Lab 10: Stored Procedure and Triggers

Stored Procedures

A stored procedure is a collection of SQL statements that are stored in Database.

One of the major advantages of stored procedures is that they can be used to encapsulate and represent business transactions.

To create a stored procedure, you use the following syntax:

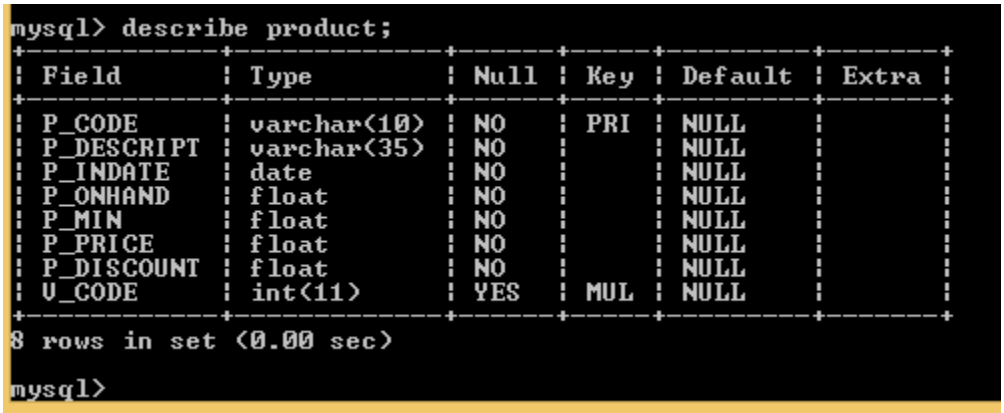
```
CREATE PROCEDURE procedure_name [([IN/OUT] argument data-type, …)]
   BEGIN
   SQL statements;
   ...
END;
```

Note the following important points about stored procedures and their syntax:

- Argument specifies the parameters that are passed to the stored procedure. A stored procedure could have zero or more arguments or parameters.
- IN/OUT indicates whether the parameter is for input, output, or both.
- data-type is one of the procedural SQL data types used in the RDBMS. The data types normally match those used in the RDBMS table creation statement.

Example 1:

Consider the following table *(script available on Google Classroom resources/sales_co.sql)*:



```
mysql> describe product;
+------------+-------------+------+-----+---------+-------+
| Field      | Type        | Null | Key | Default | Extra |
+------------+-------------+------+-----+---------+-------+
| P_CODE     | varchar(10) | NO   | PRI | NULL    |       |
| P_DESCRIPT | varchar(35) | NO   |     | NULL    |       |
| P_INDATE   | date        | NO   |     | NULL    |       |
| P_ONHAND   | float       | NO   |     | NULL    |       |
| P_MIN      | float       | NO   |     | NULL    |       |
| P_PRICE    | float       | NO   |     | NULL    |       |
| P_DISCOUNT | float       | NO   |     | NULL    |       |
| V_CODE     | int(11)     | YES  | MUL | NULL    |       |
+------------+-------------+------+-----+---------+-------+
8 rows in set (0.00 sec)

mysql>
```

```
mysql> select * from product;
+----------+-------------------------------------+----------+----------+-------+---------+-----------+--------+
| P_CODE   | P_DESCRIPT                          | P_INDATE | P_ONHAND | P_MIN | P_PRICE | P_DISCOUNT | V_CODE |
+----------+-------------------------------------+----------+----------+-------+---------+-----------+--------+
| 11QER/31 | Power painter, 15 psi., 3-nozzle    | 2003-11-03 |        8 |     5 |  109.99 |          0 |  25595 |
| 13-Q2/P2 | 7.25-in. pwr. saw blade             | 2003-12-13 |       32 |    15 |   14.99 |       0.05 |  21344 |
| 14-Q1/L3 | 9.00-in. pwr. saw blade             | 2003-11-13 |       18 |    12 |   17.49 |          0 |  21344 |
| 1546-QQ2 | Hrd. cloth, 1/4-in., 2x50           | 2004-01-04 |       15 |     8 |   39.95 |          0 |  23119 |
| 1558-QW1 | Hrd. cloth, 1/2-in., 3x50           | 2004-01-15 |       23 |     5 |   43.99 |          0 |  23119 |
| 2232/QTY | B&D jigsaw, 12-in. blade            | 2003-12-10 |        8 |     5 |  109.92 |       0.05 |  24288 |
| 2232/QWE | B&D jigsaw, 8-in. blade             | 2003-12-24 |        6 |     5 |   99.87 |       0.05 |  24288 |
| 2238/QPD | B&D cordless drill, 1/2-in.         | 2004-01-20 |       12 |     5 |   38.95 |       0.05 |  25595 |
| 23109-HB | Claw hammer                         | 2004-01-20 |       23 |    10 |    9.95 |        0.1 |  21225 |
| 23114-AA | Sledge hammer, 12 lb.               | 2004-01-20 |        8 |     5 |    14.4 |       0.05 |   NULL |
| 54778-2T | Rat-tail file, 1/8-in. fine         | 2003-12-15 |       43 |    20 |    4.99 |          0 |  21344 |
| 89-WRE-Q | Hicut chain saw, 16 in.             | 2004-02-17 |       11 |     5 |  256.99 |       0.05 |  24288 |
| PVC23DRT | PVC pipe, 3.5-in., 8-ft             | 2004-02-20 |      188 |    75 |    5.87 |          0 |   NULL |
| SM-18277 | 1.25-in. metal screw, 25            | 2004-03-01 |      172 |    75 |    6.99 |          0 |  21225 |
| SW-23116 | 2.5-in. wd. screw, 50               | 2004-02-24 |      237 |   100 |    8.45 |          0 |  21231 |
| WR3/TT3  | Steel matting, 4'x8'x1/6", .5" mesh | 2004-01-07 |       18 |     5 |  119.95 |        0.1 |  25595 |
+----------+-------------------------------------+----------+----------+-------+---------+-----------+--------+
16 rows in set (0.00 sec)
```

