A variable is a named data object whose value can change during the stored procedure execution. We typically use the variables in stored procedures to hold the immediate results. These variables are local to the stored procedure. You must declare a variable before using it.

Declaring variables

To declare a variable inside a stored procedure, you use the DECLARE statement as follows:

DECLARE variable_name datatype(size) DEFAULT default_value;

Let's examine the statement above in more detail:

- First, you specify the variable name after the DECLARE keyword. The variable name must follow the naming rules of MySQL table column names.
- Second, you specify the data type of the variable and its size. A variable can have any MySQL data types such as INT, VARCHAR , and DATETIME.
- Third, when you declare a variable, its initial value is NULL. You can assign the variable a default value using the DEFAULTkeyword.

For example, we can declare a variable named total_sale with the data type INT and default value 0 as follows:

DECLARE total_sale INT DEFAULT 0;

MySQL allows you to declare two or more variables that share the same data type using a single DECLARE statement as follows:

DECLARE x, y INT DEFAULT 0;

In this example, we declared two integer variables x and y, and set their default values to zero.

Assigning variables

Once you declared a variable, you can start using it. To assign a variable another value, you use the SET statement, for example:

DECLARE total_count INT DEFAULT 0;
SET total_count = 10;
The value of the total_count variable is 10 after the assignment.

Besides the SET statement, you can use the SELECT INTO statement to assign the result of a query, which returns a scalar value, to a variable. See the following example:

DECLARE total_products INT DEFAULT 0;
SELECT COUNT(*) INTO total_products
FROM products;
In the example above:

- First, we declared a variable named total_products and initialized its value to 0.
- Then, we used the SELECT INTO statement to assign the total_products variable the number of products that we selected from the products table in the sample database.

Variables scope

A variable has its own scope that defines its lifetime. If you declare a variable inside a stored procedure, it will be out of scope when the END statement of stored procedure reaches.

If you declare a variable inside BEGIN END block, it will be out of scope if the END is reached. You can declare two or more variables with the same name in different scopes because a variable is only effective in its own scope. However, declaring variables with the same name in different scopes is not good programming practice.

A variable whose name begins with the @ sign is a session variable. It is available and accessible until the session ends.

Introduction to MySQL stored procedure parameters

Almost stored procedures that you develop require parameters. The parameters make the stored procedure more flexible and useful. In MySQL, a parameter has one of three modes: IN,OUT, or INOUT.

- ❒ IN – is the default mode. When you define an IN parameter in a stored procedure, the calling program has to pass an argument to the stored procedure. In addition, the value of an IN parameter is protected. It means that even the value of the IN parameter is changed inside the stored procedure, its original value is retained after the stored procedure ends. In other words, the stored procedure only works on the copy of the IN parameter.
- ❒ OUT – the value of an OUT parameter can be changed inside the stored procedure and its new value is passed back to the calling program. Notice that the stored procedure cannot access the initial value of the OUT parameter when it starts.
- ❒ INOUT – an INOUT parameter is a combination of IN and OUT parameters. It means that the calling program may pass the argument, and the stored procedure can modify the INOUT parameter, and pass the new value back to the calling program.

The syntax of defining a parameter in the stored procedures is as follows:

MODE param_name param_type(param_size)

- The MODE could be IN , OUTor INOUT , depending on the purpose of the parameter in the stored procedure.
- The param_name is the name of the parameter. The name of the parameter must follow the naming rules of the column name in MySQL.
- Followed the parameter name is its data type and size. Like a variable, the data type of the parameter can be any valid MySQL data type.

Each parameter is separated by a comma (,) if the stored procedure has more than one parameter.

The syntax of defining a parameter in the stored procedures is as follows:

MODE param_name param_type(param_size)

- The MODE could be IN , OUTor INOUT , depending on the purpose of the parameter in the stored procedure.

- The param_name is the name of the parameter. The name of the parameter must follow the naming rules of the column name in MySQL.
- Followed the parameter name is its data type and size. Like a variable, the data type of the parameter can be any valid MySQL data type.

Each parameter is separated by a comma (,) if the stored procedure has more than one parameter.

