

DBMS

ASSIGNMENT: 3

VERSION: 2

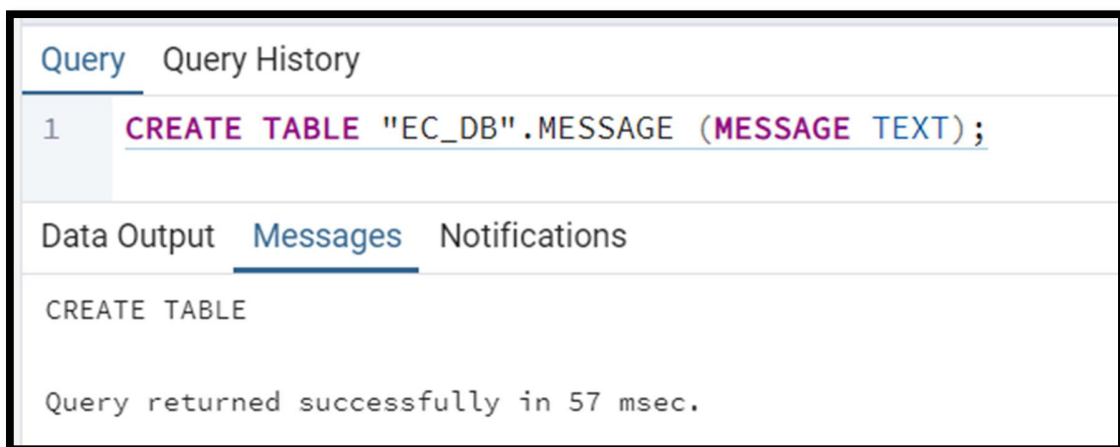
202412012

JAYESH S CHAUHAN

Q1. Create a trigger on table of your choice to check if the primary key ID already exists or not before inserting a new record. & Send a custom reply instead of an error message.

Query:

```
CREATE TABLE "EC_DB".MESSAGE (MESSAGE TEXT);
```



```
CREATE OR REPLACE FUNCTION "EC_DB".primary_key_check()
RETURNS trigger
LANGUAGE 'plpgsql'
AS $BODY$
BEGIN
    IF EXISTS (SELECT 1 FROM "EC_DB".users WHERE user_id = NEW.user_id) THEN
        INSERT INTO "EC_DB".message(message) VALUES (concat('User ID ', NEW.user_id, ' already exists.'));
        RETURN NULL;
    ELSE
        RETURN NEW;
    END IF;
END;
$BODY$;
```

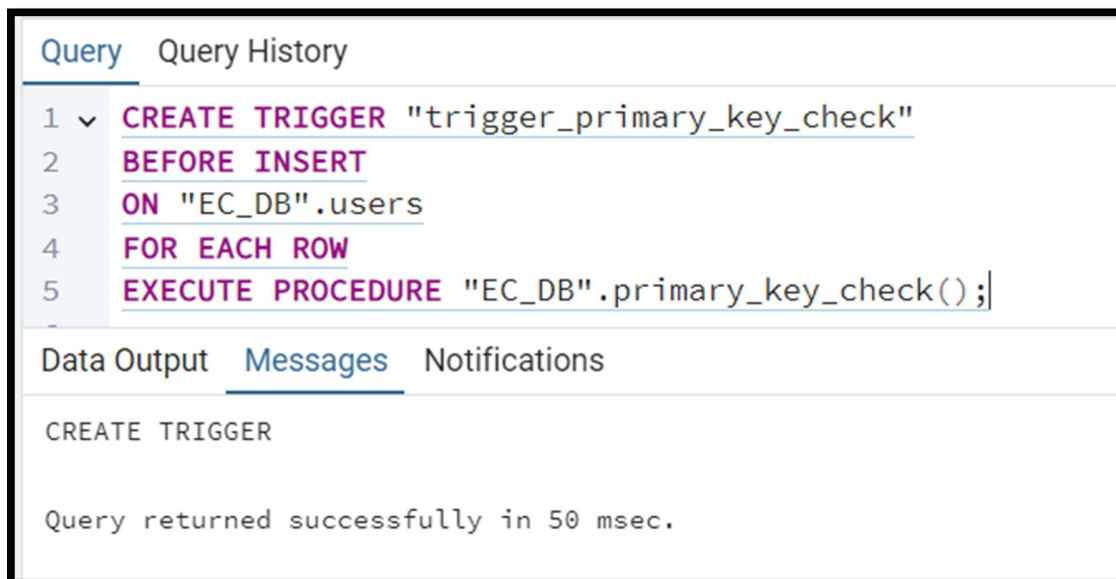


The screenshot shows a database query editor with a tab labeled 'Query'. The query is a PL/SQL function named 'primary_key_check' in the 'EC_DB' schema. The function is designed to check if a user ID already exists in the 'users' table. If it does, it returns null; otherwise, it returns the new record. The query is executed successfully, as indicated by the 'Messages' tab showing 'Query returned successfully in 102 msec.'

```
1 CREATE OR REPLACE FUNCTION "EC_DB".primary_key_check()
2 RETURNS trigger
3 LANGUAGE 'plpgsql'
4 AS $BODY$
5 BEGIN
6     IF EXISTS (SELECT 1 FROM "EC_DB".users WHERE user_id = NEW.user_id) THEN
7         INSERT INTO "EC_DB".message(message) VALUES (concat('User ID ', NEW.user_id, ' already exists.));
8         RETURN NULL;
9     ELSE
10        RETURN NEW;
11    END IF;
12 END;
13 $BODY$;
```

Query returned successfully in 102 msec.

```
CREATE TRIGGER "trigger_primary_key_check"
BEFORE INSERT
ON "EC_DB".users
FOR EACH ROW
EXECUTE PROCEDURE "EC_DB".primary_key_check();
```



The screenshot shows a database query editor with a tab labeled 'Query'. The query is a trigger named 'trigger_primary_key_check' that fires before an insert operation on the 'users' table in the 'EC_DB' schema. It calls the 'primary_key_check' function for each row. The query is executed successfully, as indicated by the 'Messages' tab showing 'Query returned successfully in 50 msec.'

```
1 CREATE TRIGGER "trigger_primary_key_check"
2 BEFORE INSERT
3 ON "EC_DB".users
4 FOR EACH ROW
5 EXECUTE PROCEDURE "EC_DB".primary_key_check();
```

Query returned successfully in 50 msec.

```
INSERT INTO "EC_DB".users (user_id, username, email, password, first_name, last_name, address,
phone_number)
VALUES (69, 'Raj', 'raj@gmail.com', 'raj123', 'Raj', 'Patel', 'Ahmedabad', '9876543210');
```



