MySQLTUTORIAL

The Ultimate Guide To MySQL Roles By Examples

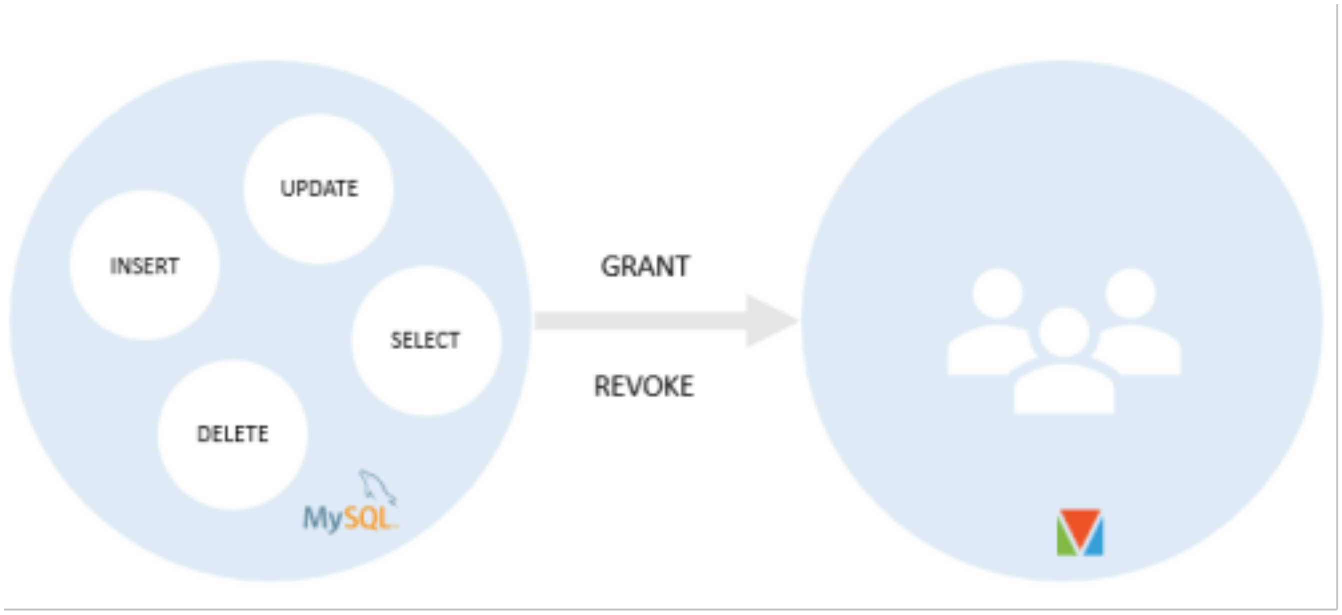
Summary: in this tutorial, you will learn how to use MySQL roles to simplify the privilege managements.

Introduction to MySQL roles

Typically, you have multiple users with the same set of privileges. Previously, the only way to grant (https://www.mysqltutorial.org/mysql-grant.aspx) and revoke (https://www.mysqltutorial.org/mysql-revoke.aspx) privileges to multiple users is to change the privileges of each user individually, which is time-consuming.

To make it easier, MySQL provided a new object called role. A role is a named collection of privileges.

Like user accounts, you can grant privileges to roles and revoke privileges from them.



If you want to grant the same set of privileges to multiple users, you follow these steps:

- First, create a new role.

- Second, grant privileges to the role.
- Third, grant the role to the users.

In case you want to change the privileges of the users, you need to change the privileges of the granted role only. The changes will take effect to all users to which the role granted.

MySQL role example

First, create a new database (https://www.mysqltutorial.org/mysql-create-table/) named CRM, which stands for customer relationship management.

```
CREATE DATABASE crm;
```

Next, use the `crm` database:

```
USE crm;
```

Then, create `customer` table inside the `CRM` database.

