CURSORS

Cursors are memory areas where Oracle executes SQL statements. In database programming cursors are internal data structures that allow processing of SQL query results .

A **cursor** is a temporary work area created in the system memory when a **SQL** statement is executed. A **cursor** contains information on a select statement and the rows of data accessed by it. This temporary work area is used to store the data retrieved from the database, and manipulate this data.

An explicit cursor enables a user to process many rows returned by a query and allows the user to write code that processes each row one at a time.

For Oracle to process a SQL statement, it needs to create an area of memory known as the context area; this will have the information needed to process the statement. This information includes the number of rows processed by the statement and a pointer to the parsed representation of the statement.

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. Cursors have two important features:

1. Cursors allows to fetch and process rows returned by a SELECT statement one row at a time.
2. A cursor is named so that it can be referenced.

Types of Cursors

There are two types of cursors:

1. Oracle automatically declares an implicit cursor every time a SQL statement is executed. The user is unaware of this and cannot control or process the information in an implicit cursor.
2. The program defines an explicit cursor for any query that returns more than one row of data. This means that the programmer has declared the cursor within the PL/SQL code block. This declaration allows the application to sequentially process each row of data as the cursor returns it.

Implicit Cursors

- Any given PL/SQL block issues an implicit cursor whenever a SQL statement is executed, as long as an explicit cursor does not exist for that SQL statement.
- A cursor is automatically associated with every DML (data manipulation) statement (UPDATE, DELETE, INSERT).
- All UPDATE and DELETE statements have cursors that identify the set of rows that will be affected by the operation.
- An INSERT statement needs a place to receive the data that is to be inserted into the database; the implicit cursor fulfills this need.
- The most recently opened cursor is called the SQL cursor.

The implicit cursor is used to process INSERT, UPDATE, DELETE, and SELECT INTO statements. During the processing of an implicit cursor, Oracle automatically performs the OPEN, FETCH, and CLOSE operations.

An implicit cursor can reveal the total rows affected by an update. Cursors have attributes such as ROWCOUNT. **SQL%ROWCOUNT** returns the number of rows updated.

```
BEGIN
      UPDATE EMPP
            SET FNAME = ©Changed©
            WHERE UPPER(FNAME) LIKE ©%T%©;
      DBMS_OUTPUT.PUT_LINE(©Rows Affected : ©||SQL%ROWCOUNT);
END;
```

Oracle automatically associates an implicit cursor with the SELECT INTO statement and fetches the values for the variables specified in INTO clause. After the SELECT INTO statement completes, Oracle closes the implicit cursor.

Explicit Cursor

The only means of generating an explicit cursor is to name the cursor in the DECLARE section of the PL/SQL block.

The advantage of declaring an explicit cursor over an indirect implicit cursor is that the explicit cursor gives the programmer more programmatic control. Also, implicit cursors are less efficient than explicit cursors, so it is harder to trap data errors.

The process of working with an explicit cursor consists of the following steps:

- Declaring the cursor

  This initializes the cursor into memory.
- Opening the cursor

  The declared cursor is opened, and memory is allotted.
- Fetching the cursor

  The declared and opened cursor can now retrieve data.
- Closing the cursor

  The declared, opened, and fetched cursor must be closed to release the memory allocation.

A cursor cannot be used unless the complete cycle of declaring, opening, fetching, and closing has been performed.

Declaring a Cursor

Declaring a cursor defines the cursor's name and associates it with a SELECT

statement.

```
CURSOR c_cursor_name IS select statement
```

Declaration of a cursor named PART_INFO cursor

```
DECLARE
      CURSOR C_PART_INFO IS
            SELECT P_CODE, DESCRIPT, V_CODE
                  FROM PRODUCT
                  WHERE QTY < 50;
...

-- code would continue here with opening, fetching
-- and closing of the cursor>
```

Any valid SELECT statement can be used to define a cursor, including joins and statements with the UNION or MINUS clause.

Record Types

A record is a composite data structure, which means that it is composed of one or more elements. Records are very much like a row of a database table, but each element of the record does not stand on its own. PL/SQL supports three kinds of records: table-based, cursor-based, and programmer-defined.

