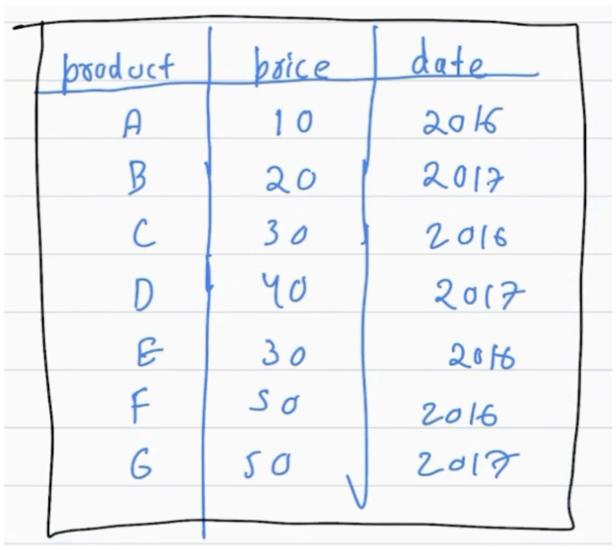
# SQL Advanced Concepts

- Partitioning
- Indexes
- Functions
- Stored Procedures

## Partitioning

Let's say we have the following data from Amazon and we want to filter out rows for the year 2016 and get a sum of price.



| product | price | date |
|---|---|---|
| A | 10 | 2016 |
| B | 20 | 2017 |
| C | 30 | 2016 |
| D | 40 | 2017 |
| E | 30 | 2016 |
| F | 50 | 2016 |
| G | 50 | 2017 |

To do that, we'll first have to go through each record, one by one, and check if the year matches to 2016.

For a record (say row 2), even if the year does not match the specified condition, we still had to check and it takes some time.

Now suppose if we had a million records in total. We'll have to check for each one of them individually. And it'll take a lot of time to process.

One good solution to resolve this problem would be to perform **Partitioning** on the table.

## What is Partitioning?

- **Partitioning** is a technique used in databases **to divide large tables into smaller, more manageable parts.**

- It can improve query performance and simplify data management. Partitioning involves splitting a table into multiple partitions based on a specific criterion, such as a range of values or a list of values.

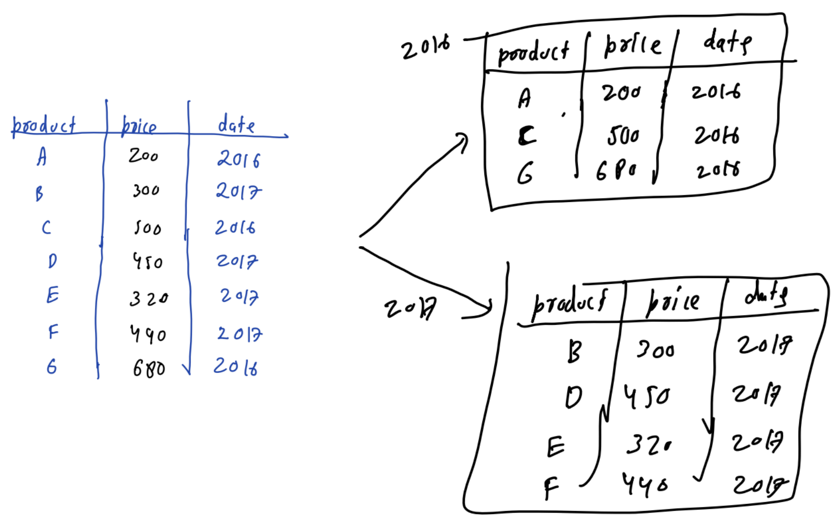
- Partitioning can be beneficial when dealing with large datasets and performing operations that involve filtering or aggregating data based on certain criteria. Instead of scanning the entire table, partitioning allows the database to only access the relevant partitions, reducing the time required for processing.

