DROP TRIGGER

In MySQL Trigger can also be drop. When Trigger drops, then it is removed from the database.

Syntax

Drop Trigger[ IF EXISTS ] Trigger\_name;

Parameter:

Trigger\_name: Name of the Trigger to be dropped

Example 1

drop Trigger student\_update;

CREATE TRIGGER 'student\_delete' AFTER DELETE ON 'student' FOR EACH ROW DELETE FROM student2 WHERE student.id=student1.id;

In MySQL, AFTER/BEFORE DELETE trigger can also be created. AFTER/BEFORE DELETE trigger means trigger will invoke after/before the record is deleted.

Syntax

CREATE TRIGGER trigger\_name

AFTER/BEFORE DELETE

ON table\_name FOR EACH ROW

BEGIN

--variable declarations

--trigger code

END;

Parameter:

trigger\_name: name of the trigger to be created.

AFTER/BEFORE DELETE: It points the trigger after/before delete query is executed.

table\_name: name of the table in which a trigger is created.

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

);

DELIMITER $$

CREATE TRIGGER before\_employee\_update

BEFORE UPDATE ON employees

FOR EACH ROW

BEGIN

INSERT INTO employees\_audit

SET action = 'update',

employeeNumber = OLD.employeeNumber,

lastname = OLD.lastname,

changedat = NOW();

END$$

DELIMITER ;

CREATE TABLE test1(a1 INT);

CREATE TABLE test2(a2 INT);

CREATE TABLE test3(a3 INT NOT NULL AUTO\_INCREMENT PRIMARY KEY);

CREATE TABLE test4(

a4 INT NOT NULL AUTO\_INCREMENT PRIMARY KEY,

b4 INT DEFAULT 0

);

delimiter |

CREATE TRIGGER testref BEFORE INSERT ON test1

FOR EACH ROW

BEGIN

INSERT INTO test2 SET a2 = NEW.a1;

DELETE FROM test3 WHERE a3 = NEW.a1;

UPDATE test4 SET b4 = b4 + 1 WHERE a4 = NEW.a1;

END;

|

delimiter ;

INSERT INTO test3 (a3) VALUES

(NULL), (NULL), (NULL), (NULL), (NULL),

(NULL), (NULL), (NULL), (NULL), (NULL);

INSERT INTO test4 (a4) VALUES

(0), (0), (0), (0), (0), (0), (0), (0), (0), (0);

mysql> INSERT INTO test1 VALUES

(1), (3), (1), (7), (1), (8), (4), (4);

Query OK, 8 rows affected (0.01 sec)

Records: 8 Duplicates: 0 Warnings: 0

SET @sum=0;

insert into accounts values(1,10000.23);

insert into accounts values(2,20000);

select @sum;

CREATE TRIGGER trg\_insert BEFORE INSERT on accounts

for EACH ROW

SET @sum=@sum+NEW.amount;

CREATE TABLE accounts(account\_nun int,amount decimal(10,2));

------Creating Stored Procedures in MySQL------

--Make sure you have version 5 of MySQL:

SELECT VERSION();

+-----------+

| VERSION() |

+-----------+

| 5.0.15-nt |

+-----------+

1 row in set (0.00 sec)

--First pick a database to use (a procedure, like a table, is associated with

--a single database.) For these examples, I will use a database that is populated

--with the tables from HW 2:

USE ozaidan\_hw2;

--Next, change the delimiter, because we will use the semicolon WITHIN the

--procedure declarations, and therefore it cannot be the delimiter anymore:

DELIMITER //

--OK, let's get started. Creating procedures is straightforward:

CREATE PROCEDURE myFirstProc()

SELECT 'Hello World!' AS Output;

//

Query OK, 0 rows affected (0.00 sec)

--Whenever you create a procedure (successfully) you should get a 'Query OK' message.

--Calling a procedure is also straightforward:

CALL myFirstProc() //

+--------------+

| Output |

+--------------+

| Hello World! |

+--------------+

1 row in set (0.00 sec)

--By the way, procedure names are NOT case sensitive:

CALL myfirstproc() //

+--------------+

| Output |

+--------------+

| Hello World! |

+--------------+

1 row in set (0.00 sec)

--Another example:

CREATE PROCEDURE ListStudents()

SELECT \*

FROM Student;

//

CALL ListStudents() //

+-------+----------+---------+------+------+-------+---------+-----------+

| StuID | LName | Fname | Age | Sex | Major | Advisor | city\_code |

+-------+----------+---------+------+------+-------+---------+-----------+

| 1001 | Smith | Linda | 18 | F | 600 | 1121 | BAL |

| 1002 | Kim | Tracy | 19 | F | 600 | 7712 | HKG |

.

.