Query Query History

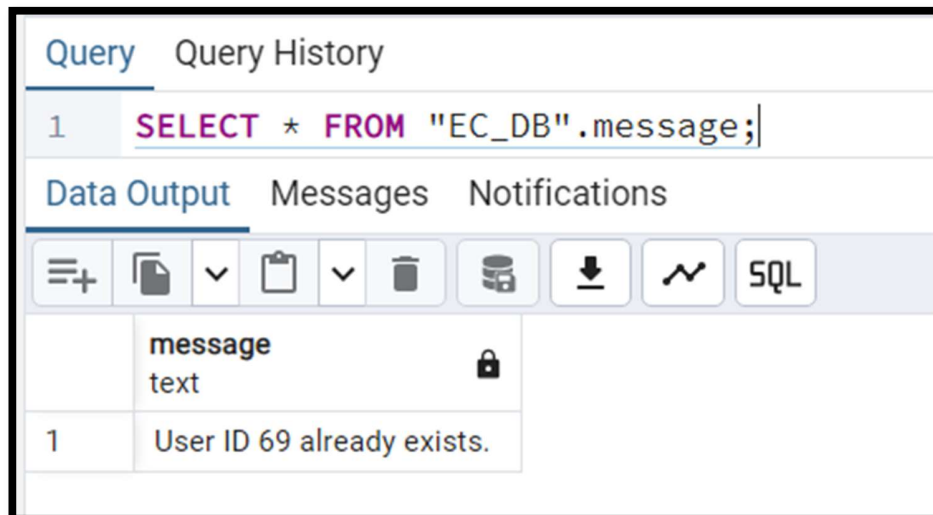
```
1 INSERT INTO "EC_DB".users (user_id, username, email, password, first_name, last_name, address, phone_number)
2 VALUES (69, 'Raj', 'raj@gmail.com', 'raj123', 'Raj', 'Patel', 'Ahmedabad', '9876543210');
```

Data Output Messages Notifications

INSERT 0 0

Query returned successfully in 89 msec.

SELECT * FROM "EC_DB".message;



Query Query History

```
1 SELECT * FROM "EC_DB".message;
```

Data Output Messages Notifications


message text

1 User ID 69 already exists.

Q2. Create a trigger on the Table of your choice to check if the Foreign key ID already exists or not before inserting a new record. & Send a custom reply instead of an error message.

Query:

CREATE TABLE "EC_DB".message2 (MESSAGE TEXT);



Query Query History

```
1 CREATE TABLE "EC_DB".message2 (MESSAGE TEXT);
```

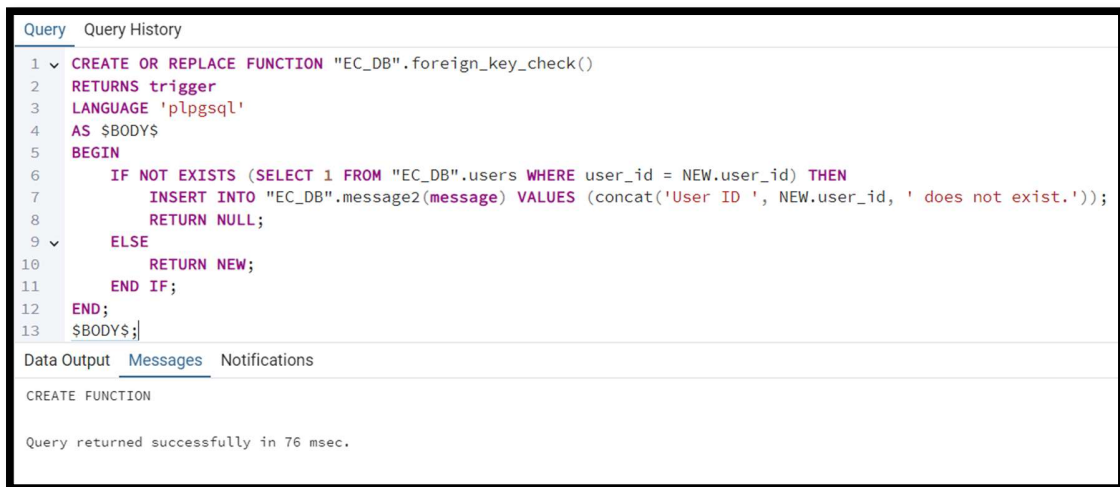
Data Output Messages Notifications

CREATE TABLE

Query returned successfully in 70 msec.

CREATE OR REPLACE FUNCTION "EC_DB".foreign_key_check()
RETURNS trigger

```
LANGUAGE 'plpgsql'
AS $BODY$
BEGIN
    IF NOT EXISTS (SELECT 1 FROM "EC_DB".users WHERE user_id = NEW.user_id) THEN
        INSERT INTO "EC_DB".message2(message) VALUES (concat('User ID ', NEW.user_id, ' does not exist.));
        RETURN NULL;
    ELSE
        RETURN NEW;
    END IF;
END;
$BODY$;
```



The screenshot shows a SQL IDE interface with a query editor and a results pane. The query editor contains the following SQL code:

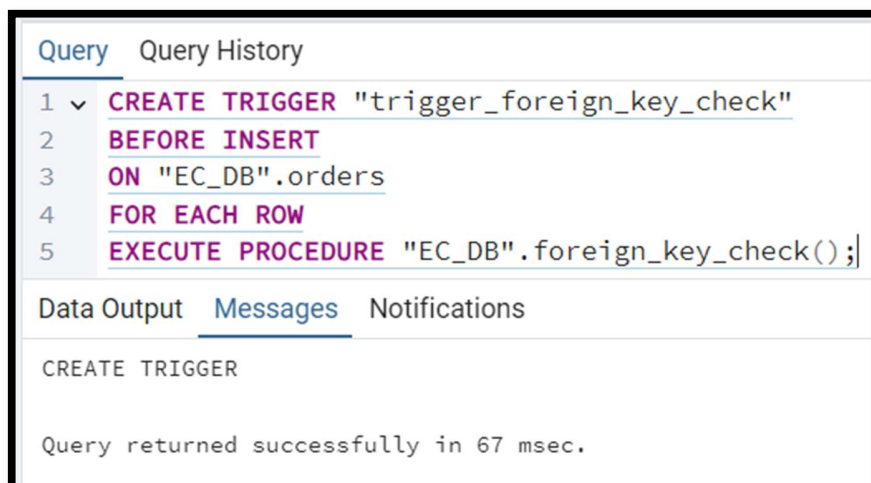
```
1 CREATE OR REPLACE FUNCTION "EC_DB".foreign_key_check()
2 RETURNS trigger
3 LANGUAGE 'plpgsql'
4 AS $BODY$
5 BEGIN
6     IF NOT EXISTS (SELECT 1 FROM "EC_DB".users WHERE user_id = NEW.user_id) THEN
7         INSERT INTO "EC_DB".message2(message) VALUES (concat('User ID ', NEW.user_id, ' does not exist.));
8         RETURN NULL;
9     ELSE
10        RETURN NEW;
11    END IF;
12 END;
13 $BODY$;
```

