Introduction

1. **About the workbook:**
   This workbook is intended to be used by S.Y.B.Sc. (Computer Science) students for the assignments in Semester–III. This workbook is designed by considering all the practical concepts / topics mentioned in syllabus.

2. **The objectives of this workbook are:**
   1) Defining the scope of the course.
   2) To have continuous assessment of the course and students.
   3) Providing ready reference for the students during practical implementation.
   4) Provide more options to students so that they can have good practice before facing the examination.
   5) Catering to the demand of slow and fast learners and accordingly providing the practice assignments to them.

3. **How to use this workbook:**
   The workbook is divided into 4 assignments. Each student is expected to complete the Lab assignments in the Lab, and try the Home assignments at home.

Instructions to the students

Please read the following instructions carefully and follow them.

1) Students are expected to carry this book every time when they come to the lab for computer science practical.
2) Students should prepare themselves beforehand for the Assignment by reading the relevant material.
3) Instructor will specify which problems to solve in the lab during the allotted slot and student should complete them and get verified by the instructor. However student should spend additional hours in Lab and at home to cover as many problems as possible given in this work book.
4) Students will be assessed for each exercise on a scale from 0 to 5.

Not done	0
Incomplete	1
Late Complete	2
Needs improvement	3
Complete	4
Well Done	5

Guidelines for Instructors

1) Explain the assignment and related concepts in around ten minutes using whiteboard if required or by demonstrating the software.
2) You should evaluate each assignment carried out by a student on a scale of 5 as specified above by ticking appropriate box.
3) The value should also be entered on assignment completion page of the respective Lab course.

Guidelines for Lab administrator

You have to ensure appropriate hardware and software is available to each student in the Lab.

The operating system and software requirements on server side and also client side are as given below:

1) Server and Client Side - (Operating System)Linux (Ubuntu/Red Hat/Fedora) – any distribution
2) Database server – PostgreSQL 7.0 onwards.

Assignment Completion Sheet

Lab Course: USCSP-233 Computer Science Laboratory			
Sr. No.	**Assignment Name**	**Marks (out of 5)**	**Teachers Sign**
1.	Stored functions		
2.	Errors and Exception handling		
3.	Cursors		
4.	Triggers		
Total (Out of 20)			
Total (Out of 10)			

Certificate

This is to certify that

Mr. /Ms.____has successfully completed USCSP-233 Computer Science Laboratory course in year______________________and his/her seat no. is______. He/she has scored Mark_________________ out of 10.

Instructor

Internal Examiner ***External Examiner***

Pre requisite

Instructions for Students:

- All Assignments of RDBMS are based on following data bases.
- Students are advised to create the following data sets and solve the remaining assignments using the same.

Instructions:-

1. Create a relational database in 3NF.
2. Insert sufficient number of records in the table.

1) Student-Teacher Database

Consider the following Entities and their Relationships for Student-Teacher database.
Student (s_no int, s_name varchar (20), s_class varchar (10), s_addr varchar (30))
Teacher (t_no int, t_name varchar (20), qualification varchar (15), experience int)

Relationship between Student and Teacher is many to many with descriptive attribute subject.

Constraints: Primary Key,
s_class should not be null.

2) Project-Employee Database

Consider the following Entities and their Relationships for Project-Employee database.
Project (pno integer, pname char (30), ptype char (20), duration integer)
Employee (eno integer, ename char (20), qualification char (15), joining_date date)

Relationship between Project and Employee is many to many with descriptive attribute start_date date, no_of_hours_worked integer.

Constraints: Primary Key,
duration should be greater than zero,
pname should not be null.

3) Person-Area Database

Consider the following Entities and their Relationships for Person-Area database.
Person (pno integer, pname varchar (20), birthdate date, income money)
Area (aname varchar (20), area_type varchar (5))

An area can have one or more persons living in it, but a person belongs to exactly one area.

Constraints: Primary Key, area_type can be either 'urban' or 'rural'

4) **Book-Author Database**

Consider the following Entities and their Relationships for Book-Author database.

Book(b_no int, b_name varchar (20), pub_name varchar (10), b_price float)
Author (a_no int, a_name varchar (20), qualification varchar (15), address varchar (15))

Relationship between Book and Author is many to many.

Constraints: Primary Key,pub_name should not be null.

5) **Movie-Actor Database**
Consider the following Entities and their Relationships for Movie-Actor database.
Movie (m_name varchar (25), release_year integer, budget money)
Actor (a_name varchar (20), role char(20), charges money, a_address varchar (20))
Producer (producer_id integer, p_name char (30), p_address varchar (20))

Each actor has acted in one or more movies. Each producer has produced many movies and each movie can be produced by more than one producers. Each movie has one or more actors acting in it, in different roles.

Constraints: Primary Key,role and p_name should not be null.

6) **Bank database**
Consider the following Entities and their Relationships for Bank database.
Branch (bid integer, brname char (30), brcity char (10))
Customer (cno integer, cname char (20), caddr char (35), city char(20))
Loan_application (lno integer, l_amt_require money, l_amt_approved money, l_date date)
The relationships are as follows:
Branch, Customer, Loan_application are related with ternary relationship.
Ternary (bid integer, cno integer, lno integer).

7) **Warehouse database.**

Consider the following Entities and their Relationships for Warehouse database.
Cities (city char(20),state char(20))

Warehouses (wid integer, wname char(30),location char(20))

Stores (sid integer , store_name char(20), location_city char(20))

Items (itemno integer, description text, weight decimal(5,2), cost decimal(5,2))

Customer (cno integer, cname char(50),addr varchar(50), cu_city char(20))

Orders (ono int,odate date)

The Relationship is as follows

Cities-warehouses 1 to m

Warehouses-stores 1 to m

Customer– orders 1 to m

Items– orders m - m relationship with descriptive attribute ordered_quantity

Stores-items m - m relationip with descriptive attribute quantity.

8) **Bus Transport System**

Consider the following database of Bus transport system. Many buses run on one route. Drivers are allotted to the buses shift-wise.

Following are the tables:

BUS(bus_no int , capacity int , depot_name varchar(20))
ROUTE (route_no int, source char(20), destination char(20),no_of_stations int)
DRIVER (driver_no int , driver_name char(20), license_no int, address char(20), d_age int , salary float)

The relationships are as follows:

BUS_ROUTE: M-1
BUS_DRIVER: M-M with descriptive attributes Date_of_duty_allotted and Shift – it can be 1 (Morning) 0r 2 (Evening).

Constraints:

1) License_no is unique.
2) Bus capacity is not null.

Assignment No. 1 Stored Functions

Aim: To study syntax, creation and deletion of stored function

Pre-requisite: Knowledge of simple SQL and nested queries.

Guidelines for Teacher / Instructor:

- Demonstration of creation of stored function in a file, how to save it, how to execute it and how todrop it on databases is expected.

Instruction for students:

- Students must read the theory and syntax for creating and dropping functions before their practical slot.

Theory: Introduction to Stored Functions or Procedures

Concept:

Stored Functions are user-defined functions which need to be called explicitly for its execution. PostgreSQL functions are also called as Stored Procedures that allows you to carry out operations that would normally take several queries.

❖ **Syntax for creating a stored function:**

```
CREATE [OR REPLACE] FUNCTION function_name (arguments) RETURNS
return_datatype AS'
DECLARE
        Variable_Declarations;
        [...]
BEGIN
        <function_body>
        [...]
RETURN {variable_name | value};
END;
'LANGUAGE 'plpgsql';
```

Where,

1. **function_name:** Specifies the name of the function.
2. **Arguments:** In PL/pgSQL functions can accept arguments/parameters of different data-types. Function arguments allow a user to pass information to a function while calling a function. Each function parameter has assigned identifier that begins with dollar ($) sign and labeled with the parameter number. Identifier $1 is used for first parameter, Identifier $2 is used for second parameter and so on.

