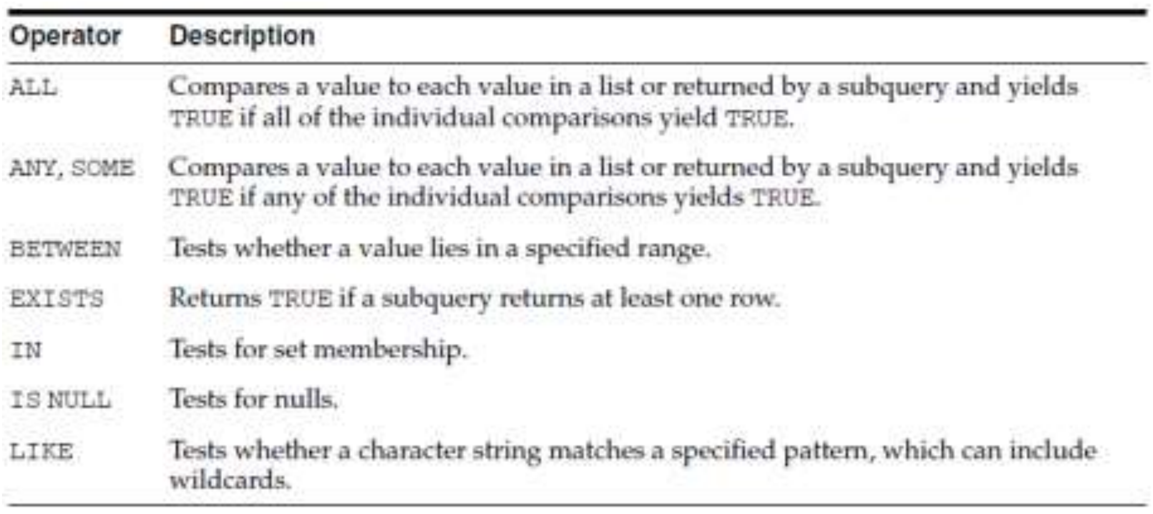
You use the COMMIT, ROLLBACK and SAVEPOINT commands to control transactions. COMMIT makes permanent any database changes made during the current transaction. ROLLBACK ends the current transaction and undoes any changes made since the transaction began. SAVEPOINT marks the current point in the processing of a transaction. Used with ROLLBACK, SAVEPOINT undoes part of a transaction.

- **SQL Functions**

Example: Calling the SQL COUNT Function in PL/SQL

```
DECLARE

job_count NUMBER;

emp_count NUMBER;

BEGIN

SELECT COUNT(DISTINCT job_id) INTO job_count FROM employees;

SELECT COUNT(*) INTO emp_count FROM employees;

END;
```

- **SQL Operators**
    1. **Comparison Operators**



Operator	Description
ALL	Compares a value to each value in a list or returned by a subquery and yields TRUE if all of the individual comparisons yield TRUE.
ANY, SOME	Compares a value to each value in a list or returned by a subquery and yields TRUE if any of the individual comparisons yields TRUE.
BETWEEN	Tests whether a value lies in a specified range.
EXISTS	Returns TRUE if a subquery returns at least one row.
IN	Tests for set membership.
IS NULL	Tests for nulls.
LIKE	Tests whether a character string matches a specified pattern, which can include wildcards.

2. **Set Operators**

Set operators combine the results of two queries into one result. INTERSECT returns all distinct rows selected by both queries. MINUS returns all distinct rows selected by the first query but not by the second. UNION returns all distinct rows selected by either query. UNION ALL returns all rows selected by either query, including all duplicates.

3. **Row Operators**

Row operators return or reference particular rows. ALL retains duplicate rows in the result of a query or in an aggregate expression. DISTINCT eliminates duplicate rows from the result of a query or from an aggregate expression. PRIOR refers to the parent row of the current row returned by a tree-structured query.

3.2 Overview of Oracle Transactions

Transaction processing with PL/SQL using SQL COMMIT, SAVEPOINT, and ROLLBACK statements that ensure the consistency of a database.