The results pane shows the following output:

```
CREATE FUNCTION

Query returned successfully in 76 msec.
```

```
CREATE TRIGGER "trigger_foreign_key_check"
BEFORE INSERT
ON "EC_DB".orders
FOR EACH ROW
EXECUTE PROCEDURE "EC_DB".foreign_key_check();
```



The screenshot shows a SQL IDE interface with a query editor and a results pane. The query editor contains the following SQL code:

```
1 CREATE TRIGGER "trigger_foreign_key_check"
2 BEFORE INSERT
3 ON "EC_DB".orders
4 FOR EACH ROW
5 EXECUTE PROCEDURE "EC_DB".foreign_key_check();
```

The results pane shows the following output:

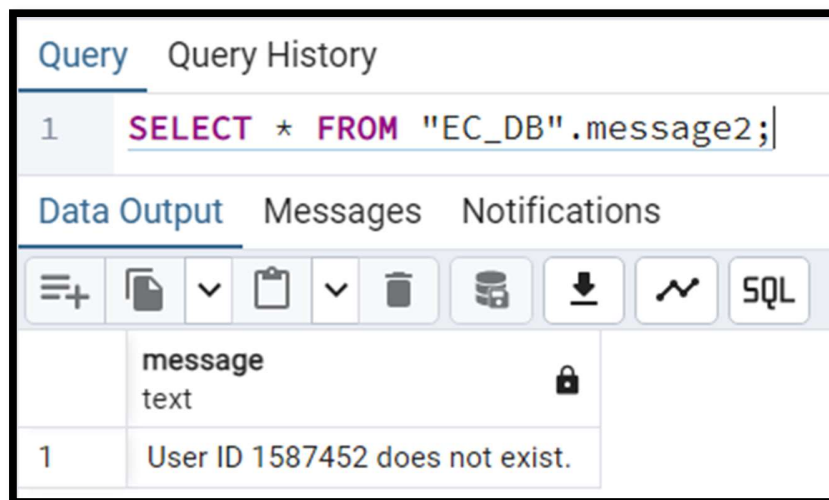
```
CREATE TRIGGER

Query returned successfully in 67 msec.
```

```
INSERT INTO "EC_DB".orders (order_id, user_id, order_date, shipping_address, total_amount, status)
VALUES (101, 1587452, '2024-08-24 10:00:00', 'Some Address', 250.00, 'Pending');
```



```
SELECT * FROM "EC_DB".message2;
```



Q3. Write a SQL function that takes a product's product_id and a discount percentage as inputs and returns the discounted price of the product.

Query:

```
CREATE OR REPLACE FUNCTION "EC_DB".discounted_price(
    p_product_id INT,
    p_discount_percentage DECIMAL
)
RETURNS DECIMAL(10, 2)
LANGUAGE 'plpgsql'
AS $BODY$
DECLARE
    count_price DECIMAL(10, 2);
    count_discounted_price DECIMAL(10, 2);
BEGIN
    SELECT price INTO count_price
    FROM "EC_DB".products
```

```
WHERE product_id = p_product_id;
count_discounted_price := count_price - (count_price * p_discount_percentage / 100);
RETURN count_discounted_price;
END;
$BODY$;
```

The screenshot shows a SQL IDE interface with a query editor and a messages pane. The query editor contains the following SQL code:

```
1 CREATE OR REPLACE FUNCTION "EC_DB".discounted_price(  
2     p_product_id INT,  
3     p_discount_percentage DECIMAL  
4 )  
5 RETURNS DECIMAL(10, 2)  
6 LANGUAGE 'plpgsql'  
7 AS $BODY$  
8 DECLARE  
9     count_price DECIMAL(10, 2);  
10    count_discounted_price DECIMAL(10, 2);  
11 BEGIN  
12     SELECT price INTO count_price  
13     FROM "EC_DB".products  
14     WHERE product_id = p_product_id;  
15     count_discounted_price := count_price - (count_price * p_discount_percentage / 100);  
16     RETURN count_discounted_price;  
17 END;  
18 $BODY$;
```

The messages pane shows the following output:

```
CREATE FUNCTION  
  
Query returned successfully in 60 msec.
```

```
SELECT "EC_DB".discounted_price(12, 20);
```

The screenshot shows a SQL IDE interface with a query editor and a data output table. The query editor contains the following SQL code:

```
1 SELECT "EC_DB".discounted_price(12, 20);
```

The data output table shows the following results:

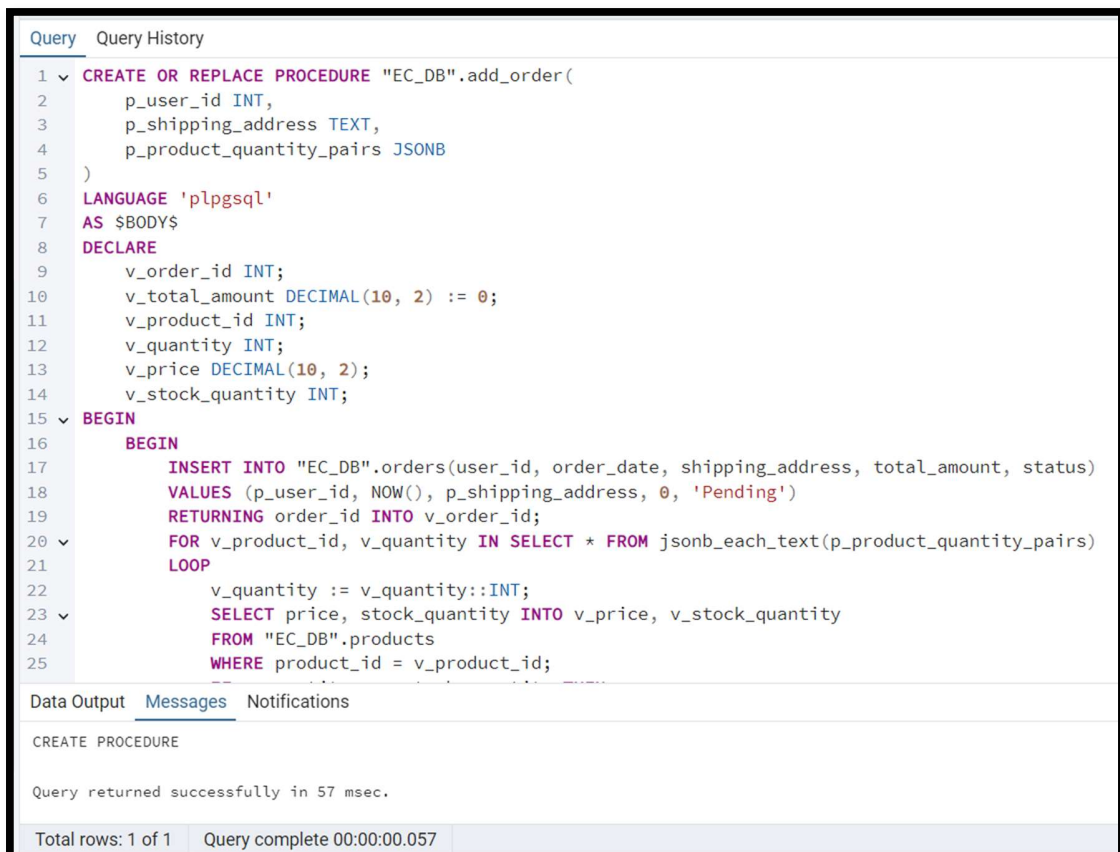
	discounted_price numeric
1	1168.74

Q4. Create a stored procedure named add_order that takes the user_id, shipping_address, and a list of product_id and quantity pairs as inputs, and inserts a new order into the orders and order_details tables. The procedure should also update the stock quantity of the products.