   PL/pgSQL allows us to create Aliases. With the help of aliases, it is possible to assign more than just one name to a variable. Aliases should be declared within the DECLARE section of a block.

Syntax:

```
variable_name ALIAS FOR $1;
```

E.g.

```
Project_name ALIAS FOR $1;
```

3. **[OR REPLACE]:** This option allows modification of an existing function.
4. **AS:** The AS keyword is used for creating a standalone function.
5. **function_body:** Contains the executable part or the program logic.
6. **Return:** The function must contain a return statement.
7. **PL/pgSQL** is the name of the language that the function is implemented in. For backward compatibility, the name can be enclosed by single quotes.
8. **Attributes:** PL/pgSQL provides two attributes to declare variables. Use attributes to assign a variable either the type of a database object, with the %TYPE attribute, or the row structure of a table with the %ROWTYPE attribute.

   a) **The %TYPE attribute:** The %TYPE is used to declare a variable with the type of a referenced database object. It is used to declare variables based on definitions of columns in a table. This attribute is used to declare the variable whose type will be same as that of a previously defined variable or a column in a table.

   Syntax: variable_name table_name.column_name%TYPE;

   E.g.dob student.date_of_birth%type;

   b) **The %ROWTYPE attribute:** It is used to declare a record variable with same structure as the rows in a table that are specified during declaration. It will have structure exactly similar to the table's row.

   Syntax: variable_name table_name%ROWTYPE;

   E.g.student_record student%rowtype;

❖ **<u>Calling Function</u>:** We can either call a function directly using SELECT or use it during assignment of a variable**.** Function can be called by using below command on terminal.
Syntax:

```
select function_name(arguments);
```

❖ **<u>Drop Function:</u>**DROP FUNCTION deletes the definition of an existing function.
Syntax:

```
DROP FUNCTION function_name (argtype[, ...]) [CASCADE | RESTRICT]
```

Where,
function_name: The name of an existing function.
1. **argtype:** The data type(s) of the function's arguments if any.
2. **CASCADE:** Automatically drop objects that depend on the function (such as triggers).
3. **RESTRICT:** Refuse to drop the function if any objects depend on it. This is the default.

Control Structures
Like the most programming languages, PL/pgSQL also provides ways for controlling flow of program execution by using conditional statements and loops.

9. Conditional Statements:

A conditional statement specifies an action (or set of actions) that should be executed depending upon the result of logical condition specified within the statement.

❖ **IF…. THEN Statement:**

It is a statement or block of statements which is executed if given condition evaluates to true. Otherwise control is passed to next statement after END IF.

Syntax:

```
IF condition THEN
        Statements;
END IF;
```

Example:

```
CREATE OR REPLACE FUNCTION if_demo () RETURNS integer AS'
DECLARE
        num1 integer = 34;
        num2 integer = 42;
BEGIN
        IF num1 > num2 THEN
                RAISE NOTICE '' num1 is greater than num2'';
        END IF;
        IF num1 < num2 THEN
                RAISE NOTICE '' num1 is less than num2'';
        END IF;
        IF num1 = num2 THEN
                RAISE NOTICE '' num1 is equal to num2'';
        END IF;
return null;
END;
' LANGUAGE 'plpgsql';
```

```
postgres=# \i c:/data/demo.sql;
CREATE FUNCTION
postgres=# select if_demo();
NOTICE: num1 is less than num2
 if_demo
------

(1 row)
```

❖ <u>**IF…. THEN…. ELSE Statement:**</u>

This statement allows you to execute a block of statements if a condition evaluates to true, otherwise a block of statements in else part will be executed.

<u>**Syntax**</u>**:**

```
IF condition THEN
        Statements
ELSE
        Statements
END IF;
```

<u>**Example**</u>**:**

```
CREATE OR REPLACE function if_else_demo(int) RETURNS void as'
DECLARE
        num alias for $1;
BEGIN
        IF (num % 2 = 0) then
             raise notice ''% is Even'',num;
        ELSE
            raise notice ''% is Odd'',num;
        END IF;
   END;
'language 'plpgsql';
```

```
postgres=# select if_else_demo(99);
NOTICE: 99 is Odd
 if_else_demo
- - - - - - - - -

(1 row)
```

❖ IF…. THEN…. ELSIF…. THEN…ELSE Statement:

IF-THEN-ELSIF provides a convenient method of checking several alternatives in turn. The IF conditions are tested successively until the first true is found, if none is true then statements in ELSE part are executed.

Syntax:

```
IF condition THEN
        Statements;
ELSIF condition THEN
        Statements;
ELSIF condition THEN
        Statements;
ELSE
        Statements;
END IF;
```

Example:

```
CREATE or replace FUNCTION elsif_demo () RETURNS void AS'
DECLARE
     x integer := 72;
     y integer := 72;
BEGIN
     IF x > y THEN
             RAISE NOTICE ''% is greater than %'',x, y;
     ELSIF x < y THEN
             RAISE NOTICE ''% is less than %'',x, y;
     ELSE
             RAISE NOTICE ''% is equal to %'',x, y;
     END IF;
END;
' LANGUAGE 'plpgsql';
```

```
postgres=# select elsif_demo();
NOTICE: 72 is equal to 72
elsif_demo
- - - - - - - -

(1 row)
```

❖ CASE Statement:

The simple form of CASE provides conditional execution based on equality of operands. The search-expression is evaluated (once) and successively compared to each expression in the WHEN clauses. If a match is found, then the corresponding statements are executed, and then control passes to the next statement after END CASE. If no match is found, the ELSE statements are executed; but if ELSE is not present, then a CASE_NOT_FOUND exception is raised.

Syntax:

```
CASE search-expression
WHEN expression [, expression [...]] THEN
```

```
        Statements
[WHEN expression [, expression [...]] THEN
        Statements
... ]
[ELSE
        Statements]
END CASE;
```

Example:

```
CREATE FUNCTION get_price(film_id1 integer) RETURNS varchar AS'
DECLARE
      rate1 float;
      price VARCHAR;
BEGIN
      SELECT INTO rate1 rate
      FROM film
      WHERE filmid = film_id1;
CASE rate1
      WHEN 0.99 THEN
                price = "AVERAGE";
      WHEN 2.99 THEN
                price = "NORMAL";
      WHEN 4.99 THEN
                price = "HIGH";
      ELSE
                price = "UNSPECIFIED";
END CASE;
RETURN price;
END;
'LANGUAGE 'plpgsql';
```

Loop Statements:

Loop condition is used to perform some task repeatedly for fixed number of times.

1. **Simple Loop:** Simple loop is an unconditional loop which starts with keyword LOOP and executes the statements within its body until terminated by an EXIT or RETURN statement.**Syntax:**

```
LOOP
      Statement; [...]
EXIT [label] [WHEN condition];
END LOOP;
```

Example:

```
create or replace function simple_loop() returns void as'
declare
        i int:= 0;
begin
        loop
                i:= i+1;
                        raise notice ''Level : %'',i;
                exit when i=5;
        end loop;
end;

'language 'plpgsql';
```

```
CREATE OR REPLACE FUNCTION exit_demo() RETURNS integer AS '
  BEGIN
        for i in 1..10 loop
           Exit When i=5;
                raise notice''Input is %'',i;
        end loop;
    return null;
    END;
    ' LANGUAGE 'plpgsql';
```

```
postgres=# select exit_demo();
NOTICE: Input is 1
```

```
postgres=# select simple_loop();
NOTICE: Level : 1
NOTICE: Level : 2
NOTICE: Level : 3
NOTICE: Level : 4
NOTICE: Level : 5
 simple_loop
---------

(1 row)
```

2. **While Loop:** The While statement repeats a sequence of statements till the condition evaluates to true. It is also called as entry controlled loop because the condition is checked before each entry tothe loop body.

