MySQL Local Variables:

MySQL STORED PROCEDURES Local Variables

Just like how we discussed the different types of supported parameters, we can also use Local Variables inside the procedures for temporary storage that have a scope limited to just that procedure itself.

Syntax

```
DECLARE {varName} DATATYPE [DEFAULT value] ;
```

Let's understand the use of local variables with the help of an example.

Suppose we want to find the count of students who is having marks below the average marks of all the students.

Let's create a proc.

```
DELIMITER //
CREATE PROCEDURE stored_proc_tutorial.spCountOfBelowAverage(OUT countBelowAverage INT)
BEGIN
    DECLARE avgMarks DECIMAL(5,2) DEFAULT 0;
    SELECT AVG(total_marks) INTO avgMarks FROM studentMarks;
    SELECT COUNT(*) INTO countBelowAverage FROM studentMarks WHERE total_marks < avgMarks
END //
DELIMITER ;
```

Here you can see that we have declared a local variable named avgMarks.

```
DECLARE avgMarks DECIMAL(5,2) DEFAULT 0;
```

This variable would hold the value of the calculated average from the first SELECT query.

```
SELECT AVG(total_marks) INTO avgMarks FROM studentMarks;
```

Calling Stored procedure from another Stored procedure:

Calling A Procedure From Another STORED PROCEDURE

We can use the concept of nested procedures when it's possible to call a procedure from within another procedure.

Let's understand this with the help of an example.

We will call a procedure from another procedure to return the overall result of a student. If student marks are above average – then the result would be PASS else – FAIL

We will create 2 procs

#1) First, is named **spGetIsAboveAverage** would return a Boolean value if the student marks are above average or not.

- Calculate the AVERAGE using the AVG function and store results in a local variable.
- Fetch marks for the student ID passed in the function call.
- Compare the studentMarks with average marks and return the result as 0 or 1.

#2) Second one is named **spGetStudentResult** – It will pass studentId as input (IN) and expect result as output (OUT) parameter.

- Calls the first procedure **spGetIsAboveAverage**
- Use the result returned from step 1) and set the result to PASS or FAIL depending on the value from step 1) being 1 or 0 respectively.

Let's see the code statements for both procedures.

Code for Procedure 1

```
DELIMITER $$
CREATE PROCEDURE stored_proc_tutorial.spGetIsAboveAverage(IN studentId INT, OUT isAboveAv
BEGIN
    DECLARE avgMarks DECIMAL(5,2) DEFAULT 0;
    DECLARE studMarks INT DEFAULT 0;
    SELECT AVG(total_marks) INTO avgMarks FROM studentMarks;
    SELECT total_marks INTO studMarks FROM studentMarks WHERE stud_id = studentId;
    IF studMarks > avgMarks THEN
        SET isAboveAverage = TRUE;
    ELSE
        SET isAboveAverage = FALSE;
    END IF;
END$$
DELIMITER ;
```

Code for Procedure 2

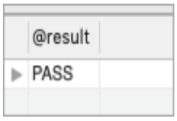
```
DELIMITER $$
CREATE PROCEDURE stored_proc_tutorial.spGetStudentResult(IN studentId INT, OUT result VAR
BEGIN
      -- nested stored procedure call
    CALL stored_proc_tutorial.spGetIsAboveAverage(studentId,@isAboveAverage);
    IF @isAboveAverage = 0 THEN
        SET result = "FAIL";
    ELSE
        SET result = "PASS";
    END IF;
END$$
DELIMITER ;
```

As you can see above, we are calling the ***spGetIsAboveAverage*** procedure from within ***spGetStudentResult***

Let's now call the procedure to fetch results. (The average marks for all the entries in studentTable is 323.6)

For PASS – We will use studentID 2 – having total marks – 450
Since 450 > 323.6 – we will expect the result to be "PASS"

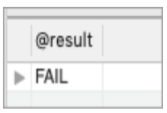
```
CALL stored_proc_tutorial.spGetStudentResult(2,@result);
SELECT @result;
```



@result
PASS

For FAIL result we will use studentId – 10 having total marks – 150
Since 150 < 323.6 – we will expect the result to be "PASS"

```
CALL stored_proc_tutorial.spGetStudentResult(10,@result);
SELECT @result;
```



@result
FAIL

Using Conditional Statements

Let's now see how we can use conditional statements like IF-ELSE or CASE etc within a procedure.