A table-based record is one whose structure is drawn from the list of columns in the table. A cursor-based record is one whose structure matches the elements of a predefined cursor. To create a table-based or cursor-based record, use the %ROWTYPE attribute:

```
record_name {table_name|cursor_name}%ROWTYPE
```

A cursor-based record can be declared only after its corresponding cursor has been declared; otherwise, a compilation error will occur.

```
DECLARE
      CURSOR C_STAFF_NAME IS
            SELECT FNAME, LNAME
                  FROM STAFF
      V_STAFF_REC C_STAFF_NAME%ROWTYPE;
```

Opening a Cursor

The OPEN cursor statement enables 4 actions automatically:

1. The variables (including bind variables) in the WHERE clause are examined.
2. Based on the values of the variables, the active set is determined, and the PL/SQL engine executes the query for that cursor. Variables are examined at cursor open time only.
3. The PL/SQL engine identifies the active set of data—the rows from all the involved tables that meet the WHERE clause criteria.
4. The active set pointer is set to the first row.

The syntax for opening a cursor is —

```
OPEN cursor_name;
```

A pointer into the active set is also established at cursor open time. The pointer determines which row is the next to be fetched by the cursor. More than one cursor

can be open at a time.

Fetching Rows in a Cursor

After the cursor has been declared and opened, data can be retrieved from the cursor. The process of getting data from the cursor is called **fetching the cursor**. There are two ways to fetch a cursor:

```
FETCH cursor_name INTO PL/SQL variables;
```

or

```
FETCH cursor_name INTO PL/SQL record;
```

When the cursor is fetched, the following occurs:

1. The FETCH command is used to retrieve one row at a time from the active set. This is generally done inside a loop. The values of each row in the active set can then be stored in the corresponding variables or PL/SQL record one at a time, performing operations on each one successively.
2. After each FETCH, the active set pointer is moved forward to the next row. Thus, each FETCH returns successive rows of the active set, until the entire set is returned. The last FETCH does not assign values to the output variables; they still contain their prior values.

```
DECLARE
      CURSOR C_PRODUCT_INFO IS                    -- declaring a cursor
            SELECT P_CODE, DESCRIPT, V_CODE
                  FROM PRODUCT;
      V_PART C_PRODUCT_INFO%ROWTYPE;              -- cursor-based record
BEGIN
      OPEN C_PRODUCT_INFO;                        -- opening a cursor
      LOOP
            FETCH C_PRODUCT_INFO INTO V_PART;     -- fetching a cursor
            EXIT WHEN C_PRODUCT_INFO%NOTFOUND;    -- cursor state variable
            DBMS_OUTPUT.PUT_LINE(V_PART.P_CODE|| ©: ©||
                  V_PART.DESCRIPT||© -- ©||V_PART.V_CODE);
      END LOOP;
      CLOSE  C_PRODUCT_INFO;                      -- closing a cursor
END;
```

Closing a Cursor

As soon as all the rows in the cursor have been processed (retrieved), the cursor should be closed. This tells the PL/SQL engine that the program is finished with the cursor, and the resources associated with it can be freed. The syntax for closing the cursor is

```
CLOSE cursor_name;
```

After a cursor is closed, we no longer can fetch from it. Likewise, it is not possible to close an already closed cursor. Trying to perform either of these actions would result in an Oracle error.

```
DECLARE
      CURSOR C_STAFF_NAME IS
            SELECT FNAME, LANME
                  FROM STAFF WHERE SID < 106;
      V_NAME C_STAFF_NAME%ROWTYPE;
BEGIN
      OPEN C_STAFF_NAME;
      LOOP
            FETCH C_STAFF_NAME INTO V_NAME;
            EXIT WHEN C_STAFF_NAME%NOTFOUND;
            DBMS_OUTPUT.PUT_LINE(©Teacher: ©||V_NAME. FNAME||
                  © ©||V_NAME.LNAME);
      END LOOP;
      CLOSE C_STAFF_NAME;
END;
```

Explicit Cursor Attributes

Following attributes of a cursor determines the result of a cursor operation when fetched or opened.

Cursor Attribute	Syntax	Description
%NOTFOUND	cur_name%NOTFOUND	A Boolean attribute that returns TRUE if the previous FETCH did not return a row and FALSE if it did.
%FOUND	cur_name%FOUND	A Boolean attribute that returns TRUE if the previous FETCH returned a row and FALSE if it did not.
%ROWCOUNT	cur_name%ROWCOUNT	The number of records fetched from a cursor at that point in time.
%ISOPEN	cur_name%ISOPEN	A Boolean attribute that returns TRUE if the cursor is open and FALSE if it is not.

