# Triggers em MySQL – Definição, Sintaxe e Criação – 44

Postado Em 13/02/2014 [data do post] por [Fábio dos Reis](http://www.bosontreinamentos.com.br/author/admin/) [autor do post] em [MySQL](http://www.bosontreinamentos.com.br/category/mysql/) [categoria do post]

[Tweet](https://twitter.com/share)

## Triggers em MySQL

Neste artigo vamos introduzir o conceito de triggers em MySQL, com exemplos de código e aplicações.

### O que é um Trigger SQL?

Um trigger (“gatilho”) é um objeto programável do banco de dados associado a uma tabela. Trata-se de um procedimento que é invocado automaticamente quando um comando DML é executado na tabela, sendo executado para cada linha afetada. Desta forma, as operações que podem disparar um trigger são:

* INSERT
* UPDATE
* DELETE

Geralmente, os triggers são empregados para verificar integridade dos dados, fazer validação dos dados e outras operações relacionadas.

### Diferença entre Trigger e Procedimento Armazenado

Tanto os triggers quanto as stored procedures são objetos programáveis de um banco de dados. Porém, eles possuem diferenças importantes entre si, que afetam o modo como são aplicados. Algumas das principais diferenças entre trigger e procedimentos armazenados são:

* Um Trigger é associado a uma tabela.
* Os triggers são armazenados no próprio banco de dados como arquivos separados.
* Triggers não são chamados diretamente, sendo invocados automaticamente, ao contrário dos procedimentos armazenados.
* Procedimentos armazenados podem trabalhar com parâmetros; Não passamos parâmetros aos triggers.
* Os triggers não retornam um conjunto de resultados (resultset) ao cliente chamador.

### Aplicações dos triggers

As principais aplicações dos triggers em bancos de dados são:

* Validação de Dados (tipos de dados, faixas de valores, etc).
* Rastreamento e registro de logs de atividades em tabelas.
* Verificação de integridade de dados e consistência
* Arquivamento de registros excluídos.

### Modos de Disparo de um Trigger

Um Trigger em MySQL pode ser disparado de dois modos diferentes:

* BEFORE – O trigger é disparado e seu código executado ANTES da execução de cada evento – por exemplo, antes de cada inserção de registros na tabela.
* AFTER – O código presente no trigger é executado após todas as ações terem sido completadas na tabela especificada.

### Sintaxe para criação de um trigger em MySQL

Para criar um trigger em MySQL usamos a seguinte sintaxe:

CREATE TRIGGER nome timing operação

ON tabela

FOR EACH ROW

declarações

Onde:

* timing pode ser BEFORE ou AFTER
* operação pode ser INSERT / UPDATE / DELETE

### Exemplo

Neste exemplo criaremos uma tabela chamada Produto, que conterá os seguintes dados:

* Nome do produto
* Identificação do produto (chave primária)
* Preço normal
* Preço com desconto a ser aplicado

Logo após criaremos um trigger de nome tr\_desconto, cuja função será aplicar um valor de desconto de 10% à coluna Preco\_Desconto quando for disparado. Ou seja, todos os produtos terão seu preço reduzido em 10% nesta coluna. O trigger será disparado ao inserir um novo registro na tabela, o que faremos no passo 3. Veja o código completo do exercício a seguir:

-- 1. Criar a tabela de exemplo:

CREATE TABLE Produto (

idProduto INT NOT NULL AUTO\_INCREMENT,

Nome\_Produto VARCHAR(45) NULL,

Preco\_Normal DECIMAL(10,2) NULL,

Preco\_Desconto DECIMAL(10,2) NULL,

PRIMARY KEY (idProduto));

-- 2. Criar o Trigger:

CREATE TRIGGER tr\_desconto BEFORE INSERT

ON Produto

FOR EACH ROW

SET NEW.Preco\_Desconto = (NEW.Preco\_Normal \* 0.90);

-- 3. Executar uma inserção que irá disparar o Trigger:

INSERT INTO Produto (Nome\_Produto, Preco\_Normal)

VALUES ("DVD", 1.00), ("Pendrive", 18.00);

-- 4. Verificar se trigger foi disparado observando o preço com desconto:

SELECT \* FROM Produto;

### Como excluir um trigger

Para excluir um trigger em MySQL usamos a declaração DROP TRIGGER, seguida do nome do trigger, como no exemplo:

DROP TRIGGER tr\_desconto;

Saiba mais sobre triggers em MySQL assistindo ao vídeo a seguir:

https://www.youtube.com/watch?v=JOnkvqUaNOU&t=3s

http://www.bosontreinamentos.com.br/mysql/mysql-triggers-definicao-sintaxe-e-criacao-44/

# MySQL – TRIGGERS

## Neste artigo, serão apresentados os principais conceitos sobre os TRIGGERS e sua aplicabilidade.

[Artigos](https://www.devmedia.com.br/artigos/) [Banco de Dados](https://www.devmedia.com.br/artigos/banco-de-dados) MySQL – TRIGGERS

Neste artigo, serão apresentados os principais conceitos sobre os TRIGGERS e sua aplicabilidade. Após a conceituação, trabalharemos um estudo de caso de um estoque de produtos, com baixa nas quantidades através destes procedimentos armazenados.

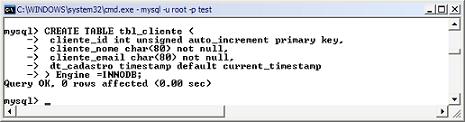
**Após ler este artigo, você estará apto à:**

* Definir o que é um TRIGGER;
* Definir dados de antes (OLD) e depois (NEW);
* Criar um TRIGGER;
* Excluir um TRIGGER;
* Restrições em relação à TRIGGERS.

### O que é um TRIGGER?

Um TRIGGER ou gatilho é um objeto de banco de dados, associado a uma tabela, definido para ser disparado, respondendo a um evento em particular. Tais eventos são os comandos da DML (Data Manipulation Language): INSERT, REPLACE, DELETE ou UPDATE. Podemos definir inúmeros TRIGGERS em uma base de dados baseados diretamente em qual dos comandos acima irá dispará-lo, sendo que, para cada um, podemos definir apenas um TRIGGER. Os TRIGGERS poderão ser disparados para trabalharem antes ou depois do evento. Veremos como definir o momento de atuação do TRIGGER mais à frente.