.

| 1034 | Epp | Eric | 18 | M | 50 | 5718 | BOS |

| 1035 | Schmidt | Sarah | 26 | F | 50 | 5718 | WAS |

+-------+----------+---------+------+------+-------+---------+-----------+

34 rows in set (0.00 sec)

--Say we only want student ID's and names. To update a procedure, we must

--first DROP it:

DROP PROCEDURE IF EXISTS ListStudents //

Query OK, 0 rows affected (0.00 sec)

--Again, whenever you drop a procedure, you should get a 'Query OK' message.

--From now on, we will always use "DROP PROCEDURE IF EXISTS procName" as

--a standard practice before declaring procedures:

DROP PROCEDURE IF EXISTS ListStudents //

CREATE PROCEDURE ListStudents()

SELECT StuID, LName, FName

FROM Student;

//

CALL ListStudents() //

+-------+----------+---------+

| StuID | LName | FName |

+-------+----------+---------+

| 1001 | Smith | Linda |

| 1002 | Kim | Tracy |

.

.

.

| 1034 | Epp | Eric |

| 1035 | Schmidt | Sarah |

+-------+----------+---------+

34 rows in set (0.00 sec)

--OK, let's use some parameters:

DROP PROCEDURE IF EXISTS sayHello //

CREATE PROCEDURE sayHello(IN name VARCHAR(20))

SELECT CONCAT('Hello ', name, '!') AS Greeting;

//

--The 'IN' keyword tells MySQL that is should be expecting an input value for

--the parameter......hunh? Why would a parameter NOT have an input value? You will

--see in a little bit. First, let's see if sayHello works:

CALL sayHello('Omar') //

+-------------+

| Greeting |

+-------------+

| Hello Omar! |

+-------------+

1 row in set (0.00 sec)

--Another example:

DROP PROCEDURE IF EXISTS saySomething //

CREATE PROCEDURE saySomething(IN phrase VARCHAR(20), IN name VARCHAR(20))

SELECT CONCAT(phrase, ' ', name, '!') AS Output;

//

CALL saySomething('Go','Blue Jays') //

CALL saySomething('Do','my homework') //

+---------------+

| Output |

+---------------+

| Go Blue Jays! |

+---------------+

1 row in set (0.00 sec)

+-----------------+

| Output |

+-----------------+

| Do my homework! |

+-----------------+

1 row in set (0.00 sec)

--and another one:

DROP PROCEDURE IF EXISTS FindStudent //

CREATE PROCEDURE FindStudent(IN id INT)

SELECT StuID, CONCAT(FName, ' ', LName) AS 'Student Name'

FROM Student

WHERE StuID = id;

//

CALL FindStudent(1001) //

+-------+--------------+

| StuID | Student Name |

+-------+--------------+

| 1001 | Linda Smith |

+-------+--------------+

1 row in set (0.00 sec)

--and yet another:

DROP PROCEDURE IF EXISTS calculate //

CREATE PROCEDURE calculate(IN x INT, IN y INT, OUT sum INT, OUT product INT)

SET sum = x + y;

SET product = x \* y;

//

ERROR 1064 (42000): You have an error in your SQL syntax; check the manual ...

--Well, that wasn't good. The reason is, we must use BEGIN/END if we have

--a compound statement:

DROP PROCEDURE IF EXISTS calculate //

CREATE PROCEDURE calculate(IN x INT, IN y INT, OUT sum INT, OUT product INT)

BEGIN

SET sum = x + y;

SET product = x \* y;

END;

//

--Did you notice the 'OUT' keyword for sum and product? This tells MySQL that those

--two parameters are not 'input' parameters but are 'output' parameters instead.

--Now, when calling the procedure, we need to provide four parameters: two input

--values, and two MySQL \*variables\* where the results will be stored:

CALL calculate(4,5,@s,@p) //

Query OK, 0 rows affected (0.00 sec)

--Here, @s and @p are MySQL variables. Notice that they start with @, although

--procedure \*parameters\* do not start with @

SELECT @s //

SELECT @p //

+------+

| @s |

+------+

| 9 |

+------+

1 row in set (0.00 sec)

+------+

| @p |

+------+

| 20 |

+------+

1 row in set (0.00 sec)

--Note: you can also have INOUT parameters, which serve as both input and output

--parameters.

--OK, let's do some interesting stuff. First off, flow control:

DROP PROCEDURE IF EXISTS mySign //

CREATE PROCEDURE mySign(IN x INT)

BEGIN

IF x > 0 THEN

SELECT x AS Number, '+' AS Sign;

ELSEIF x < 0 THEN

SELECT x AS Number, '-' AS Sign;

ELSE

SELECT x AS Number, 'Zero' AS Sign;

END IF;

END;

//

CALL mySign(2) //

CALL mySign(-5) //

CALL mySign(0) //

+--------+------+

| Number | Sign |

+--------+------+

| 2 | + |

+--------+------+

1 row in set (0.00 sec)

+--------+------+

| Number | Sign |

+--------+------+

| -5 | - |

+--------+------+

1 row in set (0.00 sec)

+--------+------+

| Number | Sign |

+--------+------+

| 0 | Zero |

+--------+------+

1 row in set (0.00 sec)