The attribute %NOTFOUND can be used to close the loop. It is recommended to add an exception clause to the end of the block to close the cursor if it is still open.

```
      ...
            EXIT WHEN C_STAFF_NAME%NOTFOUND;
      END LOOP;
      CLOSE C_STAFF_NAME;
EXCEPTION
      WHEN OTHERS THEN
            IF C_STAFF_NAME%ISOPEN THEN
                  CLOSE C_STAFF_NAME;
            END IF;
END;
```

Cursor attributes can be used with implicit cursors by using the prefix SQL, such as SQL%ROWCOUNT.

```
DECLARE
      V_ENROL STUDENT.ENROLL%TYPE;
BEGIN
      SELECT ENROLL INTO V_ENROL
            FROM STUDENT WHERE ROLL = 4004;
      IF SQL%ROWCOUNT = 1 THEN
            DBMS_OUTPUT.PUT_LINE(©Enrolment ID of 4004: ©|| V_ENROL);
      ELSIF SQL%ROWCOUNT = 0 THEN
            DBMS_OUTPUT.PUT_LINE(©Student with Roll 4004 does not exist©);
      ELSE
            DBMS_OUTPUT.PUT_LINE(©Whoa!! Multiple Enrollments©);
      END IF;
END;
```

The statements highlighted in blue are never executed because Oracle raises a **NO_DATA_FOUND** exception when a SELECT statement in a PL/SQL block doesn't return any rows. In absence of any exception handler, the above code will not dispaly any error.

```
DECLARE
      V_ENROL STUDENT.ENROLL%TYPE;
      CURSOR C_ENROLL IS
            SELECT ENROLL FROM STUDENT WHERE ROLL < 3009;
BEGIN
      OPEN C_ENROLL;
      LOOP
            FETCH C_ENROLL INTO V_ENROL;
            IF C_ENROLL%FOUND THEN
                  DBMS_OUTPUT.PUT_LINE (©Row:©||
                        TO_CHAR(C_ENROLL%ROWCOUNT)|| © :=©||V_ENROL);
            ELSE
                  EXIT;
            END IF;
      END LOOP;
      CLOSE C_ENROLL;
EXCEPTION
      WHEN OTHERS THEN
            IF C_ENROLL%ISOPEN THEN
                  CLOSE C_ENROLL;
            END IF;
END;
```

The Cursor For Loops

There is an alternative way to handle cursors. It is called the cursor FOR loop because of the simplified syntax that is used. With a cursor FOR loop, the process of opening, fetching, and closing is handled **implicitly**. This makes the blocks much easier to code and maintain.

The cursor FOR loop specifies a sequence of statements to be repeated once for each row returned by the cursor. The cursor FOR loop may be used when it is required to FETCH and PROCESS every record from a cursor till user stops processing and exits the loop.

Create a table named ITEM that includes attributes P_CODE, P_DESCRIPT, V_CODE, P_PRICE and QTY of PRODUCT table [You must ensure that ITEM table is empty].

```
DECLARE
      CURSOR C_ITEMS IS
            SELECT P_CODE, DESCRIPT,V_CODE, P_PRICE, QTY
            FROM PRODUCT;
BEGIN
      FOR IR IN C_ITEMS LOOP        -- IR is the cursor variable
            INSERT INTO ITEM
                  VALUES(IR.P_CODE, IR.DESCRIPT,
                        IR.V_CODE, IR.P_PRICE, IR.QTY );
      END LOOP;
END;
```

For example, a PL/SQL block that raises the product prize by 10 percent for products that are some kind of screws. Use a cursor FOR loop that updates the ITEM table.

```
DECLARE
      CURSOR C_RAISE_PRICE IS
            SELECT P_CODE FROM ITEM
                  WHERE UPPER(DESCRIPT) LIKE ©%SCREW%©;
BEGIN
      FOR RP IN C_RAISE_PRICE LOOP
            DBMS_OUTPUT.PUT_LINE(©PRICE FOR ITEM CODE: ©
                              ||RP.P_CODE ||© IS BEING UPDATED©);
            UPDATE ITEM
                  SET P_PRICE = P_PRICE * 1.10
                  WHERE P_CODE = RP.P_CODE;
            DBMS_OUTPUT.PUT_LINE(©ROW#: ©||C_RAISE_PRICE%ROWCOUNT);
      END LOOP;
      COMMIT;
END;
```