To illustrate stored procedures, assume that you want to create a procedure (PRC_PROD_DISCOUNT) to assign an additional 5 percent discount for all products when the quantity on hand is more than or equal to twice the minimum quantity.

```
CREATE PROCEDURE PRG_PROD()

BEGIN

UPDATE product

SET P_DISCOUNT = (P_DISCOUNT*0.05)+ P_DISCOUNT

WHERE P_ONHAND >= P_MIN*2;

END;
```

Pick a Delimiter

The delimiter is the character or string of characters which is used to complete an SQL statement. By default, we use semicolon (;) as a delimiter. But this causes problem in stored procedure because a procedure can have many statements, and everyone must end with a semicolon. So, for your delimiter, pick a string which is rarely occur within statement or within procedure. Here we have used double dollar sign i.e. $$. You can use whatever you want. To resume using ";" as a delimiter later, say "DELIMITER ; $$". See here how to change the delimiter:

```
DELIMITER $$
```

```
CREATE PROCEDURE PRG_PROD()

BEGIN
```

```
UPDATE product

SET P_DISCOUNT = (P_DISCOUNT*0.05)+ P_DISCOUNT

WHERE P_ONHAND >= P_MIN*2;

END $$

DELIMITER ;  /*to change the delimiter back*/
```

To execute the stored procedure, you must use the following syntax:

call procedure_name[(parameter_list)];

In this case we will write *call prg_prod();* to execute the procedure.

COMMENT :

The COMMENT characteristic is a MySQL extension. It is used to describe the stored routine and the information is displayed by the SHOW CREATE PROCEDURE statements.

Declare a Variable:

```
DECLARE var_name [, var_name] ... type [DEFAULT value]
```

To provide a default value for a variable, include a DEFAULT clause. The value can be specified as an expression; it need not be constant. If the DEFAULT clause is missing, the initial value is NULL.

Local variables Example

Local variables are declared within stored procedures and are only valid within the BEGIN…END block where they are declared. Local variables can have any SQL data type. The following example shows the use of local variables in a stored procedure.

```
DELIMITER $$
CREATE PROCEDURE my_procedure_Local_Variables()
BEGIN   /* declare local variables */
DECLARE a INT DEFAULT 10;
DECLARE b, c INT;    /* using the local variables */
SET a = a + 100;
SET b = 2;
```

```
SET c = a + b;
BEGIN      /* local variable in nested block */
DECLARE c INT;
SET c = 5;
/* local variable c takes precedence over the one of the
same name declared in the enclosing block. */
SELECT a, b, c;
END;
SELECT a, b, c;
END$$

DELIMITER ;

CALL my_procedure_Local_Variables();
```

Parameter IN example

In the following procedure, we have used a IN parameter 'var1' (type integer) which accept a number from the user. Within the body of the procedure, there is a SELECT statement which fetches rows from 'P' table and display the data whose p_ONHAND is equal to var1. Here is the procedure :

```
DELIMITER $$

CREATE PROCEDURE my_proc_IN (IN var1 INT)
BEGIN
  SELECT * FROM Product where P_ONHAND = var1;
END$$

DELIMITER ;

call my_proc_IN(8);
```

This procedure takes var 1 as a parameter and display that many no of rows from the table p;

```
DELIMITER $$

CREATE PROCEDURE LIMIT_ROW (IN var1 INT)
BEGIN
  SELECT * FROM Product LIMIT var1;
END$$
```

```
DELIMITER ;

call LIMIT_ROW (8);
```

Parameter OUT example

The following example shows a simple stored procedure that uses an OUT parameter.