--Before we get any further, let's introduce variables:

DROP PROCEDURE IF EXISTS mySign //

CREATE PROCEDURE mySign(IN x INT)

BEGIN

DECLARE result VARCHAR(20);

IF x > 0 THEN

SET result = '+';

ELSEIF x < 0 THEN

SET result = '-';

ELSE

SET result = 'Zero';

END IF;

SELECT x AS Number, result AS Sign;

END;

//

CALL mySign(2) //

CALL mySign(-5) //

CALL mySign(0) //

+--------+------+

| Number | Sign |

+--------+------+

| 2 | + |

+--------+------+

1 row in set (0.00 sec)

+--------+------+

| Number | Sign |

+--------+------+

| -5 | - |

+--------+------+

1 row in set (0.00 sec)

+--------+------+

| Number | Sign |

+--------+------+

| 0 | Zero |

+--------+------+

1 row in set (0.00 sec)

--Using CASE:

DROP PROCEDURE IF EXISTS digitName //

CREATE PROCEDURE digitName(IN x INT)

BEGIN

DECLARE result VARCHAR(20);

CASE x

WHEN 0 THEN SET result = 'Zero';

WHEN 1 THEN SET result = 'One';

WHEN 2 THEN SET result = 'Two';

WHEN 3 THEN SET result = 'Three';

WHEN 4 THEN SET result = 'Four';

WHEN 5 THEN SET result = 'Five';

WHEN 6 THEN SET result = 'Six';

WHEN 7 THEN SET result = 'Seven';

WHEN 8 THEN SET result = 'Eight';

WHEN 9 THEN SET result = 'Nine';

ELSE SET result = 'Not a digit';

END CASE;

SELECT x AS Digit, result AS Name;

END;

//

CALL digitName(0) //

CALL digitName(4) //

CALL digitName(100) //

+-------+------+

| Digit | Name |

+-------+------+

| 0 | Zero |

+-------+------+

1 row in set (0.00 sec)

+-------+------+

| Digit | Name |

+-------+------+

| 4 | Four |

+-------+------+

1 row in set (0.00 sec)

+-------+-------------+

| Digit | Name |

+-------+-------------+

| 100 | Not a digit |

+-------+-------------+

1 row in set (0.00 sec)

--As you'd expect, we have loops. For example, WHILE loops:

DROP PROCEDURE IF EXISTS fact //

CREATE PROCEDURE fact(IN x INT)

BEGIN

DECLARE result INT;

DECLARE i INT;

SET result = 1;

SET i = 1;

WHILE i <= x DO

SET result = result \* i;

SET i = i + 1;

END WHILE;

SELECT x AS Number, result as Factorial;

END;

//

CALL fact(1) //

CALL fact(2) //

CALL fact(4) //

CALL fact(0) //

+--------+-----------+

| Number | Factorial |

+--------+-----------+

| 1 | 1 |

+--------+-----------+

1 row in set (0.00 sec)

+--------+-----------+

| Number | Factorial |

+--------+-----------+

| 2 | 2 |

+--------+-----------+

1 row in set (0.00 sec)

+--------+-----------+

| Number | Factorial |

+--------+-----------+

| 4 | 24 |

+--------+-----------+

1 row in set (0.01 sec)

+--------+-----------+

| Number | Factorial |

+--------+-----------+

| 0 | 1 |

+--------+-----------+

1 row in set (0.00 sec)

--There is also REPEAT/UNTIL loops:

DROP PROCEDURE IF EXISTS fact //

CREATE PROCEDURE fact(IN x INT)

BEGIN

DECLARE result INT DEFAULT 1; /\* notice you can declare a variable\*/

DECLARE i INT DEFAULT 1; /\* and give it a value in one line \*/

REPEAT

SET result = result \* i;

SET i = i + 1;

UNTIL i > x

END REPEAT;

SELECT x AS Number, result as Factorial;

END;

//

CALL fact(1) //

CALL fact(2) //

CALL fact(4) //

CALL fact(0) //

+--------+-----------+

| Number | Factorial |

+--------+-----------+

| 1 | 1 |

+--------+-----------+

1 row in set (0.00 sec)

+--------+-----------+

| Number | Factorial |

+--------+-----------+

| 2 | 2 |

+--------+-----------+

1 row in set (0.00 sec)

+--------+-----------+

| Number | Factorial |

+--------+-----------+

| 4 | 24 |

+--------+-----------+

1 row in set (0.00 sec)

+--------+-----------+

| Number | Factorial |

+--------+-----------+

| 0 | 1 |

+--------+-----------+

1 row in set (0.00 sec)

--OK, do you remember this?

/\*

CREATE PROCEDURE FindStudent(IN id INT)

SELECT StuID, CONCAT(FName, ' ', LName) AS 'Student Name'

FROM Student

WHERE StuID = id;

//

\*/

--What if we only want to extract the name without printing it out?

--Obviously, we need some OUT parameters. Still, how do you extract

--information into those OUT parameters?