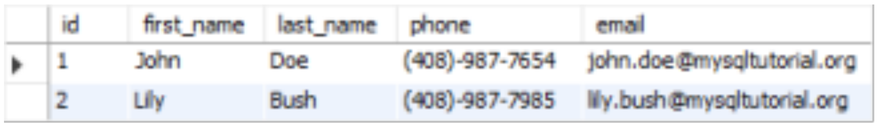
```
CREATE TABLE customers(
    id INT PRIMARY KEY AUTO_INCREMENT,
    first_name VARCHAR(255) NOT NULL,
    last_name VARCHAR(255) NOT NULL,
    phone VARCHAR(15) NOT NULL,
    email VARCHAR(255)
);
```

After that, insert data (https://www.mysqltutorial.org/mysql-insert-statement.aspx) into the `customers` table.

```
INSERT INTO customers(first_name,last_name,phone,email)
VALUES('John','Doe','(408)-987-7654','john.doe@mysqltutorial.org'),
      ('Lily','Bush','(408)-987-7985','lily.bush@mysqltutorial.org');
```

Finally, verify the insert by using the following `SELECT` (https://www.mysqltutorial.org/mysql-select-statement-query-data.aspx) statement:

```
SELECT * FROM customers;
```



id	first_name	last_name	phone	email
1	John	Doe	(408)-987-7654	john.doe@mysqltutorial.org
2	Lily	Bush	(408)-987-7985	lily.bush@mysqltutorial.org

Creating roles

Suppose you develop an application that uses the `CRM` database. To interact with the `CRM` database, you need to create accounts for developers who need full access to the database. In addition, you need to create accounts for users who need only read access and others who need both read/write access.

To avoid granting privileges to each user account individually, you create a set of roles and grant the appropriate roles to each user account.

To create new roles, you use `CREATE ROLE` statement:

```
CREATE ROLE
    crm_dev,
    crm_read,
    crm_write;
```

The role name is similar to the user account that consists of two parts: the name and host:

```
role_name@host_name
```

If you omit the host part, it defaults to '%' that means any host.

Granting privileges to roles

To grant privileges to a role, you use `GRANT` statement. The following statement grants all privileges to `crm_dev` role:

```
GRANT ALL
ON crm.*
TO crm_dev;
```

The following statement grants `SELECT` privilege to `crm_read` role:

```
GRANT SELECT
ON crm.*
TO crm_read;
```

The following statement grants `INSERT`, `UPDATE`, and `DELETE` privileges to `crm_write` role:

```
GRANT INSERT, UPDATE, DELETE
ON crm.*
TO crm_write;
```

Assigning roles to user accounts

Suppose you need one user account as the developer, one user account that can have read-only access and two user accounts that can have read/write access.

To create new users, you use `CREATE USER` (https://www.mysqltutorial.org/mysql-create-user.aspx) statements as follows:

```
-- developer user
CREATE USER crm_dev1@localhost IDENTIFIED BY 'Secure$1782';
-- read access user
CREATE USER crm_read1@localhost IDENTIFIED BY 'Secure$5432';
-- read/write users
CREATE USER crm_write1@localhost IDENTIFIED BY 'Secure$9075';
CREATE USER crm_write2@localhost IDENTIFIED BY 'Secure$3452';
```

To assign roles to users, you use `GRANT` statement.

The following statement grants the `crm_rev` role to the user account `crm_dev1@localhost` :

```
GRANT crm_dev
TO crm_dev1@localhost;
```

The following statement grants the `crm_read` role to the user account `crm_read1@localhost` :

```
GRANT crm_read
TO crm_read1@localhost;
```

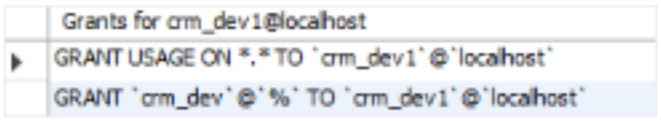
The following statement grants the `crm_read` and `crm_write` roles to the user accounts `crm_write1@localhost` and `crm_write2@localhost` :

```
GRANT crm_read,
    crm_write
TO crm_write1@localhost,
    crm_write2@localhost;
```

To verify the role assignments, you use the `SHOW GRANTS` (https://www.mysqltutorial.org/mysql-adminsitration/mysql-show-grants/) statement as the following example:

```
SHOW GRANTS FOR crm_dev1@localhost;
```

The statement returned the following result set:



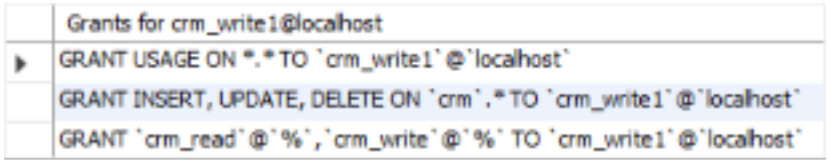
```
Grants for crm_dev1@localhost
GRANT USAGE ON *.* TO `crm_dev1`@`localhost`
GRANT `crm_dev`@`%` TO `crm_dev1`@`localhost`
```