- **COMMIT –** The COMMIT statement ends the current transaction, making any changes made during that transaction permanent, and visible to other users.

  Following is commit command's syntax,

  COMMIT;

- **ROLLBACK –** The ROLLBACK statement ends the current transaction and undoes any changes made during that transaction. If you make a mistake, such as deleting the wrong row from a table, a rollback restores the original data. If you cannot finish a transaction because an exception is raised or a SQL statement fails, a rollback lets you take corrective action and perhaps start over.

-> ROLLBACK;

->ROLLBACK TO savepoint_name;

- **SAVEPOINT –** SAVEPOINT names and marks the current point in the processing of a transaction. Savepoint let you roll back part of a transaction instead of the whole transaction. The number of active savepoint for each session is unlimited.

  ->SAVEPOINT savepoint_name;

Commit Example

```
BEGIN
   UPDATE emp_information SET emp_dept='Web Developer'
      WHERE emp_name='Saulin';
END;
COMMIT;
```

ROLLBACK Example

```
DECLARE
  emp_id  emp.empno%TYPE;
BEGIN
     UPDATE emp SET eno=1
      WHERE empname = 'Forbs ross'
 END;
ROLLBACK;
```

SAVEPOINT Example

```
BEGIN
     insert into student values(1,'ram');
            savepoint s;
            insert into student values(2,'raj');
 END;
ROLLBACK to s;
```

3.3 Concurrency Control

Concurrency control is a database management systems (DBMS) concept that is used to address occur with a multi-user system. Concurrency control, when applied to a DBMS, is meant to coordinate simultaneous transactions while preserving data integrity. The Concurrency is about to control the multi-user access of Database.

Concurrency control is a very important issue in distributed database system design. This is because concurrency allows many transactions to be executing simultaneously such that collection of manipulated data item is left in a consistent state. Database concurrency control permits users to access a database in a multiprogrammed fashion while preserving the illusion that user is executing alone on a dedicated system. Besides, it produces the same effect and has the same output on the database as some serial execution of the same transaction.

In Oracle PL/SQL, a **LOCK** is a mechanism that prevents destructive interaction between two simultaneous transactions or sessions trying to access the same database object.

A LOCK can be achieved in two ways:

***Implicit* locking or *Explicit* Locking**

Oracle server implicitly creates a deadlock situation if a transaction is done on the same table in different sessions. This default locking mechanism is called ***implicit* or *automatic* locking.**

A lock is held until the transaction is complete; this is referred to as *data concurrency*.

A key reason for locking is to ensure that all valid processes are always able to access the original data as it was at the time the query was initiated. This is referred to as *read consistency*.

With **Explicit Locking,** a table or partition can be locked using the LOCK TABLE statement in one of the specified modes. The available lock modes are **ROW EXCLUSIVE**, **SHARE UPDATE**, **ROW SHARE**, **SHARE ROW EXCLUSIVE**, **EXCLUSIVE**, **NOWAIT** and **WAIT**. Note that it is preferable to do Explicit Locking rather than relying on the implicit locking done by default by the Oracle server.

There are two types of Locks

1. Shared lock
2. Exclusive lock

Shared lock:

Shared locks are placed on resources whenever a read operation (select) is performed. Multiple shared locks can be simultaneously set on a resource.

Exclusive lock:

Exclusive locks are placed on resources whenever a write operation (INSERT, UPDATE And DELETE) are performed. Only one exclusive lock can be placed on a resource at a time.

i.e. the first user who acquires an exclusive lock will continue to have the sole ownership of the resource, and no other user can acquire an exclusive lock on that resource

Level of Locks

1. Row level
2. Table level

Row Locks (TX)

A row lock, also called a TX lock, is a lock on a single row of a table. A transaction acquires a row lock for each row modified by one of the following statements: INSERT, UPDATE, DELETE, MERGE, and SELECT ... FOR UPDATE. The row lock exists until the transaction commits or rolls back.

