**Batch: B1 Roll No.: 1711072**

**Experiment / assignment / tutorial No. 6**

**Grade: AA / AB / BB / BC / CC / CD /DD**

|  |
| --- |
| **Title: Queries based on Triggers and Cursor** |

**Objective:** To be able to use trigger and cursor on table.

**Expected Outcome of Experiment:**

CO 2: Convert entity-relationship diagrams into relational tables, populate a relational

database and formulate SQL queries on the data Use SQL for creation and query the database.

CO 3: Define and apply integrity constraints and improve database design using normalization techniques.

**Books/ Journals/ Websites referred:**

1. Dr. P.S. Deshpande, SQL and PL/SQL for Oracle 10g.Black book, Dreamtech Press

2. www.db-book.com

3. Korth, Slberchatz, Sudarshan : “Database Systems Concept”, 5th Edition , McGraw

Hill

4. Elmasri and Navathe,”Fundamentals of database Systems”, 4th Edition,PEARSON

Education.

**Resources used:** Postgresql

**Theory**

**Triggers** are database call-back functions, which are automatically performed/invoked when a specified database event occurs.

**Triggers** can be specified to fire

* Before the operation is attempted on a row (before constraints are checked and the INSERT, UPDATE or DELETE is attempted)
* After the operation has completed (after constraints are checked and the INSERT, UPDATE, or DELETE has completed)
* Instead of the operation (in the case of inserts, updates or deletes on a view)

The basic syntax of creating a trigger is as follows −

CREATE TRIGGER trigger\_name [BEFORE|AFTER|INSTEAD OF] event\_name

ON table\_name

[

-- Trigger logic goes here....

];

event\_name could be INSERT, DELETE, UPDATE, and TRUNCATE database operation on the mentioned table table\_name. You can optionally specify FOR EACH ROW after table name.

The following is the syntax of creating a trigger on an UPDATE operation on one or more specified columns of a table as follows −

CREATE TRIGGER trigger\_name [BEFORE|AFTER] UPDATE OF column\_name

ON table\_name

[

-- Trigger logic goes here....

];

**Cursors**

Rather than executing a whole query at once, it is possible to set up a cursor that encapsulates the query, and then read the query result a few rows at a time. One reason for doing this is to avoid memory overrun when the result contains a large number of rows. (However, PL/pgSQL users do not normally need to worry about that, since FOR loops automatically use a cursor internally to avoid memory problems.) A more interesting usage is to return a reference to a cursor that a function has created, allowing the caller to read the rows. This provides an efficient way to return large row sets from functions.

Before a cursor can be used to retrieve rows, it must be opened. (This is the equivalent action to the SQL command DECLARE CURSOR.) PL/pgSQL has three forms of the OPEN statement, two of which use unbound cursor variables while the third uses a bound cursor variable.

**OPEN FOR query**

Syntax: OPEN unbound\_cursorvar [ [ NO ] SCROLL ] FOR query;

example:

OPEN curs1 FOR SELECT \* FROM foo WHERE key = mykey;

**OPEN FOR EXECUTE**

Syntax: OPEN unbound\_cursorvar [ [ NO ] SCROLL ] FOR EXECUTE query\_string

[ USING expression [, ... ] ];

example:

OPEN curs1 FOR EXECUTE 'SELECT \* FROM ' || quote\_ident(tabname)

|| ' WHERE col1 = $1' USING keyvalue;

**Opening a Bound Cursor**

Syntax: OPEN bound\_cursorvar [ ( [ argument\_name := ] argument\_value [, ...] ) ];

Examples (these use the cursor declaration examples above):

OPEN curs2;

OPEN curs3(42);

OPEN curs3(key := 42);

Because variable substitution is done on a bound cursor's query, there are really two ways to pass values into the cursor: either with an explicit argument to OPEN, or implicitly by referencing a PL/pgSQL variable in the query. However, only variables declared before the bound cursor was declared will be substituted into it. In either case the value to be passed is determined at the time of the OPEN. For example, another way to get the same effect as the curs3 example above is

DECLARE

key integer;

curs4 CURSOR FOR SELECT \* FROM tenk1 WHERE unique1 = key;

BEGIN

key := 42;

OPEN curs4;

**Using Cursors**

**FETCH**

Synatx: FETCH [ direction { FROM | IN } ] cursor INTO target;

Examples:

FETCH curs1 INTO rowvar;

FETCH curs2 INTO foo, bar, baz;

FETCH LAST FROM curs3 INTO x, y;

FETCH RELATIVE -2 FROM curs4 INTO x;

**MOVE**

MOVE [ direction { FROM | IN } ] cursor;

MOVE repositions a cursor without retrieving any data. MOVE works exactly like the FETCH command, except it only repositions the cursor and does not return the row moved to. As with SELECT INTO, the special variable FOUND can be checked to see whether there was a next row to move to.

Examples:

MOVE curs1;

MOVE LAST FROM curs3;

MOVE RELATIVE -2 FROM curs4;

MOVE FORWARD 2 FROM curs4;

**UPDATE/DELETE WHERE CURRENT OF**

UPDATE table SET ... WHERE CURRENT OF cursor;

DELETE FROM table WHERE CURRENT OF cursor;

When a cursor is positioned on a table row, that row can be updated or deleted using the cursor to identify the row. There are restrictions on what the cursor's query can be (in particular, no grouping) and it's best to use FOR UPDATE in the cursor. For more information see the DECLARE reference page.

An example:

UPDATE foo SET dataval = myval WHERE CURRENT OF curs1;

**CLOSE**

CLOSE cursor;

CLOSE closes the portal underlying an open cursor. This can be used to release resources earlier than end of transaction, or to free up the cursor variable to be opened again.

An example:

CLOSE curs1;

**Implementation Screenshots (Problem Statement, Query and Screenshots of Results):**

1. CREATE OR REPLACE FUNCTION age\_constraint()

RETURNS TRIGGER AS

$$

BEGIN

IF (NEW.age<18) then

RAISE EXCEPTION 'Age below 18 not allowed to watch A rated movies.';

END IF;

RETURN new;

END

$$

LANGUAGE 'plpgsql';

CREATE TRIGGER test\_age

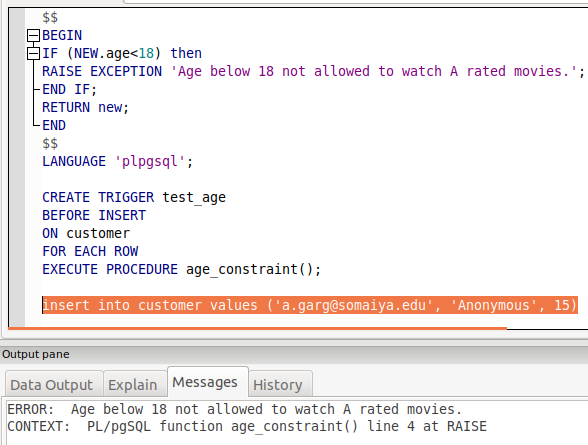
BEFORE INSERT

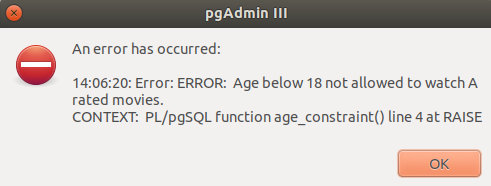
ON customer

FOR EACH ROW

EXECUTE PROCEDURE age\_constraint();

2.





# Conclusion: