

Lab - 3	SQL_Assignment_3
IT214 Database Management System, Autumn'2020; Instructor: minal_bhise@daiict, TA: mayank@daiict	

- Objectives:**
- I) Understand & Run functions.
 - II) Use triggers to execute functions.
 - III) Understand Store Procedures (For PostgreSQL 11+).

Submission: Each student team needs to upload a **single .pdf** file, which will contain the following things for all the functions and triggers listed in your specific section's lab file.

- 1) Write English query, SQL function, &/or Trigger SQL statement in the given sequence.
- 2) Screenshot of results.
- 3) Count of tuples in the results.

I. Understand & Run functions.

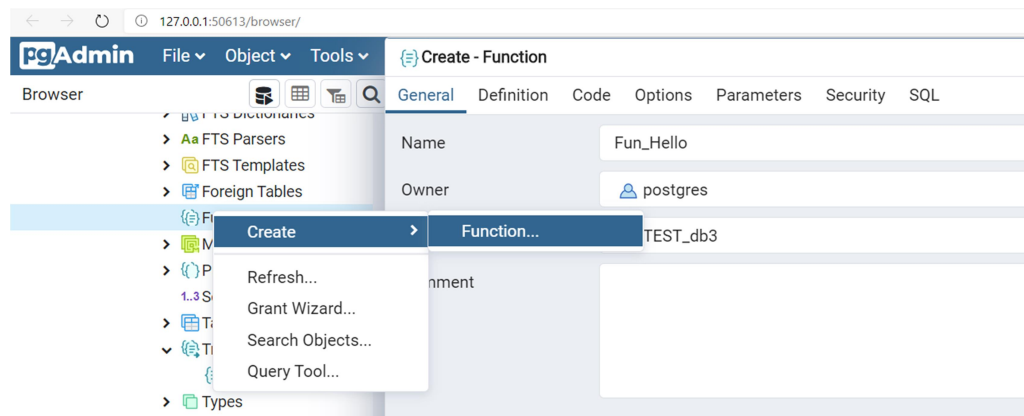
a. Create a simple function to print "Hello World" using GUI.

SQL Functions work similarly to normal coding functions.

Step1.

Using GUI – Right click on Functions => Create => Functions

Write Name – Fun_Hello



Step2.

Using GUI – Go to next tab => Definition

Choose Return Type => character varying

& Language => **plpgsql (It will throw errors if plpgsql not selected)**

21/01/2024/

Create - Function

General

Definition

Code

Options

Parameters

Security

SQL

Return type

character varying

Language

plpgsql

Arguments

+

Data type	Mode	Argument name	Default
-----------	------	---------------	---------

Step3.

Using GUI – Go to next tab => Code

Write code

BEGIN

Return 'Hello World';

END

Using GUI => Save.

Create - Function

General

Definition

Code

Options

Parameters

Security

SQL

```

1 BEGIN
2
3 Return 'Hello World';
4
5 END

```

i ?

Cancel

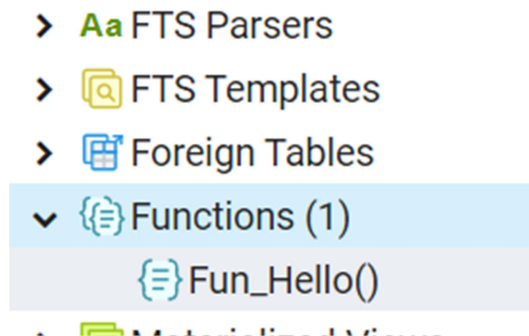
Reset

Save

Desktop

Step4.

Using GUI – Refresh to Check if a function is created under “Functions”.



b. Create simple function to print “Hello World” using SQL.

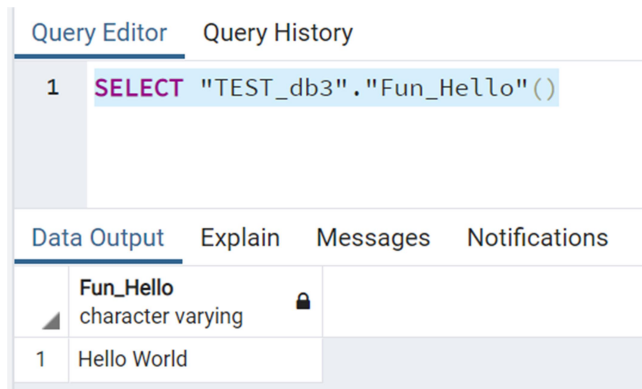
Step1.

```
set search_path to "TEST_db3";
```

```
CREATE OR REPLACE FUNCTION "TEST_db3"."Fun_Hello2"()
  RETURNS character varying
  LANGUAGE 'plpgsql'
AS $BODY$
BEGIN
return 'Hello World2';
END;
$BODY$;
```

c. Using the function. “Function calls”.

```
SELECT "TEST_db3"."Fun_Hello"();
```



d. Create functions with parameters that return the SUM of two values.



Step1. CREATE function.

```
CREATE OR REPLACE FUNCTION "TEST_db3"."Fun_SUM"(a integer, b
integer)
  RETURNS integer
  LANGUAGE 'plpgsql'
AS $BODY$
BEGIN
```

```
Return (a+b);
END
$BODY$;
```

Step2. CALL :

```
SELECT "TEST_db3"."Fun_SUM"(7,8);
```

Query Editor		Query History		
1	SELECT "TEST_db3"."Fun_SUM"(7,8);			
Data Output		Explain	Messages	Notifications
	Fun_SUM integer 			
1	15			

e. Create function with If condition to find the largest number of two.

Step1.

```
CREATE OR REPLACE function "TEST_db3"."fun_findMax"(a int, b int)
RETURNS integer
LANGUAGE 'plpgsql'
```

```
AS $BODY$
```

```
DECLARE
c integer;
BEGIN
```

```
if(a>b) then
    c=a;
else
    c=b;
end if;
```

```
RETURN c;
```

```
END;
$BODY$;
```

Step2. CALL :

```
select "TEST_db3"."fun_findMax"(14,7);
```

Data Output		Explain	
	fun_findMax integer		
1	14		

f. Create functions with a table as a return value.

Step1.

```
CREATE OR REPLACE function "TEST_db3"."fun_rtbl1"()  
RETURNS TABLE (a int, b character varying(30))  
LANGUAGE 'plpgsql'  
AS $BODY$
```

```
BEGIN  
RETURN QUERY EXECUTE format ('SELECT m_id, m_name  
FROM "TEST_db3".test_m');  
END;  
$BODY$;
```

Step2. CALL :

```
"TEST_db3"."fun_rtbl1"()
```

The screenshot shows a PostgreSQL query editor with the following SQL code:

```
1 CREATE OR REPLACE function "TEST_db3"."fun_rtbl1"()  
2 RETURNS TABLE (a int, b character varying(30))  
3 LANGUAGE 'plpgsql'  
4 AS $BODY$  
5 BEGIN  
6  
7 RETURN QUERY EXECUTE format ('SELECT m_id, m_name  
8 FROM "TEST_db3".test_m');  
9  
10 END;  
11 $BODY$;  
12  
13  
14 select "TEST_db3"."fun_rtbl1"()
```

Below the query editor, the "Data Output" tab is active, displaying the results of the function call. The results are shown in a table with 10 rows and 2 columns, labeled "fun_rtbl1 record".

fun_rtbl1 record
(1,mayank)
(2,mayank2)
(3,Mayank)
(4,Test_Proc_...
(5,TR6)
(6,tr7)
(7,TR8)
(10,Test_M1)
(11,Test_M1)
(12,Test_M1)

g. Create functions with a **Temporary table** and use of **FOR Loop**.

Step1.

```
CREATE OR REPLACE FUNCTION "TEST_db3".fun_loop2( )  
  RETURNS TABLE(a integer, b character varying)  
  LANGUAGE 'plpgsql'
```

```
AS $BODY$
```

```
DECLARE
```

```
R_LIST2 record;
```

```
BEGIN
```

```
CREATE TEMP TABLE test1 (a1 int, b1 character varying(30)) ON COMMIT  
DROP;
```

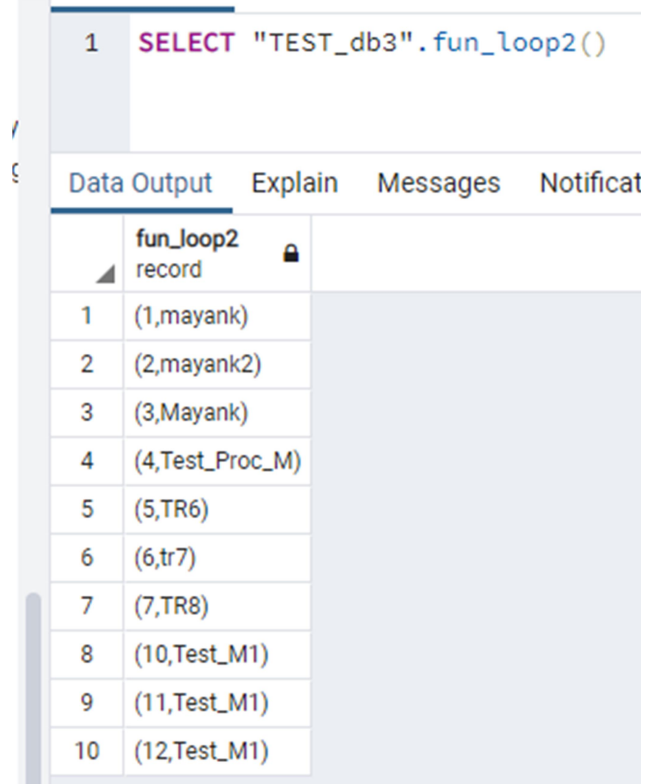
```
FOR R_LIST2 in (select m_id, m_name from "TEST_db3".test_m)  
loop
```

```
Insert into test1 (a1, b1) values (R_LIST2.m_id,R_LIST2.m_name);  
end loop;
```

```
RETURN QUERY TABLE test1;
```

```
END;
```

```
$BODY$;
```



The screenshot shows a SQL IDE interface. At the top, a query is entered: `1 SELECT "TEST_db3".fun_loop2()`. Below the query editor, there are tabs for 'Data Output', 'Explain', 'Messages', and 'Notificat'. The 'Data Output' tab is selected, showing a table with 10 rows of results. The table has two columns: 'fun_loop2' and 'record'. The results are as follows:

	fun_loop2	record
1	(1,mayank)	
2	(2,mayank2)	
3	(3,Mayank)	
4	(4,Test_Proc_M)	
5	(5,TR6)	
6	(6,tr7)	
7	(7,TR8)	
8	(10,Test_M1)	
9	(11,Test_M1)	
10	(12,Test_M1)	

II. Use triggers to execute functions.

The main difference between a normal function & a trigger function is its return type. The trigger function gets automatically called based on some events set earlier, like Insert, Update, &/or Delete on some table.

a. Create a trigger function.

Step1. Create.

```
CREATE OR REPLACE FUNCTION "TEST_db3".Trig_fun2()  
RETURNS trigger  
LANGUAGE 'plpgsql'
```

```
AS $BODY$  
BEGIN
```

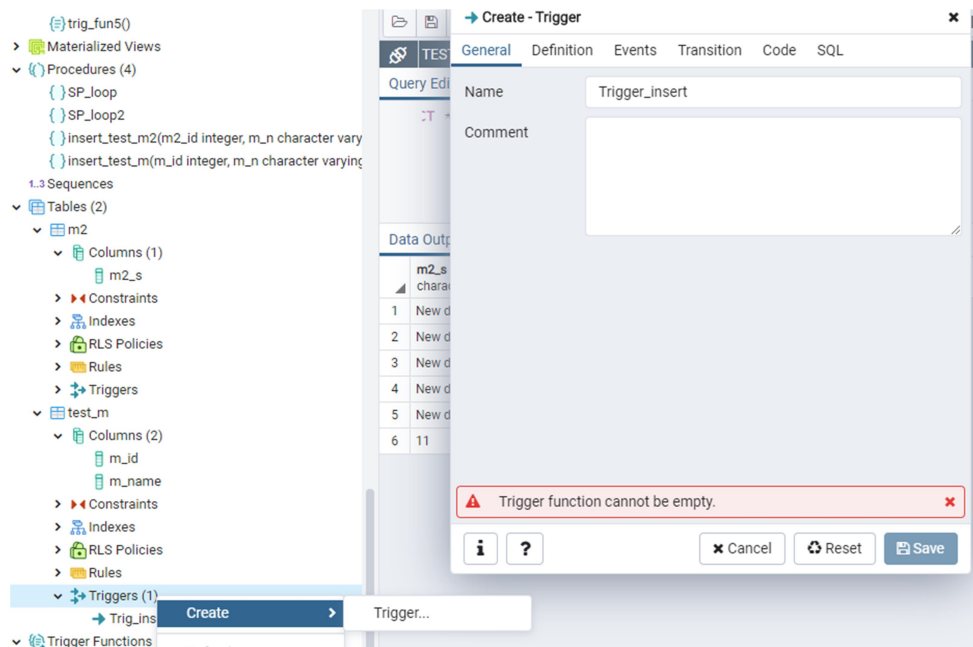
```
Insert into "TEST_db3".m2 (m2_s) values (concat('New id  
added=',NEW.m_id)); --NEW.m_id is the new data values  
RETURN NEW;  
END  
$BODY$;
```