When a transaction obtains a row lock for a row, the transaction also acquires a table lock for the table in which the row resides. The table lock prevents conflicting DDL operations that would override data changes in a current transaction.

Row-level locks serve a primary function to prevent multiple transactions from modifying the same row. Whenever a transaction needs to modify a row, a row lock is acquired by Oracle.

There is no hard limit on the exact number of row locks held by a statement or transaction. Also, unlike other database platforms, Oracle will never escalate a lock from the row level to a coarser granular level. This row locking ability provides the DBA with the finest granular level of locking possible and, as such, provides the best possible data concurrency and performance for transactions.

Table Locks (TM)

A transaction automatically acquires a table lock (TM lock) when a table is modified with the following statements: INSERT, UPDATE, DELETE, MERGE, and SELECT ... FOR UPDATE. These DML operations require table locks to reserve DML access to the table on behalf of a transaction and to prevent DDL operations that would conflict with the transaction. You can explicitly obtain a table lock using the LOCK TABLE statement.

Any table lock prevents the acquisition of an exclusive DDL lock on the same table, and thereby prevents DDL operations that require such locks. For example, a table cannot be altered or dropped if an uncommitted transaction holds a table lock for it.

3.4 Managing Cursors in PL/SQL

A Cursor is a pointer to this context area. Oracle creates context area for processing an SQL statement which contains all information about the statement.

PL/SQL allows the programmer to control the context area through the cursor. A cursor holds the rows returned by the SQL statement. The set of rows the cursor holds is referred as active set. These cursors can also be named so that they can be referred from another place of the code.

There are two types of cursors −

- Implicit cursors
- Explicit cursors

Implicit Cursors

Implicit cursors are automatically created by Oracle whenever an SQL statement is executed, when there is no explicit cursor for the statement. Programmers cannot control the implicit cursors and the information in it.

Whenever a DML statement (INSERT, UPDATE and DELETE) is issued, an implicit cursor is associated with this statement. For INSERT operations, the cursor holds the data that needs to be inserted. For UPDATE and DELETE operations, the cursor identifies the rows that would be affected.

Explicit Cursors

Programmers are allowed to create named context area to execute their DML operations to get more control over it. The explicit cursor should be defined in the declaration section of the PL/SQL block, and it is created for the 'SELECT' statement that needs to be used in the code.

Below are steps that involved in working with explicit cursors.

- **Declaring the cursor**

Declaring the cursor simply means to create one named context area for the 'SELECT' statement that is defined in the declaration part. The name of this context area is same as the cursor name.

- **Opening Cursor**

Opening the cursor will instruct the PL/SQL to allocate the memory for this cursor. It will make the cursor ready to fetch the records.

- **Fetching Data from the Cursor**

In this process, the 'SELECT' statement is executed and the rows fetched is stored in the allocated memory. These are now called as active sets. Fetching data from the cursor is a record-level activity that means we can access the data in a record-by-record way.

Each fetch statement will fetch one active set and holds the information of that particular record. This statement is same as 'SELECT' statement that fetches the record and assigns to the variable in the 'INTO' clause, but it will not throw any exceptions.

- **Closing the Cursor**

Once all the record is fetched now, we need to close the cursor so that the memory allocated to this context area will be released.

Syntax:

```
DECLARE
CURSOR <cursor_name> IS <SELECT statement^>
<cursor_variable declaration>
BEGIN
OPEN <cursor_name>;
FETCH <cursor_name> INTO <cursor_variable>;
.
.
CLOSE <cursor_name>;
END;
```

Cursor Attributes

Both Implicit cursor and the explicit cursor has certain attributes that can be accessed. These attributes give more information about the cursor operations. Below are the different cursor attributes and their usage.