Say if we split this table into two parts for 2016 & 2017.
So now we only need to check through the partition for 2016 for calculating the sum of price.

The benefit of partitioning is that it'll save a lot of time while querying data from the database.

Note that however creating a partition takes some time itself, it is a one-time activity whereas querying data is a continuous process.

We can create partitions based on other attributes like location, etc. as well.
We can also create separate partitions based on different columns.



| product | price | date |
|---|---|---|
| A | 200 | 2016 |
| B | 300 | 2017 |
| C | 500 | 2016 |
| D | 450 | 2017 |
| E | 320 | 2017 |
| F | 490 | 2017 |
| G | 680 | 2016 |

| product | price | date |
|---|---|---|
| A | 200 | 2016 |
| C | 500 | 2016 |
| G | 680 | 2016 |

| product | price | date |
|---|---|---|
| B | 300 | 2017 |
| D | 450 | 2017 |
| E | 320 | 2017 |
| F | 440 | 2017 |

<u>To create the "demo_db" database</u> -

```
CREATE DATABASE demo_db;
```

<u>To use the "demo_db" database</u> -

```
USE demo_db;
```

## How to create a partitioned table?

To create a partitioned table, you need to specify the partitioning method and define the partitions.

**(RANGE Partition)**

```
CREATE TABLE Sales (
  cust_id INT NOT NULL,
  name VARCHAR(40),
  store_id VARCHAR (20) NOT NULL,
  bill_no INT NOT NULL,
  bill_date DATE PRIMARY KEY NOT NULL,
  amount DECIMAL (8,2) NOT NULL)
PARTITION BY RANGE (year(bill_date)) (
  partition p1 VALUES LESS THAN (2016),
  partition p2 VALUES LESS THAN (2017),
  partition p3 VALUES LESS THAN (2018),
  partition p4 VALUES LESS THAN (2020));
```

**While creating a table, how would I know that I am gonna need this particular partition?**

Ideally, you should create a partition based on a column with respect to which you frequently need to filter the data.

Inserting values into the partitioned table -

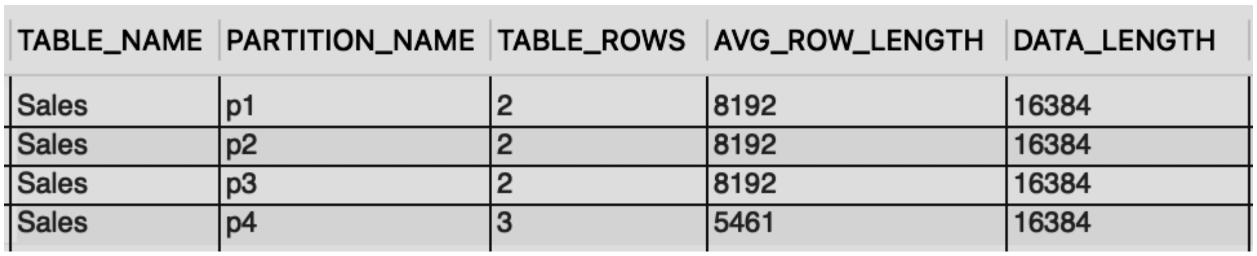
```
INSERT INTO Sales VALUES
(1, 'Mike', 'S001', 101, '2015-01-02', 125.56),
(2, 'Robert', 'S003', 103, '2015-01-25', 476.50),
(3, 'Peter', 'S012', 122, '2016-02-15', 335.00),
(4, 'Joseph', 'S345', 121, '2016-03-26', 787.00),
(5, 'Harry', 'S234', 132, '2017-04-19', 678.00),
(6, 'Stephen', 'S743', 111, '2017-05-31', 864.00),
(7, 'Jacson', 'S234', 115, '2018-06-11', 762.00),
(8, 'Smith', 'S012', 125, '2019-07-24', 300.00),
(9, 'Adam', 'S456', 119, '2019-08-02', 492.20);
```