As you can see, it just returned granted roles. To show the privileges that roles represent, you use the `USING` clause with the name of the granted roles as follows:

```
SHOW GRANTS
```

```
FOR crm_write1@localhost
USING crm_write;
```

The statement returns the following output:



	Grants for crm_write1@localhost
▶	GRANT USAGE ON *.* TO \`crm_write1\`@\`localhost\`
	GRANT INSERT, UPDATE, DELETE ON \`crm\`.* TO \`crm_write1\`@\`localhost\`
	GRANT \`crm_read\`@\`%\`,\`crm_write\`@\`%\` TO \`crm_write1\`@\`localhost\`

Setting default roles

Now if you connect to the MySQL using the `crm_read1` user account and try to access the `CRM` database:

```
>mysql -u crm_read1 -p
Enter password: ***********
mysql>USE crm;
```

The statement issued the following error message:

```
ERROR 1044 (42000): Access denied for user 'crm_read1'@'localhost' to database 'crm'
```

This is because when you granted roles to a user account, it did not automatically make the roles to become active when the user account connects to the database server.

If you invoke the `CURRENT_ROLE()` function, it will return `NONE`, meaning no active roles.

```
SELECT current_role();
```

Here is the output:

```
+----------------+
| current_role() |
+----------------+
| NONE           |
+----------------+
1 row in set (0.00 sec)
```

To specify which roles should be active each time a user account connects to the database server, you use the `SET DEFAULT ROLE` statement.

The following statement sets the default for the `crm_read1@localhost` account all its assigned roles.

```
SET DEFAULT ROLE ALL TO crm_read1@localhost;
```

Now, if you connect to the MySQL database server using the `crm_read1` user account and invoke the `CURRENT_ROLE()` function:

```
>mysql -u crm_read1 -p
Enter password: ***********
mysql> select current_role();
```

You will see the default roles for `crm_read1` user account.

```
+----------------+
| current_role() |
+----------------+
| `crm_read`@`%` |
+----------------+
1 row in set (0.00 sec)
```

You can test the privileges of `crm_read` account by switching the current database to `CRM` , executing a `SELECT` statement and a `DELETE` statement as follows:

```
mysql> use crm;
Database changed
mysql> SELECT COUNT(*) FROM customers;
```

```
+----------+
| COUNT(*) |
+----------+
|        2 |
+----------+
1 row in set (0.00 sec)

mysql> DELETE FROM customers;
ERROR 1142 (42000): DELETE command denied to user 'crm_read1'@'localhost' for table 'cu
```

It worked as expected. When we issued the `DELETE` statement, MySQL issued an error because `crm_read1` user account has only read access.

Setting active roles

A user account can modify the current user's effective privileges within the current session by specifying which granted role are active.

The following statement set the active role to `NONE`, meaning no active role.

```
SET ROLE NONE;
```

To set active roles to all granted role, you use:

```
SET ROLE ALL;
```

To set active roles to default roles that set by the `SET DEFAULT ROLE` statement, you use:

```
SET ROLE DEFAULT;
```

To set active named roles, you use:

```
SET ROLE
    granted_role_1
    [,granted_role_2, ...]
```

Revoking privileges from roles

To revoke privileges from a specific role, you use the `REVOKE` (https://www.mysqltutorial.org/mysql-revoke.aspx) statement. The `REVOKE` statement takes effect not only the role but also any account granted the role.

For example, to temporarily make all read/write users read-only, you change the `crm_write` role as follows:

```
REVOKE INSERT, UPDATE, DELETE
ON crm.*
FROM crm_write;
```

To restore the privileges, you need to re-grant them as follows:

```
GRANT INSERT, UPDATE, DELETE
ON crm.*
FOR crm_write;
```

Removing roles

To delete one or more roles, you use the `DROP ROLE` statement as follows:

```
DROP ROLE role_name[, role_name, ...];
```

Like the `REVOKE` statement, the `DROP ROLE` statement revokes roles from every user account to which they were granted.

For example, to remove the `crm_read`, `crm_write` roles, you use the following statement:

```
DROP ROLE crm_read, crm_write;
```

Copying privileges from a user account to another

MySQL treats user accounts like roles, therefore, you can grant a user account to another user account like granting a role to that user account. This allows you to copy privileges from a user to another user.

