Lecture 6: Stored procedures and triggers

Jose M. Peña
jose.m.pena@liu.se

Stored procedures

```
CREATE
    [DEFINER = { user | CURRENT_USER }]
    PROCEDURE sp_name ([proc_parameter[,...]])
    [characteristic ...] routine_body

CREATE
    [DEFINER = { user | CURRENT_USER }]
    FUNCTION sp_name ([func_parameter[,...]])
    RETURNS type
    [characteristic ...] routine_body

proc_parameter:
    [ IN | OUT | INOUT ] param_name type

func_parameter:
    param_name type

type:
    Any valid MySQL data type

characteristic:
    LANGUAGE SQL
  | [NOT] DETERMINISTIC
  | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
  | SQL SECURITY { DEFINER | INVOKER }
  | COMMENT 'string'

routine_body:
    Valid SQL procedure statement

DROP {PROCEDURE | FUNCTION} [IF EXISTS] sp_name
```

SHOW CREATE { PROCEDURE | FUNCTION } name

Must contain RETURN …;

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

SET var_name = expr [, var_name = expr] ...

SELECT col_name[,...] INTO var_name[,...] table_expr
```

```
[begin_label:] BEGIN
    [statement_list]
END [end_label]
```

Stored procedures

```
mysql> delimiter //

mysql> CREATE PROCEDURE simpleproc (OUT param1 INT)
    -> BEGIN
    ->   SELECT COUNT(*) INTO param1 FROM t;
    -> END;
    -> //
Query OK, 0 rows affected (0.00 sec)

mysql> delimiter ;

mysql> CALL simpleproc(@a);
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @a;
+------+
| @a   |
+------+
| 3    |
+------+
1 row in set (0.00 sec)
```

```
mysql> CREATE FUNCTION hello (s CHAR(20)) RETURNS CHAR(50)
    -> RETURN CONCAT('Hello, ',s,'!');
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT hello('world');
+----------------+
| hello('world') |
+----------------+
| Hello, world!  |
+----------------+
1 row in set (0.00 sec)
```

When using the `delimiter` command, you should avoid the use of the backslash ('\') character because that is the escape character for MySQL.

```
CREATE PROCEDURE p (OUT ver_param VARCHAR(25), INOUT incr_param INT)
BEGIN
  # Set value of OUT parameter
  SELECT VERSION() INTO ver_param;
  # Increment value of INOUT parameter
  SET incr_param = incr_param + 1;
END;


mysql> SET @increment = 10;
mysql> CALL p(@version, @increment);
mysql> SELECT @version, @increment;
+------------+------------+
| @version   | @increment |
+------------+------------+
| 5.0.25-log | 11         |
+------------+------------+
```

Stored procedures are stored in the server-side. Then, they help to reduce traffic between the server and the clients.

Stored procedures

- Can stored procedures return result **sets** ?
  - Procedures can. Any SELECT statement is returned.
  - Functions cannot.

MySQL Command Line Client

```
mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
mysql> create procedure showsalary(in myid int) begin select salary from emp where id=myid; select salary from emp where id=myid+1; end;//
Query OK, 0 rows affected (0.02 sec)

mysql> call showsalary(1);//
+--------+
| salary |
+--------+
|  10000 |
+--------+
1 row in set (0.00 sec)

+--------+
| salary |
+--------+
|  20000 |
+--------+
1 row in set (0.02 sec)

Query OK, 0 rows affected (0.03 sec)

mysql>
```

Flow control

```
IF search_condition THEN statement_list
    [ELSEIF search_condition THEN statement_list] ...
    [ELSE statement_list]
END IF
```

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

```
[begin_label:] REPEAT
    statement_list
UNTIL search_condition
END REPEAT [end_label]
```

```
[begin_label:] LOOP
    statement_list
END LOOP [end_label]
```

```
CASE case_value
    WHEN when_value THEN statement_list
    [WHEN when_value THEN statement_list] ...
    [ELSE statement_list]
END CASE
```

Or:

```
CASE
    WHEN search_condition THEN statement_list
    [WHEN search_condition THEN statement_list] ...
    [ELSE statement_list]
END CASE
```

`LEAVE label`

This statement is used to exit any labeled flow control construct. It can be used within `BEGIN ... END` or loop constructs (`LOOP`, `REPEAT`, `WHILE`).

`ITERATE label`

`ITERATE` can appear only within `LOOP`, `REPEAT`, and `WHILE` statements. `ITERATE` means "do the loop again."

Example:

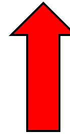
```
CREATE PROCEDURE doiterate(p1 INT)
BEGIN
  label1: LOOP
    SET p1 = p1 + 1;
    IF p1 < 10 THEN ITERATE label1; END IF;
    LEAVE label1;
  END LOOP label1;
  SET @x = p1;
END
```

Exception handlers

```
DECLARE handler_type HANDLER FOR condition_value[,...] statement

handler_type:
    CONTINUE
  | EXIT
  | UNDO

condition_value:
    SQLSTATE [VALUE] sqlstate_value
  | condition_name
  | SQLWARNING
  | NOT FOUND
  | SQLEXCEPTION
  | mysql_error_code
```



Compulsory !! E.g., just ;

See Appendix B for list

The `DECLARE ... HANDLER` statement specifies handlers that each may deal with one or more conditions. If one of these conditions occurs, the specified *statement* is executed. *statement* can be a simple statement (for example, `SET var_name = value`), or it can be a compound statement written using `BEGIN` and `END`

For a `CONTINUE` handler, execution of the current routine continues after execution of the handler statement. For an `EXIT` handler, execution terminates for the `BEGIN ... END` compound statement in which the handler is declared. (This is true even if the condition occurs in an inner block.) The `UNDO` handler type statement is not yet supported.

Control is returned to the outer block if it exists.

A *condition_value* can be any of the following values:

- An SQLSTATE value or a MySQL error code.
- A condition name previously specified with `DECLARE ... CONDITION`.
- `SQLWARNING` is shorthand for all SQLSTATE codes that begin with `01`.
- `NOT FOUND` is shorthand for all SQLSTATE codes that begin with `02`.
- `SQLEXCEPTION` is shorthand for all SQLSTATE codes not caught by `SQLWARNING` or `NOT FOUND`.

Exception handlers

```
mysql> CREATE TABLE test.t (s1 int,primary key (s1));
Query OK, 0 rows affected (0.00 sec)

mysql> delimiter //

mysql> CREATE PROCEDURE handlerdemo ()
    -> BEGIN
    ->   DECLARE CONTINUE HANDLER FOR SQLSTATE '23000' SET @x2 = 1;
    ->   SET @x = 1;
    ->   INSERT INTO test.t VALUES (1);
    ->   SET @x = 2;
    ->   INSERT INTO test.t VALUES (1);
    ->   SET @x = 3;
    -> END;
    -> //
Query OK, 0 rows affected (0.00 sec)

mysql> CALL handlerdemo()//
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @x//
    +------+
    | @x   |
    +------+
    | 3    |
    +------+
    1 row in set (0.00 sec)
```

The example associates a handler with SQLSTATE 23000, which occurs for a duplicate-key error. Notice that `@x` is `3`, which shows that MySQL executed to the end of the procedure. If the line `DECLARE CONTINUE HANDLER FOR SQLSTATE '23000' SET @x2 = 1;` had not been present, MySQL would have taken the default path (`EXIT`) after the second `INSERT` failed due to the `PRIMARY KEY` constraint, and `SELECT @x` would have returned `2`.

Cursors

1st

2nd

3rd

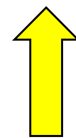
```
CREATE PROCEDURE curdemo()
BEGIN
  DECLARE done INT DEFAULT 0;
  DECLARE a CHAR(16);
  DECLARE b,c INT;
  DECLARE cur1 CURSOR FOR SELECT id,data FROM test.t1;
  DECLARE cur2 CURSOR FOR SELECT i FROM test.t2;
  DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET done = 1;

  OPEN cur1;
  OPEN cur2;

  REPEAT
    FETCH cur1 INTO a, b;
    FETCH cur2 INTO c;
    IF NOT done THEN
       IF b < c THEN
          INSERT INTO test.t3 VALUES (a,b);
       ELSE
          INSERT INTO test.t3 VALUES (a,c);
       END IF;
    END IF;
  UNTIL done END REPEAT;

  CLOSE cur1;
  CLOSE cur2;
END
```

`DECLARE` *`cursor_name`* `CURSOR FOR` *`select_statement`*



`OPEN` *`cursor_name`*

`CLOSE` *`cursor_name`*

`FETCH` *`cursor_name`* `INTO` *`var_name`* `[,` *`var_name`*`] ...`

This statement fetches the next row (if a row exists) using the specified open cursor, and advances the cursor pointer.

If no more rows are available, a No Data condition occurs with SQLSTATE value 02000. To detect this condition, you can set up a handler for it.