MySQL stored procedure parameter examples

The IN parameter example

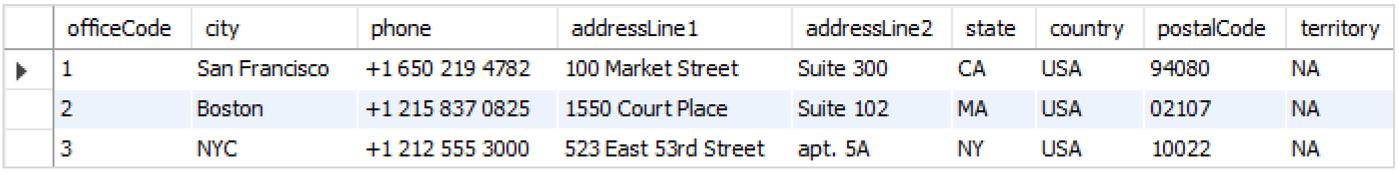
The following example illustrates how to use the IN parameter in the GetOfficeByCountry stored procedure that selects offices located in a particular country.

```
DELIMITER //
CREATE PROCEDURE GetOfficeByCountry(IN countryName VARCHAR(255))
BEGIN
SELECT *
FROM offices
WHERE country = countryName;
END //
DELIMITER ;
```

The countryName is the IN parameter of the stored procedure. Inside the stored procedure, we select all offices that locate in the country specified by the countryName parameter.

Suppose, we want to get all offices in the USA, we just need to pass a value (USA) to the stored procedure as follows:

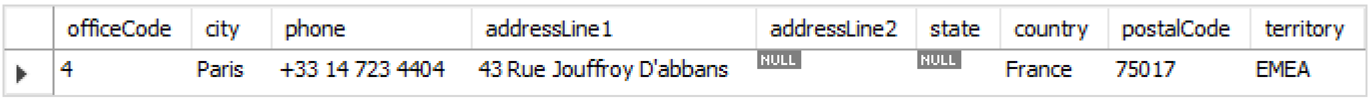
```
CALL GetOfficeByCountry('USA');
```



officeCode	city	phone	addressLine1	addressLine2	state	country	postalCode	territory
1	San Francisco	+1 650 219 4782	100 Market Street	Suite 300	CA	USA	94080	NA
2	Boston	+1 215 837 0825	1550 Court Place	Suite 102	MA	USA	02107	NA
3	NYC	+1 212 555 3000	523 East 53rd Street	apt. 5A	NY	USA	10022	NA

To get all offices in France, we pass the France literal string to the GetOfficeByCountry stored procedure as follows:

```
CALL GetOfficeByCountry('France')
```



officeCode	city	phone	addressLine1	addressLine2	state	country	postalCode	territory
4	Paris	+33 14 723 4404	43 Rue Jouffroy D'abbans	NULL	NULL	France	75017	EMEA

The OUT parameter example

The following stored procedure returns the number of orders by order status. It has two parameters:

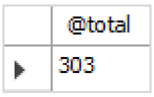
- orderStatus : the IN parameter that is the order status which we want to count the orders.
- total : the OUT parameter that stores the number of orders for a specific order status.

The following is the source code of the CountOrderByStatus stored procedure.

```
DELIMITER $$
CREATE PROCEDURE CountOrderByStatus( IN orderStatus VARCHAR(25), OUT total
INT)
BEGIN
SELECT count(orderNumber) INTO total
FROM orders
WHERE status = orderStatus;
END$$
DELIMITER ;
```

To get the number of shipped orders, we call the CountOrderByStatus stored procedure and pass the order status as Shipped, and also pass an argument (@total) to get the return value.

```
CALL CountOrderByStatus('Shipped',@total);
SELECT @total;
```

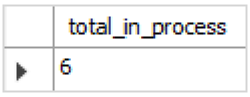


	@total
▶	303

To get the number of orders that are in-process, we call the CountOrderByStatus stored procedure as follows:

```
CALL CountOrderByStatus('in process',@total);

SELECT @total AS total_in_process;
```



	total_in_process
▶	6

The INOUT parameter example

The following example demonstrates how to use an INOUT parameter in the stored procedure.

```
DELIMITER $$
CREATE PROCEDURE set_counter(INOUT count INT(4),IN inc INT(4))
BEGIN
SET count = count + inc;
END$$
DELIMITER ;
```

How it works.

- The set_counter stored procedure accepts one INOUT parameter (count) and one IN parameter (inc).
- Inside the stored procedure, we increase the counter (count) by the value of the inc parameter.