Query:

```
CREATE OR REPLACE PROCEDURE "EC_DB".add_order(
    p_user_id INT,
    p_shipping_address TEXT,
    p_product_quantity_pairs JSONB
)
LANGUAGE 'plpgsql'
AS $BODY$
DECLARE
    v_order_id INT;
    v_total_amount DECIMAL(10, 2) := 0;
    v_product_id INT;
    v_quantity INT;
    v_price DECIMAL(10, 2);
    v_stock_quantity INT;
BEGIN
    BEGIN
        INSERT INTO "EC_DB".orders(user_id, order_date, shipping_address, total_amount, status)
        VALUES (p_user_id, NOW(), p_shipping_address, 0, 'Pending')
        RETURNING order_id INTO v_order_id;
        FOR v_product_id, v_quantity IN SELECT * FROM jsonb_each_text(p_product_quantity_pairs)
        LOOP
            v_quantity := v_quantity::INT;
            SELECT price, stock_quantity INTO v_price, v_stock_quantity
            FROM "EC_DB".products
            WHERE product_id = v_product_id;
            IF v_quantity > v_stock_quantity THEN
                RAISE EXCEPTION 'Not enough stock for product ID %', v_product_id;
            END IF;
            INSERT INTO "EC_DB".order_details(order_id, product_id, quantity, price)
            VALUES (v_order_id, v_product_id, v_quantity, v_price);
            v_total_amount := v_total_amount + (v_price * v_quantity);
            UPDATE "EC_DB".products
            SET stock_quantity = stock_quantity - v_quantity
            WHERE product_id = v_product_id;
        END LOOP;
    UPDATE "EC_DB".orders
    SET total_amount = v_total_amount
```

```
WHERE order_id = v_order_id;
COMMIT;
EXCEPTION
WHEN OTHERS THEN
    ROLLBACK;
    RAISE;
END;
END;
$BODY$;
```



The screenshot shows a SQL IDE interface with a 'Query' tab. The query editor contains a PL/SQL procedure named 'add_order' in the 'EC_DB' schema. The procedure takes four parameters: 'p_user_id' (INT), 'p_shipping_address' (TEXT), 'p_product_quantity_pairs' (JSONB), and 'v_order_id' (INT). It declares several local variables: 'v_order_id' (INT), 'v_total_amount' (DECIMAL(10, 2)), 'v_product_id' (INT), 'v_quantity' (INT), 'v_price' (DECIMAL(10, 2)), and 'v_stock_quantity' (INT). The procedure begins by inserting a new order into the 'orders' table with the provided user_id, shipping_address, and a total amount of 0, with a status of 'Pending'. It then returns the order_id to 'v_order_id'. A loop is used to process the 'p_product_quantity_pairs' JSONB, where it selects the price and stock quantity for each product_id from the 'products' table and updates the order's total amount and stock quantity accordingly. The query is executed successfully, as indicated by the 'Data Output' tab showing 'CREATE PROCEDURE' and the message 'Query returned successfully in 57 msec.' The status bar at the bottom shows 'Total rows: 1 of 1' and 'Query complete 00:00:00.057'.

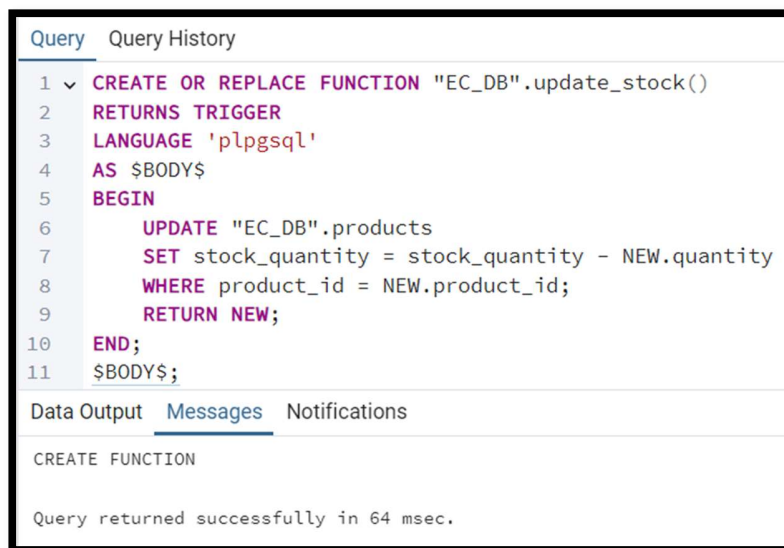
```
1 CREATE OR REPLACE PROCEDURE "EC_DB".add_order(
2     p_user_id INT,
3     p_shipping_address TEXT,
4     p_product_quantity_pairs JSONB
5 )
6 LANGUAGE 'plpgsql'
7 AS $BODY$
8 DECLARE
9     v_order_id INT;
10    v_total_amount DECIMAL(10, 2) := 0;
11    v_product_id INT;
12    v_quantity INT;
13    v_price DECIMAL(10, 2);
14    v_stock_quantity INT;
15 BEGIN
16     BEGIN
17         INSERT INTO "EC_DB".orders(user_id, order_date, shipping_address, total_amount, status)
18         VALUES (p_user_id, NOW(), p_shipping_address, 0, 'Pending')
19         RETURNING order_id INTO v_order_id;
20     FOR v_product_id, v_quantity IN SELECT * FROM jsonb_each_text(p_product_quantity_pairs)
21     LOOP
22         v_quantity := v_quantity::INT;
23         SELECT price, stock_quantity INTO v_price, v_stock_quantity
24         FROM "EC_DB".products
25         WHERE product_id = v_product_id;
```

```
CALL "EC_DB".add_order(
    60,
    '123 Main St, Anytown, USA',
    '{"60": "2"}'
);
```

Q5. Write a trigger that automatically decreases the stock quantity of a product in the products table when a new order is inserted into the order details table.

