

STORED PROCEDURES (CONT.), FUNCTIONS, TRIGGERS AND MORE ABOUT PL/SQL CONSTRUCTS IN MYSQL

PROF DR MELIKE SAH DIREKOGLU

Slide credit:

<https://dev.mysql.com/doc/refman/8.0/en/sql-compound-statements.html>.

BASIC PROGRAMMING STRUCTURES IN MYSQL

○ Stored Procedures

- Blocks of code stored in the database that are pre-compiled.
- They can operate on the tables within the database and return scalars or results sets.

○ Functions

- Can be used like a built-in function to provide expanded capability to your SQL statements.
- They can take any number of arguments and return a single value.

○ Triggers

- Kick off in response to standard database operations on a specified table.
- Can be used to automatically perform additional database operations when the triggering event occurs.



MORE ABOUT STORED PROCEDURES IN MYSQL

- A stored procedure contains a sequence of SQL commands stored in the database catalog so that it can be invoked later by a program
- Stored procedures are declared using the following syntax:

Create Procedure <proc-name>

(param_spec₁, param_spec₂, ..., param_spec_n)

begin

-- execution code

end;

where each param_spec is of the form:

[in | out | inout] <param_name> <param_type>

- in mode: allows you to pass values into the procedure,
- out mode: allows you to pass value back from procedure to the calling program



MORE ABOUT STORED PROCEDURES

- You can declare variables in stored procedures
- Can have any number of parameters.
- Each parameter has to specify whether it's in, out, or inout.
 - The typical argument list will look like
(**out** ver_param varchar(25), **inout** incr_param int ...)
 - Be careful of output parameters for side effects.
- Your varchar declarations for the parameters have to specify the maximum length.
- The individual parameters can have any supported MySQL datatype.
- They can be called using the call command, followed by the procedure name, and the arguments.
- You can use flow control statements (conditional IF-THEN-ELSE or loops such as WHILE and REPEAT)



VARIABLE DECLARATION

- DECLARE variable_name datatype(size) DEFAULT default_value;
- Variable naming rules: Identifiers can consist of any alphanumeric characters, plus the characters '_' and '\$'. Identifiers can start with any character that is legal in an identifier, including a digit. However, an identifier cannot consist entirely of digits.
- Data types: A variable can have any MySQL data types. For example:
 - Character: CHAR(n), VARCHAR(n)
 - Number: INT, SMALLINT, DECIMAL(i,j), DOUBLE
 - Date: DATE, TIME, DATETIME
 - BOOLEAN
 - <http://www.mysqltutorial.org/mysql-data-types.aspx>



VARIABLE EXAMPLES

- DECLARE x, y INT DEFAULT 0;
- DECLARE today TIMESTAMP DEFAULT CURRENT_DATE;
- DECLARE ename VARCHAR(50);
- DECLARE no_more_rows BOOLEAN;
- SET no_more_rows = TRUE;



ASSIGNING VARIABLES

- Using the SET command:

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

- Using the SELECT INTO command:

```
DECLARE total_products INT DEFAULT 0;  
SELECT COUNT(*) INTO total_products  
FROM products;
```



SELECT INTO

- SELECT columns separated by commas
- INTO variables separated by commas
- FROM tablename
- WHERE condition;

- Ex:

```
SELECT cid, cname INTO custID, customername  
FROM customer  
WHERE cid = 'c01';
```



ARITHMETIC AND STRING OPERATORS

- Arithmetic operators:

$+$, $-$, $*$, $/$

- Modulo operator:

$\%$ or mod

- Other math calculations use math functions:

Pow(x,y)

- Concatenation uses CONCAT function:

```
SELECT CONCAT('New ', 'York ', 'City');
```



MySQL COMPARISON OPERATORS

- EQUAL(=)
- LESS THAN(<)
- LESS THAN OR EQUAL(<=)
- GREATER THAN(>)
- GREATER THAN OR EQUAL(>=)
- NOT EQUAL(<>,!=)



LOGICAL OPERATORS

- Logical AND:
 - AND, &&
 - UnitsInStock < ReorderLevel AND CategoryID=1
 - UnitsInStock < ReorderLevel && CategoryID=1
- Negates value:
 - NOT, !
- Logical OR:
 - OR, ||
 - CategoryID=1 OR CategoryID=8
 - CategoryID=1 || CategoryID=8



CONDITIONS

- IF ELSE
- CASE



IF STATEMENT

- **IF statement:** The IF statement can have THEN, ELSE, and ELSEIF clauses, and it is terminated with END IF.

```
IF variable1 = 0  
THEN SELECT variable1;  
END IF;
```

Ex:

```
IF param1 = 0  
THEN SELECT 'Parameter value = 0';  
ELSE SELECT 'Parameter value <> 0';  
END IF;
```



CASE STATEMENT

- Two different syntaxes:

CASE <expression>

 WHEN <value> then

 <statements>

 WHEN <value> then

 <statements>

...