--

--Answer: something called a CURSOR:

DROP PROCEDURE IF EXISTS FindName //

CREATE PROCEDURE FindName(IN id INT, OUT fn VARCHAR(20), OUT ln VARCHAR(20))

BEGIN

DECLARE cur CURSOR FOR

SELECT FName, LName

FROM Student

WHERE StuID = id;

OPEN cur;

FETCH cur INTO fn, ln;

CLOSE cur;

END;

//

CALL FindName(1001,@f,@l) //

Query OK, 0 rows affected (0.00 sec)

--Remember that @f and @l are MySQL variables:

SELECT @f //

SELECT @l //

+-------+

| @f |

+-------+

| Linda |

+-------+

1 row in set (0.00 sec)

+-------+

| @l |

+-------+

| Smith |

+-------+

1 row in set (0.00 sec)

--What if we give an invalid student ID?

CALL FindName(0000,@f,@l) //

ERROR 1329 (02000): No data to FETCH

--MySQL complains, as expected. It would be nice to handle this more elegantly, however.

--We need an error HANDLER. Let's modify FindName:

DROP PROCEDURE IF EXISTS FindName //

CREATE PROCEDURE FindName(IN id INT, OUT fn VARCHAR(20), OUT ln VARCHAR(20))

BEGIN

DECLARE cur CURSOR FOR

SELECT FName, LName

FROM Student

WHERE StuID = id;

DECLARE EXIT HANDLER FOR NOT FOUND

SELECT 'Sorry; this ID was not found' AS 'Error Message';

OPEN cur;

FETCH cur INTO fn, ln;

CLOSE cur;

END;

//

CALL FindName(0000,@f,@l) //

+------------------------------+

| Error Message |

+------------------------------+

| Sorry; this ID was not found |

+------------------------------+

1 row in set (0.00 sec)

--Another use for handlers: multiple FETCH calls using a CONTINUE handler.

--

--In this case, we use a CONTINUE handler that, instead of exiting the procedure

--upon encountering a NOT FOUND error, simply sets a variable done = 1.

--

--Why would we do that? And how does that help us carry out multiple FETCH calls?

--

--Take a look at this procedure, which traverses all the entries of a table to

--find the maximum and minimum age:

DROP PROCEDURE IF EXISTS MaxMinAge //

CREATE PROCEDURE MaxMinAge(OUT maxAge INT, OUT minAge INT)

BEGIN

DECLARE currAge,maxSoFar,minSoFar,done INT;

DECLARE cur CURSOR FOR

SELECT Age

FROM Student;

DECLARE CONTINUE HANDLER FOR NOT FOUND

SET done = 1;

SET maxSoFar = 0;

SET minSoFar = 1000;

SET done = 0;

OPEN cur;

WHILE done = 0 DO

FETCH cur INTO currAge;

IF currAge > maxSoFar THEN

SET maxSoFar = currAge;

END IF;

IF currAge < minSoFar THEN

SET minSoFar = currAge;

END IF;

END WHILE;

CLOSE cur;

SET maxAge = maxSoFar;

SET minAge = minSoFar;

END;

//

CALL MaxMinAge(@max,@min) //

Query OK, 0 rows affected (0.00 sec)

SELECT @max //

SELECT @min //

+------+

| @max |

+------+

| 27 |

+------+

1 row in set (0.00 sec)

+------+

| @min |

+------+

| 16 |

+------+

1 row in set (0.00 sec)

--In summary, stored procedures in MySQL look like this:

DROP PROCEDURE IF EXISTS procName //

CREATE PROCEDURE procName(parameter list)

BEGIN

/\* variable declarations \*/

/\* CURSOR definitions \*/

/\* declaring handlers \*/

/\* procedure body...whatever you want it to do \*/

END;

//

--In more detail:

DROP PROCEDURE IF EXISTS procName //

CREATE PROCEDURE procName(IN/OUT/INOUT parName parType, ...)

BEGIN

/\* variable declarations \*/

DECLARE varName,... varType;

/\* e.g. DECLARE myName VARCHAR(20); DECLARE x,y,z INT; \*/

DECLARE varName varType DEFAULT value;

/\* e.g. DECLARE x INT DEFAULT 0; \*/

/\* CURSOR definitions \*/

DECLARE curName CURSOR FOR

SELECT ...

/\* e.g. DECLARE cur1 CURSOR FOR

SELECT FName, LName

FROM Student; \*/

/\* declaring handlers \*/

DECLARE EXIT/CONTINUE HANDLER FOR errorType/errorNumber

... action ...