To view the “Sales” table -

```
SELECT * FROM Sales;
```

**Information_Schema**
It is a special table inside SQL that has a lot of metadata about other tables.

```
SELECT TABLE_NAME, PARTITION_NAME, TABLE_ROWS,
AVG_ROW_LENGTH, DATA_LENGTH
FROM INFORMATION_SCHEMA.PARTITIONS
WHERE TABLE_SCHEMA = 'demo_db' AND TABLE_NAME = 'Sales';
```



| TABLE_NAME | PARTITION_NAME | TABLE_ROWS | AVG_ROW_LENGTH | DATA_LENGTH |
|---|---|---|---|---|
| Sales | p1 | 2 | 8192 | 16384 |
| Sales | p2 | 2 | 8192 | 16384 |
| Sales | p3 | 2 | 8192 | 16384 |
| Sales | p4 | 3 | 5461 | 16384 |

Suppose you want to delete all entries from partition p1.

```
ALTER TABLE Sales TRUNCATE PARTITION p1;
```

## (LIST Partition)

```
CREATE TABLE Stores_list (
  cust_name VARCHAR(40),
  bill_no VARCHAR (20) NOT NULL,
  store_id INT PRIMARY KEY NOT NULL,
  bill_date DATE NOT NULL,
  amount DECIMAL(8,2) NOT NULL
)
PARTITION BY LIST(store_id) (
  PARTITION PEast VALUES IN (101, 103, 105),
  PARTITION PWest VALUES IN (102, 104, 106),
  PARTITION pNorth VALUES IN (107, 109, 111),
  PARTITION PSouth VALUES IN (108, 110, 112));
```

## (Hash Partition)

Let's say we have 4 partitions. Then,

- 23%4 = 3
- 24%4 = 0
- 3%4 = 3
- 4%4 = 0
- 5%4 = 1
- 6%4 = 2

We can see that using the **(%)** operator always yields a value between [0-3].

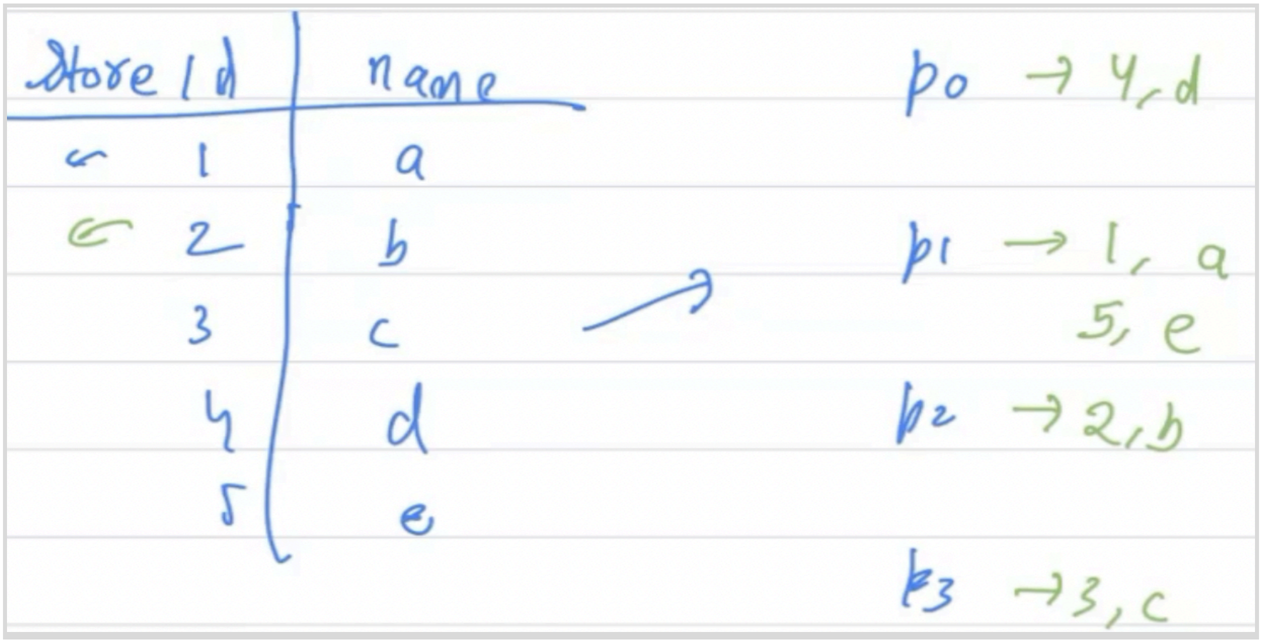
Also, everytime if we pass the same input, our output will be the same.