ELSE

 <statements>

END CASE;



CASE STATEMENT (CONTINUED)

CASE

WHEN <condition> then

<statements>

WHEN <condition> then

<statements>

...

ELSE

<statements>

END CASE;



CASE STATEMENT (IN A PROCEDURE EXAMPLE)

```
DELIMITER //  
CREATE PROCEDURE proc_CASE(IN param1 INT)  
BEGIN  
  DECLARE variable1 INT;  
  SET variable1 = param1 + 1;  
  CASE variable1  
    WHEN 0 THEN  
      INSERT INTO table1 VALUES (param1);  
    WHEN 1 THEN  
      INSERT INTO table1 VALUES (variable1);  
    ELSE  
      INSERT INTO table1 VALUES (99);  
  END CASE;  
END//  
DELIMITER;
```



LOOP CONTROL FLOW

- Iterate <label> – start the loop again
 - Can only be issued within LOOP, REPEAT, or WHILE statements
 - Works much like the “continue” statement in Java or C++.



LOOP

- [begin_label:] LOOP
 - <statement list>
- END LOOP [end_label]
- Note that the end_label has to = the begin_label
- Both are optional



REPEAT UNTIL LOOP

- [begin_label:] REPEAT
 - <statement list>
- UNTIL <search_condition>
- END REPEAT [end_label]



WHILE – DO LOOP

- [begin_label:] WHILE <condition> DO
 - <statements>
- END WHILE [end_label]



WHILE COND DO STATEMENT (IN A PROCEDURE)

DELIMITER //

```
CREATE PROCEDURE proc_WHILE (IN param1 INT)
BEGIN
```

```
    DECLARE variable1, variable2 INT;
```

```
    SET variable1 = 0;
```

```
    WHILE variable1 < param1
```

```
        DO INSERT INTO table1 VALUES (param1);
```

```
        SELECT COUNT(*) INTO variable2 FROM table1;
```

```
        SET variable1 = variable2; /*Update the control*/
```

```
    END WHILE;
```

```
END//
```

```
DELIMITER;
```



NOTES ON THE PREVIOUS EXAMPLE

- The `DELIMITER //` statement sets a session variable so that the `//` becomes the statement terminator.
- For the purposes of that session, the “;” within the stored procedure are just like any other character.
- When the stored procedure is run, however, the “;” function the way that they normally do in MySQL.
- You always want to make the delimiter a “;” again when you change it.



COMMENT SYNTAX

- From a `/*` sequence to the following `*/` sequence.
- From a `#` character to the end of the line.
- From a `--` sequence to the end of the line.
- In MySQL, the `--` (double-dash) comment style requires the second dash to be followed by at least one whitespace
 `-- Programmer: John Smith`



EXAMPLE

```
mysql> select * from employee;
```

id	name	superid	salary	bdate	dno
1	john	3	100000	1960-01-01	1
2	mary	3	50000	1964-12-01	3
3	bob	NULL	80000	1974-02-07	3
4	tom	1	50000	1978-01-17	2
5	bill	NULL	NULL	1985-01-20	1

```
mysql> select * from department;
```

dnumber	dname
1	Payroll
2	TechSupport
3	Research

- Suppose we want to keep track of the total salaries of employees working for each department

```
mysql> create table deptsal as
```

```
-> select dnumber, 0 as totalsalary from department;
```

```
Query OK, 3 rows affected (0.00 sec)
```

```
Records: 3 Duplicates: 0 Warnings: 0
```

```
mysql> select * from deptsal;
```

dnumber	totalsalary
1	0
2	0
3	0

← We need to write a procedure
to update the salaries in
the deptsal table



EXAMPLE – STEP 1

Step 1: Change the delimiter (i.e., terminating character) of SQL statement from semicolon (;) to something else (e.g., //)
So that you can distinguish between the semicolon of the SQL statements in the procedure and the terminating character of the procedure definition

```
mysql> delimiter //
```