The cursor is processed in a FOR loop. In each iteration of the loop, the SQL update statement is executed. Thus, it is clear that, the cursor need not be opened, fetched, and closed. Also a cursor attribute does not have to be used to create an exit condition for the loop that is processing the cursor.

Oracle Bulk Collect

One method of fetching data is an Oracle bulk collect. With Oracle bulk collect, the PL/SQL engine tells the SQL engine to collect many rows at once and place them in a collection.

During an Oracle bulk collect, the SQL engine retrieves all the rows and loads them into the collection and switches back to the PL/SQL engine. When rows are retrieved using Oracle bulk collect, they are retrieved with only two context switches. The larger the number of rows you would like to collect with Oracle bulk collect, the more performance improvement you will see using an Oracle bulk collect.

It should be noted that bulk collecting 100 rows may not provide much of a benefit to user, but using Oracle bulk collect to collect large numbers of rows (many hundreds) will provide increased performance.

Bulk collect is easy to use. The process is as under -

1. Define the collection or collections that will be collected using the Oracle bulk collect.
2. Define the cursor to retrieve the data in the Oracle bulk collect.
3. Bulk collect the data into the collections.

A simple Oracle bulk collect example that will list all purchases during last three days. [Let us assume PURCHASES table with one million records].

```
DECLARE
      TYPE NUM_ARRAY IS VARRAY(10000) OF NUMBER;
      TYPE STR_ARRAY IS VARRAY(10000) OF VARCHAR2(50);
      TYPE ITM_ARRAY IS VARRAY(10000) OF VARCHAR2(5);

      CODE_ARR ITM_ARRAY;
      DESC_ARR STR_ARRAY;
      QTY_ARR  NUM_ARRAY;

      CURSOR C_ALL_PURCHASES IS
            SELECT P_CODE, DESCRIPT, QTY FROM PURCHASES
                  WHERE P_DATE >= SYSDATE ± 3;;
BEGIN
      OPEN C_ALL_PURCHASES;
      FETCH C_ALL_PURCHASES
            BULK COLLECT INTO CODE_ARR, DESC_ARR, QTY_ARR;
      CLOSE C_ALL_PURCHASES;
      FOR KNT IN CODE_ARR.FIRST .. CODE_ARR.LAST LOOP
            DBMS_OUTPUT.PUT_LINE(CODE_ARR(KNT)||© : ©||
                  ||DESC_ARR(KNT)||© : ©||QTY_ARR(KNT));
      END LOOP;
END;
```

Normally a developer will use a cursor to retrieve and process multiple rows of data, one at a time, but there are performance problems when dealing with large numbers of rows using cursors. As it is seen, a cursor fetches one row at a time, holding a consistent view, until all rows have been retrieved or the cursor is closed.

A performance issue arises from the fact that there are two engines in the database, the PL/SQL engine and the SQL engine. When a cursor fetches a row of data it performs a

"context switch" to the SQL engine, and it is the SQL component that retrieves the data. The SQL engine places the data in-memory and another context switch places user back into the PL/SQL engine.

The PL/SQL engine then continues processing until the next row is required, and the process repeats. A context switch is very fast, but if performed over and over again, the constant switching can take a noticeable amount of time.

A bulk collect is a method of fetching data where the PL/SQL engine tells the SQL engine to collect many rows at once and place them in a collection. The SQL engine retrieves all the rows and loads them into the collection and switches back to the PL/SQL engine. All the rows are retrieved with only **2** context switches. The larger the number of rows processed, the more performance is gained by using a bulk collect.

NOTE: In 10g onwards, a cursor loop may cause the PL/SQL engine to automatically bulk collect 100 rows at a time, allowing user code to process rows without having to setup and execute the bulk collect operation.