Query:
CREATE OR REPLACE FUNCTION "EC_DB".update_stock()

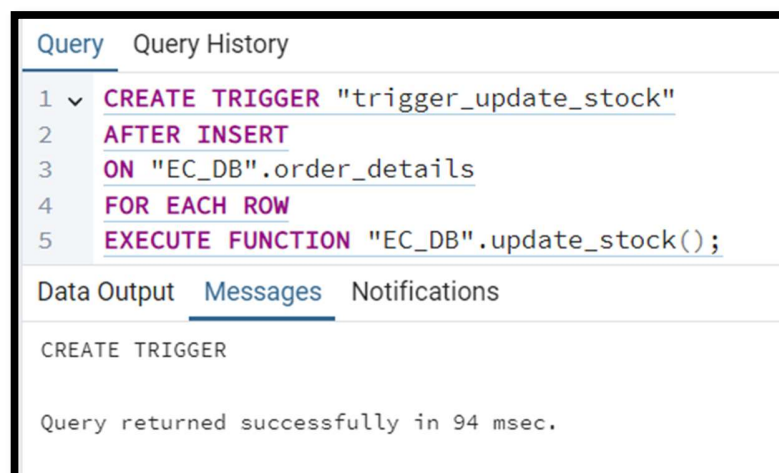

```
RETURNS TRIGGER
LANGUAGE 'plpgsql'
AS $BODY$
BEGIN
    UPDATE "EC_DB".products
    SET stock_quantity = stock_quantity - NEW.quantity
    WHERE product_id = NEW.product_id;
    RETURN NEW;
END;
$BODY$;
```



The screenshot shows a database query editor with a 'Query' tab selected. The query is a PL/pgSQL function definition for 'update_stock' in the 'EC_DB' schema. The function is designed to be a trigger, returning a trigger object. The body of the function updates the 'stock_quantity' in the 'products' table by subtracting the quantity from the 'NEW' record. The query is executed successfully, as indicated by the 'Messages' tab showing 'CREATE FUNCTION' and a status message 'Query returned successfully in 64 msec.'

```
Query Query History
1 CREATE OR REPLACE FUNCTION "EC_DB".update_stock()
2 RETURNS TRIGGER
3 LANGUAGE 'plpgsql'
4 AS $BODY$
5 BEGIN
6     UPDATE "EC_DB".products
7     SET stock_quantity = stock_quantity - NEW.quantity
8     WHERE product_id = NEW.product_id;
9     RETURN NEW;
10 END;
11 $BODY$;
Data Output Messages Notifications
CREATE FUNCTION
Query returned successfully in 64 msec.
```

```
CREATE TRIGGER "trigger_update_stock"
AFTER INSERT
ON "EC_DB".order_details
FOR EACH ROW
EXECUTE FUNCTION "EC_DB".update_stock();
```



The screenshot shows a database query editor with a 'Query' tab selected. The query is a trigger definition named 'trigger_update_stock' that fires after an insert on the 'order_details' table in the 'EC_DB' schema. The trigger calls the 'update_stock' function for each row. The query is executed successfully, as indicated by the 'Messages' tab showing 'CREATE TRIGGER' and a status message 'Query returned successfully in 94 msec.'

```
Query Query History
1 CREATE TRIGGER "trigger_update_stock"
2 AFTER INSERT
3 ON "EC_DB".order_details
4 FOR EACH ROW
5 EXECUTE FUNCTION "EC_DB".update_stock();
Data Output Messages Notifications
CREATE TRIGGER
Query returned successfully in 94 msec.
```

Before:

```
SELECT * FROM "EC_DB".order_details;
```

Data Output Messages Notifications						
	order_detail_id [PK] integer	order_id integer	product_id integer	quantity integer	price numeric (10,2)	
1	1	46	7	10	5478.67	
2	2	35	17	1	6992.10	
3	3	51	15	2	3369.91	
4	4	11	33	8	9818.68	
5	5	48	48	6	685.71	
Total rows: 300 of 300 Query complete 00:00:00.496						

```
SELECT * FROM "EC_DB".products;
```

Data Output Messages Notifications						
	product_id [PK] integer	name character varying (100)	description text	price numeric (10,2)	stock_quantity integer	category_id integer
1	1	Mobile Phone	Portable communication device with internet access and app...	7498.65	28	1
2	2	Laptop	Compact computer for personal and professional use.	2100.47	69	1
3	3	Smart Watch	Wearable device for tracking fitness and notifications.	2784.21	72	1
4	4	Headphones	Audio device for listening to music or calls privately.	6992.10	18	1
5	5	Tablet	Touchscreen device for media consumption and productivity.	1181.34	95	1
Total rows: 50 of 50 Query complete 00:00:00.086						

Apply Query:

```
INSERT INTO "EC_DB".order_details(order_detail_id, order_id, product_id, quantity, price)
VALUES (301, 50, 1, 3, 9500);
```

Query Query History	
1	INSERT INTO "EC_DB".order_details(order_detail_id, order_id, product_id, quantity, price)
2	VALUES (301, 50, 1, 3, 9500);
3	
Data Output Messages Notifications	
INSERT 0 1	
Query returned successfully in 139 msec.	

After:

```
SELECT * FROM "EC_DB".order_details;
```

Data Output Messages Notifications						
	order_detail_id [PK] integer	order_id integer	product_id integer	quantity integer	price numeric (10,2)	
297	297	43	24	4	7498.65	
298	298	56	30	9	6725.63	
299	299	14	32	4	4643.25	
300	300	2	21	7	7900.81	
301	301	50	1	3	9500.00	
Total rows: 301 of 301			Query complete 00:00:00.179			

SELECT * FROM "EC_DB".products;

Data Output Messages Notifications						
	product_id [PK] integer	name character varying (100)	description text	price numeric (10,2)	stock_quantity integer	category_id integer
46	47	Dining Table	Table used for meals in a dining area.	3303.66	32	10
47	48	Chair	Single-seat furniture for sitting.	167.62	50	10
48	49	Bed	Furniture for sleeping and resting	2431.66	97	10
49	50	Wardrobe	Storage unit for clothes and accessories.	2743.24	43	10
50	1	Mobile Phone	Portable communication device with internet access and app...	7498.65	25	1
Total rows: 50 of 50			Query complete 00:00:00.080			

Practice Set:

I. Understand & Run functions.

1. Create a simple function to print "HELLO WORLD" using GUI.

Create - Function

General

Definition

Code

Options

Parameters

Security

SQL

Name

fun_hello

Owner

postgres

✕

|

▼

Schema

EC_DB

|

▼

Comment

?

?

✕ Close

↺ Reset

💾 Save

Create - Function

×

General

Definition

Code

Options

Parameters

Security

SQL

Custom return type?

Return type

character varying

×

▼

Language

plpgsql

×

▼

Arguments

+

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

i

?