Syntax:

```
[<<Label>>]
WHILE condition LOOP
        Statements;
[...] END LOOP;
```

Example:

```
create or replace function Disp_Sum(int) returns int as'
declare
        n1 alias for $1;
        sum int= 0;
        cnt int= 1;
begin
        while (cnt <=n1) loop
                sum:= sum+cnt;
                cnt:= cnt+1;
        end loop;
raise notice "Sum of first % numbers is %",n1,sum;
return sum;
end;
'language 'plpgsql';
```

```
postgres=# select disp_sum(6);
NOTICE: Sum of first 6 numbers is 21
disp_sum
- - - - - - -
      21
(1 row)
```

3. **For** Loop:

FOR loop iterates over a range of integer values. The variable name is automatically created with integer data type and exists only inside the loop. The two expressions giving the lower and upper bound of the range are evaluated once when entering the loop. If the BY clause isn't specified the iteration step is 1, otherwise it's the value specified in the BY clause. If REVERSE is specified then the step value is subtracted after each iteration.

Syntax:

```
[<<Label>>]
FOR name IN [REVERSE] expression.. Expression [ BY expression ] LOOP
        Statements
END LOOP [label];
```

Example 1:

```
CREATE FUNCTION Example_for_loop () RETURNS integer AS '
DECLARE
        counter integer;
BEGIN
        FOR counter IN 1..6 BY 2 LOOP
                RAISE NOTICE "Counter: %", counter;
        END LOOP;
return counter;
END;
' LANGUAGE 'plpgsql';

postgres=# select example_for_loop();
NOTICE: Counter: 1
NOTICE: Counter: 3
NOTICE: Counter: 5
 example_for_loop
- - - - - - - - - - - -

(1 row)
```

Example 2:

```
CREATE OR REPLACE FUNCTION Example_for_loop () RETURNS integer AS
 ' DECLARE
        counter integer;
BEGIN
        FOR counter IN reverse 21..12 BY 2 LOOP RAISE NOTICE "Counter: %", counter;
END LOOP;
return counter;
END;
' LANGUAGE 'plpgsql';

        postgres=# select example_for_loop();
        NOTICE: Counter: 21
        NOTICE:  Counter: 19
        NOTICE:  Counter: 17
        NOTICE:  Counter: 15
        NOTICE: Counter: 13
         example_for_loop
        - - - - - - - - - - - -

        (1 row)
```

Looping Through Query Results by Using for Loop:

Using a different type of FOR loop, you can iterate through the results of a query and manipulate that data accordingly.

Syntax:

```
[<<Label>>]
FOR record _variable IN query LOOP
        Statements;
END LOOP;
```

Example:

```
CREATE or replace FUNCTION extract_title (integer) RETURNS text AS'
DECLARE
      sub_id1 ALIAS FOR $1;
      text_output TEXT;
      row_data book%ROWTYPE;
BEGIN
      FOR row_data IN SELECT * FROM book
              WHERE sub_id = sub_id1 LOOP
              text_output := row_data.title;
      END LOOP;
RETURN text_output;
END;
' LANGUAGE 'plpgsql';
```

```
postgres=# select extract_title(22);
 extract_title
----------
 RDBMS
(1 row)
```

Exit Statement:

These statements are used to exit from a loop.

4. **EXIT:** This statement is used to exit from loop without condition. Condition needs to be specified separately.

Syntax:

```
exit;
```

Example:

```
CREATE OR REPLACE FUNCTION exit_demo() RETURNS integer AS '
BEGIN
        for i in 1..10 loop
        if i = 4 then EXIT;
                else
                        raise notice''Input is %'',i;
                end if;
            end loop;
        return null;
        END;
        ' LANGUAGE 'plpgsql';
```

```
postgres=# select exit_demo();
NOTICE:  Input is 1
NOTICE:  Input is 2
NOTICE: Input is 3
exit_demo
-------

(1 row)
```

5. **exit-when:** This statement is used to exit from loop by specifying condition within exit statement.

Syntax:

Example:

```
EXIT WHEN condition;
```

```
NOTICE: Input is 2
NOTICE: Input is 3
NOTICE: Input is 4
exit_demo
-------

(1 row)
```

Example 1: Simple Function.

Consider the following Relational Database:

Student (sno, s_name, s_class, address)

Teacher (tno, t_name, qualification, experience)

Stud_teach(sno, tno, subject)

- **Write a function to count the number of the teachers who are teaching to a student named “ ”. (Accept student name as input parameter). Display appropriate message.**

```
create or replace function stud_det(varchar) returns int as'
declare
        s_name alias for $1;
        cnt int;
begin
        select into cnt count(teacher.tno) from student, teacher, stud_teach where
        student.sno=stud_teach.sno and teacher.tno=stud_teach.tno and
        sname=s_name;
        if cnt=0 then
                    raise notice ''% Student not present'',s_name;
        else
                    return cnt;
        endif;
end;

'language 'plpgsql';


postgres=# select stud_det('Gauri');
 stud_det
-------
        4
(1 row)
```

Example 2: Looping Through Query Results by Using for Loop.

❖ **Accept teacher name as input and print the names of student to whom that teacher teaches.**

```
create or replace function teach_det(varchar) returns int as'
declare
        t_name alias for $1;
        rec record;
begin
        raise notice "Teacher Name || Name of Student";
        for rec in select tname, sname from student, teacher, stud_teach where
        student.sno=stud_teach.sno and teacher.tno=stud_teach.tno and
        tname=t_name loop
                raise notice "% %",rec.tname, rec.sname;
        end loop;
return null;
end;
'language 'plpgsql';
```

```
postgres=# select teach_det('Shriprada');
NOTICE: Teacher Name || Name of Student
NOTICE:    Shriprada              Faisal
NOTICE:    Shriprada              Umar
NOTICE:    Shriprada              Zeba
 teach_det
---------------
(1 row)
```

Exercise

Lab Work

1) **Project-Employee Database**

Consider the following Entities and their Relationships for Project-Employee database.

Project (pno integer, pname char (30), ptype char (20), duration integer)

Employee (eno integer, ename char (20), qualification char (15), joining_date date)

Relationship between Project and Employee is many to many with descriptive attribute start_date date, no_of_hours_worked integer.

Constraints: Primary Key,
duration should be greater than zero,
pname should not be null.

1. Write a stored function to find the number of employees whose joining date is before '01/01/2007'.
2. Write a stored function to accept eno as input parameter and count number of projects on which that employee is working.
3. Write a stored function to display all projects started after date "01/01/2019".

2) Person-Area Database

Consider the following Entities and their Relationships for Person-Areadatabase.

Person (pno integer, pname varchar (20), birthdate date, income money)

Area (aname varchar (20), area_type varchar (5))

An area can have one or more persons living in it, but a person belongs to exactly one area.

Constraints: Primary Key,area_type can be either 'urban' or 'rural'.

1. Write a stored function to print total number of persons of a particular area. Accept area name as input parameter.
2. Write a stored function to update the income of all persons living in urban area by 20%.