Bulk Collect without Cursor For Loop

```
DECLARE
      TYPE SALES_TAB IS TABLE OF SALES%ROWTYPE;
      T_SAL SALES_TAB;
BEGIN
      SELECT * BULK COLLECT INTO T_SAL FROM SALES;
      DBMS_OUTPUT.PUT_LINE(T_SAL.COUNT);
END;
/
```

Chunking Bulk Collections Using the LIMIT Clause

PL/SQL collections are essentially arrays in memory, so massive collections can have a detrimental effect on system performance due to the amount of memory they require. In some situations, it may be necessary to split the data being processed into chunks to make the code more memory-friendly. This "chunking" can be achieved using the LIMIT clause of the BULK COLLECT syntax.

```
DECLARE
      TYPE T_BULK_COLL_ARR IS TABLE OF TEST_TBL%ROWTYPE;
      TBL T_BULK_COLL_ARR;
      CURSOR C_RES IS
            SELECT * FROM TEST_TBL;
BEGIN
      OPEN C_RES;
      LOOP
            FETCH C_RES BULK COLLECT INTO TBL LIMIT 10000;
            -- PROCESS CONTENTS OF COLLECTION HERE.
            DBMS_OUTPUT.PUT_LINE(TBL.COUNT || © ROWS©);
            EXIT WHEN C_RES%NOTFOUND;
      END LOOP;
      CLOSE C_DATA;
END;
/
```

Different Approaches to Bulk Insert

[Source: https://www.akadia.com/services/ora_bulk_insert.html]

```
CREATE TABLE USER_TBL AS
      SELECT * FROM ALL_OBJECT
            WHERE 1 = 2;
```

The Old Fashioned Way

A straightforward code; unfortunately, it takes a lot of time to run - it is "old-fashioned" code, so let's improve it using collections and bulk processing.

```
CREATE OR REPLACE PROCEDURE TEST_PROC IS
BEGIN
      FOR X IN (SELECT * FROM ALL_OBJECTS)
      LOOP
            INSERT INTO USER_TBL(OWNER, OBJECT_NAME, SUBOBJECT_NAME, OBJECT_ID,
                  DATA_OBJECT_ID, OBJECT_TYPE, CREATED, LAST_DDL_TIME, TIMESTAMP,
                  STATUS, TEMPORARY, GENERATED, SECONDARY)
            VALUES (X.OWNER, X.OBJECT_NAME, X.SUBOBJECT_NAME, X.OBJECT_ID,
                  X.DATA_OBJECT_ID, X.OBJECT_TYPE, X.CREATED, X.LAST_DDL_TIME,
                  X.TIMESTAMP, X.STATUS, X.TEMPORARY, X.GENERATED, X.SECONDARY);
      END LOOP;
      COMMIT;
END TEST_PROC;
/

SET TIMING ON;
EXEC TEST_PROC;
      PL/SQL procedure successfully completed.
      Elapsed: 00:00:12.84
```

USING Bulk Collect

Converting to collections and bulk processing can increase the volume and complexity of the user code. Collections, an evolution of PL/SQL tables that allows user to manipulate many variables at onc, as a unit. Collections, coupled with BULK_COLLECT and FORALL (Oracle 8i onwards), can dramatically increase the performance of data manipulation code within PL/SQL.

```
CREATE OR REPLACE PROCEDURE TEST_PROC
            (FETCH_SIZE IN PLS_INTEGER DEFAULT 100) IS
      TYPE ARRAY IS TABLE OF ALL_OBJECTS%ROWTYPE;
      L_DATA ARRAY;
      CURSOR BULK_CUR IS SELECT * FROM ALL_OBJECTS;
BEGIN
      OPEN C;
      LOOP
            FETCH BULK_CUR BULK COLLECT INTO L_DATA LIMIT FETCH_SIZE;
            FORALL NDX IN 1 .. L_DATA.COUNT
                  INSERT INTO USER_TBL VALUES L_DATA(NDX);
            EXIT WHEN BULK_CUR%NOTFOUND;
END LOOP;
CLOSE BULK_CUR;
END TEST_PROC;
/

EXEC TEST_PROC;
      PL/SQL procedure successfully completed.
      Elapsed: 00:00:02.34
```

