STORED PROCEDURES & TRIGGERS

SE 2222 - Week 9

Stored Procedures

- A stored procedure is a prepared SQL code that you can save, so the code can be reused over and over again.

- You can also pass parameters to a stored procedure, so that the stored procedure can act based on the parameter value(s) that is passed.

- They work like functions from a programming language in that you can use if / else sentences , for – while loop sentences etc..

Changing MySQL Delimiter

- When you write SQL statements, you use the semicolon (;) to separate two statements like the following example:
  - SELECT * FROM products;
  - SELECT * FROM customers;

- A MySQL client program such as MySQL Workbench uses the (;) delimiter to separate statements and executes each statement separately.
- A stored procedure, however, consists of multiple statements separated by a semicolon (;).
- If you use a MySQL client program to define a stored procedure that contains semicolon characters, the MySQL client program will not treat the whole stored procedure as a single statement, but many statements.
- Therefore, you must redefine the delimiter temporarily so that you can pass the whole stored procedure to the server as a single statement.

Changing Delimiter

- To redefine the default delimiter, you use the DELIMITER command:

```
DELIMITER delimiter_character
```

- For example, this statement changes the delimiter to //:

```
DELIMITER //
```

- Once change the delimiter, you can use the new delimiter to end a statement as follows:

```
DELIMITER //

SELECT * FROM customers //

SELECT * FROM products //
```

- To change the delimiter back to semicolon, you use this statement:

```
DELIMITER ;
```

Creating a Stored Procedure

General Syntax

```
DELIMITER $$

CREATE PROCEDURE sp_name()
BEGIN
   -- statements
END $$

DELIMITER ;
```

To execute a stored procedure, you use the CALL statement:

```
CALL stored_procedure_name(argument_list);
```

- This query returns all products in the products table from the sample database.

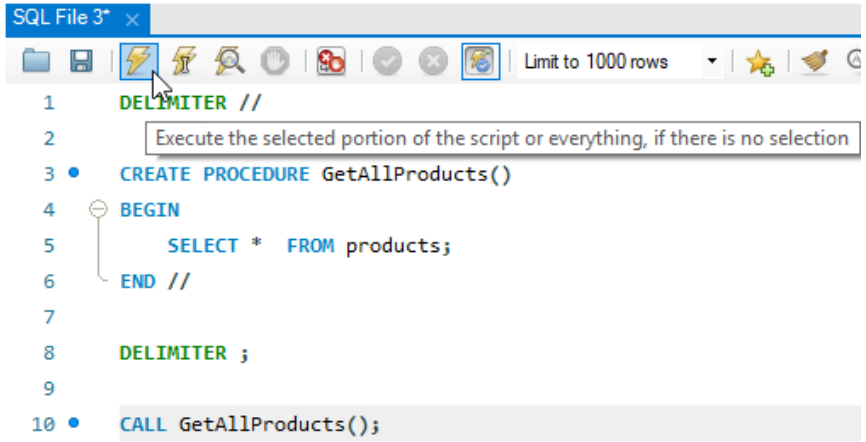
```
SELECT * FROM products;
```

- If we want to create this as a function :

```
DELIMITER //

CREATE PROCEDURE GetAllProducts()
BEGIN
    SELECT *  FROM products;
END //

DELIMITER ;
```



Deleting a Stored Procedure

- To drop a stored procedure , we must type :

```
DROP PROCEDURE [IF EXISTS] stored_procedure_name;
```

Declaring a variable

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];
```

In this syntax:

- First, specify the name of the variable after the `DECLARE` keyword. The variable name must follow the naming rules of MySQL table column names.
- Second, specify the data type and length of the variable. A variable can have any MySQL data types such as `INT`, `VARCHAR`, and `DATETIME`.
- Third, assign a variable a default value using the `DEFAULT` option. If you declare a variable without specifying a default value, its value is `NULL`.

The following example declares a variable named `totalSale` with the data type `DEC(10,2)` and default value `0.0` as follows:

```
DECLARE totalSale DEC(10,2) DEFAULT 0.0;
```

Assigning variables

Assigning variables

Once a variable is declared, it is ready to use. To assign a variable a value, you use the `SET` statement:

```
SET variable_name = value;
```

For example:

```
DECLARE total INT DEFAULT 0;
SET total = 10;
```

The value of the `total` variable is `10` after the assignment.

In addition to the `SET` statement, you can use the `SELECT INTO` statement to assign the result of a query to a variable as shown in the following example:

```
DECLARE productCount INT DEFAULT 0;

SELECT COUNT(*)
INTO productCount
FROM products;
```

Example

```
DELIMITER $$

CREATE PROCEDURE GetTotalOrder()
BEGIN
    DECLARE totalOrder INT DEFAULT 0;

    SELECT COUNT(*)
    INTO totalOrder
    FROM orders;

    SELECT totalOrder;
END$$

DELIMITER ;
```

Defining Parameters

- In MySQL, a parameter has one of three modes: **IN,OUT, or INOUT.**

IN parameters

- 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 parameters

- 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 parameters

- 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.

Here is the basic syntax of defining a parameter in stored procedures:

```
1 [IN | OUT | INOUT] parameter_name datatype[(length)]
```

IN PARAMETER EXAMPLE