EXAMPLE – STEP 2

Step 2:

1. Define a procedure called updateSalary which takes as input a department number.
2. The body of the procedure is an SQL command to update the totalsalary column of the deptsal table.
3. Terminate the procedure definition using the delimiter you had defined in step 1 (//)

```
mysql> delimiter //  
mysql> create procedure updateSalary (IN param1 int)  
-> begin  
->     update deptsal  
->     set totalsalary = (select sum(salary) from employee where dno = param1)  
->     where dnumber = param1;  
-> end; //  
Query OK, 0 rows affected (0.01 sec)
```

EXAMPLE – STEP 3

Step 3: Change the delimiter back to semicolon (;)

```
mysql> delimiter //
```

```
mysql> create procedure updateSalary (IN param1 int)
```

```
    -> begin
```

```
    ->     update deptsal
```

```
    ->     set totalsalary = (select sum(salary) from employee where dno = param1)
```

```
    ->     where dnumber = param1;
```

```
    -> end; //
```

```
Query OK, 0 rows affected (0.01 sec)
```

```
mysql> delimiter ;
```



EXAMPLE – STEP 4

Step 4: Call the procedure to update the totalsalary for each department

```
mysql> call updateSalary(1);  
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> call updateSalary(2);  
Query OK, 1 row affected (0.00 sec)
```

```
mysql> call updateSalary(3);  
Query OK, 1 row affected (0.00 sec)
```



EXAMPLE – STEP 5

Step 5: Show the updated total salary in the deptsal table

```
mysql> select * from deptsal;
```

dnnumber	totalsalary
1	100000
2	50000
3	130000

3 rows in set (0.00 sec)



STORED PROCEDURES IN MYSQL

- Use **show procedure status** to display the list of stored procedures you have created

```
mysql> show procedure status;
```

! Db	! Name	! Type	! Definer	! Modified	! Created	! Security_
type	! Comment	! character_set_client	! collation_connection	! Database Collation		
! ptan	! updateSalary0	! PROCEDURE	! ptan@%	! 2010-03-16 12:21:55	! 2010-03-16 12:21:55	! DEFINER
!	!	! latin1	! latin1_swedish_ci	! latin1_swedish_ci	!	!

! row in set (0.02 sec)

- Or use Routines tab
- Use **drop procedure** to remove a stored procedure

```
mysql> drop procedure updateSalary;  
Query OK, 0 rows affected (0.00 sec)
```



USING CURSORS IN STORED PROCEDURES IN MySQL

- MySQL also supports cursors in stored procedures.
 - A cursor is used to iterate through a set of rows returned by a query so that we can process each individual row.
- To learn more about stored procedures, go to:
<http://www.mysqltutorial.org/mysql-stored-procedure-tutorial.aspx>



EXAMPLE USING CURSORS

- The previous procedure updates one row in deptsal table based on input parameter
- Suppose we want to **update all the rows in deptsal simultaneously**
 - First, let's reset the totalsalary in deptsal to zero (Part 1)

```
mysql> update deptsal set totalsalary = 0;  
Query OK, 0 rows affected (0.00 sec)  
Rows matched: 3  Changed: 0  Warnings: 0
```

```
mysql> select * from deptsal;
```

dnumber	totalsalary
1	0
2	0
3	0

```
3 rows in set (0.00 sec)
```


EXAMPLE USING CURSORS – PART 2

```
mysql> delimiter $$
mysql> drop procedure if exists updateSalary$$
Query OK, 0 rows affected (0.00 sec)
```

Drop the old procedure

```
mysql> create procedure updateSalary()
-> begin
->     declare done int default 0;
->     declare current_dnum int;
->     declare dnumcur cursor for select dnumber from deptsal;
->     declare continue handler for not found set done = 1;
->
->     open dnumcur;
->
->     repeat
->         fetch dnumcur into current_dnum;
->         update deptsal
->         set totalsalary = (select sum(salary) from employee
->                             where dno = current_dnum)
->         where dnumber = current_dnum;
->     until done
->     end repeat;
->
->     close dnumcur;
-> end$$
Query OK, 0 rows affected (0.00 sec)
```