For example, we want to write a procedure to take studentId and depending on the studentMarks we need to return the class according to the below criteria.

Marks >= 400 : Class – First Class
Marks >= 300 and Marks < 400 – Second Class
Marks < 300 – Failed

Let's try creating such a procedure using the IF-ELSE statements.

```
DELIMITER $$
CREATE PROCEDURE stored_proc_tutorial.spGetStudentClass(IN studentId INT, OUT class VARCH
BEGIN
    DECLARE marks INT DEFAULT 0;
    SELECT total_marks INTO marks FROM studentMarks WHERE stud_id = studentId;
        IF marks >= 400 THEN
        SET class = "First Class";
    ELSEIF marks >=300 AND marks < 400 THEN
        SET class = "Second Class";
    ELSE
        SET class = "Failed";
    END IF;
END$$
DELIMITER ;
```

As seen above, we have used

- IF – ELSEIF block within the body to query student marks and SET the OUT parameter named class which is returned to the procedure caller.
- studentId is passed as an IN parameter.
- We've declared a local variable named 'marks' which would fetch the total_marks from the studentMarks table for the given ID and store it temporarily.
- The local variable is used in comparison with the IF ELSE-IF block.

Let's try running the above procedure for different inputs and see the result.

For student ID – 1 – total_marks are 450 – hence the expected result is FIRST CLASS.

```
CALL stored_proc_tutorial.spGetStudentClass(1,@class);
SELECT @class;
```

@class
First Class

For student ID – 6 – total_marks is 380 – Hence the expected result is SECOND CLASS.

```
CALL stored_proc_tutorial.spGetStudentClass(6,@class);
SELECT @class;
```

@class
Second Class

For student ID – 11 – total_marks are 220 – Hence expected result is FAILED.

Error Handling In STORED PROCEDURES

Like any other programming language, MySQL can also throw errors and exceptions while executing queries.

So, while executing individual queries on the terminal or client, you can either get a success response or an error that defines what went wrong.

Similarly, in MySQL procedures, these errors can occur.

We can add handlers to it to handle any generic or specific error code that was thrown during procedure execution. This error handler can direct the execution engine to either continue the procedure execution or exit with some message.

Declaring a Handler for Error

To handle exceptions or errors, you would need to declare a HANDLER within the procedure body.

The syntax for declaring handler is

```
DECLARE {action} HANDLER FOR {condition} {statement}
```

{action} can have values

- **CONTINUE:** This would still continue executing the current procedure.
- **EXIT:** This procedure execution would halt and the flow would be terminated.

{condition}: It's the event that would cause the HANDLER to be invoked.

- Ex – and MySQL error code – ex. MySQL would throw error code 1062 for a duplicate PRIMARY KEY violation
- SQLWARNING / NOTFOUND – etc are used for more generic cases. **For example**, SQLWARNING condition is invoked whenever the MySQL engine issues any warning for the executed statement. **Example,** UPDATES are done without a WHERE clause.

{statement} – Can be one or multiple statements, which we want to execute when the handler is executing. It would be enclosed within BEGIN and END keywords similar to the actual PROCEDURE body.

Note: It's important to note that, you can declare multiple handlers in a given MySQL procedure body definition. This is analogous to having multiple catch blocks for different types of exceptions in a lot of programming languages like Java, C#, etc.

Let's see an example of HANDLER declaration for a DUPLICATE KEY INSERT (which is error code 1062).

```
DECLARE EXIT HANDLER FOR 1062
BEGIN
  SELECT 'DUPLICATE KEY ERROR' AS errorMessage;
END;
```

As seen above, we have created a handler for error 1062. As soon as this condition is invoked, the statements between the BEGIN and END block would be executed.