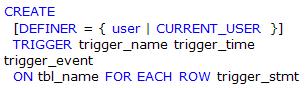
Utilizaremos a seguinte tabela da **Figura 01** para nossos testes:



**Figura 01.** Criando a tabela de testes – tbl\_cliente.

Baseado na tabela **tbl\_cliente**, podemos definir os TRIGGERS para serem disparados, por exemplo, antes (BEFORE) ou depois (AFTER) de um INSERT. Agora sabemos então que para cada momento BEFORE ou AFTER, podemos ter um TRIGGER a ser disparado para defender alguma lógica.

A sintaxe geral de definição de um TRIGGER é a seguinte:



- DEFINER: Quando o TRIGGER for disparado, esta opção será checada para checar com quais privilégios este será disparado. Utilizará os privilégios do usuário informado em user (‘wagner’@’localhost’) ou os privilégios do usuário atual (CURRENT\_USER). Caso essa sentença seja omitida da criação do TRIGGER, o valor padrão desta opção é CURRENT\_USER();

* trigger\_name: define o nome do procedimento, por exemplo, trg\_test;
* trigger\_time: define se o TRIGGER será ativado antes (BEFORE) ou depois (AFTER) do comando que o disparou;
* trigger\_event: aqui se define qual será o evento, INSERT, REPLACE, DELETE ou UPDATE;
* tbl\_name: nome da tabela onde o TRIGGER ficará “pendurado” aguardando o trigger\_event;
* trigger\_stmt: as definições do que o o TRIGGER deverá fazer quando for disparado.

### Definir dados de antes (OLD) e depois (NEW)

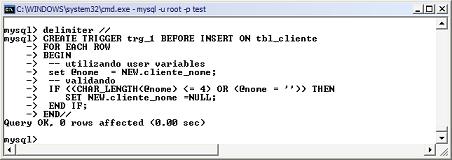
Em meio aos TRIGGERS temos dois operadores importantíssimos que nos possibilitam acessar as colunas da tabela alvo do comando DML, ou seja, podemos acessar os valores que serão enviados para a tabela tbl\_cliente antes (BEFORE) ou depois (AFTER) de um UPDATE, por exemplo. Tais operadores nos permitirão então, ter dois momentos, o antes e o depois e também examinar os valores para que sejam ou não inseridos, atualizados ou excluídos da tabela.

Antes mesmo de analisarmos os operadores, temos que analisar vejamos as seguintes diretrizes:

* INSERT: o operador NEW.nome\_coluna, nos permite verificar o valor enviado para ser inserido em uma coluna de uma tabela. OLD.nome\_coluna não está disponível.
* DELETE: o operador OLD.nome\_coluna nos permite verificar o valor excluído ou a ser excluído. NEW.nome\_coluna não está disponível.
* UPDATE: tanto OLD.nome\_coluna quanto NEW.nome\_coluna estão disponíveis, antes da atualização (BEFORE) ou depois da atualização (AFTER) de uma linha.

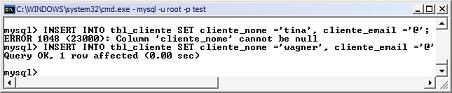
Percebemos então que, ao inserir uma nova linha em uma tabela, temos os valores das colunas disponível através do operador NEW.nome\_coluna, quando excluímos uma linha, temos ainda os valores das colunas da linha excluída através do operador OLD.nome\_coluna e temos os dois operadores disponíveis no UPDATE e no REPLACE pois as duas declarações consistem em um DELETE seguido por um INSERT. (REPLACE, este comando apresenta outros detalhes a serem analisados numa próxima oportunidade).

Criemos então um primeiro TRIGGER, bem básico que não fará nada mais que validar se os dados foram passados em uma declaração INSERT antes (BEFORE) que sejam cadastrados na tabela de exemplo. Validaremos o nome com quantidade de caracteres maior ou igual a 4 (quatro). (**Figura 02**)



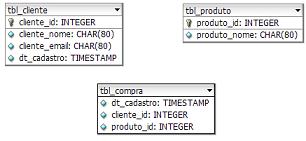
**Figura 02** – TRIGGER para conferir se o CHAR\_LENTH() de nome é maior que 4 (quatro).

Ao tentarmos inserir um valor cujo número de caracteres é menor ou igual a 0 ou nada, o TRIGGER será disparado e setará o valor enviado para NULL através do operador NEW.nome\_coluna. Como na tabela de exemplo a coluna cliente\_nome foi configurada com a restrição NOT NULL, ou seja, não aceitará valores nulos, uma mensagem de erro será enviada e o INSERT falhará, como mostra a **Figura 03**.

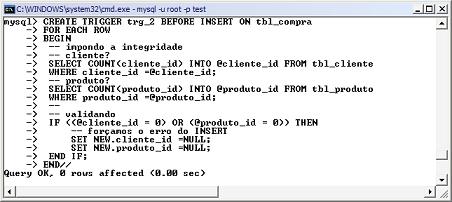


**Figura 03** – O INSERT falhará após uma tentativa de inserir um valor não permitido.

Podemos criar uma restrição de integridade referencial com TRIGGERS. Por exemplo, usaremos tabelas MyISAM (que não dão suporte a criação de relacionamento com chaves estrangeiras) e, aproveitando nosso exemplo, como clientes compram produtos, podemos criar uma tabela de produtos e restringir que somente os produtos cadastrados podem ser comprados, inseridos na tabela de compra e que somente clientes cadastrados podem efetuar compras. Criemos então a tabela de produtos e compras, acompanhe o seguinte modelo:

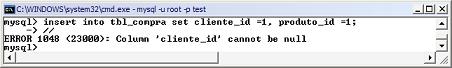


Perceba que o modelo não conta com relacionamentos, os quais defenderemos com TRIGGERS. Um INSERT que seja endereçado à tabela tbl\_compra (que tem as colunas cliente\_id e produto\_id definidas como NOT NULL), disparará um TRIGGER, tendo a responsabilidade de conferir se o produto a ser inserido existe na tabela de produtos e se o cliente que efetua a compra está na tabela tbl\_cliente. Vamos ao TRIGGER (**Figura 04**):