So if we fix the no. of partitions (say N), we can easily fix the output to be between [0 to N-1].

Similarly, in hash partitioning, we define beforehand that we have this many no. of partitions. Now the hash function sends the input to one of these partitions based on the property of that hash function.

## Creating a hash partitioned table -

```
CREATE TABLE Stores_hash (
cust_name VARCHAR(40),
bill_no VARCHAR (20) NOT NULL,
store_id INT PRIMARY KEY NOT NULL,
bill_date DATE NOT NULL,
amount DECIMAL(8,2) NOT NULL
)
PARTITION BY HASH(store_id)
PARTITIONS 4;
```



Here, if we're looking for the store_id 5, we'll have to go through each of these records, one by one in order to reach 5.

Now suppose we have hash partitioning and we know the no. of partitions is 4. Then we can directly use 5%4=1 i.e. p1.
So we only need to look into partition 1 which is way faster as compared to going through all the records.

---

## Indexes

Suppose you're looking for a particular record in a database. To fetch that record, you'll have to scan the whole disk first. Then get that record into the RAM and from there into the CPU.

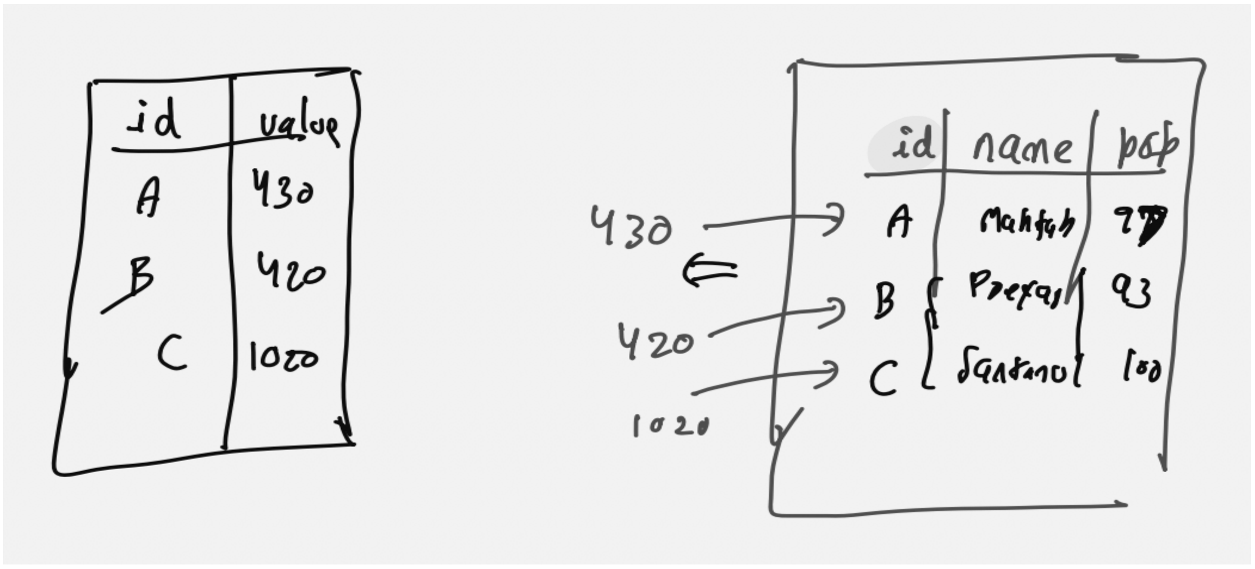
What if we want to reduce the number of disk accesses?