Suppose you need another developer account for the `CRM` database:

First, create the new user account:

```
CREATE USER crm_dev2@localhost
IDENTIFIED BY 'Secure$6275';
```

Second, copy privileges from the `crm_dev1` user account to `crm_dev2` user account as follows:

```
GRANT crm_dev1@localhost
TO crm_dev2@localhost;
```

In this tutorial, you have learned how to use MySQL roles to make it easier to manage privileges of user accounts.

SQL

1. DDL
2. DML
3. DQL
4. DCL
5. TCL

DDL

1.

```
CREATE TABLE professor(
empid int PRIMARY KEY,
name varchar(10)
);
```

```
CREATE TABLE course(
courseID integer PRIMARY KEY,
title varchar(30) UNIQUE NOT NULL,
ects integer
);
```

```
CREATE TABLE teaches(
title varchar(30) REFERENCES course(title) ON
DELETE CASCADE,
empid integer REFERENCES professor(empid) ON
DELETE SET NULL,
semester char(1)
);
```

2.

```
ALTER TABLE professor
ADD COLUMN (office integer);
```

```
ALTER TABLE professor
DROP COLUMN rank;
```

```
ALTER TABLE professor
MODIFY COLUMN name type varchar(30);
```

3.

```
DROP TABLE professor;
```

```
TRUNCATE TABLE professor;
```

DML

1.

```
INSERT INTO professor VALUES
(2136, 'Curie', 'C4');
```

```
INSERT INTO student (studid, name) VALUES
(29999, 'James'),
(31555, 'Christian');
```

2.

```
DELETE FROM student
WHERE semester > 13;
```

```
DELETE FROM student;
```

3.

```
UPDATE student
SET semester=semester+1
WHERE semester < 18;
```

DCL

1.

```
GRANT select (empid), update (office)
ON professor
TO some_user, another_user;
```

2.

```
REVOKE ALL PRIVILEGES
ON professor
FROM some_user, another_user;
```

3.

```
Privileges (rights) on tables, columns,...:
select, insert, update, delete, rule,
references, trigger
```

DQL

JOIN

1. NATURAL JOIN
2. CROSS JOIN
3. JOIN.......ON
4. JOIN.........USING(....)

1.

```
SELECT *
FROM professor NATURAL JOIN course;
```

2.

```
SELECT * FROM professor CROSS JOIN course;
```

3.

```
SELECT *
FROM professor JOIN course
ON professor.empid= course.taughtby
```

4.

```
SELECT *
FROM professor JOINcourse USING(empid);
```

5.

```
SELECT professor.empid, name, title
FROM professor, course
WHERE professor.empid= course.empid;
```

SET OPERATION

UNION:

```
(SELECT name FROM professor)
UNION
(SELECT name FROM professor);
```

SUBQUERY

UNCORRELATED:

```
SELECT empid
FROM professor
WHERE building IN
          (SELECT building
          FROM professor
          WHERE dept= 'PHI');
```

JOIN:

```
SELECT DISTINCT p1.empid
FROM professor p1, professor p2
WHERE p1.building = p2.building AND p2.dept =
'PHI';
```

ANY:

```
SELECT name
FROM professor
WHERE rank > ANY
        (SELECT rank
        FROM professor);
```

ALL:

```
SELECT name
FROM professor
WHERE rank >= ALL
       (SELECT rank
        FROM professor);
```

NOT IN:

```
SELECT empid
FROM professor
WHERE empid NOT IN
             (SELECT taughtby
              FROM course);
```

Correlated Subqueries:

```
SELECT name
FROM professor p
WHERE EXISTS (SELECT *
             FROM course c
             WHERE c.taughtby=
             p.empid);
```

IN vs. EXISTS:

```
SELECT name
FROM professor
WHERE empid IN (SELECT taughtby
               FROM course);
```

```
SELECT name
FROM professor
WHERE EXISTS (SELECT *
            FROM course c
            WHERE taughtby=
            empid);
```

Join Variants

Standard formulation (CROSS JOIN):

```
SELECT *
FROM R1, R2
WHERE R1.A = R2.B;
```

Alternative formulation (INNER JOIN):

```
SELECT *
FROM R1 JOIN R2
ON R1.A = R2.B;
```

NATURAL JOIN:

```
SELECT *
FROM R1 NATURAL JOIN R2;
```

Other variants:

```
LEFT, RIGHT, or FULL OUTER JOIN
```

```
SELECT *
FROM student NATURAL JOIN grades;
```

```
SELECT s.studid, name, semester, courseid,
empid, grade
FROM student s JOIN grades g
ON s.studid= g.studid;
```