**Figura 04** – TRIGGER para impor integridade referencial ao campos que se relacionam com outras tabelas.

Como somente criamos a tabela e não inserimos nenhum cliente e nem produto algum, qualquer tentativa de inserção na tabela tbl\_compra falhará, como mostra a figura seguinte (**Figura 05**):



**Figura 05** – Implementando restrição de integridade referencial com TRIGGERS.

Uma outra aplicação para se trabalhar com TRIGGERS é ter uma tabela para e-mail para ser utilizada por algum sistema de newsletter da empresa. Podemos facilmente nesse caso, implementar um TRIGGER para colher os e-mail do cadastro de clientes e estes serem inseridos em uma outra tabela, por exemplo, uma tabela com a seguinte estrutura (**Figura 06**):



**Figura 06** – Tabela para cadastrar e-mail automaticamente com o auxílio de TRIGGERS.

Definindo o TRIGGER para pegar os valores inseridos na coluna cliente\_email da tabela tbl\_cliente e inserir também na tabela tbl\_newsletter. Segue definido na **Figura 07**.



**Figura 07** – Inserção automática de e-mail em outra tabela.

Testando o TRIGGER na **Figura 08**:



**Figura 08** – Os endereços de e-mails de clientes sendo inseridos automaticamente por meio de TRIGGERS.

### Excluir um TRIGGER

Para excluir um TRIGGER, utilize a sintaxe DROP TRIGGER trigger\_name, como mostra o exemplo da **Figura 09**.



**Figura 09** – Excluindo um TRIGGER.

### Restrições em relação à TRIGGERS

A implementação deste recurso atualmente no MySQL tem várias limitações, a conferir as principais:

* Não se pode chamar diretamente um TRIGGER com CALL, como se faz com um Stored Procedures;
* Não é permitido iniciar ou finalizar transações em meio à TRIGGERS;
* Não se pode criar um TRIGGERS para uma tabela temporária – TEMPORARY TABLE;
* TRIGGERS ainda não podem ser implementadas com a intenção de devolver para o usuário ou para uma aplicação mensagens de erros.

Apresentamos neste artigo, os principais conceitos de implementações de TRIGGERS, com os operadores OLD e NEW. No próximo artigo mostrarei como impedir uma exclusão de um dado baseado em outras tabelas relacionadas e como construir um **log** no banco de dados com um TRIGGER disparado com a instrução UPDATE, onde temos acesso ao antes e o depois das alterações.

<https://www.devmedia.com.br/mysql-triggers/8088>

# MySQL Básico: Triggers

## Veja neste artigo como utilizar triggers no banco de dados MySQL para automatizar ações com base em eventos ocorridos nas tabelas, como inclusão e exclusão de registros.

[Artigos](https://www.devmedia.com.br/artigos/) [Banco de Dados](https://www.devmedia.com.br/artigos/banco-de-dados) MySQL Básico: Triggers

É muito comum, em aplicações que utilizam [bancos de dados](http://www.devmedia.com.br/cursos/banco-de-dados), que ações sejam disparadas em resposta ou como consequência de outras, realizando operações de cálculo, validações e, em geral, surtindo alterações na base de dados.

Em muitos casos, os programadores optam por executarem tais ações a partir da própria aplicação, executando várias instruções SQL em sequência para obter o resultado esperado. De fato essa é uma solução que pode até ser tida como mais segura, por certos pontos de vista, mas tende a tornar ainda mais “pesada” a execução de certas tarefas, requisitando mais recursos da máquina cliente.

 Conheça mais sobre [Comandos Básicos SQL](http://www.devmedia.com.br/comandos-basicos-em-sql-insert-update-delete-e-select/37170)

A “solução” (ou pelo menos uma forma alternativa) a essa está na utilização de TRIGGERS no banco de dados, automatizando certas ações com base em eventos ocorridos.

Triggers (“gatilhos” em português) são objetos do banco de dados que, relacionados a certa tabela, permitem a realização de processamentos em consequência de uma determinada ação como, por exemplo, a inserção de um registro.

Os triggers podem ser executados ANTES ou DEPOIS das operações de INSERT, UPDATE e DELETE de registros.

Observação: [o suporte a triggers foi incluído na versão 5.0.2 do MySQL](http://www.devmedia.com.br/mysql-triggers/8088).

### Prós e Contras das Triggers

Os principais pontos positivos sobre os triggers são:

* Parte do processamento que seria executado na aplicação passa para o banco, poupando recursos da máquina cliente.
* Facilita a manutenção, sem que seja necessário alterar o código fonte da aplicação.

Já contra sua utilização existem as seguintes considerações:

* Alguém que tenha acesso não autorizado ao banco de dados poderá visualizar e alterar o processamento realizado pelos gatilhos.
* Requer maior conhecimento de manipulação do banco de dados (SQL) para realizar as operações internamente.

A seguir é explicado o processo de criação de triggers, a sintaxe utilizada e o significado de cada instrução.

### Sintaxe da Trigger

A sintaxe dos comandos para criar um novo trigger no [MySQL](http://www.devmedia.com.br/guias/mysql/19) é a seguinte:

CREATE TRIGGER nome momento evento

ON tabela

FOR EACH ROW

BEGIN

/\*corpo do código\*/

END

Listagem 1. Sintaxe para criação de trigger

Onde se tem os seguintes parâmetros:

* nome: nome do gatilho, segue as mesmas regras de nomeação dos demais objetos do banco.
* momento: quando o gatilho será executado. Os valores válidos são BEFORE (antes) e AFTER (depois).
* evento: evento que vai disparar o gatilho. Os valores possíveis são INSERT, UPDATE e DELETE. Vale salientar que os comandos LOAD DATA e REPLACE também disparam os eventos de inserção e exclusão de registros, com isso, os gatilhos também são executados.
* tabela: nome da tabela a qual o gatilho está associado.

Não é possível criar mais de um trigger para o mesmo evento e momento de execução na mesma tabela. Por exemplo, não se pode criar dois gatilhos AFTER INSERT na mesma tabela.

### Os registros NEW e OLD

Como os triggers, são executados em conjunto com operações de inclusão e exclusão, é necessário poder acessar os registros que estão sendo incluídos ou removidos. Isso pode ser feito através das palavras NEW e OLD.