See how we call the set_counter stored procedure:

```
SET @counter = 1;
CALL set_counter(@counter,1); -- 2
CALL set_counter(@counter,1); -- 3
CALL set_counter(@counter,5); -- 8
```

```
SELECT @counter; -- 8
```

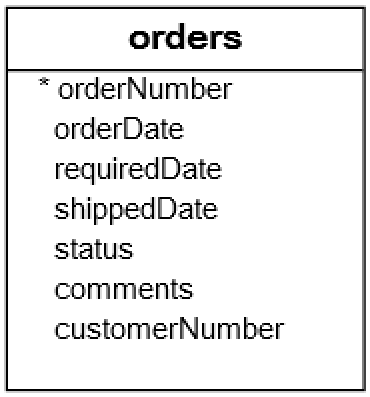
In this tutorial, we have shown you how to define parameters for a stored procedure, and introduced you to different parameter modes: IN, OUT, and INOUT.

MySQL Stored Procedures That Return Multiple Values

MySQL stored function returns only one value. To develop stored programs that return multiple values, you need to use stored procedures with INOUT or OUT parameters.

Stored procedures that return multiple values example

Let's take a look at the orders table in the sample database.



The following stored procedure accepts customer number and returns the total number of orders that were shipped, canceled, resolved, and disputed.

```
DELIMITER $$

CREATE PROCEDURE get_order_by_cust(

IN cust_no INT,

OUT shipped INT,

OUT canceled INT,

OUT resolved INT,

OUT disputed INT)

BEGIN

-- shipped

SELECT count(*) INTO shipped

FROM orders

WHERE customerNumber = cust_no AND status = 'Shipped';

-- canceled
```

SELECT count(*) INTO canceled

FROM orders

WHERE customerNumber = cust_no AND status = 'Canceled';

-- resolved

SELECT count(*) INTO resolved

FROM orders

WHERE customerNumber = cust_no AND status = 'Resolved';

-- disputed

SELECT count(*) INTO disputed

FROM orders

WHERE customerNumber = cust_no AND status = 'Disputed';

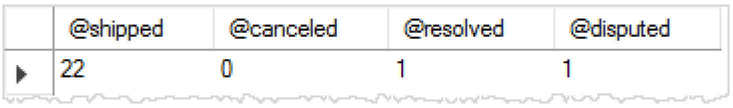
END

In addition to the IN parameter, the stored procedure takes four additional OUT parameters: shipped, canceled, resolved, and disputed. Inside the stored procedure, you use a SELECT statement with the COUNT function to get the corresponding total of orders based on the order's status and assign it to the respective parameter.

To use the get_order_by_cust stored procedure, you pass customer number and four user-defined variables to get the out values.

After executing the stored procedure, you use the SELECT statement to output the variable values.

CALL get_order_by_cust(141,@shipped,@canceled,@resolved,@disputed);
SELECT @shipped,@canceled,@resolved,@disputed;



@shipped	@canceled	@resolved	@disputed
22	0	1	1

Stored Procedures and Stored Functions in MySQL

Example cases discussed here are:

- CASE 1: A Stored Procedure that Accept No Parameters
- CASE 2: A Stored Procedure that Accept Parameters (IN, OUT, INOUT)
- CASE 3: A Stored Procedure that Accept Parameters, Return ResultSet
- CASE 4: A Stored Function that Accept No Parameters
- CASE 5: A Stored Function that Accept Parameters

Prerequisites

```
DROP TABLE IF EXISTS emp;

CREATE TABLE emp(`first name` VARCHAR(20), id INT PRIMARY KEY);
```

```
insert into emp values('HJK', 1);

insert into emp values('ABC', 2);

insert into emp values('DEF', 3);
```

Verify Using:

```
select * from emp;
```

CASE 1: A Stored Procedure that Accept No Parameters

```
DELIMITER |

CREATE PROCEDURE sample_sp_no_param ()

BEGIN

UPDATE emp SET `first name`= 'ChangedHJK' where id = 1;

END |

DELIMITER ;
```

Execute and Verify Commands

```
CALL sample_sp_no_param;

select * from emp;
```

CASE 2: A Stored Procedure that Accept Parameters (IN, OUT, INOUT)

```
DELIMITER |

CREATE PROCEDURE sample_sp_with_params (IN empId INT UNSIGNED, OUT
oldName VARCHAR(20), INOUT newName VARCHAR(20))

BEGIN

SELECT `first name` into oldName FROM emp where id = empId;

UPDATE emp SET `first name`= newName where id = empId;

END |

DELIMITER ;
```