/\* e.g. DECLARE EXIT HANDLER FOR NOT FOUND

SELECT 'Sorry; this ID was not found' AS 'Error Message'; \*/

/\* e.g. DECLARE CONTINUE HANDLER FOR NOT FOUND

SET done = 1; \*/

/\* procedure body...whatever you want it to do \*/

/\* IF statement \*/

IF cond1 THEN

action1

ELSEIF cond2 THEN

action2

ELSEIF cond3 THEN

action3

ELSE

elseaction

END IF;

/\* e.g. IF x > 0 THEN

SET result = '+';

ELSEIF x < 0 THEN

SET result = '-';

ELSE

SET result = 'Zero';

END IF; \*/

/\* CASE statement \*/

CASE varName

WHEN val1 THEN action1

WHEN val2 THEN action2

ELSE elseaction

END CASE;

/\* e.g. CASE position

WHEN 1 THEN SET result = 'Gold Medal';

WHEN 2 THEN SET result = 'Silver Medal';

WHEN 3 THEN SET result = 'Bronze Medal';

ELSE SET result = 'No Medal!';

END CASE; \*/

/\* WHILE loop \*/

WHILE cond DO

action1

action2

...

END WHILE

/\* e.g. WHILE i < 5 DO

SET result = result + i;

SET i = i + 1;

END WHILE; \*/

/\* REPEAT/UNTIL loop \*/

REPEAT

action1

action2

...

UNTIL cond

END REPEAT;

/\* e.g. REPEAT

SET result = result + i;

SET i = i + 1;

UNTIL i >= 5

END REPEAT; \*/

/\* using a CURSOR \*/

OPEN curName;

.

.

FETCH curName INTO var1, var2, ...;

.

.

CLOSE curName;

/\* e.g. Assume cur1 has id's, first names, and last names

let's find the name of the student whose StuID is x:

OPEN cur1;

SET found = 0;

WHILE found = 0 DO

FETCH cur1 INTO nextID, nextFName, nextLName;

IF nextID = x THEN

SET result = CONCAT(nextFName, ' ', nextLName);

SET found = 1;

END IF;

END WHILE;

CLOSE cur1; \*/

END;

//

CASE 4: A Stored Function that Accept No Parameters

DELIMITER |

CREATE FUNCTION sample\_fn\_no\_param ()

RETURNS INT

BEGIN

DECLARE count INT;

SELECT COUNT(\*) INTO count FROM emp;

RETURN count;

END

|

DELIMITER ;

Execute and Verify Commands

select sample\_fn\_no\_param ();

A FUNCTION is always returns a value using the return statement. A PROCEDURE may return one or more values through parameters or may not return at all.

b. Functions are normally used for computations where as procedures are normally used for executing business logic.

c. A Function returns 1 value only. Procedure can return multiple values (max 1024).

d. Stored procedure returns always integer value by default zero. Whereas function returns type could be scalar or table or table values

e. Stored procedure is precompiled execution plan where as functions are not.

f. A function can call directly by SQL statement like select func\_name from dual while procedure cannot.

g.Stored procedure has the security and reduces the network traffic and also we can call stored procedure in any no. of applications at a time.

h. A Function can be used in the SQL Queries while a procedure cannot be used in SQL queries .that cause a major

difference b/w function and procedures.

CREATE FUNCTION hello (s CHAR(20))

RETURNS CHAR(50) DETERMINISTIC

RETURN CONCAT('Hello, ',s,'!');

END

SELECT hello('world');

1. Deterministic functions :  
Deterministic functions always result in the same output every time they are called with a fixed set of input values and given the same condition of the database. For example, AVG() function always results the same result given the qualifications stated above.

2. Nondeterministic functions :  
Nondeterministic functions result in different output each time they are called with a fixed set of input values even if the database state that they access remains the same. For example, GETDATE() function, results the current date and time value, always a different value.

Non-deterministic means a value like Now() or Rand() determines results. Your query is deterministic.

DELIMITER //

CREATE FUNCTION CalcIncome ( starting\_value INT )

RETURNS INT

BEGIN

DECLARE income INT;

SET income = 0;

label1: WHILE income <= 3000 DO

SET income = income + starting\_value;

END WHILE label1;

RETURN income;

END; //

DELIMITER ;

You could then reference your new function as follows:

SELECT CalcIncome (1000);

In MySQL, a function is a stored program that you can pass parameters into and then return a value.

CREATE FUNCTION function\_name [ (parameter datatype [, parameter datatype]) ]

RETURNS return\_datatype

BEGIN

Declaration\_section

Executable\_section

END;

Parameter:

Function\_name: name of the function

Parameter: number of parameter. It can be one or more than one.

return\_datatype: return value datatype of the function

declaration\_section: all variables are declared.

executable\_section: code for the function is written here.

