Triggers

Banco de Dados

Prof. Igor Avila Pereira
igor.pereira@riogrande.ifrs.edu.br

Divisão de Computação
Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Sul (IFRS)
Câmpus Rio Grande

Agenda

Triggers (Gatilhos)

Uma função (stored procedure) pode ser criada para executar antes (**BEFORE**) ou depois (**AFTER**) de um **INSERT**, **UPDATE** ou **DELETE**

- Uma vez para cada registro modificado
- Ou por instrução SQL.

Logo que ocorre um desses eventos, a função é disparada automaticamente para tratar o evento.

Esta função deve ser declarada como uma função que não recebe argumentos e que retorna o tipo **TRIGGER**.

Criação

Após criar a função deseja, estabelecemos o gatilho pelo comando **CREATE TRIGGER**.

Sintaxe:

```
CREATE TRIGGER nome_trigger
{ BEFORE | AFTER } { evento [ OR ... ] }
ON tabela [ FOR [ EACH ] { ROW | STATEMENT } ]
EXECUTE PROCEDURE nome_da_função ( argumentos )
```

***Uma função pode ser utilizada por vários gatilhos.**

Triggers (Gatilhos)

A trigger fica associado à tabela especificada e executa a função especificada **nome_da_função** quando determinados eventos ocorrerem.

O gatilho pode ser especificado para disparar:

- Antes das restrições serem verificadas e o comando INSERT, UPDATE ou DELETE ser tentado;
- Ou após a operação estar completa (após as restrições serem verificadas e o INSERT, UPDATE ou DELETE ter completado).

Tipos de Triggers

Trigger-por-instrução

É disparada somente uma vez quando a instrução é executada.

- Deve sempre retornar NULL.

Trigger-por-linha

É disparada uma vez para cada registro afetado pela instrução que disparou a trigger.

- Pode retornar uma linha da tabela.

Variáveis disponíveis

Variáveis disponíveis no ambiente de programação:

- **NEW**: Tipo de dado RECORD; variável contendo a nova linha do banco de dados, para as operações de INSERT/UPDATE.
- **OLD**: Tipo de dado RECORD; variável contendo a antiga linha do banco de dados, para as operações de UPDATE/DELETE.

Variáveis disponíveis

- **TG_NAME:** Tipo de dado **name**; variável contendo o nome do gatilho disparado.
- **TG_WHEN:** Tipo de dado **text**; uma cadeia de caracteres contendo BEFORE ou AFTER, dependendo da definição do gatilho.
- **TG_LEVEL**: Tipo de dado text; uma cadeia de caracteres contendo ROW ou STATEMENT, dependendo da definição do gatilho.
  - **STATEMENT**: Para que execute somente uma vez por comando SQL.
  - **ROW**: Para que execute a trigger para cada linha afetada por um comando.

Variáveis disponíveis

- **TG_OP**: Tipo de dado text; uma cadeia de caracteres contendo INSERT, UPDATE, ou DELETE, informando para qual operação o gatilho foi disparado.
- **TG_RELID:** Tipo de dado oid; o ID de objeto da tabela que causou o disparo do gatilho.
- **TG_RELNAME**: Tipo de dado name; o nome da tabela que causou o disparo do gatilho (**obsoleto**).
- **TG_TABLE_NAME**: Nome da tabela que disparou o gatilho.

Variáveis disponíveis

- **TG_NAME_SCHEMA**: Nome do schema onde está a tabela que disparou a trigger.
- **TG_NARGS**: Tipo de dado integer; o número de argumentos fornecidos ao procedimento de gatilho na instrução CREATE TRIGGER.
- **TG_ARGV[]**: Tipo de dado matriz de text; os argumentos da instrução CREATE TRIGGER. O contador do índice começa por 0. Índices inválidos (menor que 0 ou maior ou igual a tg_nargs) resultam em um valor nulo.

Ordem de Execução

Ordem de Execução

Podemos ter mais de uma trigger associada ao mesmo evento e momento, neste caso a ordem de execução das triggers é definida pela ordem alfabética de seus nomes.

Triggers Recursivas

Se uma função de trigger executar comandos SQL, estes comandos podem disparar triggers novamente.

- Isto é conhecido como cascatear triggers.
- Não existe limitação direta do número de níveis de cascateamento.

É possível que o cascateamento cause chamadas recursivas da mesma trigger;

Exemplo

Uma trigger para INSERT pode executar um comando que insere uma linha adicional na mesma tabela, fazendo com que o trigger para INSERT seja disparado novamente. É responsabilidade do programador evitar recursões infinitas nestes casos.