In this case, the SELECT statement would return the errorMessage as 'DUPLICATE KEY ERROR'.

We can add multiple statements as needed within this BEGIN..END block.

Using Handler Within STORED PROCEDURE Body

Let's now understand how these handlers can be used within the body of the MySQL procedure.

We will understand this with the help of an example.

Let's create a procedure that would insert a record in the studentMarks table and have IN parameters as studentId, total_marks, and grade. We are also adding an OUT parameter named rowCount which would return the total count of records in the studentMarks table.

Let's also add the Error Handler for Duplicate Key record i.e. if someone invokes it for inserting a record with an existing studentID, then the Error handler would be invoked and will return an appropriate error.

```
DELIMITER $$
CREATE PROCEDURE stored_proc_tutorial.spInsertStudentData(IN studentId INT,
IN total_marks INT,
IN grade VARCHAR(20),
OUT rowCount INT)
BEGIN

-- error Handler declaration for duplicate key
DECLARE EXIT HANDLER FOR 1062
    BEGIN
    SELECT 'DUPLICATE KEY ERROR' AS errorMessage;
    END;

-- main procedure statements
INSERT INTO studentMarks(stud_id, total_marks, grade) VALUES(studentId,total_marks,grade)
    SELECT COUNT(*) FROM studentMarks INTO rowCount;
END$$
DELIMITER ;
```

Let's now call this Procedure with an existing student id.

```
CALL stored_proc_tutorial.spInsertStudentData(1,450,'A+',@rowCount);
```

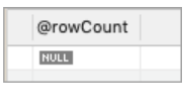
The output would display an error message defined in the error handler.

Also since it's an exit handler, the main procedure flow would not continue. Hence, in this case the rowCount OUT parameter would not be updated as this statement is after the INSERT statement that had generated the error condition.

errorMessage
DUPLICATE KEY ERROR

So, if you try to retrieve the value of rowCount, you would get NULL.

```
SELECT @rowCount;
```



@rowCount
NULL

Let's DROP this procedure, and re-create with CONTINUE action instead of EXIT for the error handler.

```
DROP PROCEDURE stored_proc_tutorial.spInsertStudentData


DELIMITER $$
CREATE PROCEDURE stored_proc_tutorial.spInsertStudentData(IN studentId INT,
IN total_marks INT,
IN grade VARCHAR(20),
OUT rowCount INT)
BEGIN
    DECLARE CONTINUE HANDLER FOR 1062
    BEGIN
        SELECT 'DUPLICATE KEY ERROR' AS errorMessage;
    END;
    INSERT INTO studentMarks(stud_id, total_marks, grade) VALUES(studentId,total_marks,gr
    SELECT COUNT(*) FROM studentMarks INTO rowCount;
END$$
DELIMITER ;
```

As you can see above, instead of `DECLARE EXIT', we are now using 'DECLARE CONTINUE' which would result in continuing the execution after handling the error code.

Let's try to call this procedure with an existing student ID.

```
CALL stored_proc_tutorial.spInsertStudentData(1,450,'A+',@rowCount);
```

You will still see the same error, but the rowCount would be updated this time, as we have used CONTINUE action instead of EXIT.

Let's try to fetch the value of the rowCount OUT parameter.

```
SELECT @rowCount;
```

@rowCount
13

Introduction to MySQL WHILE loop statement

The WHILE loop is a loop statement that executes a block of code repeatedly as long as a condition is true.