Step2. Create Trigger CALL

Create a trigger to call for the above-created function.

GUI => Right Click on Triggers => Create => Trigger...

Add Name => Trigger_Insert



Step3. Select Trigger Function

Go to Definition tab => Select the Trigger Function

The screenshot shows the 'Create - Trigger' dialog box with the 'Definition' tab selected. The 'Row trigger?' checkbox is checked (Yes). The 'Constraint trigger?' checkbox is unchecked (No). The 'Deferrable?' checkbox is unchecked (No). The 'Deferred?' checkbox is unchecked (No). The 'Trigger function' field contains the text '"TEST_db3".trig_fun2'. Below it, the 'Arguments' list shows four items: '"TEST_db3"."Trig_fun1"', '"TEST_db3".trig_fun2', '"TEST_db3".trig_fun2', and '"TEST_db3".trig_fun6'. A red error message at the bottom states 'Specify at least one event.' The dialog has buttons for 'Cancel', 'Reset', and 'Save'.

Step4. Select the events when the function needs to be called.

Fires(Executes) when ? **BEFORE** or **AFTER** => **AFTER**

& **Switch on (Yes)** the events when you want to run the trigger function.

The screenshot shows the 'Create - Trigger' dialog box with the 'Events' tab selected. The 'Fires' dropdown menu is set to 'AFTER'. The 'Events' section has four checkboxes: 'INSERT' (checked Yes), 'UPDATE' (unchecked No), 'DELETE' (unchecked No), and 'TRUNCATE' (unchecked No). The 'When' field contains the number '1'. The 'Columns' field is empty. The dialog has buttons for 'Cancel', 'Reset', and 'Save'.

Step2. OR Add the trigger function call using below given code.

```
CREATE TRIGGER "Trigger_insert"  
AFTER INSERT  
ON "TEST_db3".test_m  
FOR EACH ROW  
EXECUTE PROCEDURE "TEST_db3".trig_fun2();
```

Step5. Auto call to trigger a function when inserts happen on the table where we had created the trigger call.

```
INSERT INTO "TEST_db3".test_m(  
m_id, m_name)  
VALUES (12, 'Test_12');
```

Step6. Check that the trigger function added a new record automatically in the m2 table.

```
SELECT * FROM "TEST_db3".m2;
```

Query Editor		Query History	
1 SELECT * FROM "TEST_db3".m2			
2			
Data Output		Explain	Messages
m2_s character varying (20)			
1	New data added		
2	New data added		
3	New data added		
4	New data added		
5	New data added		
6	11		
7	New id added=12		

III. Understand Store Procedures. (For PostgreSQL 11+).

Store procedures are similar to normal functions. Except minor difference in the create statement and calling.

a. Create a simple stored procedure to insert records in a table.

Step1. CREATE stored procedure.

```
CREATE OR REPLACE PROCEDURE "TEST_db3".insert_test_m(  
    m_id integer,  
    m_n character varying)  
LANGUAGE 'plpgsql'  
AS $BODY$  
BEGIN  
  
    INSERT INTO "TEST_db3".test_m (m_id, m_name) values (M_id, m_n);  
    COMMIT;  
  
END  
$BODY$;
```

Step2. CALL the stored procedure.

```
CALL "TEST_db3".insert_test_m(15, 'Test15');
```

It will insert the new records in the test_m table with ID 15 and the name 'Test15'.

b. One more sample stored procedure to list all records as a message.

Step1. CREATE stored procedure.