Home Assignments

1) Bus Transport Database

Consider the following Entities and their Relationships for Bus Transport database.

Bus (bus_no int ,b_capacity int , depot_name varchar(20))

Route (route_no int, source char (20), destination char (20), no_of_stations int)

Driver (driver_no int ,driver_name char(20), license_no int, address char(20), d_age int , salary float)

Relationship between Bus and Route is many to one and relationship between Bus and Driver is many to many with descriptive attributes date_of_duty_allotted and shift.

Constraints: Primary Key, license_no is unique, b_capacity should not be null, shift can be 1 (Morning) or 2(Evening).

1. Write a stored function to accept route no and display bus information running on that route.
2. Write a stored function to accept shift and depot name and display driver details who having duty allocated after '01/07/2020'.
3. Write a stored function to accept source name and display count of buses running from source place.
4. Write a stored function to accept depot name and display driver details having age more than 50.

2) Bank Database

Consider the following Entities and their Relationships for Bank database.

Branch (br_id integer, br_name char (30), br_city char (10))

Customer (cno integer, c_name char (20), caddr char (35), city char (20))

Loan_application(lno integer, l_amt_required money, l_amt_approved money, l_date date)

Relationship between Branch, Customer and Loan_application is Ternary.
Ternary (br_id integer, cno integer, lno integer)
Constraints: Primary Key,
l_amt_required should be greater than zero.

1. Write a stored function to accept branch name and display customer details whose loan amount required is more than loan approved.
2. Write a stored function to accept branch name and display customer name, loan number, loan amount approved on or after 01/06/2019.
3. Write a stored function to display total loan amount approved by all branches after date 30/05/2019.
4. Write a stored function to display customer details who have applied for loan more than one branches.

3) Business trip database

Consider Business trip database that keeps track of the business trips of salesman in an office.

Following are the tables:

Salesman (sno integer, s_name char (30), start_year integer, dept_no varchar(10))
Trip(tno integer, from_city char (20), to_citychar (20),departure_date date, return_date date)
Dept(dept_no varchar (10), dept_name char(20))
Expense(eid integer, amount money)

Relationships:

Dept-Salesman: 1 to M
Salesman-Trip: 1 to M
Trip-Expense: 1 to 1

Execute the following stored functions.

a) Write a stored function to find a business trip having maximum expenses.
b) Write a stored function to count the total number of business trips from 'Pune' to 'Mumbai'.

4) Railway Reservation Database

Consider a Railway reservation system for passengers. The bogie capacity of all the bogies of a train is same.

TRAIN (train_no int, train_name varchar(20), depart_time time , arrival_time time, source_stn varchar (20),dest_stn varchar (20), no_of_res_bogies int ,bogie_capacity int)

PASSENGER (passenger_id int, passenger_name varchar(20), address varchar(30), age int ,gender char)

Relationships:

Train _Passenger: M-M relationship named ticket with descriptive attributes as follows:
TICKET(train_no int, passenger_id int, ticket_no int ,bogie_no int, no_of_berths int ,tdate date , ticket_amt decimal(7,2),status char)

Constraints:

The status of a berth can be 'W' (waiting) or 'C' (confirmed).

Execute the following stored functions.

1. Write a stored function to calculate the ticket amount paid by all the passengers on 12/12/2019 forall the trains.
2. Write a stored function to update the status of the ticket from ‘waiting’ to ‘confirm’ forpassengernamed “Mr.Mohite”.

Assignment Evaluation

0: Not Done [ ]	1: Incomplete [ ]	2:Late Complete[ ]
3: Needs Improvement [ ]	4: Complete [ ]	5: Well Done [ ]

Signature of the instructor:______________________Date:______________________

Assignment No. 2 Error and Exception Handling

Aim: To study how to handle different errors and exceptions that arises while using database.

Pre-requisite: Knowledge of how to create stored functions.

Guidelines for Teacher / Instructor:

- Demonstration of creation of raise statement on databases is expected.

Instruction for students:

- Students must read the theory and syntax for handling error before their practical slot.

Theory: Error and Exception Handling

Concept:

Any error occurring in a PL/pgSQL function aborts execution of the function. Errors can be trapped and recovered by using a Begin block with an Exception clause.

Syntax:

```
Declare
        Declarations
Begin
        Statements
        Exception
When condition then
        Handler statements
End;
```

If no errors occur, this form of block simply executes all the statements, and then control passes to the next statement. But if error occurs within the statements, further processing of the statements is stopped and control passes to the exception list.

Raise Statement: It is used to raise errors, report messages and exceptions during a PL/pgSQL function's execution.
This statement can raise built in exceptions, such as division_by_zero, not found etc.

Syntax:

```
RAISE level "format string" [, expression [, ...]];
```

Where,

- **level:** The level option specifies the severity of an error.
  Level can be:
  - **DEBUG:** DEBUG level statements send the specified text as a message to the PostgreSQL logand the client program if the client is running in debug mode.

- **NOTICE:** This level statement sends the specified text as a message to the client program.
- **EXCEPTION:** This statement sends the specified text as error message. It causes the current transaction to be aborted.

- **format:** The format is a string that specifies the message. The format uses percentage (%) placeholders that will be substituted by the next arguments. The number of placeholders must match the number of arguments; otherwise PostgreSQL will report the following error message
  example 1: ERROR: too many parameters specified for RAISE.
  example 2: ERROR: control reached end of function without RETURN.

Example:

❖ **In the example given below, the first raise statement gives a debug level message and sends specified text to PostgreSQL log. The second statements send a notice to the user. The third raise statement displays an error and throws an exception, which causes the function to end.**

```
CREATE FUNCTION raise_test(int, int) RETURNS void
AS' DECLARE
        x1 alias for $1;
        x2 alias for $2;
        div INTEGER;
BEGIN
        RAISE DEBUG "The raise_test() function begins from here.";
        if x2 != 0 then
                div = x1 / x2;
                raise notice "Division of % and % is %",x1,x2,div;

        else
                RAISE EXCEPTION “Transaction aborted due to division by zero
        exception”;
        end if;
END;
'LANGUAGE 'plpgsql';
```

```
postgres=# select raise_test(24,3);
NOTICE: Division of 24 and 3 is 8
raise_test
--------

(1 row)
postgres=# select raise_test(18,0);
ERROR: Transaction aborted due to division by zero exception. CONTEXT: PL/pgSQL function
raise_test(integer,integer) line 12 at RAISE
```

❖ **Consider the database**
PROJECT (pno, p_name, ptype, duration)
EMPLOYEE(eno, e_name, qualification, joindate)
PROJ_EMP (pno, eno, start_date, no_of_hours_worked)

Write a stored function to accept project name as input and print the names of employees working on that project. Raise an exception for an invalid project name.

```
create or replace function chk_emp(varchar) returns int as'
declare
      rec record;
      pname1 alias for $1;
      ecount int;
begin
      for rec in select pname from project loop
              if(rec.pname<>pname1)then
                      continue;
              else
                      select into ecount count(proj_emp.eno)
                      from employee,project, proj_emp where
                      project.pno=proj_emp.pno and
                      employee.eno=proj_emp.eno and
                      pname=pname1;
              end if;
      end loop;

      if ecount>0 then
              raise notice"Employee count is : %",ecount;
      else
              raise notice"Invalid Project Name";
      end if;
return null;
end;
```

```
        'language 'plpgsql';

postgres=# select chk_emp('system');
NOTICE: Employee count is : 4
chk_emp
------

(1 row)


postgres=# select chk_emp('web design');
NOTICE: Invalid Project Name
chk_emp
------

(1 row)
```

Exercise

Lab_Work

1. Bus Driver Database :

BUS (bus_no int , capacity int , depot_name varchar(20))
ROUTE (route_no int, source char(20), destination char(20),no_of_stations int)
DRIVER (driver_no int , driver_name char(20), license_no int, address char(20), d_age int , salary float)

The relationships are as follows:
BUS_ROUTE: M-1
BUS_DRIVER: M-M with descriptive attributes Date of duty allotted and Shift – it can be 1 (Morning) or 2 (Evening).