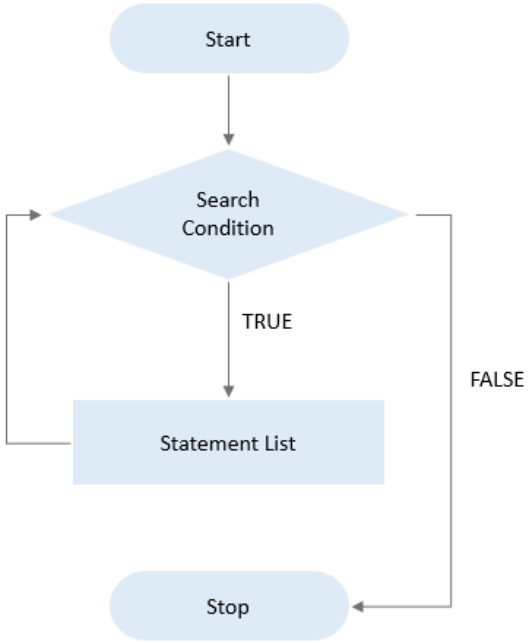
Here is the basic syntax of the WHILE statement:

```
[begin_label:] WHILE search_condition DO
    statement_list
END WHILE [end_label]
```

In this syntax:

- First, specify a search condition after the WHILE keyword.
- The WHILE checks the search_condition at the beginning of each iteration.
- If the search_condition evaluates to TRUE, the WHILE executes the statement_list as long as the search_condition is TRUE.
- The WHILE loop is called a pretest loop because it checks the search_condition before the statement_list executes.
- Second, specify one or more statements that will execute between the DO and END WHILE keywords.
- Third, specify optional labels for the WHILE statement at the beginning and end of the loop construct.

The following flowchart illustrates the MySQL WHILE loop statement:



MySQL WHILE loop statement example

First, create a table named calendars which stores dates and derived date information such as day, month, quarter, and year:

```
CREATE TABLE calendars(
    id INT AUTO_INCREMENT,
    fulldate DATE UNIQUE,
    day TINYINT NOT NULL,
    month TINYINT NOT NULL,
    quarter TINYINT NOT NULL,
    year INT NOT NULL,
    PRIMARY KEY(id)
);
```

Second, create a new stored procedure to insert a date into the `calendars` table:

```
DELIMITER $$

CREATE PROCEDURE InsertCalendar(dt DATE)
BEGIN
    INSERT INTO calendars(
        fulldate,
        day,
        month,
        quarter,
        year
    )
    VALUES(
        dt,
        EXTRACT(DAY FROM dt),
        EXTRACT(MONTH FROM dt),
        EXTRACT(QUARTER FROM dt),
        EXTRACT(YEAR FROM dt)
    );
END$$

DELIMITER ;
```

Third, create a new stored procedure `LoadCalendars()` that loads a number of days starting from a start date into the `calendars` table.

```
DELIMITER $$

CREATE PROCEDURE LoadCalendars(
    startDate DATE,
    day INT
)
BEGIN

    DECLARE counter INT DEFAULT 1;
    DECLARE dt DATE DEFAULT startDate;

    WHILE counter <= day DO
        CALL InsertCalendar(dt);
        SET counter = counter + 1;
        SET dt = DATE_ADD(dt,INTERVAL 1 day);
    END WHILE;

END$$

DELIMITER ;
```

The stored procedure `LoadCalendars()` accepts two arguments:

- `startDate` is the start date inserted into the `calendars` table.
- `day` is the number of days that will be loaded starting from the `startDate`.

In the `LoadCalendars()` stored procedure:

First, declare a `counter` and `dt` variables for keeping immediate values. The default values of `counter` and `dt` are `1` and `startDate` respectively.

Then, check if the `counter` is less than or equal `day`, if yes:

- Call the stored procedure `InsertCalendar()` to insert a row into the `calendars` table.
- Increase the `counter` by one. Also, increase the `dt` by one day using the `DATE_ADD()` function.

The `WHILE` loop repeatedly inserts dates into the `calendars` table until the `counter` is equal to `day`.