Use cursor to iterate the rows

```
mysql> delimiter ;
```

EXAMPLE USING CURSORS – PART 3

- Call procedure

```
mysql> select * from deptsal;
```

dnumber	totalsalary
1	0
2	0
3	0

```
3 rows in set (0.01 sec)
```

```
mysql> call updateSalary;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> select * from deptsal;
```

dnumber	totalsalary
1	100000
2	50000
3	130000

```
3 rows in set (0.00 sec)
```



ANOTHER EXAMPLE

- Create a procedure to give a raise to all employees

```
mysql> select * from emp;
```

id	name	superid	salary	bdate	dno
1	john	3	100000	1960-01-01	1
2	mary	3	50000	1964-12-01	3
3	bob	NULL	80000	1974-02-07	3
4	tom	1	50000	1978-01-17	2
5	bill	NULL	NULL	1985-01-20	1
6	lucy	NULL	90000	1981-01-01	1
7	george	NULL	45000	1971-11-11	NULL

```
7 rows in set (0.00 sec)
```



ANOTHER EXAMPLE – PART 2

```
mysql> delimiter |
mysql> create procedure giveRaise (in amount double)
-> begin
->     declare done int default 0;
->     declare eid int;
->     declare sal int;
->     declare emprec cursor for select id, salary from employee;
->     declare continue handler for not found set done = 1;
->
->     open emprec;
->     repeat
->         fetch emprec into eid, sal;
->         update employee
->             set salary = sal + round(sal * amount)
->             where id = eid;
->     until done
->     end repeat;
-> end |
Query OK, 0 rows affected (0.00 sec)
```



ANOTHER EXAMPLE – PART 3

```
mysql> delimiter ;
```

```
mysql> call giveRaise(0.1);
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> select * from employee;
```

id	name	superid	salary	bdate	dno
1	john	3	110000	1960-01-01	1
2	mary	3	55000	1964-12-01	3
3	bob	NULL	88000	1974-02-07	3
4	tom	1	55000	1978-01-17	2
5	bill	NULL	NULL	1985-01-20	1

```
5 rows in set (0.00 sec)
```



FUNCTIONS

- Your user-defined functions can act just like a function defined in the database.
- They take arguments and return a single output.
- The general syntax is: create function <name> (<arg1> <type1>, [<arg2> <type2> [...]]) returns <return type> [deterministic]
 - Deterministic means that the output from the function is strictly a consequence of the arguments.
 - Same values input → same values output.
 - Like a static method in Java.
 - Note that the arguments cannot be changed and the new values passed back to the caller.
- Follow that with begin ... end and you have a function.



FUNCTIONS

- You need ADMIN privilege to create functions on mysql-user server
- Functions are declared using the following syntax:

```
function <function-name> (param_spec1, ..., param_speck)  
    returns <return_type>  
    [not] deterministic allow optimization if same output  
    for the same input (use RAND not deterministic )
```

Begin

-- execution code

end;



STORED FUNCTIONS

- Stored functions differ from stored procedures in that stored functions actually **return a value**.
- Stored functions have **only input parameters** (if any parameters at all), so the IN , OUT , and INOUT keywords aren't used.
- Stored functions have **no output parameters**; instead, you use a RETURN statement to return a value whose type is determined by the RETURNS type statement, which precedes the body of the function.



CREATING STORED FUNCTION

- Use SQL tab
- Or use PhPMyAdmin Routines Tab → Add routine

Create new routine

Details ⓘ

Routine name

Type **PROCEDURE**

Direction Name Type

Parameters IN INT

Add parameter

1

Definition

Is deterministic ☐

Definer

Security type **DEFINER**

SQL data access **CONTAINS SQL**

Comment

Go Close

Select function

SYNTAX OF A STORED FUNCTION

DELIMITER //

CREATE FUNCTION

FunctionName([parameters]) RETURNS
DATATYPE

DETERMINISTIC

SQL SECURITY DEFINER

COMMENT 'A function'

BEGIN

SQL Statements

Return Variable;

END //

DELIMITER ;



OPTIONAL CHARACTERISTICS

- Type: Procedure/Function
- Deterministic : If the procedure always returns the same results, given the same input. The default value is NOT DETERMINISTIC.
- SQL Security : At call time, check privileges of the user.
 - INVOKER is the user who calls the procedure.
 - DEFINER is the creator of the procedure. The default value is DEFINER.
- Comment : For documentation purposes; the default value is ""