If we can somehow find out that a particular record is present at this location, the disk access will reduce significantly.

This is the basic idea behind **Indexes**.

- Indexes are data structures in a database that provide quick access to specific data based on the values in one or more columns. They work similarly to indexes in books, allowing you to find information more efficiently by referencing the index rather than scanning the entire table.
- Creating indexes on frequently queried columns can significantly improve query performance. When a query includes conditions or joins on indexed columns, the database can use the index to locate the relevant data more quickly, reducing the time required for query execution.

Whatever data you're looking for, an index will tell you where that data is stored and that it'll do with the help of a hash function.



Now don't you think since it occupies comparatively lesser space, we can keep this index inside the RAM for faster access.

The only drawback is that as soon as you insert a new record into the database, you'll have to add a corresponding entry into the index as well.

So the tradeoff here is that an index makes the reading faster but at the same time slows down the write operation.

To use the “demo_db” database -

```
USE demo_db;
```

To create the “names” table -

```
CREATE TABLE `names` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `first_name` varchar(50) CHARACTER SET latin1 DEFAULT NULL,
  `last_name` varchar(50) CHARACTER SET latin1 DEFAULT NULL,
PRIMARY KEY (`id`)
);
```

Inserting records into the “names” table -

```
INSERT INTO names VALUES (1, 'shivank', 'agrawal');
INSERT INTO names VALUES (2, 'shiva', 'agrawal');
INSERT INTO names VALUES (3, 'shi', 'agrawal');
INSERT INTO names VALUES (4, 'sh', 'agrawal');
INSERT INTO names VALUES (5, 'shivank1', 'agrawal');
INSERT INTO names VALUES (6, 'shivank', 'agrawal');
```

To view the “names” table -

```
SELECT * FROM names;
```

## EXPLAIN in MySQL

The EXPLAIN keyword is used with a SQL query to obtain the following information about the query execution plan in MySQL :

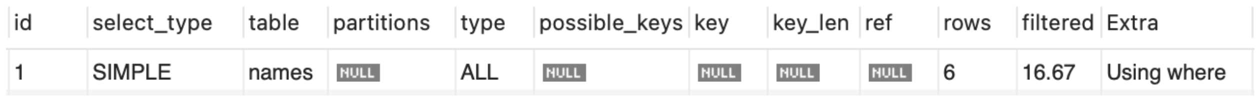
*id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | extra*

- The ID of the query
- The type of your SELECT (if you are running a SELECT)
- The table on which your query was running
- Partitions accessed by your query

- Types of JOINs used (if any)
- Indexes from which MySQL could choose
- Indexes MySQL actually used
- The length of the index chosen by MySQL
- The number of rows accessed by the query
- Columns compared to the index
- The percentage of rows filtered by a specified condition
- Any extra information relevant to the query

Before creating an index -

```
EXPLAIN SELECT * FROM names WHERE first_name='shivank';
```



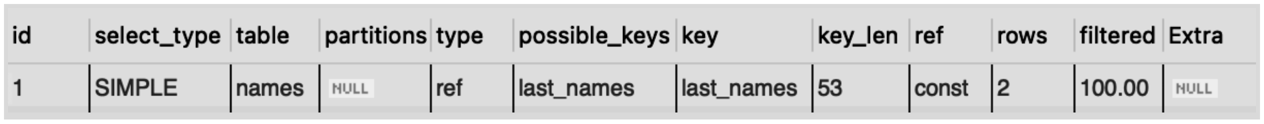
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | SIMPLE | names | NULL | ALL | NULL | NULL | NULL | NULL | 6 | 16.67 | Using where |

