Artigo

Trabalhando com Triggers no PostgreSQL

Neste artigo veremos na prática a utilização das triggers no banco de dados PostgreSQL 9.4, além de aprendermos a utilizar as rules.

A medida que a base de dados vai crescendo, o mesmo ocorre com a sua complexidade, e devido a isto, fica difícil implementar ou mesmo solucionar problemas que o banco deve executar antes ou depois de um evento específico.

tabelas. Neste artigo aprenderemos a trabalhar com as triggers e sua relação com as Rules e as Views.

DevCast: **Você usa Triggers?**

Uma visão geral sobre as triggers

Triggers, em termos de banco de dados, são as operações realizadas de forma espontânea para eventos específicos. Quando tratamos dos eventos, estes podem ser tanto um `INSERT` quanto um `UPDATE`, ou mesmo um `DELETE`. Assim, podemos definir determinadas operações que serão realizadas sempre que o evento ocorrer.

Quando nos referirmos a uma operação com uma **trigger**, esta é conhecida por trigger de função ou trigger function. Lembre-se que trigger e função de trigger são duas coisas diferentes, onde a primeira pode ser criada utilizando a instrução `CREATE TRIGGER`, enquanto que a última é definida pelo comando `CREATE FUNCTION`. Em linhas gerais, com as triggers definimos qual tarefa executar, e com as triggers de função definimos como essa tarefa será realizada.

Ao temos uma grande quantidade de acessos ao banco de dados por múltiplas aplicações, a utilização das triggers é de grande utilidade, e com isso, podemos manter a integridade de dados complexos, além de podermos acompanhar as

ser definidas em linguagens compatíveis ao PostgreSQL, como `PL/pgSQL`, `PL/Python`, **PL/Java** dentre outros. Para esse artigo usaremos a linguagem `PL/pgSQL`.

```
1  CREATE OR REPLACE FUNCTION trigger_function_name
2  RETURNS trigger AS $ExemploFuncao$
3  BEGIN
4  /* Aqui definimos nossos códigos.*/
5  RETURN NEW;
6  END;
7  $ExemploFuncao
```

Listagem 1. Criação da estrutura de uma trigger function

```
1  CREATE [ CONSTRAINT ] TRIGGER NAME { BEFORE | AFTER | INSTEAD OF } { e
2  ON table_NAME
3  [ FROM referenced_table_NAME ]
4  [ NOT DEFERRABLE | [ DEFERRABLE ] { INITIALLY IMMEDIATE | INITIALLY DE
5  [ FOR [ EACH ] { ROW | STATEMENT } ]
6  [ WHEN ( condition ) ]
7  EXECUTE PROCEDURE function_NAME ( arguments )
8  -- Quando um evento for declarado com:
9  INSERT
10 UPDATE [ OF column_NAME [, ... ] ]
11 DELETE
12 TRUNCATE
```

Listagem 2. Sintaxe de uma trigger simples

especial chamada de `TriggerData`. Repare também que o seu tipo de retorno é a trigger, onde ela é chamada automaticamente no momento da ocorrência dos eventos (que podem ser `INSERT`, `UPDATE`, `DELETE` ou `TRUNCATE`). Com o PostgreSQL temos dois tipos de trigger disponíveis: trigger de nível de linha (`row-level Trigger`) e a trigger a nível de instrução (statement level trigger). Ambos são especificados com a utilização das cláusulas `FOR EACH ROW` (nível gatilho de linha) e `FOR EACH STATEMENT`, respectivamente. A utilização delas pode ser definida de acordo com a quantidade de vezes que a trigger deverá ser executada. Por exemplo, se uma instrução `UPDATE` for executada, e esta afetar seis linhas, temos que a trigger de nível de linha será executada seis vezes, enquanto que a trigger a nível de instrução será chamada apenas uma vez por instrução SQL.

Quando utilizamos triggers podemos conectá-las tanto a tabelas quanto a Views, de forma que as triggers são executadas para as tabelas em duas situações: `BEFORE` e `AFTER`, para qualquer uma das instruções DML (`INSERT`, `UPDATE`, `DELETE`), além de também possibilitar a sua execução utilizando a declaração `TRUNCATE`.

Quando temos a trigger definida com a instrução `INSTEAD OF` podemos utilizar as DML's para as Views. As triggers serão disparadas antes ou depois das instruções DML, mas podem ser definidas apenas a nível de instrução. Já quando utilizamos o `INSTEAD OF` nas instruções DML, podemos executá-las apenas a nível de linha.

Com relação aos demais parâmetros, temos o `NAME`, que é utilizado para atribuirmos um nome para a trigger, o qual deve ser distinto das demais triggers criadas para a mesma tabela. A instrução table_NAME apresenta o nome da tabela em uso.