The following statement calls the stored procedure `LoadCalendars()` to load 31 days into the `calendars` table starting from `January 1st 2019`.

```
CALL LoadCalendars('2019-01-01',31);
```

id	fulldate	day	month	quarter	year
1	2019-01-01	1	1	1	2019
2	2019-01-02	2	1	1	2019
3	2019-01-03	3	1	1	2019
4	2019-01-04	4	1	1	2019
5	2019-01-05	5	1	1	2019
6	2019-01-06	6	1	1	2019
7	2019-01-07	7	1	1	2019
8	2019-01-08	8	1	1	2019
9	2019-01-09	9	1	1	2019
10	2019-01-10	10	1	1	2019
11	2019-01-11	11	1	1	2019
12	2019-01-12	12	1	1	2019
13	2019-01-13	13	1	1	2019
14	2019-01-14	14	1	1	2019
15	2019-01-15	15	1	1	2019

Chunkwise deletion

```
DELIMITER $$

USE `mydb`$$

DROP PROCEDURE IF EXISTS `sp_delete`$$


CREATE DEFINER=`root`@`localhost` PROCEDURE `sp_delete`()

BEGIN

 DECLARE lastRows INT DEFAULT 0;

 DECLARE startRows INT DEFAULT 0;

   SELECT COUNT(*) FROM AllRows INTO lastRows;

   SET startRows=0;

WHILE startRows <lastRows DO

       delete from tbl where  created_at< DATE_SUB(NOW(), INTERVAL 1 MONTH) Limit
startRows,1000;

   SET startRows= startRows+1000;

END WHILE;


END$$

DELIMITER ;
```

Call sp_delete();

Chunkwise insertion

```
USE `inventory`;
DROP procedure IF EXISTS `insert_data`;

DELIMITER $$
USE `inventory`$$
CREATE PROCEDURE `insert_data` ()
BEGIN
DECLARE i INT DEFAULT 0;
DECLARE limitSize INT DEFAULT 1000000;
DECLARE total_count bigint;
SET total_count = 10000000;
        WHILE i <= total_count DO
        INSERT INTO offline_order_status_june_22_new
        SELECT * FROM offline_order_status_backup_june22 LIMIT i,limitSize;
SET i = i + limitSize;
        END WHILE;
END $$
DELIMITER ;

call insert_data();
```

Cursors:

```
USE `inventory`$$

DROP PROCEDURE IF EXISTS state_update;

DELIMITER $$

CREATE PROCEDURE state_update()

BEGIN

DECLARE done INT DEFAULT 1;

DECLARE i,p INT;

DECLARE var_state_abbr,var_statename VARCHAR(50);

DECLARE crsr1 CURSOR FOR SELECT statename,stateabbr FROM

`db_area`.`state_abbrevations`;

DECLARE CONTINUE HANDLER FOR NOT FOUND SET done =0;

OPEN crsr1;

        WHILE done=1  DO

                FETCH crsr1 INTO var_statename,var_state_abbr;

                UPDATE `db_company`.temp_grouped_state_city SET
                multiple_states=REPLACE(multiple_states,CONCAT(',',var_state_abbr,','),
                CONCAT(',',var_statename,','));

        END WHILE;

UPDATE `db_company`.temp_grouped_state_city SET multiple_states = TRIM(BOTH ',' FROM

multiple_states);

CLOSE crsr1;

END $$

DELIMITER ;
```

Procedure for a dynamic query(prepare statement) for column add in all table of a single schema using the cursor

```
DELIMITER $$
USE `inventory`$$
DROP PROCEDURE IF EXISTS convert_database_to_utf8 ;
CREATE PROCEDURE convert_database_to_utf8()
BEGIN
   DECLARE table_name VARCHAR(255);
   DECLARE done INT DEFAULT FALSE;

   DECLARE cur CURSOR FOR
      SELECT t.table_name FROM information_schema.tables t WHERE t.table_schema =
DATABASE() AND t.table_type='BASE TABLE';
   DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;
   OPEN cur;
      tables_loop: LOOP
         FETCH cur INTO table_name;

         IF done THEN
            LEAVE tables_loop;
         END IF;

         SET @sql = CONCAT("ALTER TABLE ", table_name, "add column status int(2);");
         PREPARE stmt FROM @sql;
         EXECUTE stmt;
         DROP PREPARE stmt;
      END LOOP;
   CLOSE cur;
END $$
DELIMITER ;
call convert_database_to_utf8();
```