Using the product table again:

```
DELIMITER $$;

CREATE PROCEDURE PRG_AVG_PRICE(out avg_price decimal)
BEGIN
SELECT AVG(P_PRICE) INTO avg_price FROM Product;
END$$
```

In order to execute the procedure, write:

```
call prg_avg_price(@out);
```

and then:

```
SELECT @out;
```

To print the output.

DROP PROCEDURE

This statement is used to drop a stored procedure or function. That is, the specified routine is removed from the server.

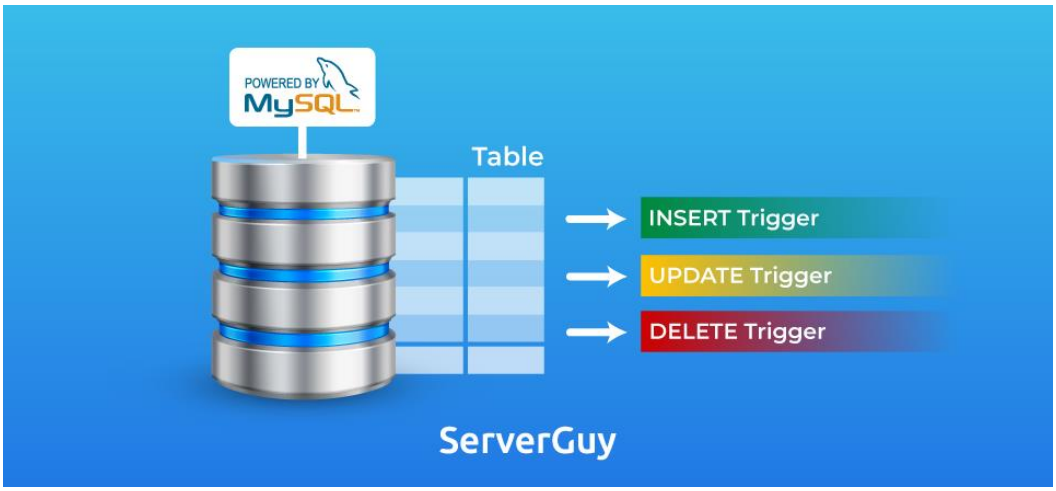
```
DROP PROCEDURE procedure_name   ;
```

Triggers

Triggers are the SQL code that are automatically executed in response to a certain events on a particular table. In other words

A trigger is procedural SQL code that is automatically invoked by the RDBMS upon the occurrence of a given data manipulation event. It is useful to remember that:

- A trigger is invoked before or after a data row is inserted, updated, or deleted.
- A trigger is associated with a database table.
- Each database table may have one or more triggers.
- A trigger is executed as part of the transaction that triggered it.



In order to explain triggers, let us create an example where previously a view was making changes/updates to the original table and we want to restrict this behaviour

```
DELIMITER //
mysql> CREATE TRIGGER prvent_update
   -> BEFORE UPDATE ON product
   -> FOR EACH ROW
   -> BEGIN
   -> SIGNAL SQLSTATE '45000'
   -> SET message_text = 'Update on this tables are not allowed';
   -> END;
   -> //
```

```
mysql> DELIMITER ;
```

However the given example will not only restrict view to make updates, but it will restrict direct updates to the product table as well

Task: Create a trigger that restricts only view to make updates to the product table. Direct updates to the product table must be allowed. (Hint Use IF statement)

Another Example where we create a database for a blogging application. Two tables are required:

1. `blog`: stores a unique post ID, the title, content, and a deleted tag.
2. `audit`: stores a basic set of historical changes with a record ID, the blog post ID, the change type (NEW, EDIT or DELETE) and the date/time of that change.