Em gatilhos executados após a inserção de registros, a palavra reservada NEW dá acesso ao novo registro. Pode-se acessar as colunas da tabela como atributo do registro NEW, como veremos nos exemplos.

O operador OLD funciona de forma semelhante, porém em gatilhos que são executados com a exclusão de dados, o OLD dá acesso ao registro que está sendo removido.

### Utilização do trigger

Para exemplificar e tornar mais clara a utilização de gatilhos, simularemos a seguinte situação: um mercado que, ao realizar vendas, precisa que o estoque dos produtos seja automaticamente reduzido. A devolução do estoque deve também ser automática no caso de remoção de produtos da venda.

Como se trata de um ambiente hipotético, teremos apenas duas tabelas de estrutura simples, cujo script de criação é mostrado na listagem a seguir.

CREATE TABLE Produtos

(

Referencia VARCHAR(3) PRIMARY KEY,

Descricao VARCHAR(50) UNIQUE,

Estoque INT NOT NULL DEFAULT 0

);

INSERT INTO Produtos VALUES ('001', 'Feijão', 10);

INSERT INTO Produtos VALUES ('002', 'Arroz', 5);

INSERT INTO Produtos VALUES ('003', 'Farinha', 15);

CREATE TABLE ItensVenda

(

Venda INT,

Produto VARCHAR(3),

Quantidade INT

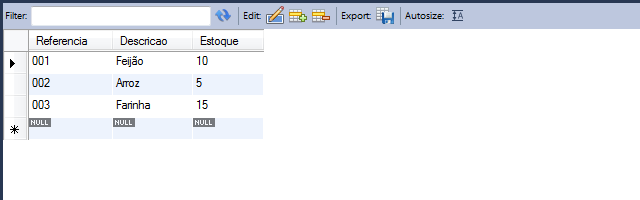
);

Listagem 2. Criação das tabelas utilizadas

Ao inserir e remover registro da tabela ItensVenda, o estoque do produto referenciado deve ser alterado na tabela Produtos. Para isso, serão criados dois triggers: um AFTER INSERT para dar baixa no estoque e um AFTER DELETE para fazer a devolução da quantidade do produto.

Observação: como usaremos instruções que requerem ponto e vírgula no final, alteraremos o delimitador de instruções para $$ e depois de criar os triggers, voltaremos para o padrão. Essa alteração não está diretamente ligada aos triggers.

Apenas para registrar e conferir, a imagem a seguir mostra um select feito sobre a tabela Produtos após a inserção dos registros de exemplo.

  
Figura 1. Registros iniciais da tabela Produtos

Na Listagem 3 são criados os gatilhos para executar as ações já discutidas.

DELIMITER $

CREATE TRIGGER Tgr\_ItensVenda\_Insert AFTER INSERT

ON ItensVenda

FOR EACH ROW

BEGIN

UPDATE Produtos SET Estoque = Estoque - NEW.Quantidade

WHERE Referencia = NEW.Produto;

END$

CREATE TRIGGER Tgr\_ItensVenda\_Delete AFTER DELETE

ON ItensVenda

FOR EACH ROW

BEGIN

UPDATE Produtos SET Estoque = Estoque + OLD.Quantidade

WHERE Referencia = OLD.Produto;

END$

DELIMITER ;

Listagem 3. Criação dos triggers

No primeiro gatilho, foi utilizado o registro NEW para obter as informações da linha que está sendo inserida na tabela. O mesmo é feito no segundo gatilho, onde se obtém os dados que estão sendo apagados da tabela através do registro OLD.

Tendo criado os triggers, podemos testá-los inserindo dados na tabela ItensVenda. Nesse caso, vamos simular uma venda de número 1 que ontem três unidades do produto 001, uma unidade do produto 002 e cinco unidades do produto 003.

INSERT INTO ItensVenda VALUES (1, '001',3);

INSERT INTO ItensVenda VALUES (1, '002',1);

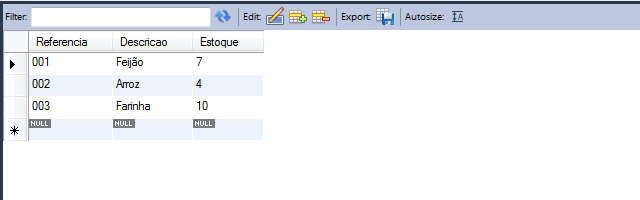
INSERT INTO ItensVenda VALUES (1, '003',5);

Listagem 4. Inserindo dados na tabela

DELETE FROM ItensVenda WHERE Venda = 1 AND Produto = '001';

Listagem 5. Excluindo dados da tabela ItensVenda

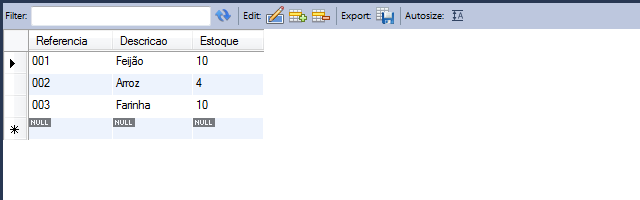
Agora, fazendo uma consulta à tabela Produtos, obtemos o resultado exibido na Figura 2.

  
Figura 2. Baixa no estoque após a inserção na tabela ItensVenda

Nota-se que o estoque dos produtos foi corretamente reduzido, de acordo com as quantidades “vendidas”.

Agora para testar o trigger da exclusão, removeremos o produto 001 dos itens vendidos. Com isso, o seu estoque deve ser alterado para o valor inicial, ou seja, 10.

Executando novamente um select na tabela Produtos, veremos que apenas o produto 001 teve o estoque atualizado, voltando a 10, como vemos na Figura 3.

  
Figura 3.Devolução do estoque após exclusão de registro na tabela ItensVenda

Com isso confirmamos que os gatilhos estão funcionando da forma esperada.