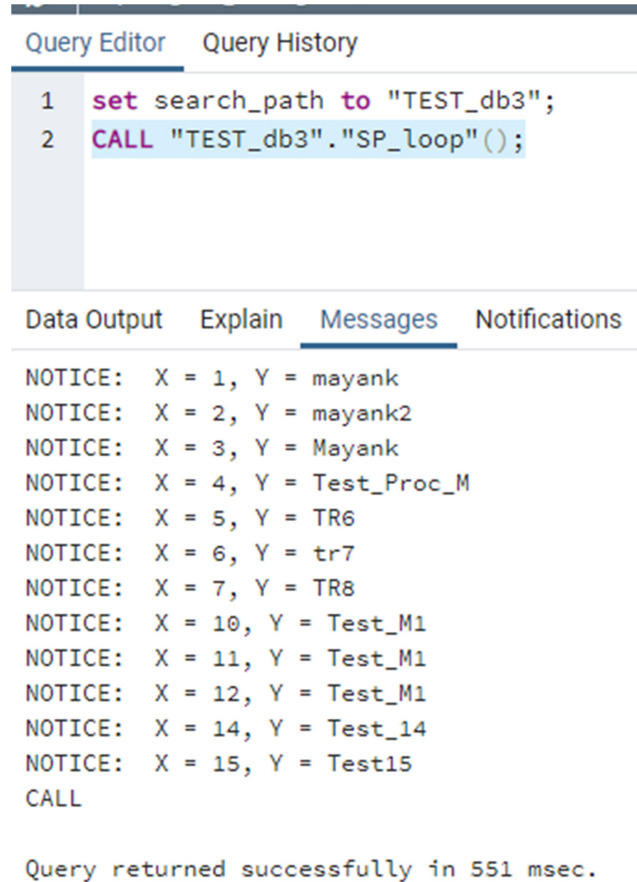
```
CREATE OR REPLACE PROCEDURE "TEST_db3"."SP_loop2"()  
LANGUAGE 'plpgsql'  
AS $BODY$  
DECLARE  
    R_LIST2 record;  
  
BEGIN  
    FOR R_LIST2 in (select m_id, m_name from sv_db3.test_m)  
    loop  
        RAISE NOTICE 'X = %, Y = %', R_LIST2.m_id, R_LIST2.m_name;  
    end loop;
```

END;

\$BODY\$;

Step2. CALL the stored procedure.

CALL "TEST_db3"."SP_loop"();



The screenshot displays a database query editor interface. At the top, there are two tabs: "Query Editor" and "Query History". The "Query Editor" tab is active, showing two lines of SQL code:
1 set search_path to "TEST_db3";
2 CALL "TEST_db3"."SP_loop"();
Below the code editor, there are four tabs: "Data Output", "Explain", "Messages", and "Notifications". The "Messages" tab is selected, showing a series of "NOTICE" messages. Each message indicates the current values of variables X and Y during the execution of the stored procedure. The messages are:
NOTICE: X = 1, Y = mayank
NOTICE: X = 2, Y = mayank2
NOTICE: X = 3, Y = Mayank
NOTICE: X = 4, Y = Test_Proc_M
NOTICE: X = 5, Y = TR6
NOTICE: X = 6, Y = tr7
NOTICE: X = 7, Y = TR8
NOTICE: X = 10, Y = Test_M1
NOTICE: X = 11, Y = Test_M1
NOTICE: X = 12, Y = Test_M1
NOTICE: X = 14, Y = Test_14
NOTICE: X = 15, Y = Test15
At the bottom of the "Messages" tab, the text "CALL" is visible. Below the "Messages" tab, a status message states: "Query returned successfully in 551 msec."

```
1 set search_path to "TEST_db3";  
2 CALL "TEST_db3"."SP_loop"();
```

Data Output Explain Messages Notifications

NOTICE: X = 1, Y = mayank
NOTICE: X = 2, Y = mayank2
NOTICE: X = 3, Y = Mayank
NOTICE: X = 4, Y = Test_Proc_M
NOTICE: X = 5, Y = TR6
NOTICE: X = 6, Y = tr7
NOTICE: X = 7, Y = TR8
NOTICE: X = 10, Y = Test_M1
NOTICE: X = 11, Y = Test_M1
NOTICE: X = 12, Y = Test_M1
NOTICE: X = 14, Y = Test_14
NOTICE: X = 15, Y = Test15
CALL

Query returned successfully in 551 msec.