Aggregate functions

Aggregate functions: `AVG, MAX, MIN, COUNT, SUM`

```
SELECT DISTINCT studid, name, semester
FROM student NATURAL JOIN grades
WHERE grade > (SELECT AVG(grade)
              FROM grades);
```

```
SELECT *
FROM student s
WHERE 1 = (SELECT COUNT(*)
          FROM takes t
          WHERE t.studid=
          s.studid);

SELECT empid, name, (SELECT
           SUM(ects)
           FROM course
           WHERE taughtby= empid)
           AS teachingLoad
FROM professor;


SELECT AVG(teachingLoad) AS result
FROM (SELECT SUM(ects) AS
     teachingLoad
     FROM course c
     GROUP BY taughtby) as r;
```

Grouping

```
SELECT empid, name, SUM(ects)
FROM course, professor
WHERE taughtby= empid AND rank = 'C4'
GROUP BY taughtby, name
HAVING AVG(ects) > 3;
```

LIMIT:

```
 SELECT *
 FROM student
 ORDER BY semester DESC
 LIMIT 5, 4;
```

RECURSION

```
WITH RECURSIVE transitiveCourse(pred, succ,
depth) AS (
  SELECT predecessor, successor, 0
  FROM requires
UNION
  SELECT DISTINCT t.pred, r.successor,
  t.depth+1
  FROM transitiveCourset, requires r
  WHERE t.succ= r.predecessor AND
  t.depth<1
)
SELECT c2.title
FROM transitiveCoursetc, course c1, course c2
WHERE tc.succ= c1.courseid AND c1.title =
'Theory of Science'
AND tc.pred= c2.courseid;
```

Views:

1.

```
CREATE VIEW profsAndtheirCoursesAS
SELECT c.title, p.name
FROM professor p, course c
WHERE p.empid= c.taughtby;

SELECT * FROM profsAndtheirCourses;
```

2.

```
CREATE VIEW ectsPerStudAS
SELECT s.name, s.studid, SUM(c.ects) AS sum
FROM student s, takes t, course c
WHERE t.courseid= c.courseidAND s.studid=
t.studid
GROUP BY s.name, t.studid;

SELECT sum FROM ectsPerStud;
```

3.

```
REPLACE VIEW profsAndtheirCoursesAS
SELECT c.title, p.name
FROM professor p, course c
WHERE p.empid= c.taughtby;
```

4.

```
CREATE VIEW howToughAS
SELECT empid, AVG(grade) AS avgGrade
FROM grades
GROUP BY empid;

UPDATE howTough
SET avgGrade= 1.0
WHERE empid= (SELECT empid
FROM professor
WHERE name = 'Socrates');
```

DATABASE

```
CREATE DATABASE sabbir;

SHOW DATABASES;

SHOW tables;

SHOW tables FROM sabbir;
```

TABLE

```
DESC sabbir_table;
```

USER

```
CREATE USER 'sabbir@'localhost' IDENTIFIED BY
'password';


mysql -u sabbir -p
Password: password
```

```
 SELECT user FROM mysql.user;

GRANT ALL ON sabbir.sabbir_table TO
'sabbir@'localhost';

 GRANT ALL ON *.* TO '*'@'localhost';


 DROP USER 'sabbir@'localhost';
```

TRIGGER

before insert:

```
 create table employee (id int, name varchar(20),
 salary int);

 delimiter $

 create trigger bef_ins
 before insert on employee
 for each row
 if new.salary is null then set new.salary =
 10000;
 end if; $

 delimiter ;

 insert into employee
 values (2, 'kona', 20000),
 (3, 'sabbir', null),
 (4, 'saifuddin', 10000);
```

//after insert

```
 create table text (id int, text varchar (30));

 delimiter #

 create trigger aft_ins
 after insert on employee
 for each row
 begin
 if new.salary is null
 then insert into t_aft_ins(text) values
 (concat(new.name, ', insert salary'));
 end if;
 end#

 delimiter ;

 insert into employee
 values (8, 'kona', 20000),
 (9, 'sabbir', null),
 (10, 'saifuddin', 10000);
```

Before update:

```
 delimiter #

 create trigger bef_up
 before update on employee
 for each row
 if new.salary< old.salary then set new.salary =
 old.salary*1.1;
 end if; #

 delimiter ;
```