Abaixo temos um comando que exibe as triggers que foram criadas:

SHOW TRIGGERS

Listagem 6. Exibição das triggers criadas

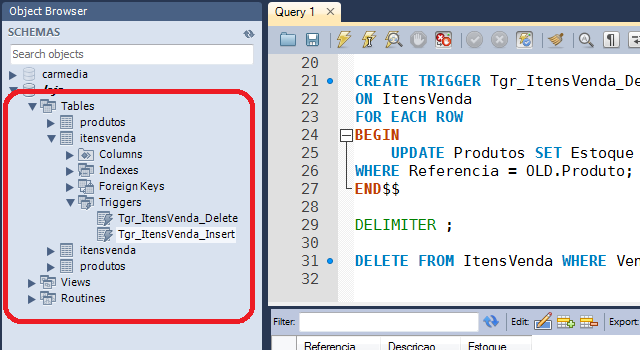
A seguir vemos a exclusão de uma trigger no MySQL:

DROP TRIGGER Tgr\_ItensVenda\_Insert

Listagem 7. Exclusão de trigger

### Informações adicionais

Apenas a nível de informação, vale o seguinte comentário: na nova versão do MySQL, no MySQL Workbench é possível visualizar os gatilhos relacionados a uma tabela através do Object browser, como mostra a figura a seguir.

  
Figura 4.Object Browser no MySQL Workbench

Em ambientes reais, [triggers podem ser utilizados para operações mais complexas](http://www.devmedia.com.br/desenvolvendo-triggers-em-sql-server-oracle-firebird-e-postgres/5625), por exemplo, antes de vender um item, verificar se há estoque disponível e só então permitir a saída do produto.

Com isso finalizamos este artigo, onde foram apresentados os triggers no banco de dados MySQL.

Até a próxima oportunidade.

https://www.devmedia.com.br/mysql-basico-triggers/37462

# How To Use MySQL Triggers

Introduction

MySQL triggers apply restrictions to tables when adding, updating, or removing table rows.

Columns in MySQL apply a slight amount of value limitations. For example, setting a column [data type](https://phoenixnap.com/kb/mysql-data-types) as tiny int and not null requires a small number value input. Still, more restrictions are needed to maintain the integrity of data.

Os triggers podem ser executados ANTES ou DEPOIS das operações de INSERT, UPDATE e DELETE de registros.

This tutorial shows you how to use MySQL triggers and provides examples for each type of trigger.

Prerequisites

* A system running MySQL on a [database server](https://phoenixnap.com/servers/database)
* [MySQL user account](https://phoenixnap.com/kb/how-to-create-new-mysql-user-account-grant-privileges) with root privileges
* Knowledge of basic MySQL commands (refer to our downloadable [MySQL commands](https://phoenixnap.com/kb/mysql-commands-cheat-sheet) cheat sheet)

## What is a Trigger in MySQL?

Um gatilho/trigger é um objeto nomeado do MySQL que se ativa quando um evento ocorre numa tabela. Os gatilhos são um tipo particular de procedimento armazenado associado a uma tabela específica.

Os gatilhos permitem o acesso a valores da tabela para fins de comparação utilizando NEW e OLD. A disponibilidade dos modificadores depende do evento de disparo que se utiliza:

|  |  |  |
| --- | --- | --- |
| Trigger Event | OLD | NEW |
| INSERT | No | Yes |
| UPDATE | Yes | Yes |
| DELETE | Yes | No |

A verificação ou modificação de um valor ao tentar inserir dados torna a coluna NEW.<nome> modificador disponível. Isto acontece porque uma tabela é atualizada com um novo conteúdo. Em contraste, um OLD.<nome de coluna > valor não existe para uma declaração de inserção porque não existe qualquer informação no seu lugar de antemão para ser alterada.

Quando se atualiza um registro da tabela, ambos os modificadores estão disponíveis. Há OLD.<nome da coluna> dados que queremos atualizar para NEW.<nome da coluna> dados.

Finalmente, ao remover um registro, a coluna OLD.<nome> modificador acessa o valor removido. O NEW.<nome da coluna> não existe porque nada substitui o valor antigo no momento da remoção.

### MySQL Trigger Example

As an example of an applied trigger, inserting new values into the table person yields a different result than the original input:

create table person(

id int primary key auto\_increment,

name varchar(50),

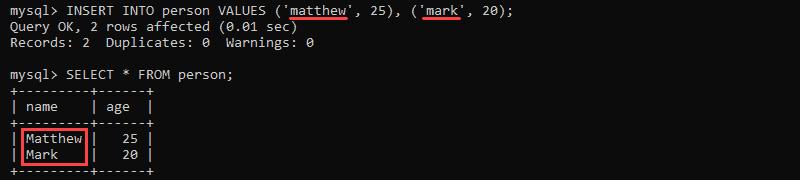
age int

);

insert into person (name, age) values

('Ribamar', 64),

('Elias', 21);



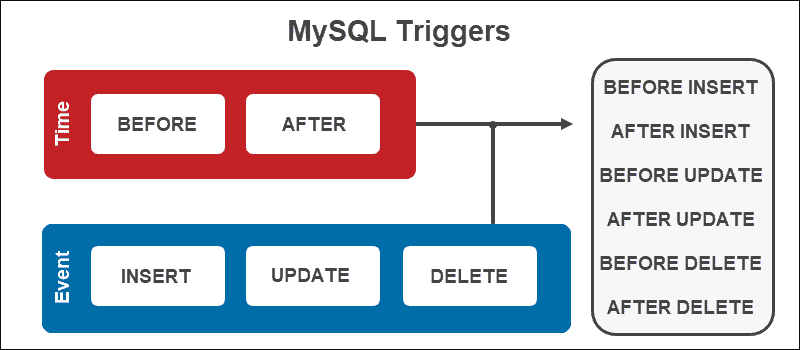
Notice the inserted names were initially lowercase. When selecting the table, the first letter shows as capitalized. Even though there is no indication of anything different from a regular insert statement, the trigger fired before the insert statement to capitalize the first letter of the name.

## Using MySQL Triggers

Every trigger associated with a table has a unique name and function based on two factors:

1. Time. BEFORE or AFTER a specific row event.

2. Event. INSERT, UPDATE or DELETE.



MySQL triggers fire depending on the activation time and the event for a total of six unique trigger combinations. The before statements help to check data and make changes before making commitments, whereas the after statements commit the data first and then execute statements.

The execution of a set of actions happens automatically, affecting all inserted, deleted, or updated rows in the statement.