The following SQL creates the `blog` and indexes the deleted column:

```
CREATE TABLE `blog` (
`id` mediumint(8) unsigned NOT NULL AUTO_INCREMENT,
`title` text,
`content` text,
`deleted` tinyint(1) unsigned NOT NULL DEFAULT '0',
PRIMARY KEY (`id`),
INDEX (deleted)
);
```

The following SQL creates the `audit` table. All columns are indexed, and a foreign key is defined for audit.blog_id which references blog.id. Therefore, when we physically DELETE a blog entry, its full audit history is also removed.

```
CREATE TABLE `audit` (
`id` mediumint(8) unsigned NOT NULL AUTO_INCREMENT,
`blog_id` mediumint(8) unsigned NOT NULL,
`changetype` enum('NEW','EDIT','DELETE') NOT NULL,
`changetime` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
PRIMARY KEY (`id`),
INDEX (`blog_id`),
INDEX (`changetype`),
INDEX (`changetime`),
CONSTRAINT `FK_audit_blog_id` FOREIGN KEY (`blog_id`) REFERENCES `blog`
(`id`) ON DELETE CASCADE ON UPDATE CASCADE
);
```

When a record is inserted into the blog table, we want to add a new entry into the audit table containing the blog ID and a type of 'NEW' (or 'DELETE' if it was deleted immediately).

When a record is updated in the blog table, we want to add a new entry into the audit table containing the blog ID and a type of 'EDIT' or 'DELETE' if the deleted flag is set.

Note that the changetime field will automatically be set to the current time.

Each trigger requires:

1. A unique name. It is preferred to use a name which describes the table and action, e.g. blog_before_insert or blog_after_update.
2. The table which triggers the event. A single trigger can only monitor a single table.
3. When the trigger occurs. This can either be BEFORE or AFTER an INSERT, UPDATE or DELETE. A BEFORE trigger must be used if you need to modify incoming data. An AFTER trigger must be used if you want to reference the new/changed record as a foreign key for a record in another table.
4. The trigger body; a set of SQL commands to run. Note that you can refer to columns in the subject table using OLD.col_name (the previous value) or NEW.col_name (the new value). The value for NEW.col_name can be changed in BEFORE INSERT and UPDATE triggers.

The basic trigger syntax is:

```
CREATE TRIGGER `event_name` BEFORE/AFTER INSERT/UPDATE/DELETE
```

```
ON `database`.`table`
FOR EACH ROW BEGIN
      -- trigger body
      -- this code is applied to every
      -- inserted/updated/deleted row
END;
```

We require two triggers — AFTER INSERT and AFTER UPDATE on the blog table. It's not necessary to define a DELETE trigger since a post is marked as deleted by setting its deleted field to true.

Our trigger body requires a number of SQL commands separated by a semi-colon (;). To create the full trigger code, we must change delimiter to something else such as $$.

Our AFTER-INSERT trigger can now be defined. It determines whether the deleted flag is set, sets the @changetype variable accordingly, and inserts a new record into the audit table:

```
DELIMITER $$
CREATE
      TRIGGER `blog_after_insert` AFTER INSERT
      ON fb.`blog`
      FOR EACH ROW BEGIN
      INSERT INTO audit (blog_id, changetype) VALUES (NEW.id, 'NEW');
END  $$
DELIMITER ;
```

The AFTER UPDATE trigger is almost identical:

```
DELIMITER $$

CREATE
      TRIGGER `blog_after_update` AFTER UPDATE
      ON fb.`blog`
      FOR EACH ROW BEGIN
            IF NEW.deleted THEN
                  SET @changetype = 'DELETE';
            ELSE
                  SET @changetype = 'EDIT';
```

```
            END IF;

            INSERT INTO audit (blog_id, changetype) VALUES (NEW.id,
@changetype);
END $$

DELIMITER ;
```

Let us see what happens when we insert a new post into our blog table:

```
INSERT INTO blog (title, content) VALUES ('Article One', 'Initial text.');
```

Check both the blog and the audit table.

Now let us update our blog text:

```
UPDATE blog SET content = 'Edited text' WHERE id = 1;
```

Check the blog and audit tables again.

Finally, let us mark the post as deleted:

```
UPDATE blog SET deleted = 1 WHERE id = 1;
```

The `audit` table is updated accordingly and we have a record of when changes occurred.

In the above example, there is new keyword 'NEW' which is a MySQL extension to triggers. There is two MySQL extension to triggers 'OLD' and 'NEW'. OLD and NEW are not case sensitive.

Within the trigger body, the OLD and NEW keywords enable you to access columns in the rows affected by a trigger
In an INSERT trigger, only NEW.col_name can be used.
In a UPDATE trigger, you can use OLD.col_name to refer to the columns of a row before it is updated and NEW.col_name to refer to the columns of the row after it is updated.
In a DELETE trigger, only OLD.col_name can be used; there is no new row.

For more details:
https://www.w3resource.com/mysql/mysql-procedure.php#SP
https://www.w3resource.com/mysql/mysql-triggers.php