EXAMPLE STORED FUNCTION

```
DELIMITER //  
CREATE FUNCTION hello (s CHAR(20))  
RETURNS CHAR(50) DETERMINISTIC  
BEGIN  
RETURN CONCAT('Hello, ',s,'!');  
END //  
DELIMITER ;
```

- Click SQL tab and write:
- **SELECT** hello('world');

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



EXAMPLE STORED FUNCTION

```
DELIMITER //
DROP FUNCTION IF EXISTS empTax;
CREATE FUNCTION empTax(Salary Decimal(10,2)) RETURNS
    Decimal(10,2)
BEGIN
    Declare tax decimal(10,2);
    if salary < 3000.00 then
        set tax=salary*0.1;
    elseif Salary <5000.00 then
        set tax=Salary*0.2;
    else
        set tax=Salary*0.3;
    end if;
    return tax;
END
//
DELIMITER;
```



CALLING THE STORED FUNCTION WITHIN AN SQL QUERY

```
select sname, emptax(Salary) as tax from salesreps;
```

```
+-----+-----+
```

```
| sname | tax   |
```

```
+-----+-----+
```


```
| PETER | 1950.00 |
```

```
| PAUL  | 2160.00 |
```

```
| MARY  | 2250.00 |
```

```
+-----+-----+
```

```
3 rows in set (0.00 sec)
```



Salary column pass to the emptax function as a parameter and the function returns the tax value for each person



A FUNCTION WITH REPEAT UNTIL LOOP

```
DELIMITER //
CREATE FUNCTION CalcIncome ( starting_value INT )
RETURNS INT
BEGIN
    DECLARE income INT;
    SET income = 0;
    label1: REPEAT
        SET income = income + starting_value;
        UNTIL income >= 4000
    END REPEAT label1;
    RETURN income;
END; //
DELIMITER ;
```



ANOTHER EXAMPLE OF FUNCTIONS

```
mysql> select * from employee;
```

id	name	superid	salary	bdate	dno
1	john	3	100000	1960-01-01	1
2	mary	3	50000	1964-12-01	3
3	bob	NULL	80000	1974-02-07	3
4	tom	1	50000	1970-01-17	2
5	bill	NULL	NULL	1985-01-20	1

```
5 rows in set (0.00 sec)
```

```
mysql> delimiter ;
```

```
mysql> create function giveRaise (oldval double, amount double
```

```
-> returns double
```

```
-> deterministic
```

```
-> begin
```

```
->     declare newval double;
```

```
->     set newval = oldval * (1 + amount);
```

```
->     return newval;
```

```
-> end ;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> delimiter ;
```


ANOTHER EXAMPLE OF FUNCTIONS

```
mysql> select name, salary, giveRaise(salary, 0.1) as newsal  
-> from employee;
```

name	salary	newsal
john	100000	110000
mary	50000	55000
bob	80000	88000
tom	50000	55000
bill	NULL	NULL

```
5 rows in set (0.00 sec)
```



MORE FUNCTION EXAMPLES

MySQL stored function – Example 1

MySQL> Delimiter //

To set the delimiter

MySQL> Create function Add1(a int, b int) returns int

Begin

Declare int;

Set c= a+b;

Return c;

End;

//

MySQL> Select Add1(10,20) ;

To run the Function



MySQL stored function – Example 2

MySQL> Delimiter //

To set the delimiter

MySQL>Create function Max1 (n1 Int,n2 Int) Returns Varchar(10)
DETERMINISTIC

BEGIN

DECLARE Maximum1 varchar(10);

If n1>n2 then

Set Maximum1= 'Maximum number is n1' ;

Else

Set Maximum1= 'Maximum number is n2';

End if;

Return Maximum;

End;

//

MySQL> Select Max1(10,20) ;

To run the Function

MySQL> Delimiter // ← To set the delimiter

MySQL> Create function Rname(rno1 int) returns varchar(20)

Begin

Declare sname varchar(20);

Select name into sname from Stud where rno=rno1;

Return sname;

End;

//

MySQL> Select Rname(1) ; ← To run the Function

MySQL stored function – Example 4

MySQL> Delimiter //

To set the delimiter

MySQL> Create function Fgrade(rno1 int) returns varchar(20)

DETERMINISTIC

Begin

Declare grade varchar(20);

Declare mark1 int;

Select mark into mark1 from Stud where rno=rno1;

If (mark1 >75) then

Set grade= 'Distinction'

Elseif (mark1 >=60 and mark1 <75) then

Set grade='First Class'