Cursor Attribute	**Description**
%FOUND	It returns the Boolean result 'TRUE' if the most recent fetch operation fetched a record successfully, else it will return FALSE.
%NOTFOUND	This works oppositely to %FOUND it will return 'TRUE' if the most recent fetch operation could not able to fetch any record.
%ISOPEN	It returns Boolean result 'TRUE' if the given cursor is already opened, else it returns 'FALSE'
%ROWCOUNT	It returns the numerical value. It gives the actual count of records that got affected by the DML activity.

Example

```
DECLARE
   c_id customers.id%type;
   c_name customerS.No.ame%type;
   c_addr customers.address%type;
   CURSOR c_customers is
      SELECT id, name, address FROM customers;
BEGIN
   OPEN c_customers;
   LOOP
   FETCH c_customers into c_id, c_name, c_addr;
      EXIT WHEN c_customers%notfound;
      dbms_output.put_line(c_id || ' ' || c_name || ' ' || c_addr);
   END LOOP;
   CLOSE c_customers;
END;
```

Cursor FOR Loop statement

"FOR LOOP" statement can be used for working with cursors. We can give the cursor name instead of range limit in the FOR loop statement so that the loop will work from the first record of the cursor to the last record of the cursor. The cursor variable, opening of cursor, fetching and closing of the cursor will be done implicitly by the FOR loop.

Syntax:

```
DECLARE
CURSOR <cursor_name> IS <SELECT statement>;
BEGIN
  FOR I IN <cursor_name>
  LOOP
  .
  .
  END LOOP;
END;
```

Example

```
DECLARE
CURSOR c1 IS SELECT emp_name FROM employee;
BEGIN
FOR lv_emp_name IN c1
LOOP
Dbms_output.put_line(‘Employee Fetched:‘||lv_emp_name);
END LOOP;
END;
```

Parameterized cursor

Parameterized cursors are static cursors that can accept passed-in parameter values when they are opened.

An explicit cursor may accept a list of parameters. Each time you open the cursor, you can pass different arguments to the cursor, which results in different result sets.

Syntax:

```
CURSOR cursor_name (parameter_list)
IS
cursor_query;
```

To open a cursor with parameters, you use the following syntax:

```
  OPEN cursor_name (value_list);
```

Example 1:

```
DECLARE
   my_record      emp%ROWTYPE;
   CURSOR c1 (max_wage NUMBER) IS
      SELECT * FROM emp WHERE sal < max_wage;
BEGIN
   OPEN c1(2000);
   LOOP
      FETCH c1 INTO my_record;
      EXIT WHEN c1%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE('Name = ' || my_record.ename || ', salary = '
         || my_record.sal);
   END LOOP;
   CLOSE c1;
END;
```

Example 2:

```
DECLARE
   r_product products%rowtype;
   CURSOR c_product (low_price NUMBER, high_price NUMBER)
   IS
      SELECT *
      FROM products
      WHERE list_price BETWEEN low_price AND high_price;
BEGIN
   -- show mass products
```

```
    dbms_output.put_line('Mass products: ');
    OPEN c_product(50,100);
    LOOP
        FETCH c_product INTO r_product;
        EXIT WHEN c_product%notfound;
        dbms_output.put_line(r_product.product_name || ': ' ||r_product.list_price);
    END LOOP;
    CLOSE c_product;

    -- show luxury products
    dbms_output.put_line('Luxury products: ');
    OPEN c_product(800,1000);
    LOOP
        FETCH c_product INTO r_product;
        EXIT WHEN c_product%notfound;
        dbms_output.put_line(r_product.product_name || ': ' ||r_product.list_price);
    END LOOP;
    CLOSE c_product;
END;
```

In this example:

- First, declare a cursor that accepts two parameters low price and high price. The cursor retrieves products whose prices are between the low and high prices.
- Second, open the cursor and pass the low and high prices as 50 and 100 respectively. Then fetch each row in the cursor and show the product’s information, and close the cursor.
- Third, open the cursor for the second time but with different arguments, 800 for the low price and 100 for the high price. Then the rest is fetching data, printing out product’s information, and closing the cursor.