×

Close

↺

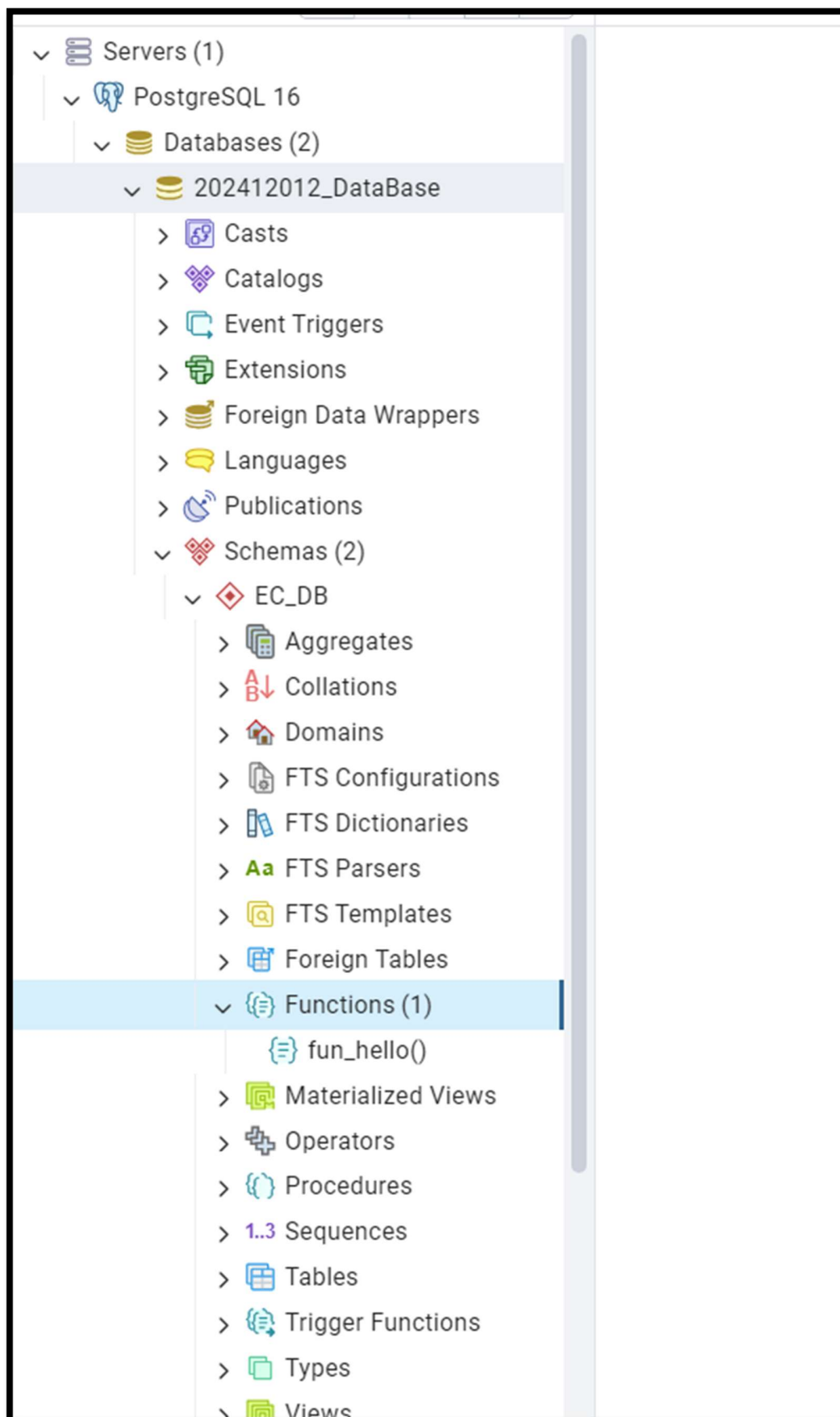
Reset

Save

A screenshot of the 'Create - Function' dialog box in SQL Server Enterprise Manager. The dialog has a title bar with a minus, maximize, and close button. Below the title bar are tabs: 'General', 'Definition', 'Code' (which is selected and underlined), 'Options', 'Parameters', 'Security', and 'SQL'. The main area of the dialog shows a code editor with the following text:

```
1  BEGIN
2  RETURN 'Hello World!';
3  END
```

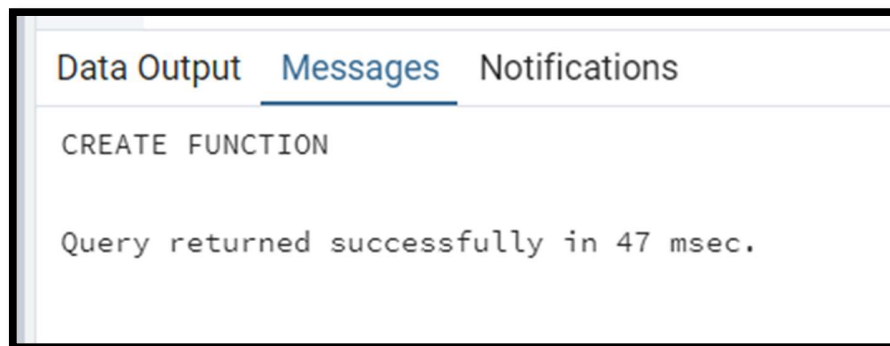
On the left side of the code editor is a vertical line with a small downward arrow at the top. At the bottom of the dialog are three buttons: 'Close' (with a close icon), 'Reset' (with a circular arrow icon), and 'Save' (with a floppy disk icon). There are also two small circular icons on the bottom left, one with an 'i' and one with a '?'. The background of the dialog is light gray.



2. Create a simple function to print "HELLO WORLD" using SQL.

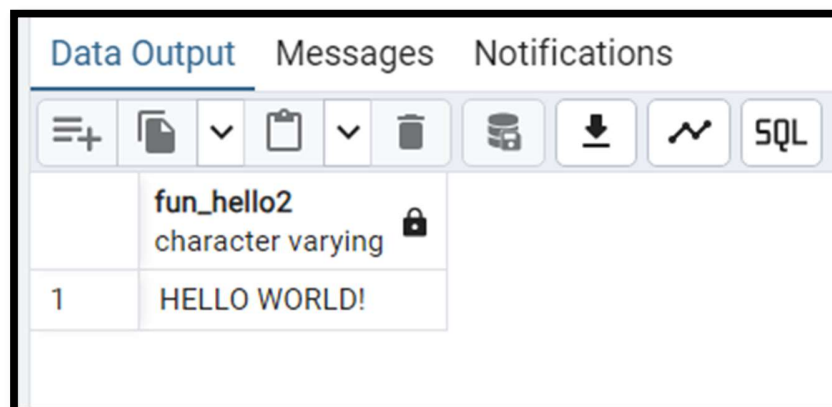
```
set search_path to "EC_DB";
```

```
CREATE OR REPLACE FUNCTION "EC_DB"."fun_hello2 "()  
RETURNS character varying  
LANGUAGE 'plpgsql'  
AS $BODY$ BEGIN  
Return 'HELLO WORLD!';  
END;  
$BODY$;
```