CREATE PROCEDURE fact(IN x INT)

BEGIN

DECLARE result INT;

DECLARE i INT;

SET result = 1;

SET i = 1;

WHILE i <= x DO

SET result = result \* i;

SET i = i + 1;

END WHILE;

SELECT x AS Number, result as Factorial;

END;

CALL fact(1) //

CALL fact(2) //

CALL fact(4) //

CALL fact(0) //

DROP PROCEDURE IF EXISTS fact //

CREATE PROCEDURE fact(IN x INT)

BEGIN

DECLARE result INT DEFAULT 1; /\* notice you can declare a variable\*/

DECLARE i INT DEFAULT 1; /\* and give it a value in one line \*/

REPEAT

SET result = result \* i;

SET i = i + 1;

UNTIL i > x

END REPEAT;

SELECT x AS Number, result as Factorial;

END;

//

CALL fact(1) //

CALL fact(2) //

CALL fact(4) //

CALL fact(0) //

CREATE PROCEDURE FindStudent(IN id INT)

SELECT StuID, CONCAT(FName, ' ', LName) AS 'Student Name'

FROM Student

WHERE StuID = id;

#CREATE TABLE Department(DepartmentID INT PRIMARY KEY,DepartmentName VARCHAR(30))

#CREATE cType DeptType AS TABLE(DeptId INT, DeptName VARCHAR(30));

CASE 3: A Stored Procedure that Accept Parameters, Return ResultSet

DELIMITER |

CREATE PROCEDURE sample\_sp\_with\_params\_resultset (IN empId INT UNSIGNED, OUT oldName VARCHAR(20), INOUT newName VARCHAR(20))

BEGIN

SELECT `first name` into oldName FROM emp where id = empId;

UPDATE emp SET `first name`= newName where id = empId;

select \* from emp;

END

|

DELIMITER ;

Execute and Verify Commands

set @inout='updatedHJKS';

CALL sample\_sp\_with\_params\_resultset (1,@out,@inout);

You can verify the values of OUT and INOUT parameters as:

select @out,@inout;

CASE 2: A Stored Procedure that Accept Parameters (IN, OUT, INOUT)

DELIMITER |

CREATE PROCEDURE sample\_sp\_with\_params (IN empId INT UNSIGNED, OUT oldName VARCHAR(20), INOUT newName VARCHAR(20))

BEGIN

SELECT `first name` into oldName FROM emp where id = empId;

UPDATE emp SET `first name`= newName where id = empId;

END

|

DELIMITER ;

Execute and Verify Commands

set @inout='updatedHJK';

CALL sample\_sp\_with\_params(1,@out,@inout);

select @out,@inout;

select \* from emp;

CASE 1: A Stored Procedure that Accept No Parameters

DELIMITER |

CREATE PROCEDURE sample\_sp\_no\_param ()

BEGIN

UPDATE emp SET `first name`= 'ChangedHJK' where id = 1;

END

|

DELIMITER ;

Execute and Verify Commands

CALL sample\_sp\_no\_param;

select \* from emp;

CREATE TABLE emp(`first name` VARCHAR(20), id INT PRIMARY KEY);

insert into emp values('HJK', 1);

insert into emp values('ABC', 2);

insert into emp values('DEF', 3);

Verify Using:

select \* from emp;

CASE 1: A Stored Procedure that Accept No Parameters

CASE 2: A Stored Procedure that Accept Parameters (IN, OUT, INOUT)

CASE 3: A Stored Procedure that Accept Parameters, Return ResultSet

CASE 4: A Stored Function that Accept No Parameters

CASE 5: A Stored Function that Accept Parameters

https://quizizz.com/join?gc=029726

create table Atable (aid int primary key);

alter table atable add name varchar(20);

insert into atable values (1,'one');

create table Btable (bid int primary key);

alter table btable add name varchar(20);

insert into btable values (11,'eleven');

create table ctable (aid int,bid int,foreign key(aid) references atable(aid),foreign key(bid) references btable(bid));

insert into ctable values(1,11);

select a.aid "A",b.bid "B" from atable a,btable b, ctable c where b.bid=c.bid and c.aid = a.aid;

select a.name,b.name from atable a,btable b, ctable c where b.bid=c.bid and c.aid = a.aid;

1] create table Atable (aid int primary key)

insert into atable values (1);

2] create table Btable (bid int primary key);

insert into btable values (11);

3] create table ctable (aid int,bid int,foreign key(aid) references atable(aid),foreign key(bid) references btable(bid));

4] insert into ctable values(1,11);

5] select a.aid "A",b.bid "B" from atable a,btable b, ctable c where b.bid=c.bid and c.aid = a.aid;

truncate ctable

truncate atable

truncate btable

alter table atable add name varchar(20);

alter table btable add name varchar(20);

insert into atable values (1,'one');

insert into btable values (11,'eleven');

insert into ctable values (1,11);

insert into ctable values (1,12);

select a.name,b.name from atable a,btable b, ctable c where b.bid=c.bid and c.aid = a.aid;

1] create database university;

2] use university;

3] create table students (s\_id INT(10) NOT NULL AUTO\_INCREMENT, s\_firstname VARCHAR(30) NOT NULL, s\_lastname VARCHAR(30) NOT NULL, s\_email VARCHAR(40), PRIMARY KEY (s\_id));

4] insert into students (s\_firstname,s\_lastname,s\_email) values ('Shankar', 'Bhat', 'shankar@example.com');