Elseif (marks <60) then

Set grade='Pass Class'

End if;

Return grade;

End;

//

MySQL> Select Fgrade(1) ;

To run the Function



SQL TRIGGERS

- To monitor a database and take a corrective action when a condition occurs
 - Examples:
 - Charge \$10 overdraft fee if the balance of an account after a withdrawal transaction is less than \$500
 - Limit the salary increase of an employee to no more than 5% raise

```
CREATE TRIGGER trigger-name  
    trigger-time trigger-event  
    ON table-name  
    FOR EACH ROW  
        trigger-action;
```

trigger-time \in {BEFORE, AFTER}

trigger-event \in {INSERT,DELETE,UPDATE}



TRIGGERS

- Please see: <https://dev.mysql.com/doc/refman/8.0/en/create-trigger.html> for the complete specification for triggers.

CREATE

[DEFINER = { user | CURRENT_USER }]

TRIGGER trigger_name

trigger_time trigger_event

ON tbl_name FOR EACH ROW

[trigger_order]

trigger_body

trigger_time: { BEFORE | AFTER }

trigger_event: { INSERT | UPDATE | DELETE }

trigger_order: { FOLLOWS | PRECEDES }

other_trigger_name



SQL TRIGGERS: AN EXAMPLE

- We want to create a trigger to update the total salary of a department when a new employee is hired

```
mysql> select * from employee;
```

id	name	superid	salary	bdate	dno
1	john	3	100000	1960-01-01	1
2	mary	3	50000	1964-12-01	3
3	bob	NULL	80000	1974-02-07	3
4	tom	1	50000	1970-01-17	2
5	bill	NULL	NULL	1985-01-20	1

```
5 rows in set (0.00 sec)
```

```
mysql> select * from deptsal;
```

dnumber	totalsalary
1	100000
2	50000
3	130000

```
3 rows in set (0.00 sec)
```


SQL TRIGGERS: AN EXAMPLE – PART 1

- Create a trigger to update the total salary of a department when a new employee is hired:

```
mysql> delimiter ;
mysql> create trigger update_salary
-> after insert on employee
-> for each row
-> begin
->     if new.dno is not null then
->         update deptsal
->         set totalsalary = totalsalary + new.salary
->         where dnumber = new.dno;
->     end if;
-> end ;
Query OK, 0 rows affected (0.06 sec)
mysql> delimiter ;
```

- The keyword “new” refers to the new row inserted



SQL TRIGGERS: AN EXAMPLE – PART 2

```
mysql> select * from deptsal;
```

dnumber	totalsalary
1	100000
2	50000
3	130000

```
3 rows in set (0.00 sec)
```

```
mysql> insert into employee values (6,'lucy',null,90000,'1981-01-01',1);  
Query OK, 1 row affected (0.08 sec)
```

```
mysql> select * from deptsal;
```

dnumber	totalsalary
1	190000
2	50000
3	130000

```
3 rows in set (0.00 sec)
```

← Totalsalary of department 1 increases by 90000

```
mysql> insert into employee values (7,'george',null,45000,'1971-11-11',null);  
Query OK, 1 row affected (0.02 sec)
```

```
mysql> select * from deptsal;
```

dnumber	totalsalary
1	190000
2	50000
3	130000

```
3 rows in set (0.00 sec)
```

totalsalary did not change, why?

```
mysql> drop trigger update_salary;  
Query OK, 0 rows affected (0.00 sec)
```

To remove the trigger



SQL TRIGGERS: ANOTHER EXAMPLE – PART 3

- A trigger to update the total salary of a department when an employee tuple is modified:

```
mysql> delimiter ;
mysql> create trigger update_salary2
-> after update on employee
-> for each row
-> begin
->     if old.dno is not null then
->         update deptsal
->         set totalsalary = totalsalary - old.salary
->         where dnumber = old.dno;
->     end if;
->     if new.dno is not null then
->         update deptsal
->         set totalsalary = totalsalary + new.salary
->         where dnumber = new.dno;
->     end if;
-> end ;
Query OK, 0 rows affected (0.06 sec)
```