```
DELIMITER //

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

DELIMITER ;
```

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

OUT PARAMETER EXAMPLE

```
DELIMITER $$

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

DELIMITER ;
```

INOUT PARAMETER EXAMPLE

```
DELIMITER $$

CREATE PROCEDURE SetCounter(
    INOUT counter INT,
    IN inc INT
)
BEGIN
    SET counter = counter + inc;
END$$

DELIMITER ;
```

IF – THEN & IF – ELSE STATEMENT

```
IF condition THEN
    statements;
END IF;
```

```
DELIMITER $$

CREATE PROCEDURE GetCustomerLevel(
    IN  pCustomerNumber INT,
    OUT pCustomerLevel  VARCHAR(20))
BEGIN
    DECLARE credit DECIMAL(10,2) DEFAULT 0;

    SELECT creditLimit
    INTO credit
    FROM customers
    WHERE customerNumber = pCustomerNumber;

    IF credit > 50000 THEN
        SET pCustomerLevel = 'PLATINUM';
    END IF;
END$$

DELIMITER ;
```

```
IF condition THEN
    statements;
ELSE
    else-statements;
END IF;
```

```
DELIMITER $$

CREATE PROCEDURE GetCustomerLevel(
    IN  pCustomerNumber INT,
    OUT pCustomerLevel  VARCHAR(20))
BEGIN
    DECLARE credit DECIMAL DEFAULT 0;

    SELECT creditLimit
    INTO credit
    FROM customers
    WHERE customerNumber = pCustomerNumber;

    IF credit > 50000 THEN
        SET pCustomerLevel = 'PLATINUM';
    ELSE
        SET pCustomerLevel = 'NOT PLATINUM';
    END IF;
END$$

DELIMITER ;
```

LOOP STATEMENT

```
[label]: LOOP
    ...
    -- terminate the loop
    IF condition THEN
        LEAVE [label];
    END IF;
    ...
END LOOP;
```

```
DELIMITER $$
CREATE PROCEDURE LoopDemo()
BEGIN
    DECLARE x  INT;
    DECLARE str  VARCHAR(255);

    SET x = 1;
    SET str =  '';

    loop_label:  LOOP
        IF  x > 10 THEN
            LEAVE  loop_label;
        END  IF;

        SET  x = x + 1;
        IF  (x mod 2) THEN
            ITERATE  loop_label;
        ELSE
            SET  str = CONCAT(str,x,',');
        END  IF;
    END LOOP;
    SELECT str;
END$$

DELIMITER ;
```

WHILE STATEMENT

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

```
CREATE PROCEDURE `FindEmployeeOffice`(
out output text
)
BEGIN
declare counter int default 0;
declare OutputValues text default '';
declare temp int;
while counter < 10 do
select officeCode
into temp
from employees limit 1;
set counter = counter + 1;
set OutputValues = concat(OutputValues,' ',temp);
end while;
set output = OutputValues;
END
```

SQL TRIGGERS

- In MySQL, a trigger is a stored program invoked automatically in response to an event such as insert, update, or delete that occurs in the associated table. For example, you can define a trigger that is invoked automatically before a new row is inserted into a table.

```
CREATE TRIGGER trigger_name
{BEFORE | AFTER} {INSERT | UPDATE| DELETE }
ON table_name FOR EACH ROW
trigger_body;
```

OLD AND NEW VALUES FOR OPERATIONS

To distinguish between the value of the columns `BEFORE` and `AFTER` the DML has fired, you use the `NEW` and `OLD` modifiers.

For example, if you update the column description, in the trigger body, you can access the value of the description before the update `OLD.description` and the new value `NEW.description`.

The following table illustrates the availability of the `OLD` and `NEW` modifiers:

Trigger Event	`OLD`	`NEW`
`INSERT`	No	Yes
`UPDATE`	Yes	Yes
`DELETE`	Yes	No

Example

First, create a new table named `employees_audit` to keep the changes to the `employees` table:

```
CREATE TABLE employees_audit (
    id INT AUTO_INCREMENT PRIMARY KEY,
    employeeNumber INT NOT NULL,
    lastname VARCHAR(50) NOT NULL,
    changedat DATETIME DEFAULT NULL,
    action VARCHAR(50) DEFAULT NULL
);
```

Next, create a `BEFORE UPDATE` trigger that is invoked before a change is made to the `employees` table.

```
CREATE TRIGGER before_employee_update
    BEFORE UPDATE ON employees
    FOR EACH ROW
 INSERT INTO employees_audit
 SET action = 'update',
     employeeNumber = OLD.employeeNumber,
     lastname = OLD.lastname,
     changedat = NOW();
```

Inside the body of the trigger, we used the `OLD` keyword to access values of the columns `employeeNumber` and `lastname` of the row affected by the trigger.

Continue on…

- http://www.mysqltutorial.org/mysql-triggers/mysql-drop-trigger/

Labwork Questions

Question #1

- Create a procedure that lists all start cities alphabetically and call it.

Question #2

- Create a procedure that inserts the same two new rows with the given parameters.

Question #3

- Insert anything to the table without any procedure and with the procedure at question #2.

Question #4

- Create a trigger such that when a new row is added to the passenger table, the same rows are added three times into the «passenger_backup» table.

Question #5

- Try the previous trigger and observe the change (show every rows before and after trigger triggered).