MySQL – TRIGGERS

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

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:

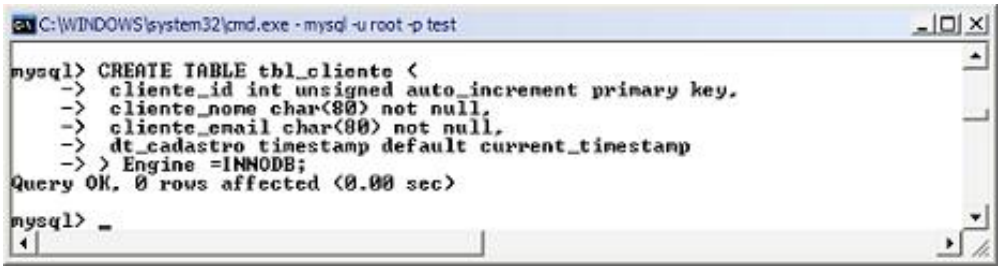


Figura 01. Criando a tabela de testes – tbl_cliente.

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:

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

- 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 (BEFORE) ou depois (AFTER) da atualização 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**)

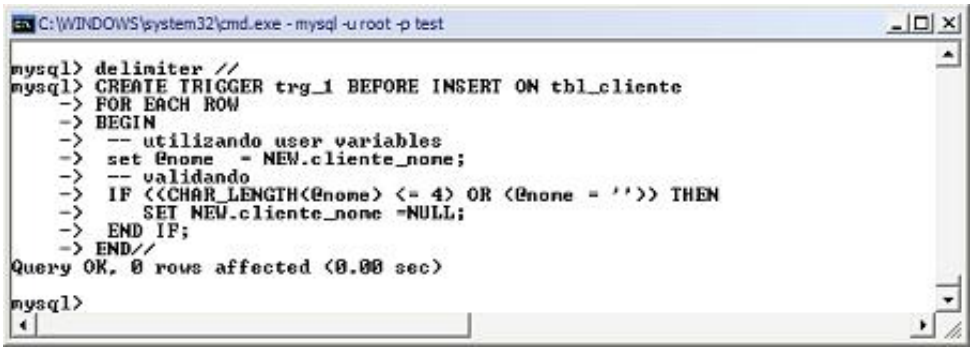


Figura 02 – TRIGGER para conferir se o CHAR_LENTH() de nome é maior que 4 (quatro).

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**.

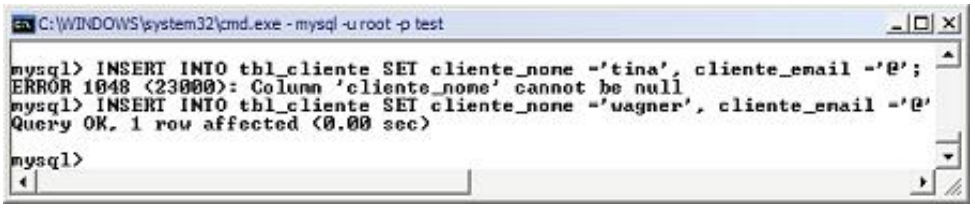
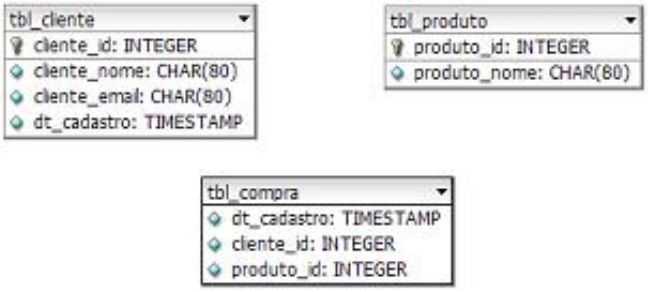


Figura 03 – O INSERT falhará após uma tentativa de inserir um valor não permitido.

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**):

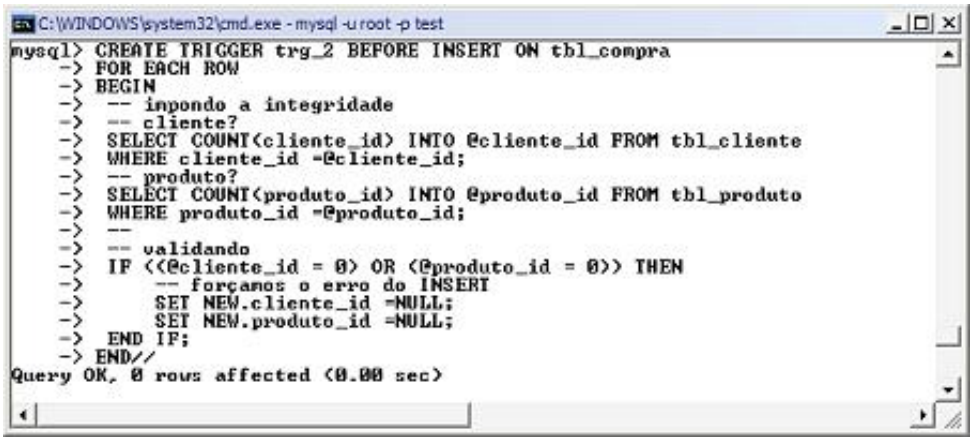


Figura 04 – TRIGGER para impor integridade referencial ao campos que se relacionam com outras tabelas.

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**.

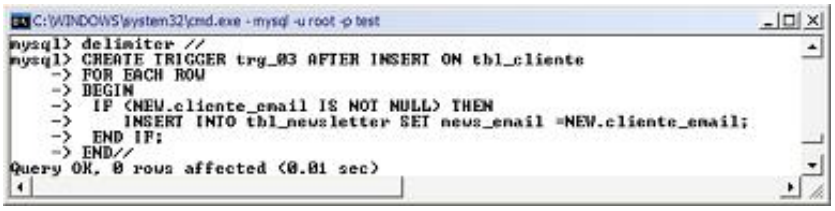


Figura 07 – Inserção automática de e-mail em outra tabela.

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.

Conclusão

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.

Happy MySQL'ing

por Wagner Bianchi

Wagner Bianchi é Tecnólogo em Gerenciamento de Bancos de Dados pela Faculdade Infórium de Tecnologia, Pós-Graduando em Administração Estratégica de Empresas (Executivo Jr.) pela Fundação Getúlio Vargas no Minas Business Institute, Consultor em Desenvolvimento de Sistemas pela INFODBA C&T, empresa onde é Sócio-Diretor e Analista de Sistemas em projetos OpenSource e também alocado em várias empresas. Atua também como Analista de Sistemas em Projetos da UFMG (Inova Tecnologia) onde desenvolve produtos voltados para a área de CMS e Gestão da Informação. Certificado MySQL 5.0 Developer I (CMDEV I) e MCDBA.

www.devmedia.com.br/articles/viewcomp.asp?comp=8088