3. Using the function. "Function calls".

```
SELECT "EC_DB"."fun_hello2"();
```

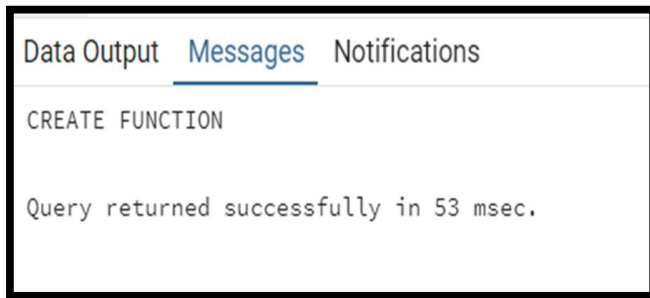


4. Create functions with parameters that return the SUM of two numbers.

```
CREATE OR REPLACE FUNCTION "EC_DB"."Fun_Sum"(a integer, b integer)  
RETURNS INTEGER
```

```
LANGUAGE 'plpgsql'  
AS $BODY$ BEGIN  
return (a+b);  
END;  
$BODY$;
```

```
SELECT "EC_DB"."Fun_Sum"(2, 3)
```



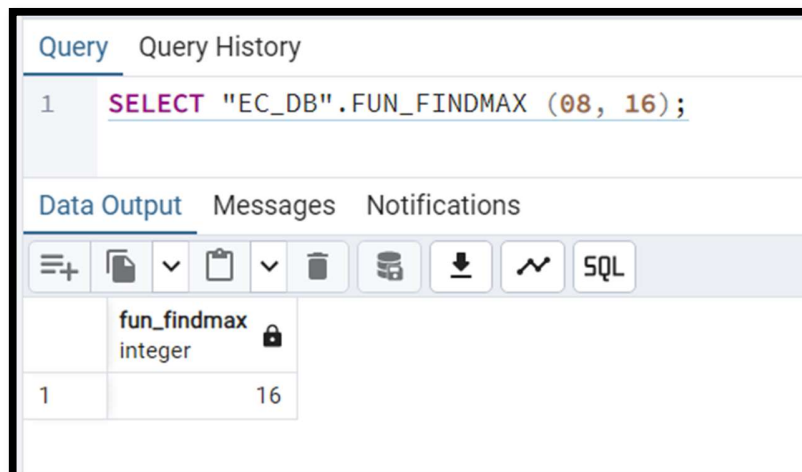
	Fun_Sum integer
1	5

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

```
CREATE  
OR REPLACE FUNCTION "EC_DB".FUN_FINDMAX (A INT, B INT) RETURNS INTEGER LANGUAGE 'plpgsql' AS  
$BODY$  
DECLARE C INTEGER;  
BEGIN IF (A > B) THEN ELSE C = B;  
END IF;  
RETURN C;  
END;  
$BODY$;
```



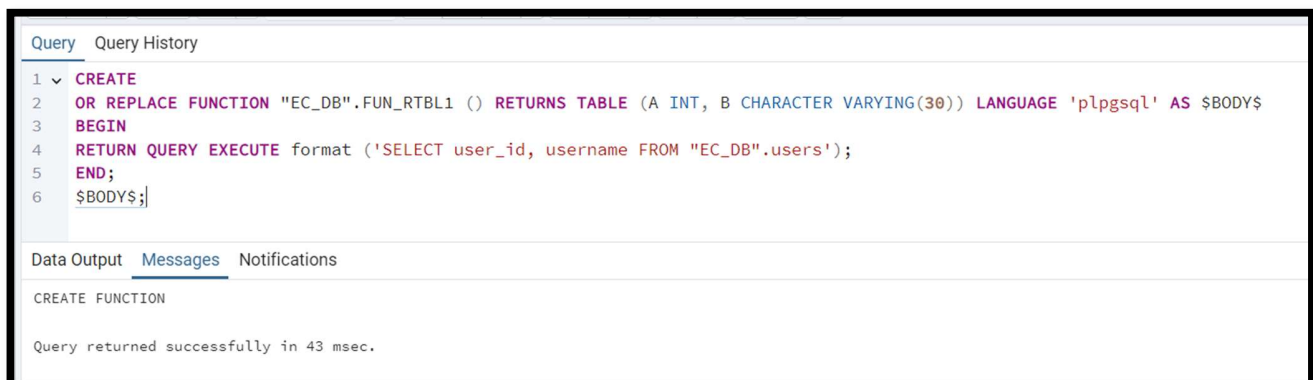
```
SELECT "EC_DB".FUN_FINDMAX (08, 16);
```



fun_findmax	
integer	
1	16

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

```
CREATE  
OR REPLACE FUNCTION "EC_DB".FUN_RTBL1 () RETURNS TABLE (A INT, B CHARACTER VARYING(30))  
LANGUAGE 'plpgsql' AS $BODY$  
BEGIN  
RETURN QUERY EXECUTE format ('SELECT user_id, username FROM "EC_DB".users');  
END;  
$BODY$;
```



fun_findmax	
integer	
1	16

```
SELECT "EC_DB".FUN_RTBL1 ()
```


Data Output			Messages	Notifications
	fun_rtbl1	record		
1	(1,mamooty17)			
2	(2,stuvantara)			
3	(3,hrishita94)			
4	(4,jivin46)			
5	(5,adash)			
6	(6,bkapoor)			
7	(7,birhrishita)			
8	(8,rgola)			
9	(9,dasguptaneyesa)			
10	(10,siyengar)			
Total rows: 100 of 100			Query complete 00:00:00.099	

7. Create functions with a Temporary table and use of FOR loop.

```
CREATE
OR REPLACE FUNCTION "EC_DB".FUN_LOOP () RETURNS TABLE (A INTEGER, B CHARACTER VARYING)
LANGUAGE 'plpgsql' AS $BODY$
DECLARE
R_LIST2 record;
BEGIN
CREATE TEMP TABLE test1 (al int, bl character varying (30)) ON COMMIT DROP;
FOR R_LIST2 in (select user_id, username from "EC_DB".users)
loop
Insert into test1 (al, bl) values (R_LIST2.user_id,R_LIST2.username); end loop;
RETURN QUERY TABLE test1;
END;
$BODY$;
```

```
Query Query History
1 CREATE
2 OR REPLACE FUNCTION "EC_DB".FUN_LOOP () RETURNS TABLE (A INTEGER, B CHARACTER VARYING) LANGUAGE 'plpgsql' AS $BODY$
3 DECLARE
4   R_LIST2 record;
5 BEGIN
6 CREATE TEMP TABLE test1 (a1 int, b1 character varying (30)) ON COMMIT DROP;
7 FOR R_LIST2 in (select user_id, username from "EC_DB".users)
8 loop
9   Insert into test1 (a1, b1) values (R_LIST2.user_id,R_LIST2.username); end loop;
10 RETURN QUERY TABLE test1;
11 END;
12 $BODY$;

Data Output Messages Notifications
CREATE FUNCTION

Query returned successfully in 61 msec.
```