### Create Triggers

Use the CREATE TRIGGER statement syntax to create a new trigger:

CREATE TRIGGER <trigger name> <trigger time > <trigger event>

ON <table name>

FOR EACH ROW

<trigger body>;

The best practice is to name the trigger with the following information:

<trigger time>\_<table name>\_<trigger event>

For example, if a trigger fires before insert on a table named employee, the best convention is to call the trigger:

before\_employee\_insert

Alternatively, a common practice is to use the following format:

<table name>\_<first letter of trigger time><first letter of trigger name>

The before insert trigger name for the table employee looks like this:

employee\_bi

The trigger executes at a specific time of an event on a table defined by <table name> for each row affected by the function.

### Delete Triggers

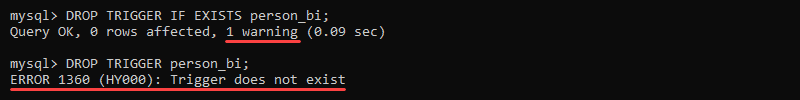
To delete a trigger, use the DROP TRIGGER statement:

DROP TRIGGER <trigger name>;



Alternatively, use:

**DROP TRIGGER IF EXISTS <trigger name>;**



The error message does not display because there is no trigger, so no warning prints.

### Create Example Database

[Create a database](https://phoenixnap.com/kb/how-to-create-mysql-database-workbench) for the trigger example codes with the following structure:

1. [Create a table](https://phoenixnap.com/kb/how-to-create-a-table-in-mysql) called person with name and age for columns.

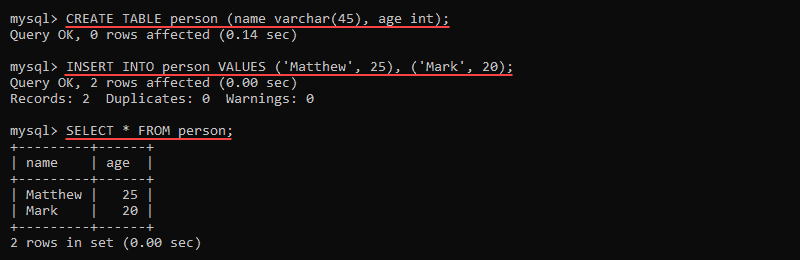
CREATE TABLE person (name varchar(45), age int);

Insert sample data into the table:

INSERT INTO person VALUES ('Matthew', 25), ('Mark', 20);

Select the table to see the result:

SELECT \* FROM person;



2. Create a table called average\_age with a column called average:

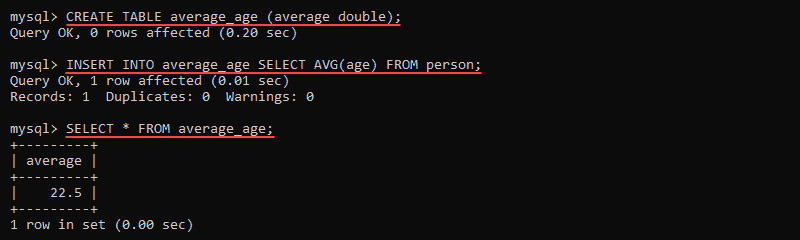
CREATE TABLE average\_age (average double);

Insert the average age value into the table:

INSERT INTO average\_age SELECT AVG(age) FROM person;

Select the table to see the result:

SELECT \* FROM average\_age;



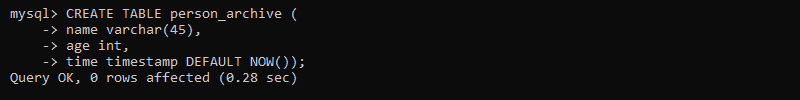
3. Create a table called person\_archive with name, age, and time columns:

CREATE TABLE person\_archive (

name varchar(45),

age int,

time timestamp DEFAULT NOW());



### Create a BEFORE INSERT Trigger

To create a BEFORE INSERT trigger, use:

CREATE TRIGGER <trigger name> BEFORE INSERT

ON <table name>

FOR EACH ROW

<trigger body>;

The BEFORE INSERT trigger gives control over data modification before committing into a database table. Capitalizing names for consistency, checking the length of an input, or catching faulty inputs with BEFORE INSERT triggers further provides value limitations before entering new data.

BEFORE INSERT Trigger Example

Create a BEFORE INSERT and BEFORE UPDATE trigger to check the age value before inserting/updating data into the person table:

delimiter //

CREATE TRIGGER person\_bi BEFORE INSERT

ON person

FOR EACH ROW

IF NEW.age < 18 THEN

SIGNAL SQLSTATE '50001' SET MESSAGE\_TEXT = 'A pessoa precisa ter mais que 18 anos';

END IF; //

delimiter ;

delimiter //

CREATE TRIGGER person\_bu BEFORE UPDATE

ON person

FOR EACH ROW

IF NEW.age < 18 THEN

SIGNAL SQLSTATE '50001' SET MESSAGE\_TEXT = 'A pessoa precisa ter mais que 18 anos';

END IF; //

delimiter ;

Inserting data activates the trigger and checks the value of age before committing the information:

INSERT INTO person VALUES ('John', 14);



The console displays the descriptive error message. The data does not insert into the table because of the failed trigger check.

### Create an AFTER INSERT Trigger

Create an AFTER INSERT trigger with:

CREATE TRIGGER <trigger name> AFTER INSERT

ON <table name>

FOR EACH ROW

<trigger body>;

The AFTER INSERT trigger is useful when the entered row generates a value needed to update another table.

AFTER INSERT Trigger Example

Inserting a new row into the person table does not automatically update the average in the average\_age table. Create an AFTER INSERT trigger on the person table to update the average\_age table after insert:

delimiter //

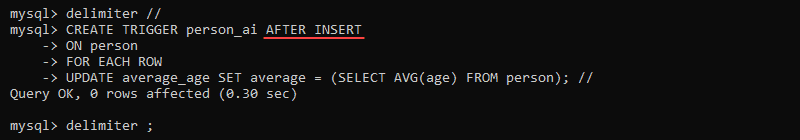
CREATE TRIGGER person\_ai AFTER INSERT

ON person

FOR EACH ROW

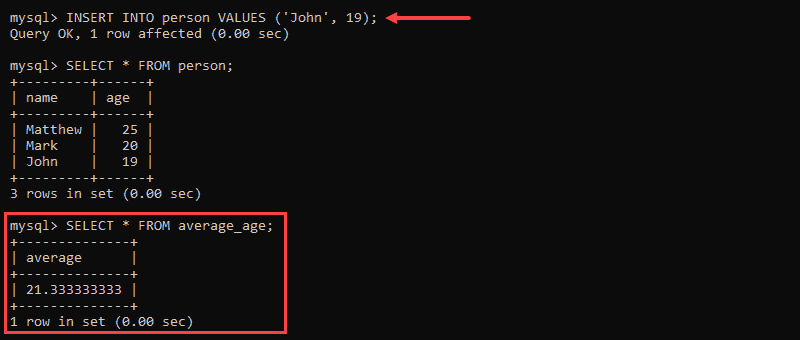
UPDATE average\_age SET average = (SELECT AVG(age) FROM person); //

delimiter ;



Inserting a new row into the person table activates the trigger:

INSERT INTO person VALUES ('John', 19);



The data successfully commits to the person table and updates the average\_age table with the correct average value.