Creating an index -

```
CREATE INDEX first_name_index ON names(first_name);
```

After creating an index -

```
EXPLAIN SELECT * FROM names WHERE first_name='shivank';
```



| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | SIMPLE | names | NULL | ref | last_names | last_names | 53 | const | 2 | 100.00 | NULL |

**How would you delete an existing index?**

```
DROP INDEX first_name_index;
```

## 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 logins with their email, they get a message saying "Hi".

For Shivank - "Hi Shivank"
For Suraaj - "Hi Suraaj"
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
```

**Note:**

The "**DETERMINISTIC**" refers to a property of a function which says :

- If you provide the same inputs to a deterministic function multiple times, you will always get the same result.
- In other words, the function always produces the same output for a given set of input parameters.

**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 - link

```
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? (starting from Oct 2018 to Sept 2019)

**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;
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**.

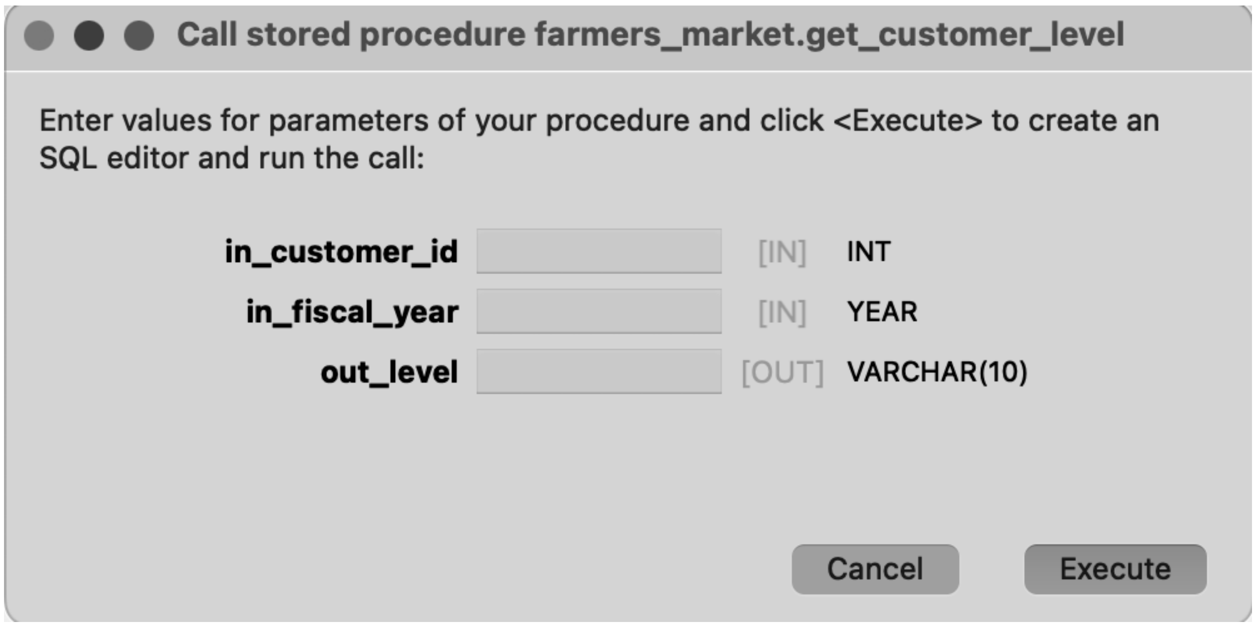
```
CREATE PROCEDURE `get_customer_level`
        (IN in_customer_id INT,
        IN in_fiscal_year YEAR,
        OUT out_level VARCHAR(10))

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



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

| | | | |
|---|---|---|---|
| in_customer_id | | [IN] | INT |
| in_fiscal_year | | [IN] | YEAR |
| out_level | | [OUT] | VARCHAR(10) |

---

**-> 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.