

## **Functions**

Functions in SQL allow you to encapsulate a set of SQL statements into a reusable code block. They can accept parameters and return values, providing flexibility and modularity to your SQL code.

### **Example -**

Consider Scaler's portal. Whenever a user logs in with their email, they get a message saying "Hi".

For Shivank - "Hi Shivank"

For Suraj - "Hi Suraj"

For Varma - "Hi Varma"

Instead of manually responding "Hi" to each particular user, we can create a function that greets them automatically.

```
def greet(name):  
    print("Hi" + name)
```

### **Syntax:**

```
CREATE FUNCTION func_name(parameter data_type)  
RETURNS data_type  
DECLARE variable_name data_type  
BEGIN  
    SELECT...;  
RETURN variable_name  
END
```

**Question:** Write a function such that whatever input the user gives is incremented by 100. Say I/P=200, then O/P=200+100=300.

**Step 1:** Create a new database / Use an existing database

To use the "demo\_db" database -

```
USE demo_db;
```

### **Step 2:**

- Go to the Database Schema
- Then scroll down to the Functions
- Right Click and select the Create Function option

Now you can define your function.

```
Create function `add` (ip int)
Returns int
Deterministic
Begin
    Declare op int;
    Set op=ip+100;
    Return op;
Return op;
End
```

### Step 3:

Click on Apply and your function will be formatted in the following manner.

```
CREATE DEFINER=`root`@`localhost` FUNCTION `add` (ip int)
RETURNS int
    DETERMINISTIC
Begin
    Declare op int;
    Set op=ip+100;
    Return op;
Return op;
End
```

### Step 4:

- Refresh your Database Schema
- Go to the Function
- Click the ⚡ icon on the right side
- Provide an input to you function

-> **Input:** 200

The function will be called - `SELECT demo_db.add(200);`  
and you'll get the result.

-> **Output:** 300

---

To use the "farmers\_market" database -  
`Use farmers_market;`

To view the "customer\_purchases" table -  
`Select * from customer_purchases;`

This table has all the records related to the customer entity.

Let's say the company's financial year starts as :

- Oct-Dec 22, then
- Jan-Mar 23, then
- Apr-Jun 23 and
- Jul-Sep 23

**Question:** Fetch the data for the customer with customer\_id=7 and FY 2019? i.e. Oct18 to Sep19

**Query:**

```
Select *  
from customer_purchases  
where customer_id=7  
And year(date_add(market_date, interval 3 month))=2019;
```

**Do you see a problem here?**

For every year you have to change the year and rewrite this query again and again.

**What's the solution then?**

Creating a user-defined function to get the fiscal year.

```
Create function `get_fiscal_year` (market_date DATE)  
Returns int  
Deterministic  
Begin  
    Declare fiscal_year int;  
    Set fiscal_year=year(date_add(market_date, interval 3 month));  
    Return fiscal_year;  
Return fiscal_year;  
End
```

Follow the same steps as we did for the previous question.

-> **Input:** '2019-11-01'

Function Call - SELECT farmers\_market.get\_fiscal\_year('2019-11-01');

-> **Output:** 2020

**Query:**

```
Select *  
from customer_purchases
```

```
where customer_id=7  
And get_fiscal_year(market_date)=2019;
```

Question: We need to see the total amount spent by our customer with customer\_id 7 during the fiscal year 2020.

**Query:**

```
SELECT  
    customer_id,  
    ROUND(SUM(quantity * cost_to_customer_per_qty), 2) AS  
total_spent  
FROM customer_purchases  
WHERE customer_id = 7  
AND get_fiscal_year(market_date) = 2020  
GROUP BY customer_id;
```

---

## **Stored Procedures**

**Question:** Let's say we want to assign some badge to the customers based on their total purchase amount in a particular fiscal year.

- If the total purchase amount is greater than 250, the customer gets a Gold badge.
- Otherwise (purchase amount is less than 250) he/she gets a Silver badge.

**Step 1:** Create a new database / Use an existing database

To use the "demo\_db" database -

```
USE demo_db;
```

**Step 2:**

- Go to the Database Schema
- Then scroll down to the Stored Procedures
- Right Click and select the Create Stored Procedure option