SQL TRIGGERS: AN EXAMPLE – PART 4

```
mysql> delimiter ;
mysql> select * from employee;
```

id	name	superid	salary	bdate	dno
1	john	3	100000	1960-01-01	1
2	mary	3	50000	1964-12-01	3
3	bob	NULL	80000	1974-02-07	3
4	tom	1	50000	1970-01-17	2
5	bill	NULL	NULL	1985-01-20	1
6	lucy	NULL	90000	1981-01-01	1
7	george	NULL	45000	1971-11-11	NULL

```
7 rows in set (0.00 sec)
```

```
mysql> select * from deptsal;
```

dnumber	totalsalary
1	190000
2	50000
3	130000

```
3 rows in set (0.00 sec)
```

```
mysql> update employee set salary = 100000 where id = 6;
```

```
Query OK, 1 row affected (0.03 sec)
```

```
Rows matched: 1  Changed: 1  Warnings: 0
```

```
mysql> select * from deptsal;
```

dnumber	totalsalary
1	200000
2	50000
3	130000

```
3 rows in set (0.00 sec)
```

SQL TRIGGERS: ANOTHER EXAMPLE – PART 5

- A trigger to update the total salary of a department when an employee tuple is deleted:

```
mysql> delimiter !
mysql> create trigger update_salary3
-> before delete on employee
-> for each row
-> begin
->         if (old.dno is not null) then
->             update deptsal
->             set totalsalary = totalsalary - old.salary
->             where dnumber = old.dno;
->         end if;
-> end !
Query OK, 0 rows affected (0.08 sec)
mysql> delimiter ;
```

SQL TRIGGERS: ANOTHER EXAMPLE – PART 6

```
mysql> select * from employee;
```

id	name	superid	salary	bdate	dno
1	john	3	100000	1960-01-01	1
2	mary	3	50000	1964-12-01	3
3	bob	NULL	80000	1974-02-07	3
4	tom	1	50000	1970-01-17	2
5	bill	NULL	NULL	1985-01-20	1
6	lucy	NULL	100000	1981-01-01	1
7	george	NULL	45000	1971-11-11	NULL

```
7 rows in set (0.00 sec)
```

```
mysql> select * from deptsal;
```

dnumber	totalsalary
1	200000
2	50000
3	130000

```
3 rows in set (0.00 sec)
```

```
mysql> delete from employee where id = 6;  
Query OK, 1 row affected (0.02 sec)
```

```
mysql> delete from employee where id = 7;  
Query OK, 1 row affected (0.03 sec)
```

```
mysql> select * from deptsal;
```

dnumber	totalsalary
1	100000
2	50000
3	130000

```
3 rows in set (0.00 sec)
```

A FEW THINGS TO NOTE

- A given trigger can only have one event.
- If you have the same or similar processing that has to go on during insert and delete, then it's best to have that in a procedure or function and then call it from the trigger.
- A good naming standard for a trigger is `<table_name>_event` if you have the room for that in the name.
- Just like a function or a procedure, the trigger body will need a `begin ... end` unless it is a single statement trigger.



THE SPECIAL POWERS OF A TRIGGER

- While in the body of a trigger, there are potentially two sets of column values available to you, with special syntax for denoting them.
 - old.<column name> will give you the value of the column before the DML statement executed.
 - new.<column name> will give you the value of that column **after** the DML statement executed.
- Insert triggers have no old values available, and delete triggers have no new values available for obvious reasons. **Only update triggers have both the old and the new values available.**
- Only triggers can access these values this way.



CHANGING COLUMNS IN A TRIGGER

- In the body of a trigger, it is possible to change the values for the columns in the current row.
- Just use the “set” verb to change them.
- You can only do this for an update or insert trigger.
- You can only change the values of new.<column name> since there is no point to changing the old values.



MORE EXAMPLES

- Simplified example of a parent table: hospital_room as the parent and hospital_bed as the child.
- The room has a column: max_beds that dictates the maximum number of beds for that room.
- The hospital_bed table has a before insert trigger that checks to make sure that the hospital room does not already have its allotted number of beds.