Constraints:
1. License_no is unique. 2. Bus capacity is not null

1. Write a stored function to display the all Dates on which a driver has driven any bus. (Acct driver name as input parameter).Raise an exception in case of invalid driver name.
2. Write a stored function to display the details of the buses that run on route_no = "______". (accept route_no as input parameter). Raise an error in case of route number is not present.

2. Student Teacher Database

Student (sno integer, s_name char(30), s_class char(10), s_addr Char(50))
Teacher (tno integer, t_name char (20), qualification char (15),experience integer)

The relationship is as follows:
Student-Teacher: M-M with descriptive attribute Subject.

1. Write a stored function to find the names of the teachers teaching to a student named "________". (Accept student name as input parameter). Raise an exception if student name does not exist.
2. Write a stored function to count the number of the students who are studying subject named "________" (Accept subject name as input parameter). Display error message if subject name is not valid.

Home Assignments

1. Person-Area Database

Person (pno integer, pname varchar (20), birthdate date, income money)

Area (aname varchar (20), area-type varchar (5))

An area can have one or more persons living in it, but a person belongs to exactly one area. The attribute'area_type' can have values either 'urban' or 'rural'.

1. Write a stored function to print total number of person of a particular area. (Accept area_name as input parameter). Display appropriate message for invalid area name.
2. Write a stored function to print sum of income of person living in "______" area type. (Accept area type as input parameter). Display appropriate message for invalid area type.
3. Write a stored function to display details of person along with area name whose birthday falls in the month of "_________". (Accept month as input parameter). Display error message for invalid month name.

2. Railway Reservation System Database

TRAIN: (train_no int, train_name varchar(20), depart_time time , arrival_time time, source_stn varchar (20),dest_stn varchar (20), no_of_res_bogies int ,bogie_capacity int)

PASSENGER : (passenger_id int, passenger_name varchar(20), address varchar(30), age int ,gender char)

Relationships:

Train _Passenger: M-M relationship named ticket with descriptive attributes as follows

TICKET: (train_no int, passenger_id int, ticket_no int ,bogie_no int, no_of_berths int ,tdate date , ticket_amt decimal(7,2),status char)

Constraints: The status of a berth can be 'W' (waiting) or 'C' (confirmed).

1. Write a stored function to print the details of train wise confirmed bookings on date "____________" (Accept date as input parameter).Raise an error in case of invalid date.
2. Write a stored function to accept date and passenger name and display no of berths reserved and ticket amount paid by him. Raise exception if passenger name is invalid.
3. Write a stored function to display the ticket details of a train. (Accept train name as input

parameter).Raise an exception in case of invalid train name.

3. **Bank Database**

Branch (bid integer, br_name char (30), br_city char (10))
Customer (cno integer, cname char (20), caddr char (35), city char(20))
Loan_application (lno integer, l_amt_require money, l_amt_approved Money, l_date date)

The relationships are as follows:

Branch, customer, loan_application are related with ternary relationship.
Ternary (bid integer, cno integer, lno integer).

1. Write a stored function to accept customer name as input and display the loan details of that customer. (Accept customer name as input parameter). Raise an exception for an invalid customername.
2. Write a stored function to display details of customers of particular branch. (Accept branch nameas input parameter) Display appropriate error message if branch name is invalid.

4. **Movie-Actor Database**

Movies (m_name varchar (25), release_year integer, budget money)
Actor (a_name char (30), role char (30), charges money, a_address varchar(30))
Producer (producer_id integer, name char (30), p_address varchar (30))

The relationships are as follows:

Each actor has acted in one or more movies. Each producer has produced many movies and each movie can be produced by more than one producers. Each movie has one or more actors acting in it, in differentroles.

1. Write a stored function to accept movie name as input and display the details of actors for that movie and sort it by their charges in descending order. (Accept movie name as input parameter).Raise an exception for an invalid movie name.
2. Write a stored function to accept actor / actress name as input and display the names of movies inwhich that actor has acted in. (Accept actor name as input parameter). Raise an exception for an invalid actor name.
3. Write a stored function to accept producer name as input and display the count of movies he/shehas produced. (Accept producer name as input parameter). Raise an exception for an invalid producer name.

Assignment Evaluation

0: Not Done [ ]	1: Incomplete [ ]	2: Late Complete []
3: Needs Improvement [ ]	4: Complete [ ]	5: Well Done [ ]

Signature of Instructor:____________________________ **Date:**___________________

Assignment No. 3 - Cursors

Aim: Learn how to create and execute cursors.

Pre-requisite: Knowledge of stored functions, control statements, SQL and Nested Queries must.

Guidelines for Teachers / Instructors:

- Demonstration of creation and execution of cursors on database is expected.

Instructions for Students:

- Students must read the theory and syntax for creating cursors with and without parameterbefore his/her practical slot.

- Solve SET A, B or C assigned by instructor in allocated slots only.

Theory:Introduction to Cursor

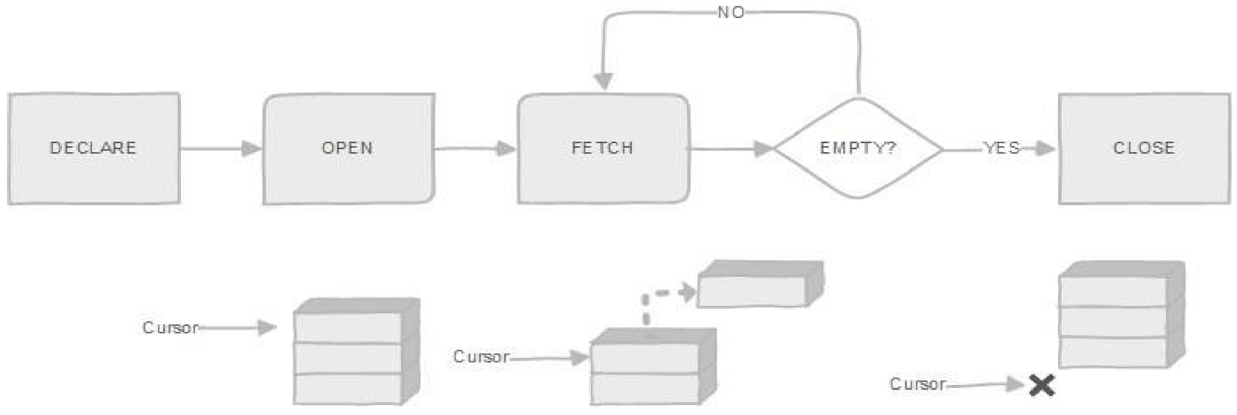
Concept:

In PL/pgSQL, a cursor allows to encapsulate the query rather than executing a whole query at once. After encapsulating query it is possible to read few rows from result set. The main 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 loopsautomatically use a cursor internally to avoid memory problems.)

A PL/pgSQL cursor allows user to access or retrieve multiple rows of data from a table, so thatuser can process different operations on each individual row at a time.