Eliminate CURSOR LOOP at all

The CURSOR Loop can be completely eliminated making the procedure compact and precise. This although do not impove performance.

```
CREATE OR REPLACE PROCEDURE TEST_PROC IS
    TYPE OBJ_TBL IS TABLE OF ALL_OBJECTS%ROWTYPE;
    USR_OBJECTS OBJ_TBL;
BEGIN
      SELECT * BULK COLLECT INTO USR_OBJECTS
            FROM ALL_OBJECTS;
      FORALL KNT IN USR_OBJECTS.FIRST .. USR_OBJECTS.LAST
            INSERT INTO USER_TBL VALUES USR_OBJECTS(KNT) ;
END;
/

EXEC TEST_PROC;
      PL/SQL procedure successfully completed.
      Elapsed: 00:00:03.51
```

Cursor Parameter Modes

The syntax for cursor parameters is very similar to that of procedures and functions, with the restriction that a cursor parameter can be an IN parameter only. OUT or IN OUT modes for cursor parameters is not allowed.

The IN and IN OUT modes are used to pass values out of a procedure through that parameter. This doesn't make sense for a cursor. Values cannot be passed back out of a cursor through the parameter list. Information is retrieved from a cursor only by fetching a record and copying values from the column list with an INTO clause.

Default Values for Parameters

Cursor parameters can be assigned default values. Here is an example of a parameterized cursor with a default value:

```
CURSOR EMP_CUR (ENO NUMBER := 0) IS
      SELECT EID, FNAME
            FROM EMPLOYEE
            WHERE EID = ENO;
```

To initiate a cursor to some EID (say 9001), the cursor may be opned as -

```
OPEN EMP_CUR(9001);
FETCH EMP_CUR INTO V_EID, V_NAME;
```

Because the ENO parameter has a default value, user can open and fetch from the cursor without specifying a parameter. If the value for the parameter is not specified, the cursor uses the default value.

```
OPEN EMP_CUR;
FETCH EMP_CUR INTO V_EID, V_NAME;
```

NOTE : Other details with examples has been discussed in Lab.

Implicit SQL Cursor Attributes

When the RDBMS opens an implicit cursor to process user request (whether it is a query or an INSERT or an UPDATE), it makes cursor attributes available to user with the SQL cursor. This is not a cursor in the way of an explicit cursor. User **cannot open, fetch from, or close** the SQL cursor, but user can access information about the most recently executed SQL statement through SQL cursor attributes.

The SQL cursor has attributes as an explicit cursor: SQL%FOUND, SQL%NOTFOUND, SQL%ROWCOUNT, SQL%ISOPEN

Differences Between Implicit and Explicit Cursor Attributes

The values returned by implicit cursor attributes differ from those of explicit cursor attributes in the following ways:

- If the RDBMS has not opened an implicit SQL cursor in the session, then the SQL %ROWCOUNT attribute returns NULL instead of raising the INVALID_CURSOR error. References to the other attributes (ISOPEN, FOUND, NOTFOUND) all return FALSE. User will never raise the INVALID_CURSOR error with an implicit cursor attribute reference.
- The %ISOPEN attribute will always return FALSE -- before and after the SQL statement. After the statement is executed (whether it is SELECT, UPDATE, DELETE, or INSERT), the implicit cursor will already have been opened and closed implicitly. An implicit cursor can never be open outside of the statement itself.
- SQL cursor attributes are always set according to the results of the most recently executed SQL statement. That SQL statement might have been executed in a stored procedure which your program called; user might not even be aware of that SQL statement execution. If user plan to make use of a SQL cursor attribute, make sure that attribute is referenced immediately after the SQL statement is executed.
- The %FOUND attribute returns TRUE if an UPDATE, DELETE, or INSERT affected at least one record. It will return FALSE if those statements failed to affect any records. It returns TRUE if an implicit SELECT returns one row.
- The behavior of the %NOTFOUND attribute for UPDATE, DELETE, or INSERT statements is the opposite of %FOUND. The situation for an implicit SELECT is a bit different: when using an implicit SELECT statement, never rely on the %NOTFOUND and %FOUND attributes.
- When an implicit SELECT statement does not return any rows, PL/SQL immediately raises the NO_DATA_FOUND exception. When an implicit SELECT statement returns more than one row, PL/SQL immediately raises the TOO_MANY_ROWS exception. In either case, once the exception is raised, control shifts to the exception section of the PL/SQL block.