THE TRIGGER

```
CREATE DEFINER=`root`@`localhost`  
TRIGGER `programming`.`hospital_bed_BEFORE_INSERT`  
BEFORE INSERT ON `hospital_bed` FOR EACH ROW  
BEGIN  
    declare max_beds_per_room int;  
    declare current_count int;  
    select    max_beds into max_beds_per_room  
    from      hospital_room  
    where     hospital_room_no = new.room_id;  
    select    count(*) into current_count  
    from      hospital_bed  
    where     room_id = new.room_id;  
    if current_count >= max_beds_per_room then  
        signal sqlstate '45000' set message_text='Too many beds in that  
room already!';  
    end if;  
END;
```



FIRING THE TRIGGER

```
insert into hospital_bed (room_id, hospital_bed_id)
values ('323B', 1);
```

```
insert into hospital_bed (room_id, hospital_bed_id)
values ('323B', 2);
```

```
insert into hospital_bed (room_id, hospital_bed_id)
values ('323B', 3);
```

```
insert into hospital_bed (room_id, hospital_bed_id)
values ('323B', 4);
```

```
insert into hospital_bed (room_id, hospital_bed_id)
values ('323B', 5);
```

Error Code: 1644. Too many beds in that room already!



USING A STORED PROCEDURE INSTEAD

```
CREATE DEFINER=`root`@`localhost` PROCEDURE `too_many_beds`(in room_id
varchar(45))
BEGIN
    declare max_beds_per_room int;
    declare current_count int;
    declare room_count int;
    -- see if the hospital room exists
    select      count(*) into room_count
    from        hospital_room
    where       hospital_room_no = room_id;
    if room_count = 1 then -- we can see if room for 1 more bed
    begin
        select      max_beds into max_beds_per_room
        from        hospital_room
        where       hospital_room_no = room_id;
        -- count the beds in this room
        select      count(*) into current_count
        from        hospital_bed
        where       room_id = room_id;
        if current_count >= max_beds_per_room then
            -- flag an error to abort if necessary
            signal sqlstate '45000' set message_text='Too many beds in
that room already!';
        end if;
    end;
    end if;
END
```



COMMENTS ON THE PROCEDURE

- Because that is in isolation from the beds table, we have to check to make sure that the room number is viable.
- As a stored procedure, this can be called directly from the command line as a means of unit testing.
- I'm still not too sure how exacting the typing of the parameters has to be. For instance, does that one argument have to be exactly a varchar(45) in order for it to work, or not?



VIEWING YOUR TRIGGERS

- MySQL has a schema that has tables for all of the information that is needed to define and run the data in the database. This is meta data.
- `select * from information_schema.triggers where trigger_schema='<your schema name>';` -- retrieve the trigger information for the triggers in <your schema name>.
- Alternatively, you can use the “show triggers” command (this is not SQL) that will display a report of your triggers from the default schema.

```
mysql> show triggers;
```



VIEWING YOUR TRIGGERS (CONTINUED)

- If you're using MySQL Workbench, the IDE provides access to your triggers:
 - In the navigator pane, right click the table that has the trigger.
 - Select "Alter Table"
 - This will open up a rather lavish dialog which has tabs down near the bottom. One of those tabs is "Triggers". Select that.
 - That will open up **another** dialog, and over to the left will be the list of events that you can define triggers for.
 - At this point, you can right click one of those events and it will pop up a menu that will give you the option to create a **new** trigger for that event.
 - Or you can double click an existing trigger to get into an editor on that particular trigger. This will allow you to update the trigger in place as it were, rather than drop and recreate it.



DYNAMIC SQL

- Sometimes you need to operate against a table or columns that are not known at compile time. MySQL has a process using set, prepare, execute, and deallocate.

```
CREATE DEFINER='root'@'localhost' PROCEDURE `dynamic`(in  
tableName varchar(40))  
begin  
    set @statement = concat('select * from ', tableName);  
    prepare stmt from @statement;  
    execute stmt;  
    set @statement = concat('select count(*) from ', tableName, ' into  
@count');  
    prepare stmt from @statement;  
    execute stmt;  
    select concat('Count was: ', @count, ' from table: ', tableName);  
    deallocate prepare stmt;  
end
```



DYNAMIC SQL (CONTINUED)

- The @ in front of a name makes it a user variable, which is shared between the command session and the stored procedure.
- concat will take any number of arguments.
- Just like the Java API, you can have bind variables in the SQL that you submit, then use the using clause in the execute statement.
 - The bind variables have to map one for one to the variables in the using clause: execute stmt using @var1, @var2, ...



CREDITS

Presentation taken from:

- www.cse.msu.edu/~pramanik/teaching/courses/cse480/14s/lectures/12/lecture13.ppt by Sakti Pramanik at Michigan State University
- MySQL Procedural Language by David Brown at California State University Long Beach



MORE TRIGGER EXAMPLES

- <https://phoenixnap.com/kb/mysql-trigger>