We use cursors when we want to divide a large result set into parts and process eachpart individually. If we process it at once, we may have a memory overflow error.

The following diagram illustrates how to use a cursor in PostgreSQL:



\

➢ ***Steps for cursor:***

1. First, declare a cursor.
2. Next, open the cursor.
3. Then, fetch rows from the result set into a target.
4. After that, check if there are more rows left to fetch. If yes, go to step 3, otherwise go to step 5.
5. Finally, close the cursor.

We will examine each step in more detail in the following sections.

1. Declaration:

We can declare a cursor variable by using two ways:

a) Bound Cursor Variable:

To create a bound cursor variable, use the following cursor declaration.

Syntax:

```
DECLARE
cursor_name CURSOR [(arguments)]
```

For Example:

i)
```
declare
movie_cursor cursor for
             select *
             from movie;
```
Here, the movie_cursor is a cursor that encapsulates all rows in the movie table.

j)
```
declare
movie_cursor1 cursor(year integer) for
              select *
              from movie
             where rel_year = year;
```

Here, the movie_cursor1 is a cursor that encapsulates movie with a particular release year in the movie table.

b) Unbound Cursor Variable:

To create an unbound cursor variable, PL/pgSQL provides a special type called REFCURSOR i.e. Declare the cursor variable of type REFCURSOR.

Syntax:
```
DECLARE
cursor_name REFCURSOR;
```

For Example: declare my_cursor **refcursor** ;

2. ***Opening Cursor:***
Cursors must be opened before they can be used to fetch rows.PL/pgSQL has three forms of the OPEN statement.

a) **Opening unbound cursors :**
We open an unbound cursor using the following syntax:

i) Because the unbound cursor variable is not bounded to any query when we declared it, we have to specify the query when we open it.

Syntax : **OPEN** unbound_cur_variable **FOR** query;
For Example: **open** my_cursor **for**
select *
from city
where country = p_country;

ii) PostgreSQL allows you to open a cursor and bound it to a dynamic query.

Syntax :**OPEN** unbound_cur_variable **FOR EXECUTE** query_string
USING expression [,…]];

For Example : query_str = 'select * from city order by $1;
Open cur_city **for execute** query_str **using** sort_field;

In the above example, we build a dynamic query that sorts rows based on asort_field parameter and open the cursor that executes the dynamic query.

b) **Opening a Bound Cursor**
Because a bound cursor already bounds to a query when we declared it, so when we open it, we just need to pass the arguments to the query if necessary.

Syntax: **OPEN** bound_cursor_name [(argument values)];
For Example: i)**open** movie_cursor;
ii) **open** movie_cursor1 (year :=2020);

3. ***Fetching Rows:***
After opening a cursor, we can manipulate it using FETCH, MOVE, UPDATE, DELETE statement.

I. Fetching the next row:
The FETCH statement gets the next row from the cursor and assigns it a target_variable which could be a record, a row variable, or a comma-separated list of variables. If no more rows found, the target_variable is set to NULL(s).

Syntax: fetch[direction{from | in}] cur_variable **into** target_variable;

The *direction* clause can be any of the following variants:

NEXT, PRIOR, FIRST, LAST, ABSOLUTE*count*, RELATIVE*count*, FORWARD, or BACKWARD.

By default, a cursor gets the NEXT row if we don't specify the direction explicitly.

For Example:i)**fetch** movie_cursor **into** row_movie;
ii)**fetch last** from row_movie **into** title, rel_year;

II. Move :

If you want to move the cursor only without retrieving any row, you use the MOVE statement.

Syntax: move[direction { from | in }] cursor_variable

The *direction* clause can be any of the variants NEXT, PRIOR, FIRST, LAST, ABSOLUTE*count*, RELATIVE*count*, ALL, FORWARD [*count*| ALL], or BACKWARD [*count* | ALL].

For Example : i) **move** movie_cursor1;
ii) **move last** from movie_cursor;
iii)**move relative** -1 from movie_cursor;
iv)**move forward** 3 from movie_cursor;

III. Deleting or updating the row

Once a cursor is positioned, we can delete or update row identifying by the cursor using DELETE WHERE CURRENT OF or UPDATE WHERE CURRENT OF statement as follows:

Syntax: update table_name
set column_name = value,…
where current of cursor_variable;

delete from table_name
where current of cursor_variable;

For Example: update movie
set rel_year =p_year
where current of movie_cursor;

4. **Closing cursor :**

To close an opening cursor, we use **CLOSE** statement as follows:

Syntax : **close** cursor_variable;

For Example : close movie_cursor;

The CLOSE statement releases resources or frees up cursor variable to allow it to be opened again using OPEN statement.

Examples for Practice:

1) **Using Parameterized Cursor :**

   Consider the following Relational Database:
   Doctor (d_no, d_name, d_city)
   Hospital (h_no, h_name, h_city)
   DH(d_no, h_no**)**

- **Display Hospital wise doctor details.**

```
 create or replace function Doc_cursor() returns
void as'
declare
c1 cursor for select * from Hospital;
c2 cursor(hno Hospital.h_no%type) for
select h_name,DH.d_no,d_name
 from Hospital,Doctor, DH
 where Hospital.h_no=DH.h_no
 and  Doctor.d_no=DH.d_no
 and DH.h_no=hno;
 rec1 Hospital %rowtype;
 rec2 record;
  begin
      open c1;
            raise notice ''Hospital Name Doctor No Doctor Name '';
            loop
                  fetch c1 into rec1;
                  exit when not found;
                  open c2(rec1.h_no);
                      loop
                            fetch c2 into rec2;
                            exit when not found;
                            raise notice ''% %%'',rec2.h_name,rec2.d_no,rec2.d_name;
                         endloop;
                  close c2;
             end loop;
      close c1;
end;'language 'plpgsql';
```

```
pgsql=# select Doc_cursor ();
```

2) **Using Multiple Cursors:**

   Consider the following Relational Database:

Route (rno, source, destination, no_of_stations)
Bus (bno, capacity, depot_name, rno)

- **Display the details of the buses that run on route_no=1 and route_no=2 using multiple cursors.**

```
create or replace function Disp_route() returns void as' declarec1 cursor for
select * from bus where rno=1;
c2 cursor for select * from bus where rno=2;
rec1 record;rec2 record;
begin
      open c1;
          raise notice "Details of Buses run on route 1 are:";
          raise notice "Bus_No Capacity Depot Name";
    loop
         fetch c1 into rec1;
         exit when not found;
          raise notice " % % % ",rec1.bno,rec1.capacity,rec1.depot_name;
    end loop;
    open c2;
            raise notice "Details of Buses run on route 2 are:";
           raise notice "Bus No Capacity Depot Name";
           loop
                 fetch c2 into rec2;
                 exit when not found;
                 raise notice "% % % ",rec2.bno,rec2.capacity,rec2.depot_name;
          end loop;
    closec2;
close c1;
end;'language 'plpgsql';
```

```
pgsql=# select Disp_route();
```

3) **Consider the relation**
Employee (<u>eno</u>, ename, deptno,salary).