Now, we'll write our **Stored Procedure**.

```
CREATE PROCEDURE `get_customer_level`  
    (IN in_customer_id INT,  
    IN in_fiscal_year YEAR,  
    OUT out_level VARCHAR(10))
```

```

BEGIN
    DECLARE sales DECIMAL(10, 2) DEFAULT 0;

    SELECT
        SUM(ROUND(quantity * cost_to_customer_per_qty, 2))
    INTO sales
    FROM customer_purchases
    WHERE
        customer_id = in_customer_id
        AND get_fiscal_year(market_date) = in_fiscal_year;

    IF sales > 250 THEN
        SET out_level = 'GOLD';
    ELSE
        SET out_level = 'SILVER';
    END IF;
END

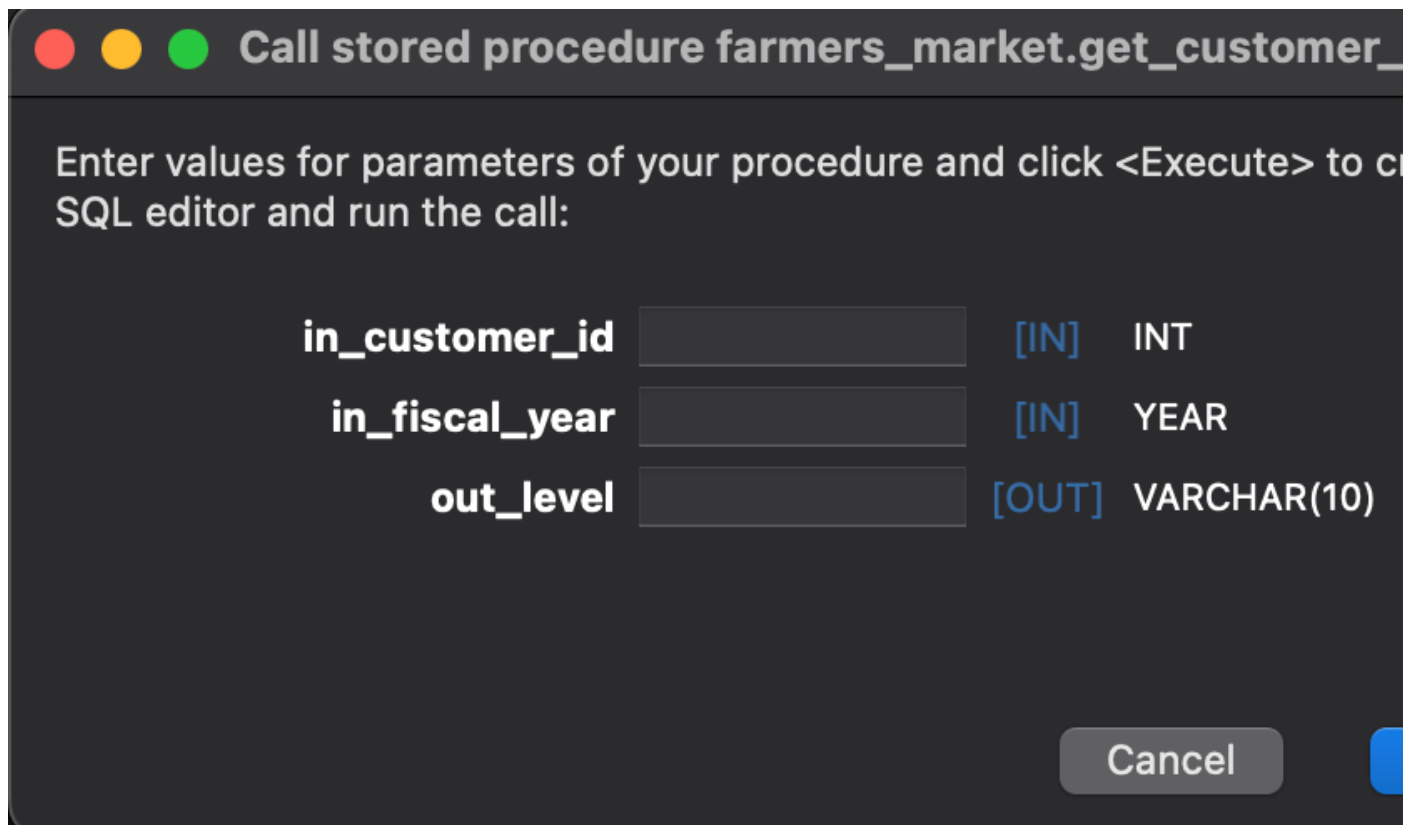
```

Notice that we're not using the RETURN statement anywhere.  
Infact, the RETURN statement is not allowed in Stored Procedures.

Instead, we use the parameters to provide input(s) and store the output(s).

### Step 3:

- Click on Apply.
- Refresh your Database Schema
- Go to the Stored Procedure
- Click the ⚡ icon on the right side
- Provide input to your Stored Procedure



Call stored procedure farmers\_market.get\_customer\_level

Enter values for parameters of your procedure and click <Execute> to call the procedure in the SQL editor and run the call:

in_customer_id	<input type="text"/>	[IN]	INT
in_fiscal_year	<input type="text"/>	[IN]	YEAR
out_level	<input type="text"/>	[OUT]	VARCHAR(10)

Cancel

---

**-> Input:**

- in\_customer\_id = 7
- in\_fiscal\_year = 2019

You can leave the out\_level as blank  
Click on Execute

**-> Stored Procedure Call**

```
set @out_level = '0';  
call farmers_market.get_customer_level(7, 2019, @out_level);  
select @out_level;
```

**-> Output:** Gold

---

**-> Input:**

- in\_customer\_id = 7
- in\_fiscal\_year = 2020

You can leave the out\_level as blank  
Click on Execute

**-> Stored Procedure Call**

```
set @out_level = '0';  
call farmers_market.get_customer_level(7, 2020, @out_level);  
select @out_level;
```

**-> Output:** Silver

---

## **Function vs. Stored Procedure**

- A function must return a value but in Stored Procedure it is optional. Even a procedure can return zero or n values.
- Functions can have only input parameters for it whereas Procedures can have input or output parameters.
- Functions can be called from Procedure whereas Procedures cannot be called from a Function.
- The procedure allows SELECT as well as DML(INSERT/UPDATE/DELETE) statement in it whereas Function allows only SELECT statement in it.
- Procedures cannot be utilized in a SELECT statement whereas Function can be embedded in a SELECT statement.
- Stored Procedures cannot be used in the SQL statements anywhere in the WHERE/HAVING/SELECT section whereas Function can be.