### Create a BEFORE UPDATE Trigger

Make a BEFORE UPDATE trigger with:

CREATE TRIGGER <trigger name> BEFORE UPDATE

ON <table name>

FOR EACH ROW

<trigger body>;

The BEFORE UPDATE triggers go together with the BEFORE INSERT triggers. If any restrictions exist before inserting data, the limits should be there before updating as well.

BEFORE UPDATE Trigger Example

If there is an age restriction for the person table before inserting data, the age restriction should also exist before updating information. Without the BEFORE UPDATE trigger, the age check trigger is easy to avoid. Nothing restricts editing to a faulty value.

Add a BEFORE UPDATE trigger to the person table with the same body as the BEFORE INSERT trigger:

delimiter //

CREATE TRIGGER person\_bu BEFORE UPDATE

ON person

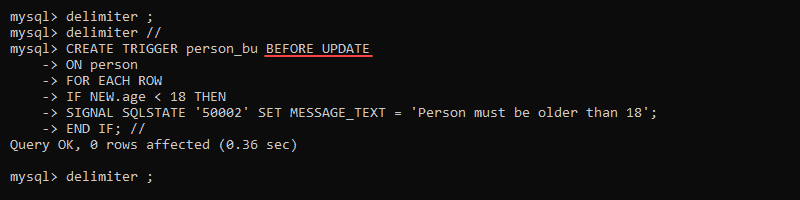
FOR EACH ROW

IF NEW.age < 18 THEN

SIGNAL SQLSTATE '50002' SET MESSAGE\_TEXT = 'Person must be older than 18.';

END IF; //

delimiter ;



Updating an existing value activates the trigger check:

UPDATE person SET age = 17 WHERE name = 'John';



Updating the age to a value less than 18 displays the error message, and the information does not update.

### Create an AFTER UPDATE Trigger

Use the following code block to create an AFTER UPDATE trigger:

CREATE TRIGGER <trigger name> AFTER UPDATE

ON <table name>

FOR EACH ROW

<trigger body>;

The AFTER UPDATE trigger helps keep track of committed changes to data. Most often, any changes after inserting information also happen after updating data.

AFTER UPDATE Trigger Example

Any successful updates to the age data in the table person should also update the intermediate average value calculated in the average\_age table.

Create an AFTER UPDATE trigger to update the average\_age table after updating a row in the person table:

delimiter //

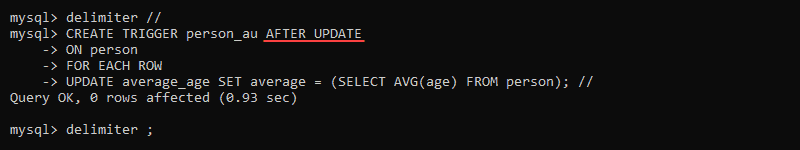
CREATE TRIGGER person\_au AFTER UPDATE

ON person

FOR EACH ROW

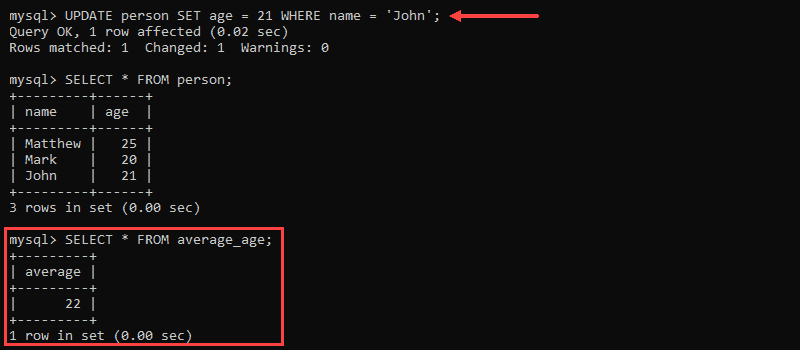
UPDATE average\_age SET average = (SELECT AVG(age) FROM person); //

delimiter ;



Updating existing data changes the value in the person table:

UPDATE person SET age = 21 WHERE name = 'John';



Updating the table person also updates the average in the average\_age table.

### Create a BEFORE DELETE Trigger

To create a BEFORE DELETE trigger, use:

CREATE TRIGGER <trigger name> BEFORE DELETE

ON <table name>

FOR EACH ROW

<trigger body>;

The BEFORE DELETE trigger is essential for security reasons. If a parent table has any children attached, the trigger helps block deletion and prevents orphaned tables. The trigger also allows archiving data before deletion.