5] insert into students (s\_firstname,s\_lastname,s\_email) values ('Venkat', 'Rao', 'venkat@example.com');

6] insert into students (s\_firstname,s\_lastname,s\_email) values ('Mohan', 'Nair', 'mohan@example.com');

7] insert into students (s\_firstname,s\_lastname,s\_email) values ('Abhijeet', 'Patel', 'abhi@example.com');

8] select \* from students;

9] select \* from students where s\_id='3';

10] create table students (s\_firstname VARCHAR(30) NOT NULL, s\_lastname VARCHAR(30) NOT NULL, s\_email VARCHAR(40), s\_phone BIGINT(10) NOT NULL, PRIMARY KEY (s\_phone, s\_firstname));

11] insert into students (s\_firstname,s\_lastname,s\_email,s\_phone) values ('Shankar', 'Bhat', 'shankar@example.com', '7303075409');

12] insert into students (s\_firstname,s\_lastname,s\_email,s\_phone) values ('Venkat', 'Rao', 'venkat@example.com', '7404076894');

13] insert into students (s\_firstname,s\_lastname,s\_email,s\_phone) values ('Mohan', 'Nair', 'mohan@example.com', '7404076892');

14] insert into students (s\_firstname,s\_lastname,s\_email,s\_phone) values ('Abhijeet', 'Patel', 'abhi@example.com', '7404076991');

15] insert into students (s\_firstname,s\_lastname,s\_email,s\_phone) values ('Manoj', 'Nair', 'manoj@example.com', '7404076892');

16] select \* from students;

17] select \* from students where s\_firstname='Mohan' AND s\_phone='7404076892';

18] ERROR [ insert into students (s\_firstname,s\_lastname,s\_email,s\_phone) values ('Manoj', 'Pillai', 'manoj@example.com', '7404076892');]

19] ALTER TABLE students ADD PRIMARY KEY (s\_id);

20] ALTER TABLE students ADD CONSTRAINT pk\_students PRIMARY KEY (s\_phone,s\_firstname);

21] select \* from students;

22] create table courses (c\_id INT(10) NOT NULL AUTO\_INCREMENT, c\_name VARCHAR(30) NOT NULL, PRIMARY KEY (c\_id));

23] insert into courses (c\_name) values ('Computer Science');

24] insert into courses (c\_name) values ('Economics');

25] insert into courses (c\_name) values ('Arts');

26] insert into courses (c\_name) values ('Chemistry');

27] insert into courses (c\_name) values ('Astro Physics');

28] select \* from courses;

29] create table enrollment (e\_id INT(10) NOT NULL AUTO\_INCREMENT, e\_StudentID integer, e\_CourseID integer, e\_year YEAR, PRIMARY KEY (e\_id), FOREIGN KEY (e\_StudentID) REFERENCES students(s\_id), FOREIGN KEY (e\_CourseID) REFERENCES courses(c\_id));

30] insert into enrollment (e\_StudentID, e\_CourseID, e\_year) values (1, 3, 2016);

31] insert into enrollment (e\_StudentID, e\_CourseID, e\_year) values (2, 4, 2016);

32] insert into enrollment (e\_StudentID, e\_CourseID, e\_year) values (3, 2, 2016);

33] insert into enrollment (e\_StudentID, e\_CourseID, e\_year) values (4, 1, 2016);

34] ERROR [insert into enrollment (e\_StudentID, e\_CourseID, e\_year) values (5, 1, 2016);]

35] select \* from enrollment;

select d.departmentid,d.departmentname,e.name from department d left join employee e on d.departmentid=e.deptid;

https://www.kaggle.com/datasets/anushabellam/cars-cars-2

Renamed the github repository to .......

https://github.com/sbmadake/MySQLLearning

Syntax:

ALTER TABLE table\_name

RENAME TO new\_table\_name;

Example:

In this example, the table name cus\_tbl is renamed as cus\_table.

ALTER TABLE cus\_tbl

RENAME TO cus\_table;

Syntax:

ALTER TABLE table\_name

CHANGE COLUMN old\_name new\_name

column\_definition

[ FIRST | AFTER column\_name ]

Example:

In this example, we will change the column name "cus\_surname" to "cus\_title".

Use the following query to do this:

ALTER TABLE cus\_tbl

CHANGE COLUMN cus\_surname cus\_title

varchar(20) NOT NULL;

Syntax:

ALTER TABLE table\_name

DROP COLUMN column\_name;

Let's take an example to drop the column name "cus\_address" from the table "cus\_tbl".

Use the following query to do this:

ALTER TABLE cus\_tbl

DROP COLUMN cus\_address;

The MODIFY command is used to change the column definition of the table.

Syntax:

ALTER TABLE table\_name

MODIFY column\_name column\_definition

[ FIRST | AFTER column\_name ];

Example:

In this example, we modify the column cus\_surname to be a data type of varchar(50) and force the column to allow NULL values.