- **Write a cursor to print the details of the employee along with commission earned for each employee. Commission is 20% of salary for employees of dept no = 5; its 50% of salary for employees of deptno=8; its 30% of salary for employees of deptno=10.**

```
   create function cursor_demo( ) returns integer as'
   declare
  emp_recEmployee%rowtype
 C1 cursor for Select *
                  from employee;
Comm Number (6,2);
begin
      open C1;
            loop
                  fetch C1 into emp_rec;
```

```
                        if emp_rec.deptno = 5 then
                            Comm:=emp_rec.salary * 0.2;
                            else if emp_rec.deptno = 8 then
                                    Comm :=emp_rec.salary * 0.5;
                                    else If emp_rec.deptno = 10 then
                                            Comm :=emp_rec.salary * 0.3;
                                     end if;
                             end if;
                       endif;
                       raise notice "emp_rec.ename||emp_rec.deptno||emp_rec.salary||comm. ";
                       exit when not found;
                end loop;
        close C1;
  end;'language 'plpgsql';
```

Exercises

Lab Work **(Number of Slots – 2)**

1) **Movie - Actor Database:**

Movie (m_name char (25), release_year integer, budget money)
Actor (a_name varchar (20), role char (20), charges money, a_address varchar (20))
Producer (producer_id integer, p_name char (30), p_address varchar (20))

Each actor has acted in one or more movies. Each producer has produced many movies and each movie can be produced by more than one producers. Each movie has one or more actors acting in it, in different roles.

Constraints: Primary Key,
role should be 'Main','Supportive','Villan','Comedy'
p_name should not be null.
budget,charges > 0

a) Write a cursor to pass a_name as a parameter to a function and returns the names of movies in which given actor is acting.
b) Write a cursor to display producer names who have produced movies in which 'Amitabh' is acted.

2) **Railway Reservation Database**

Consider the following Entities and their Relationships Railway -Reservation Database:

Train (tno int, tname varchar (20), depart_time time, arrival_time time, source_stn char (10), dest_stn char (10), no_of_res_bogies int ,bogie_capacity int)

Passenger (passenger_id int, passenger_name varchar (20), address varchar (30), age int, gender char)

Relationship between Train and Passenger is many to many with descriptive attribute ticket.

Ticket (train_no int, passenger_id int, ticket_no int,bogie_no int, no_of_berths int, tdate date, ticket_amt decimal (7,2),status char)

Constraints: Primary Key, Status of a berth can be 'W' (waiting) or 'C' (confirmed).

a) Write a stored function using cursor to accept date and displays the names of all passengers who have booked ticket for that date.
b) Write a stored function using cursors to accept a train name and date, and prints details of all tickets booked for that train on that date.

Home Assignments

1. **Bank Database**

Consider the following Entities and their Relationships for Bank database.

Branch (br_id integer, br_name char (30), br_city char (10))

Customer (cno integer, c_name char (20), caddr char (35), city char (20))

Loan_application(lno integer, l_amt_required money, l_amt_approved money, l_date date)

Relationship between Branch, Customer and Loan_application is Ternary.

Ternary (br_id integer, cno integer, lno integer)

Constraints: Primary Key,
l_amt_required should be greater than zero.

a) Write a cursor to accept br_name from user and display the name of customers who have approved loan amount greater than 50000 from the given branch.
b) Write a stored function using cursor to print the loan details of all loans taken by a customer. (Accept Customer name as input parameter).

2. **Student-Teacher Database**

Consider the following Entities and their Relationships for Student-Teacher database.

Student (s_no integer, s_name char (20), address char (25), class char (10))

Teacher (t_no integer,t_namechar (10), qualification char (10),experience integer)

Relationship between Student and Teacher is many to many with descriptive attribute subject and marks_scored.

Constraints: Primary Key,s_name,t_name should not be null,marks_scored> 0

a. Write a cursor which will display s_na names of teachers who teach 'RDBMS' subject.
b. Write a cursor to accept the class from user and display the student names,subject and marks of that class.

3. **Using Business-Trip Database:**

Consider the following Entities and their Relationships for Business-Trips database.
Salesman (<u>sno</u> integer, s_namevarchar (30), start_year integer)
Trip (<u>tno</u> integer, from_citychar (20), to_citychar (20), departure_date date, return_date date)
Dept(<u>dept_no</u> varchar (10), dept_name char (20))
Expense (<u>eid</u> integer, amount money)

Relationship between Dept and Salesman is one to many, Salesman and Trip is one to many, and Trip and Expense is one to one.

Constraints: Primary Key,
s_name,dept_name should not be null.
amount > 0

a. Write a stored function with cursor, which accepts dept_no as input and prints the names of all salesmen working in that department.
b. Write a cursor to accept departure_date as parameter and display the details of trips for given departure_date.

4. **Warehouse Database**

Consider the following Entities and their Relationships for Warehousedatabase.
Cities (<u>city</u> char (20), state char (20))
Warehouses (<u>wid</u> integer, wname char (30), location char (20))
Stores (<u>sid</u> integer, store_name char (20), location_city char (20))
Items (<u>itemno</u> integer, description text, weight decimal (5, 2), cost decimal (5, 2))
Customer (<u>cno</u> integer, cname char (50), addrvarchar (50), c_city char (20))
Orders (<u>ono</u> int, odate date)

Relationship between:

Cities-Warehouses is 1 - M
Warehouses-Stores is 1- M
Customer-Orders is 1- M
Items-Orders is M – M with descriptive attribute ordered_quantity,
Stores-Items isM – M with descriptive attribute quantity.

Constraints: Primary Key, wname should not be null.

a. Write a cursor to accept a city from the user and list all warehouses in that city.
b. Write a cursor to find the name of customer who orderd items between Rs. 50000 to Rs. 100000

Assignment Evaluation

0: Not Done [ ]	1: Incomplete [ ]	2:LateComplete[]
3: Needs Improvement [ ]	4: Complete [ ]	5: Well Done [ ]

Signature of Instructor:____________________________ **Date:**____________________

Assignment No. 4 - Triggers

Aim: To study creating, execution and dropping of triggers.

Pre-requisite: Knowledge of simple SQL, Stored functions and Errors and Exception handling must.

Guidelines for Teachers / Instructors:

- Demonstration of creation of triggers, execution and dropping of triggers on database is expected.

Instructions for Students:

- Students must read the theory and syntax for creating and dropping triggers before his/her practical slot.
- Solve SET A, B or C assigned by instructor in allocated slots only.

Concept

- PL/pgSQL can be used to define trigger procedures. PostgreSQL **Triggers** are databasecallback functions, which are automatically invoked when a specified database event (insert, delete, and update) occurs.
- A trigger procedure is created with the **CREATE FUNCTION** command, declaring it as a function with no arguments and a return type of trigger.
- A trigger that is marked FOR EACH ROW is called once for every row that the operation modifies. In contrast, a trigger that is marked FOR EACH STATEMENT only executes once for any given operation, regardless of how many rows it modifies.

Syntax of Creating Trigger:

```
CREATETRIGGER trigger_name
{BEFORE |AFTER} {event_ name} ON table_name
FOR EACH {ROW | STATEMENT}
EXECUTE PROCEDURE function_name (arguments);
```

Parameters Used:

1. **trigger_name:** User defined name of the trigger.
2. **before/after:** Determines whether the function is called before or after an event.
3. **event_name:** Database events can be Insert, Update and Delete.
4. **table_name:** The name of the table the trigger is for.
5. **for each row /for each statement:** specifies whether the trigger procedure should be fired once for every row affected by the trigger event or just once per SQL statement. If neither is specified, **for each** statement is the default.
6. **function_name:** Name of the function to execute.

Syntax of Drop Trigger:
DROP TRIGGER trigger_name ON table_name;
Parameters Used:

1. **trigger_name:** The name of the trigger to remove.
2. **table_name:** The name of the table for which trigger is defined.