Alterando uma Trigger

Sintaxe:

```
ALTER TRIGGER nome_trigger
        ON tabela RENAME TO novo_nome
```

- **nome**: é nome do gatilho existente a ser alterado.
- **tabela**: é o nome da tabela onde o gatilho atua.
- **novo_nome**: é o novo nome do gatilho.

Excluindo uma Trigger

Para excluir uma trigger basta executar o comando abaixo:

```
DROP TRIGGER nome_trigger
        ON tabela [ CASCADE | RESTRICT ]
```

- **nome**: é o nome do gatilho a ser removido.
- **tabela**: é o nome da tabela para a qual o gatilho está definido.
- **[CASCADE — RESTRICT]** indica se ao remover a trigger vamos remover também todos os objetos que dependem dela **(CASCADE)** ou recusaremos sua exclusão **(RESTRICT)**.

Desabilitando uma Trigger

Para desabilitar uma trigger execute o comando abaixo:

```
ALTER TABLE tabela DISABLE TRIGGER nome_trigger
```

Para desabilitar todas as triggers da tabela, execute o seguinte comando:

```
ALTER TABLE nome_tabela DISABLE TRIGGER ALL
```

Habilitar uma Trigger

Para habilitar uma trigger basta alterar o parâmetro DISABLE para ENABLE, observe abaixo:

```
ALTER TABLE tabela ENABLE TRIGGER nome_trigger
```

Agora para habilitar todas as triggers da tabela:

```
ALTER TABLE tabela ENABLE TRIGGER nome_trigger
```

Obtendo informações sobre as triggers

Opção 1:

```
SELECT * FROM INFORMATION_SCHEMA.TRIGGERS
```

Opção 2:

```
SELECT * FROM PG_CATALOG.PG_TRIGGER
```

Exemplos

```
CREATE TABLE empregados(
 codigo int4 NOT NULL,
 nome VARCHAR,
 salario int4,
 departamento_cod int4,
 ultima_data TIMESTAMP,
 ultimo_usuario VARCHAR(50),
 CONSTRAINT empregados_pkey PRIMARY KEY (codigo) )

 CREATE FUNCTION empregados_gatilho() RETURNS TRIGGER AS $empregados_gatilho$
   BEGIN
       -- Verificar se foi fornecido o nome e o salário do empregado
       IF NEW.nome IS NULL THEN
           RAISE EXCEPTION 'O nome do empregado não pode ser nulo';
       END IF;
       IF NEW.salario IS NULL THEN
           RAISE EXCEPTION '% não pode ter um salário nulo', NEW.nome;
       END IF;
       -- Quem paga para trabalhar?
       IF NEW.salario < 0 THEN
           RAISE EXCEPTION '% não pode ter um salário negativo', NEW.nome;
       END IF;
       -- Registrar quem alterou a folha de pagamento e quando
       NEW.ultima_data := 'now';
       NEW.ultimo_usuario := CURRENT_USER;
       RETURN NEW;
   END;
 $empregados_gatilho$ LANGUAGE plpgsql;

 CREATE TRIGGER empregados_gatilho BEFORE INSERT OR UPDATE ON empregados
   FOR EACH ROW EXECUTE PROCEDURE empregados_gatilho();
```

Exemplos

```
INSERT INTO empregados (codigo,nome, salario) VALUES (5,'João',1000);
INSERT INTO empregados (codigo,nome, salario) VALUES (6,'José',1500);
INSERT INTO empregados (codigo,nome, salario) VALUES (7,'Maria',2500);
SELECT * FROM empregados;
INSERT INTO empregados (codigo,nome, salario) VALUES (5,NULL,1000);
```

Figura: Casos de Teste

Exemplos

```
CREATE TABLE empregados (
  nome  VARCHAR NOT NULL,
  salario     INTEGER
);
CREATE TABLE empregados_audit(
  operacao    CHAR(1)   NOT NULL,
  usuario     VARCHAR      NOT NULL,
  DATA        TIMESTAMP NOT NULL,
  nome    VARCHAR      NOT NULL,
  salario     INTEGER
);
CREATE OR REPLACE FUNCTION processa_emp_audit() RETURNS TRIGGER AS $emp_audit$
  BEGIN
      -- Cria uma linha na tabela emp_audit para refletir a operação
      -- realizada na tabela emp. Utiliza a variável especial TG_OP
      -- para descobrir a operação sendo realizada.
      IF (TG_OP = 'DELETE') THEN
          INSERT INTO emp_audit SELECT 'E', USER, now(), OLD.*;
          RETURN OLD;
      ELSIF (TG_OP = 'UPDATE') THEN
          INSERT INTO emp_audit SELECT 'A', USER, now(), NEW.*;
          RETURN NEW;
      ELSIF (TG_OP = 'INSERT') THEN
          INSERT INTO emp_audit SELECT 'I', USER, now(), NEW.*;
          RETURN NEW;
      END IF;
      RETURN NULL; -- o resultado é ignorado uma vez que este é um gatilho AFTER
  END;
  $emp_audit$ LANGUAGE plpgsql;

CREATE TRIGGER emp_audit
  AFTER INSERT OR UPDATE OR DELETE ON empregados
    FOR EACH ROW EXECUTE PROCEDURE processa_emp_audit();
```