Use the following query to do this:

ALTER TABLE cus\_tbl

MODIFY cus\_surname varchar(50) NULL;

Syntax:

ALTER TABLE table\_name

ADD new\_column\_name column\_definition

[ FIRST | AFTER column\_name ],

ADD new\_column\_name column\_definition

[ FIRST | AFTER column\_name ],

...

;

Example:

In this example, we add two new columns "cus\_address", and cus\_salary in the existing table "cus\_tbl". cus\_address is added after cus\_surname column and cus\_salary is added after cus\_age column.

Use the following query to do this:

ALTER TABLE cus\_tbl

ADD cus\_address varchar(100) NOT NULL

AFTER cus\_surname,

ADD cus\_salary int(100) NOT NULL

AFTER cus\_age ;

Example:

In this example, we add a new column "cus\_age" in the existing table "cus\_tbl".

Use the following query to do this:

ALTER TABLE cus\_tbl

ADD cus\_age varchar(40) NOT NULL;

MySQL ALTER statement is used when you want to change the name of your table or any table field. It is also used to add or delete an existing column in a table.

The ALTER statement is always used with "ADD", "DROP" and "MODIFY" commands according to the situation.

1) ADD a column in the table

Syntax:

ALTER TABLE table\_name

ADD new\_column\_name column\_definition

[ FIRST | AFTER column\_name ];

Parameters

table\_name: It specifies the name of the table that you want to modify.

new\_column\_name: It specifies the name of the new column that you want to add to the table.

column\_definition: It specifies the data type and definition of the column (NULL or NOT NULL, etc).

FIRST | AFTER column\_name: It is optional. It tells MySQL where in the table to create the column. If this parameter is not specified, the new column will be added to the end of the table.

Books- It contains information about the books belongs to the library

Column Name Data Type Description

Book\_No Number(6) Book identification number

Book\_Name VarChar2(30) Name of the book

Author\_name Varchar2(30) Author of the book

Cost Number(7,2) Cost of the book

Category Char(10) Category like System , Fiction ,Database etc.

1) Insert data in Book table as follows:

Book\_No Book Name Author Cost Category

101 Let us C Denis Ritchie 450 System

102 Oracle – Complete Ref Loni 550 Database

103 Mastering SQL Loni 250 Database

104 PL SQL-Ref Scott Urman 750 Database

1] select all the records

2] select records whose cost between >=500 and <=700

3] Find record with book name staring with O letter

4] find records whose cost is < avg cost

5] create new table with same structure

4] create new table as copy of books

5] create new table with database as category

6] find record with max cost

7] find record with min cost

8] find name of the author who has published more than one book

create table shirts (name varchar(30),size enum('xsmall','small','medium','large'));

insert into shirts values('POLO','small');

insert into shirts values('corcs','large');

insert into shirts values('JP','asdf');

create table member (Member\_Id int(5) primary key,Member\_Name varchar(20),Member\_address varchar(10),Acc\_Open\_Date date,Membership\_type enum('Lifetime','Annual','Half Yearly','Quarterly'),Fees\_paid int(5),Max\_Books\_Allowed int(2),Penalty\_Amount double)

Before we proceed to explain the MySQL database system, let us revise a few definitions related to the database.

Database − A database is a collection of tables, with related data.

Table − A table is a matrix with data. A table in a database looks like a simple spreadsheet.

Column − One column (data element) contains data of one and the same kind, for example the column postcode.

Row − A row (= tuple, entry or record) is a group of related data, for example the data of one subscription.

Redundancy − Storing data twice, redundantly to make the system faster.

Primary Key − A primary key is unique. A key value can not occur twice in one table. With a key, you can only find one row.

Foreign Key − A foreign key is the linking pin between two tables.

Compound Key − A compound key (composite key) is a key that consists of multiple columns, because one column is not sufficiently unique.

Index − An index in a database resembles an index at the back of a book.

Referential Integrity − Referential Integrity makes sure that a foreign key value always points to an existing row.

What is a Database?

A database is a separate application that stores a collection of data. Each database has one or more distinct APIs for creating, accessing, managing, searching and replicating the data it holds.

Other kinds of data stores can also be used, such as files on the file system or large hash tables in memory but data fetching and writing would not be so fast and easy with those type of systems.

Nowadays, we use relational database management systems (RDBMS) to store and manage huge volume of data. This is called relational database because all the data is stored into different tables and relations are established using primary keys or other keys known as Foreign Keys.

MySQL is a very popular, open source database.

Officially pronounced “my Ess Que Ell” (not my sequel).

Handles very large databases; very fast performance.

Why are we using MySQL?

Free (much cheaper than Oracle!)

Each student can install MySQL locally.

Easy to use Shell for creating tables, querying tables, etc.

Easy to use with Java JDBC

MySQL is a relational database management system based on SQL – Structured Query Language. The application is used for a wide range of purposes, including data warehousing, e-commerce, and logging applications.

The most common use for mySQL however, is for the purpose of a web database. It can be used to store anything from a single record of information to an entire inventory of available products for an online store.