Variables used in Trigger: When a PL/pgSQL function is called as a trigger, several special variables are created automatically in the top-level block. They are:

Variable Name	Description
NEW	Data type RECORD; variable holding the new database row for INSERT/UPDATE operations in row-level triggers. This variable is NULL in statement-level triggers and for DELETE operations.
OLD	Data type RECORD; variable holding the old database row for UPDATE/DELETE operations in row-level triggers. This variable is NULL in statement-level triggers and for INSERT operations
TG_NAME	Data type name; variable that contains the name of the trigger actually fired.
TG_WHEN	Data type text; a string of BEFORE, AFTER, or INSTEAD OF, depending on the trigger's definition.
TG_LEVEL	Data type text; a string of either ROW or STATEMENT depending on the trigger's definition.
TG_OP	Data type text; a string of INSERT, UPDATE, DELETE, or TRUNCATE telling for which operation the trigger was fired.
TG_TABLE_NAME	Data type name; the name of the table that caused the trigger invocation.
TG_TABLE_SCHEMA	Data type name; the name of the schema of the table that caused the trigger invocation.
TG_NARGS	Data type integer; the number of arguments given to the trigger procedure in the CREATE TRIGGER statement.
TG_ARGV []	Data type array of text; the arguments from the CREATE TRIGGER statement.

Example 1:
Consider the following Relational Database:
Department (dno, dname)
Employee (eno, ename, sal, dno)

Delete Employee record from Employee table and display message to user ‘Employee record being deleted’.

Function:
```
create or replace function Emp_Del() returns trigger as'
declare
begin
raise notice "Employee record being deleted";
return old;
end;'
language 'plpgsql';
```

Trigger:
```
create trigger trig_del
before delete on Employee
for each row
execute procedure Emp_Del();
```

After creating function and trigger type delete query on terminal.

```
Lab=# delete from Employee;
```

The trigger gets fired and executes procedure/function Emp_Del() and text message displayed to the user.

Example 2:
Consider the following Relational Database:
Student (rollno, s_name , class)
Subject (scode , subject_name)
Stud_Sub(rollno, scode, marks)

The below example ensures that if student marks entered less than 0 or greater than 100, trigger gets fired.

Create Function:
```
create or replace function stud_marks() returns trigger as'
declare
begin
if(NEW.marks_scored < 0 or NEW.marks_scored >100) then
raise exception "Student marks should not be less than 0 or greater than 100";
end if;
return new;
end;'
language 'plpgsql';
```

Create Trigger:
```
create trigger trig_stud_marks
before insert on stud_sub
for each row
execute procedure stud_marks();
```

After creating function and trigger type insert query on terminal.

```
Lab=# insert into stud_sub values(1,301,101);
Lab=# insert into stud_sub values(1,301,-1);
```

The trigger gets fired and executes procedure/function stud_marks() and error message displayed to the user if student marks entered less than 0 or greater than 100.

Exercises

Lab Assignment

1) Bank Database

Consider the following Entities and their Relationships for Bank database.

Branch (br_id integer, br_name char (30), br_city char (10))

Customer (cno integer, c_name char (20), caddr char (35), city char (20))

Loan_application (lno integer, l_amt_required money, l_amt_approved money, l_date date)

Relationship between Branch, Customer and Loan_application is Ternary.

Ternary (br_id integer, cno integer, lno integer)

Constraints: Primary Key,
l_amt_required should be greater than zero.

Create trigger for the following:

1. Write a trigger before insert record of customer. If the customer number is less than or equal to zero and customer name is null then give the appropriate message.
2. Write a trigger which will execute when you update customer number from customer. Display message "You can't change existing customer number".
3. Write a trigger to validate the loan amount approved. It must be less than the loan amount required.

2) Student Competition Database

Consider the following Entities and their Relationships for Student-Competition database.

Student (sreg_no int ,s_name varchar(20), s_class char(10))

Competition (c_no int ,c_name varchar(20), c_type char(10))

Relationship between Student and Competition is many to many with descriptive attributes rank and year.

Constraints: Primary Key,
c_type should not be null,
c_type can be 'sport' or 'academic'.

Create trigger for the following:

1. Write a trigger that restricts insertion of rank value greater than 3. (Raise user defined exception and give appropriate massage)
2. Write a trigger on relationship table .If the year entered is greater than current year, it should display message "Year is Invalid".

Home Assignmants:

1) Student-Teacher Database
Consider the following Entities and their Relationships for Student-Teacher database.
Student (s_no int, s_name varchar (20), s_class varchar (10), s_addr varchar (30))
Teacher (t_no int, t_name varchar (20), qualification varchar (15), experience int)

Relationship between Student and Teacher is many to many with descriptive attribute subject.

Constraints: Primary Key,
s_class should not be null.

Create trigger for the following:

1. Write a trigger before insert the record of Student. If the sno is less than or equal to zero give the message "Invalid Number".
2. Write a trigger before update a student's s_class from student table. Display appropriate message.
3. Write a trigger before inserting into a teacher table to check experience. Experience should be minimum 2 year. Display appropriate message.

2) Project-Employee Database Consider the following Entities and their Relationships for Project-Employee database.
Project (pno integer, pname char (30), ptype char (20), duration integer)
Employee (eno integer, ename char (20), qualification char (15), joining_date date)

Relationship between Project and Employee is many to many with descriptive attribute start_date date, no_of_hours_worked integer.
Constraints: Primary Key,
pname should not be null.

Create trigger for the following:

1. Write a trigger before inserting into an employee table to check current date should be always greater than joining date. Display appropriate message.
2. Write a trigger before inserting into a project table to check duration should be always greater than zero. Display appropriate message.
3. Write a trigger before deleting an employee record from employee table. Raise a notice and display the message "Employee record is being deleted".

3) **Railway Reservation Database**
Consider the following Entities and their Relationships for Railway Reservation database.Train (tno int, tname varchar (20), depart_time time, arrival_time time, source_stn char (10),dest_stn char (10), no_of_res_bogies int ,bogie_capacity int)
Passenger (passenger_id int, passenger_name varchar (20), address varchar (30), age int, genderchar)

Relationship between Train and Passenger is many to many with descriptive attribute ticket.

Ticket (train_no int, passenger_id int, ticket_no int,bogie_no int, no_of_berths int, tdate date,ticket_amt decimal (7,2),status char)

Constraints: Primary Key,
Status of a berth can be 'W' (waiting) or 'C' (confirmed)

Create trigger for the following:

1. Write a trigger to restrict the bogie capacity of any train to 30.
2. Write a trigger after insert on passenger to display message “Age above 5 will be charged full fare” if age of passenger is more than 5.

4) **Bus Transport Database**
Consider the following Entities and their Relationships for Bus Transport database.
Bus (bus_no int , b_capacity int , depot_name varchar(20))
Route (route_no int, source char (20), destination char (20), no_of_stations int)
Driver (driver_no int ,driver_name char(20), license_no int, address char(20), d_age int , salary float)

Relationship between Bus and Route is many to one and relationship between Bus and Driver is many to many with descriptive attributes date_of_duty_allotted and shift.

Constraints: Primary Key, license_no is unique, b_capacity should not be null,shift can be 1 (Morning) or 2(Evening).

1. Write a trigger after insert or update the record of driver if the age is between 18 and 50 give the message “valid entry” otherwise give appropriate message.
2. Write a trigger which will prevent deleting drivers living in _______.

Assignment Evaluation

0: Not Done [ ]	1: Incomplete [ ]	2: Late Complete [ ]
3: Needs Improvement [ ]	4: Complete [ ]	5: Well done [ ]

Signature of the instructor:_____________________Date:_____________________