SELECT "EC_DB".fun_loop()

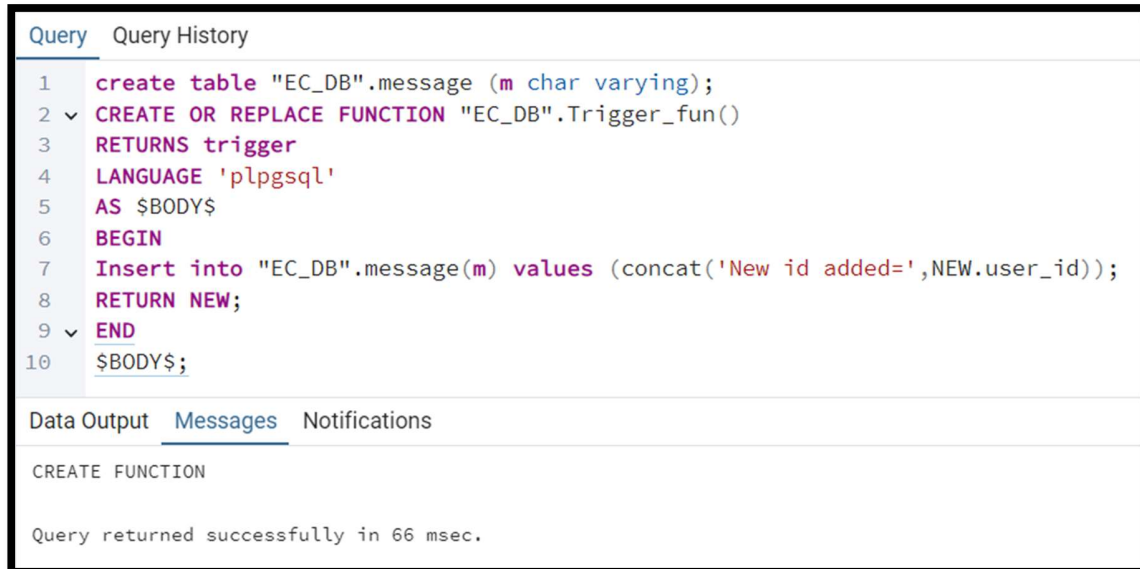
Data Output		Messages	Notifications
<div><div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div></div><div>SQL</div></div>			
	fun_loop record		
1	(1,mamooty17)		
2	(2,stuvantara)		
3	(3,hrishita94)		
4	(4,jivin46)		
5	(5,adash)		
6	(6,bkapoor)		
7	(7,birhrishita)		
8	(8,rgola)		
9	(9,dasguptaneysa)		
10	(10,siyengar)		
Total rows: 100 of 100		Query complete 00:00:00.059	

II. Use triggers to execute functions.

1. Create a trigger function.

```
create table "EC_DB".message (m char varying);
CREATE OR REPLACE FUNCTION "EC_DB".Trigger_fun()
RETURNS trigger
LANGUAGE 'plpgsql'
```

```
AS $BODY$  
BEGIN  
Insert into "EC_DB".message(m) values (concat('New id added=',NEW.user_id));  
RETURN NEW;  
END  
$BODY$;
```

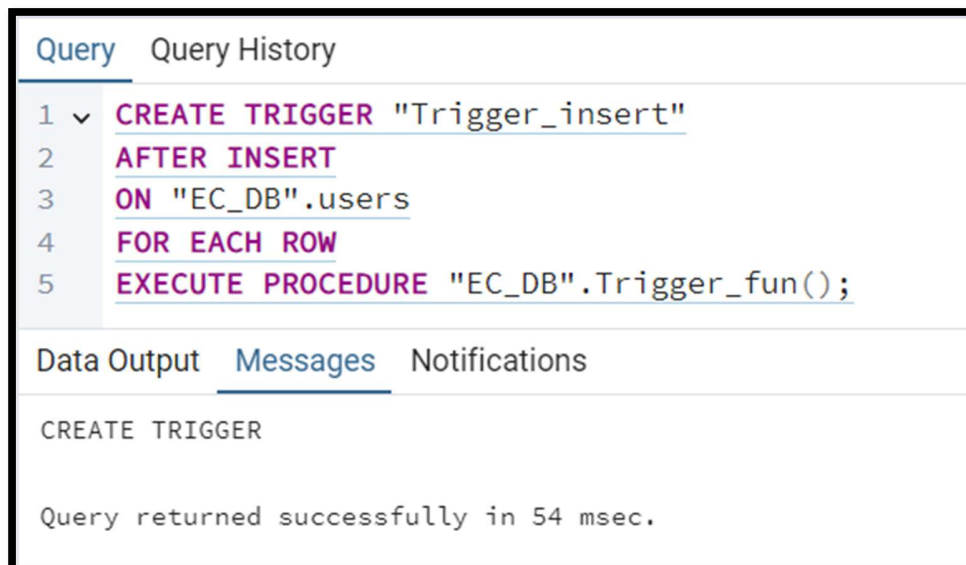


The screenshot shows a SQL query editor with a tab labeled "Query" and "Query History". The query is as follows:

```
1 create table "EC_DB".message (m char varying);  
2 CREATE OR REPLACE FUNCTION "EC_DB".Trigger_fun()  
3 RETURNS trigger  
4 LANGUAGE 'plpgsql'  
5 AS $BODY$  
6 BEGIN  
7 Insert into "EC_DB".message(m) values (concat('New id added=',NEW.user_id));  
8 RETURN NEW;  
9 END  
10 $BODY$;
```

Below the query, there are tabs for "Data Output", "Messages", and "Notifications". The "Messages" tab is selected, showing the text "CREATE FUNCTION". At the bottom, a status message reads: "Query returned successfully in 66 msec."

```
CREATE TRIGGER "Trigger_insert"  
AFTER INSERT  
ON "EC_DB".users  
FOR EACH ROW  
EXECUTE PROCEDURE "EC_DB".Trigger_fun();
```



The screenshot shows a SQL query editor with a tab labeled "Query" and "Query History". The query is as follows:

```
1 CREATE TRIGGER "Trigger_insert"  
2 AFTER INSERT  
3 ON "EC_DB".users  
4 FOR EACH ROW  
5 EXECUTE PROCEDURE "EC_DB".Trigger_fun();
```

Below the query, there are tabs for "Data Output", "Messages", and "Notifications". The "Messages" tab is selected, showing the text "CREATE TRIGGER". At the bottom, a status message reads: "Query returned successfully in 54 msec."

```
INSERT into "EC_DB".users values(0810, 'Jayesh', 'jayesh@gmail.com', 'jay123', 'jayesh', 'chauhan', 'mumbai', '9874521245');
```



```
SELECT * FROM "EC_DB".message;
```