```
DROP TRIGGER [schema_name.]trigger_name
```

Triggers

```
[begin_label:] BEGIN
    [statement_list]
END [end_label]
```

```
CREATE
    [DEFINER = { user | CURRENT_USER }]
    TRIGGER trigger_name trigger_time trigger_event
    ON tbl_name FOR EACH ROW trigger_stmt
```

This statement creates a new trigger. A trigger is a named database object that is associated with a table, and that activates when a particular event occurs for the table.

The trigger becomes associated with the table named *tbl_name*, which must refer to a permanent table. You cannot associate a trigger with a `TEMPORARY` table or a view.

trigger_time is the trigger action time. It can be `BEFORE` or `AFTER` to indicate that the trigger activates before or after the statement that activated it.

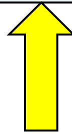
trigger_event indicates the kind of statement that activates the trigger. The *trigger_event* can be one of the following:

- `INSERT`: The trigger is activated whenever a new row is inserted into the table; for example, through `INSERT`, `LOAD DATA`, and `REPLACE` statements.
- `UPDATE`: The trigger is activated whenever a row is modified; for example, through `UPDATE` statements.
- `DELETE`: The trigger is activated whenever a row is deleted from the table; for example, through `DELETE` and `REPLACE` statements. However, `DROP TABLE` and `TRUNCATE` statements on the table do *not* activate this trigger, because they do not use `DELETE`.

Note: Currently, triggers are not activated by cascaded foreign key actions. This limitation will be lifted as soon as possible.

Triggers

```
CREATE
    [DEFINER = { user | CURRENT_USER }]
    TRIGGER trigger_name trigger_time trigger_event
    ON tbl_name FOR EACH ROW trigger_stmt
```



No sense for INSERT

You can refer to columns in the subject table (the table associated with the trigger) by using the aliases `OLD` and `NEW`. `OLD.col_name` refers to a column of an existing row before it is updated or deleted. `NEW.col_name` refers to the column of a new row to be inserted or an existing row after it is updated.

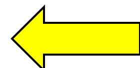
No sense for DELETE

A column named with `OLD` is read-only. You can refer to it (if you have the `SELECT` privilege), but not modify it. A column named with `NEW` can be referred to if you have the `SELECT` privilege for it. In a `BEFORE` trigger, you can also change its value with `SET NEW.col_name = value` if you have the `UPDATE` privilege for it. This means you can use a trigger to modify the values to be inserted into a new row or that are used to update a row.

- If a `BEFORE` trigger fails, the operation on the corresponding row is not performed.
- A `BEFORE` trigger is activated by the *attempt* to insert or modify the row, regardless of whether the attempt subsequently succeeds.
- An `AFTER` trigger is executed only if the `BEFORE` trigger (if any) and the row operation both execute successfully.
- An error during either a `BEFORE` or `AFTER` trigger results in failure of the entire statement that caused trigger invocation.

There cannot be two triggers for a given table that have the same trigger action time and event.

SHOW TRIGGERS;



Triggers

```
MySQL Command Line Client
Query OK, 0 rows affected (0.00 sec)

mysql> create trigger newemp before insert on emp for each row begin declare m int; select count(*) into m from emp where id=new.id; if m>0
then begin select max(id) into m from emp; set new.id=m+1; end; end if; end;//
Query OK, 0 rows affected (0.00 sec)

mysql>
mysql>
mysql>
mysql> insert into emp values(3,3);//
Query OK, 1 row affected (0.05 sec)

mysql> select * from emp;//
+----+--------+
| id | salary |
+----+--------+
|  1 |  10000 |
|  2 |  20000 |
|  3 |      3 |
|  4 |      3 |
+----+--------+
4 rows in set (0.00 sec)

mysql> insert into emp values(11,3);//
Query OK, 1 row affected (0.05 sec)

mysql> select * from emp;//
+----+--------+
| id | salary |
+----+--------+
|  1 |  10000 |
|  2 |  20000 |
|  3 |      3 |
|  4 |      3 |
| 11 |      3 |
+----+--------+
5 rows in set (0.00 sec)

mysql>
```

Exercise: Create a procedure, function and trigger to obtain the **derived attribute** representing the average salary of the employees of a department.