BEFORE DELETE Trigger Example

Archive deleted data by creating a BEFORE DELETE trigger on the table person and insert the values into the person\_archive table:

delimiter //

CREATE TRIGGER person\_bd BEFORE DELETE

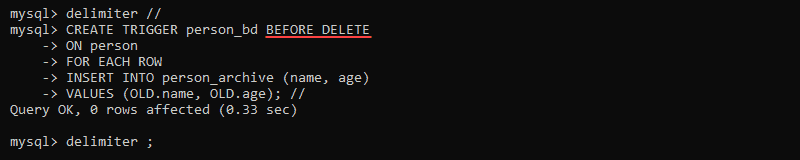
ON person

FOR EACH ROW

INSERT INTO person\_archive (name, age)

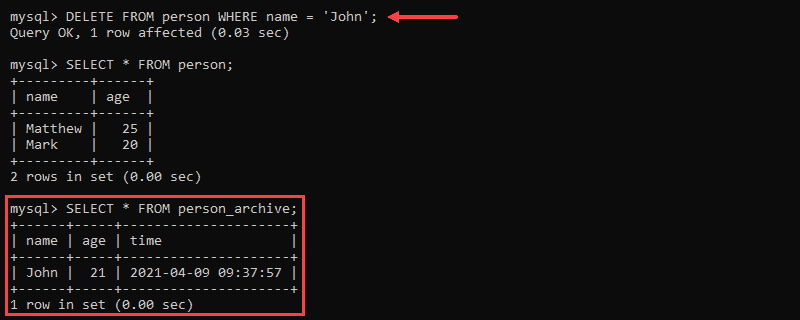
VALUES (OLD.name, OLD.age); //

delimiter ;



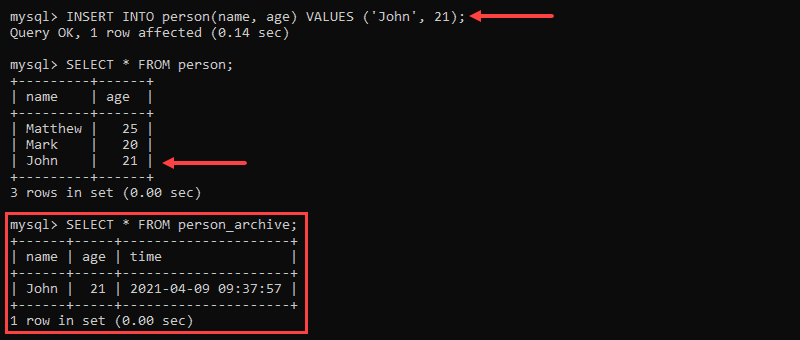
Deleting data from the table person archives the data into the person\_archive table before deleting:

DELETE FROM person WHERE name = 'John';



Inserting the value back into the person table keeps the log of the deleted data in the person\_archive table:

INSERT INTO person VALUES ('John', 21);



The BEFORE DELETE trigger is useful for logging any table change attempts.

### Create an AFTER DELETE Trigger

Make an AFTER DELETE trigger with:

CREATE TRIGGER <trigger name> AFTER DELETE

ON <table name>

FOR EACH ROW

<trigger body>;

The AFTER DELETE triggers maintain information updates that require the data row to disappear before making the updates.

AFTER DELETE Trigger Example

Create an AFTER DELETE trigger on the table person to update the average\_age table with the new information:

delimiter //

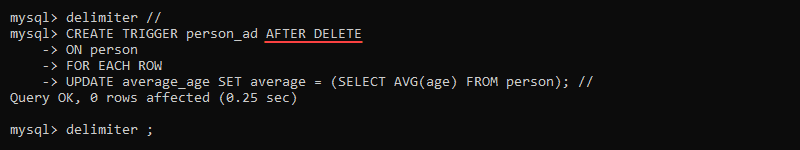
CREATE TRIGGER person\_ad AFTER DELETE

ON person

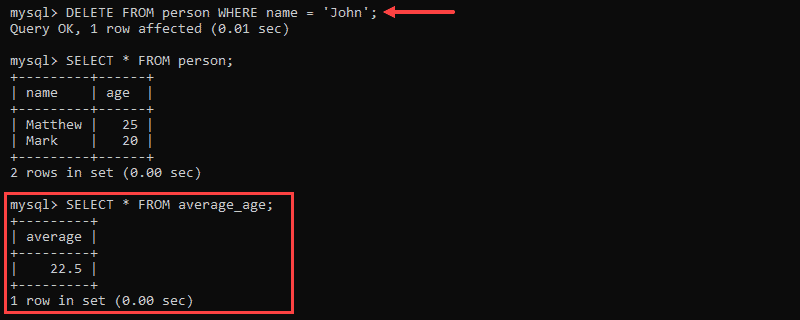
FOR EACH ROW

UPDATE average\_age SET average = (SELECT AVG(person.age) FROM person); //

delimiter ;



Deleting a record from the table person updates the average\_age table with the new average:



Without the AFTER DELETE trigger, the information does not update automatically.

### Create Multiple Triggers

MySQL does not support having multiple triggers fire at the same time. However, adding multiple logical operations to the same trigger is possible. Use the BEGIN and END delimiters to indicate the trigger body:

CREATE TRIGGER <trigger name> <trigger time > <trigger event>

ON <table name>

FOR EACH ROW

BEGIN

<trigger body>;

END;

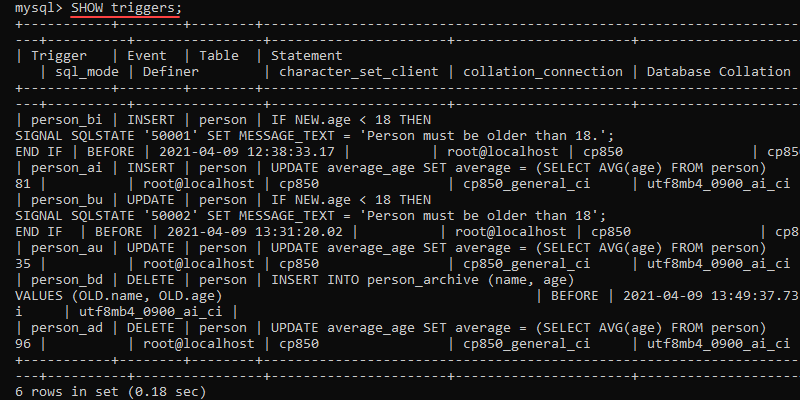
Make sure to change the default delimiter before creating a trigger with multiple operations.

### Show Triggers

List all the triggers in a database with:

SHOW triggers;

The output shows a list of all the triggers, including the name and statement contents:



Other information displays as well, such as the creation time and the user who created the trigger.

Conclusion

MySQL triggers provide further validation and control of data before or after specific events happen. Whether you are trying to prevent an error or add restrictions for consistency, triggers help control data input, update, and removal.

Keep in mind the trigger checks happen row-wise, which causes performance to slow down with massive queries. For more materials on this topic, check out our article on [how to improve MySQL performance with tuning](https://phoenixnap.com/kb/improve-mysql-performance-tuning-optimization).

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