Execute and Verify Commands

```
set @inout='updatedHJK';
```

```
CALL sample_sp_with_params(1,@out,@inout);

select @out,@inout;

select * from emp;
```

CASE 3: A Stored Procedure that Accept Parameters, Return ResultSet

```
DELIMITER |

CREATE PROCEDURE sample_sp_with_params_resultset (IN empId INT
UNSIGNED, OUT oldName VARCHAR(20), INOUT newName VARCHAR(20))

BEGIN

SELECT `first name` into oldName FROM emp where id = empId;

UPDATE emp SET `first name`= newName where id = empId;

select * from emp;

END |

DELIMITER ;
```

Execute and Verify Commands

```
set @inout='updatedHJKS';

CALL sample_sp_with_params_resultset (1,@out,@inout);
```

You can verify the values of OUT and INOUT parameters as:

```
select @out,@inout;
```

CASE 4: A Stored Function that Accept No Parameters

```
DELIMITER |

CREATE FUNCTION sample_fn_no_param ()

RETURNS INT

BEGIN

DECLARE count INT;

SELECT COUNT(*) INTO count FROM emp;

RETURN count;

END |
DELIMITER ;
```

Execute and Verify Commands

```
select sample_fn_no_param ();
```

CASE 5: A Stored Function that Accept Parameters

```
DELIMITER |

CREATE FUNCTION sample_fn_with_params (empId INT UNSIGNED, newName
VARCHAR(20))

RETURNS VARCHAR(20)

BEGIN

DECLARE oldName VARCHAR(20);

SELECT `first name` into oldName FROM emp where id = empId;

UPDATE emp SET `first name`= newName where id = empId;

RETURN oldName;

END |

DELIMITER ;
```

Execute and Verify Commands

```
select sample_fn_with_params(2,'UpdatedABC');
```

Drop Commands

```
DROP  PROCEDURE IF EXISTS sample_sp_no_param;

DROP PROCEDURE IF EXISTS sample_sp_with_params;

DROP PROCEDURE IF EXISTS sample_sp_with_params_resultset;

DROP FUNCTION IF EXISTS sample_fn_no_param;

DROP FUNCTION IF EXISTS sample_fn_with_params;
```

MySQL Stored Procedure Advantages

Followings are the advantages of using MySQL Stored Procedures:

1. **Increasing the performance of applications**: As we know that after creating the stored procedure it is compiled and stored in the database. But MySQL implements stored procedures slightly different which helps in increasing the performance of the applications.

MySQL stored procedures are compiled on demand. After compiling a stored procedure, MySQL puts it into a cache. And MySQL maintains its own stored procedure cache for every single connection. If an application uses a stored procedure multiple times in a single connection, the compiled version is used; otherwise, the stored procedure works like a query.

2. **Fast**: MySQL Stored procedures are fast because MySQL server takes some advantage of caching. Another reason for its speed is that it makes the reduction in network traffic. Suppose, if we have a repetitive task that requires checking, looping, multiple statements, and no user interaction, does it with a single call to a procedure that's stored on the server.
3. **Portable**: MySQL Stored procedures are portable because when we write our stored procedure in SQL, we know that it will run on every platform that MySQL runs on, without obliging us to install an additional runtime-environment package or set permissions for program execution in the operating system.
4. **Reusable and transparent**: Stored procedures expose the database interface to all applications so that developers don't have to develop functions that are already supported in stored procedures. Hence, we can say that MySQL stored procedures are reusable and transparent.
5. **Secure**: MySQL stored procedures are secure because the database administrator can grant appropriate permissions to applications that access stored procedures in the database without giving any permissions on the underlying database tables.

MySQL Stored Procedure Disadvantages

Followings are the advantages of using MySQL Stored Procedures:

1. **Memory usage increased**: If we use many stored procedures, the memory usage of every connection that is using those stored procedures will increase substantially.
2. **Restricted for complex business logic**: Actually, stored procedure's constructs are not designed for developing complex and flexible business logic.
3. **Difficult to debug**: It is difficult to debug stored procedures. Only a few database management systems allow you to debug stored procedures. Unfortunately, MySQL does not provide facilities for debugging stored procedures.
4. **Difficult to maintain**: It is not easy to develop and maintain stored procedures. Developing and maintaining stored procedures are often required a specialized skill set that not all application developers possess. This may lead to problems in both application development and maintenance phases.