Exemplos

```sql
INSERT INTO empregados (nome, salario) VALUES ('João',1000);
INSERT INTO empregados (nome, salario) VALUES ('José',1500);
INSERT INTO empregados (nome, salario) VALUES ('Maria',250);
UPDATE empregados SET salario = 2500 WHERE nome = 'Maria';
DELETE FROM empregados WHERE nome = 'João';
SELECT * FROM empregados;
SELECT * FROM empregados_audit;
```

Figura: Casos de Teste

Exemplos

```
CREATE TABLE empregados (
  codigo          serial  PRIMARY KEY,
  nome    VARCHAR    NOT NULL,
  salario     INTEGER
);
CREATE TABLE empregados_audit(
  usuario         VARCHAR      NOT NULL,
  DATA            TIMESTAMP NOT NULL,
  id              INTEGER   NOT NULL,
  coluna          text      NOT NULL,
  valor_antigo    text      NOT NULL,
  valor_novo      text      NOT NULL
);
CREATE OR REPLACE FUNCTION processa_emp_audit() RETURNS TRIGGER AS $emp_audit$
  BEGIN
      -- Não permitir atualizar a chave primária
      IF (NEW.codigo <> OLD.codigo) THEN
          RAISE EXCEPTION 'Não é permitido atualizar o campo codigo';
      END IF;
      -- Inserir linhas na tabela emp_audit para refletir as alterações
      -- realizada na tabela emp.
      IF (NEW.nome <> OLD.nome) THEN
         INSERT INTO emp_audit SELECT CURRENT_USER, CURRENT_TIMESTAMP,
                     NEW.id, 'nome', OLD.nome, NEW.nome;
      END IF;
      IF (NEW.salario <> OLD.salario) THEN
         INSERT INTO emp_audit SELECT CURRENT_USER, CURRENT_TIMESTAMP,
                     NEW.codigo, 'salario', OLD.salario, NEW.salario;
      END IF;
      RETURN NULL; -- o resultado é ignorado uma vez que este é um gatilho AFTER
  END;
$emp_audit$ LANGUAGE plpgsql;
```

Exemplos

```
CREATE TRIGGER emp_audit
AFTER UPDATE ON empregados
FOR EACH ROW EXECUTE PROCEDURE processa_emp_audit();



INSERT INTO empregados (nome, salario) VALUES ('João',1000);
INSERT INTO empregados (nome, salario) VALUES ('José',1500);
INSERT INTO empregados (nome, salario) VALUES ('Maria',2500);
UPDATE empregados SET salario = 2500 WHERE id = 2;
UPDATE empregados SET nome = 'Maria Cecília' WHERE id = 3;
UPDATE empregados SET codigo=100 WHERE codigo=1;
ERRO:  Não é permitido atualizar o campo codigo
SELECT * FROM empregados;
SELECT * FROM empregados_audit;
```

Figura: Casos de Teste

Exemplos

```
CREATE TABLE teste1 (
campoA INT NOT NULL,
campoB INT NOT NULL
);

CREATE TABLE teste2 (
campoC INT NOT NULL,
campoD INT NOT NULL
);

CREATE LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION insert_minha_function() RETURNS TRIGGER AS '
BEGIN
INSERT INTO teste2 (campoC, campoD) values (5,25);
RETURN new;
END
'LANGUAGE plpgsql;

CREATE TRIGGER insert_minha_trigger AFTER INSERT ON teste1
  FOR EACH ROW EXECUTE PROCEDURE insert_minha_function();

CREATE OR REPLACE FUNCTION delete_minha_function() RETURNS TRIGGER AS '
BEGIN
INSERT INTO teste2 (campoC, campoD) values (6,26);
RETURN old;
END
'LANGUAGE plpgsql;

CREATE TRIGGER delete_minha_trigger BEFORE DELETE ON teste1
  FOR EACH ROW EXECUTE PROCEDURE delete_minha_function();
```

Triggers

Banco de Dados

Prof. Igor Avila Pereira
igor.pereira@riogrande.ifrs.edu.br

Divisão de Computação
Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Sul (IFRS)
Câmpus Rio Grande