Before delete:

```
 create table employee2 (id int, name
 varchar(20), salary int);

 delimiter  #
 create trigger bef_del
 before delete on employee
 for each row
 begin
 insert into employee2 values (old.id, old.name,
 old.salary);
 end #

 delimiter ;


 delete from employee where id=2;
```

```
## DROP TABLE IF EXISTS SalaryArchives;
```

General Information

```
For connection:
mysql -u root -p --port 3307

//////////////

cheking status:
status;

select user from MySQl.user;

SELECT USER();
SELECT DATABASE();
show variables where variable_name ='port';
show variables where variable_name ='hostname';

/////////////////////

clear screen:
system cls;

Create user:
create user 'test_DB_user'@'localhost' IDENTIFIED BY 'password' PASSWORD
EXPIRE NEVER;

create database:
create database test_db

CREATE VIEW [Brazil Customers] AS
SELECT CustomerName, ContactName
FROM Customers
WHERE Country = 'Brazil';

SELECT * FROM [Brazil Customers];

CREATE OR REPLACE VIEW profsAndtheirCourses AS SELECT c.title, p.name FROM
professor p, course c WHERE p.empid = c.taughtby;

SELECT empid, name, SUM(ects) FROM course, professor WHERE taughtby = empid
AND rank = 'C4' GROUP BY taughtby, name HAVING AVG(ects) > 3;

GRANT INSERT, UPDATE, DELETE
ON employees
TO bob@localhost;
```

Creating AFTER INSERT trigger example

```
DELIMITER $$

CREATE TRIGGER after_members_insert
AFTER INSERT
ON members FOR EACH ROW
BEGIN
    IF NEW.birthDate IS NULL THEN
```

```
        INSERT INTO reminders(memberId, message)
        VALUES(new.id,CONCAT('Hi ', NEW.name, ', please update your date of
birth.'));
    END IF;
END$$

DELIMITER ;
```

Introduction to MySQL CREATE TRIGGER statement

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

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

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

```
## SHOW TRIGGERS;
```

MySQL BEFORE INSERT Trigger

```
DELIMITER $$

CREATE TRIGGER before_workcenters_insert
BEFORE INSERT
ON WorkCenters FOR EACH ROW
BEGIN
    DECLARE rowcount INT;

    SELECT COUNT(*)
    INTO rowcount
    FROM WorkCenterStats;

    IF rowcount > 0 THEN
        UPDATE WorkCenterStats
        SET totalCapacity = totalCapacity + new.capacity;
    ELSE
        INSERT INTO WorkCenterStats(totalCapacity)
        VALUES(new.capacity);
    END IF;

END $$

DELIMITER ;
```

Creating `BEFORE UPDATE` trigger example

The following statement creates a `BEFORE UPDATE` trigger on the `sales` table.

```
DELIMITER $$

CREATE TRIGGER before_sales_update
BEFORE UPDATE
ON sales FOR EACH ROW
BEGIN
    DECLARE errorMessage VARCHAR(255);
    SET errorMessage = CONCAT('The new quantity ',
                        NEW.quantity,
                        ' cannot be 3 times greater than the current quantity
',
                        OLD.quantity);

    IF new.quantity > old.quantity * 3 THEN
        SIGNAL SQLSTATE '45000'
            SET MESSAGE_TEXT = errorMessage;
    END IF;
END $$

DELIMITER ;
```

Let's examine the trigger in details:

….., declare a variable and set its value to an error message. Note that, in the `BEFORE TRIGGER`, you can access both old and new values of the columns via `OLD` and `NEW` modifiers.

```
DECLARE errorMessage VARCHAR(255);
SET errorMessage = CONCAT('The new quantity ',
                        NEW.quantity,
                        ' cannot be 3 times greater than the current quantity
',
                        OLD.quantity);
```

Note that we use the `CONCAT()` function to form the error message.

Finally, use the `IF-THEN` statement to check if the new value is 3 times greater than old value, then raise an error by using the `SIGNAL` statement:

```
IF new.quantity > old.quantity * 3 THEN
        SIGNAL SQLSTATE '45000'
            SET MESSAGE_TEXT = errorMessage;
```

```
END IF;
```

Creating AFTER DELETE trigger example

The following AFTER DELETE trigger updates the total salary in the SalaryBudgets table after a row is deleted from the Salaries table:

```
CREATE TRIGGER after_salaries_delete
AFTER DELETE
ON Salaries FOR EACH ROW
UPDATE SalaryBudgets
SET total = total - old.salary;
```