A expressão condition é uma expressão booleana que determina se a trigger function será executada. Se a condição `WHEN` for especificada, a função será chamada se a condição retornar true. Além disso, ela pode ser referida a colunas que contenham os valores antigos e se quer passar os novos valores. Para isso são utilizadas as instruções `OLD.column_NAME` ou `NEW.column_NAME`, respectivamente. Lembre-se que as `function_names` são funções fornecidas pelos usuários.

Por último, temos os arguments, que são listas opcionais de argumentos separados por vírgulas que podem ser fornecidos para a função quando a trigger for executada.

Para demonstrarmos o procedimento de criação das triggers e sua utilização criaremos alguns exemplos simples.

Para isso, criaremos uma nova base de dados, a qual chamaremos de `DbTeste` e em seguida criaremos uma tabela `Funcionarios` que conterá os campos da **Listagem 3**. Neste nosso exemplo queremos manter atualizados todos os registros adicionados para uma possível `funcionarios_auditoria`.

```
1  CREATE TABLE funcionarios
2  (
3    nome character varying(100) NOT NULL,
4    email character varying(200) NOT NULL,
5    telefone character(14) NOT NULL,
6    profissao character varying(150) NOT NULL,
7    endereco character varying(100) NOT NULL,
8    salario real
```


Após a criação da nossa tabela principal criaremos uma nova tabela com o nome de `funcionarios_funcionarios_auditoria`, que será responsável por manter o histórico das alterações realizadas nos registros, como podemos ver na **Listagem 4**.

```
1  CREATE TABLE funcionarios_funcionarios_auditoria (
2      codigo_func INT NOT NULL,
3      data_alteracao TEXT NOT NULL
4  );
```

Listagem 4. Criação da tabela `funcionarios_funcionarios_auditoria`

Observe que há apenas dois campos: o código do funcionário e a data da alteração/criação do registro, que receberá uma data no formato Timestamp no momento em que o registro for criado na tabela de Funcionários.

Para darmos seguimento aos nossos testes iremos inserir alguns dados, como podemos ver na **Listagem 5**.

```
1  INSERT INTO FUNCIONARIOS (codigo, nome, email, telefone, profissao, en
2  VALUES (1, 'Edson Dionisio', 'edson.dionisio@gmail.com', '(81)99740280
3  2000.00);
4  INSERT INTO FUNCIONARIOS (codigo, nome, email, telefone, profissao, en
5  VALUES (2, 'Marilia Késsia', 'mkessia.dionisio@gmail.com', '(81)997402
6  desenvolvimento', 'rua testes', 6000.00);
```


```
13 INSERT INTO FUNCIONARIOS (codigo, nome, email, telefone, profissao, en
14 VALUES (5, 'Maria das Dores', 'maria@gmail.com', '(81)997407845', 'Sec
15 'rua testes', 1800.00);
```

Listagem 5. Inserindo dados na tabela de `funcionarios`

As triggers functions recebem, através de uma entrada especial, uma estrutura `TriggerData`, que possui um conjunto de variáveis locais que podemos usar nas nossas triggers functions. Dentre as variáveis presentes nesta estrutura temos as variáveis `OLD` e `NEW`, além de outras que começam com o prefixo `TG_`, como `TG_TABLE_NAME`.

A variável `NEW` é do tipo `RECORD` e contém uma nova linha a ser armazenada com base nos comandos `INSERT/UPDATE` das triggers a nível de linha.

Já a variável OLD também é do tipo `RECORD` e armazena a linha antiga quando utilizada com os comandos `UPDATE/DELETE` nas triggers de linha.

Após a criação das nossas tabelas definiremos uma trigger function, a qual chamaremos de `funcionario_log_func`, e será responsável por registrar as alterações feitas na tabela de `funcionarios_auditoria` depois de uma operação de `INSERT` na tabela funcionários, como apresentada pela **Listagem 6**.

```
1 CREATE OR REPLACE FUNCTION funcionario_log_func()
```


```
8  RETURN NEW;
9  END;
10 $teste_trigger
```

Listagem 6. Criação da trigger function `funcionario_log_func`

Com a nossa função criada, definiremos agora a nossa trigger e em seguida, a associaremos a tabela de funcionários, como podemos ver na **Listagem 7**.

```
1 CREATE TRIGGER log_trigger
2 AFTER INSERT ON funcionarios
3 FOR EACH ROW
4 EXECUTE PROCEDURE funcionario_log_func();
```

Listagem 7. Criação da trigger log_trigger

Ao inserirmos um novo registro na nossa tabela de funcionários, podemos ver que um novo registro foi criado também na tabela de `funcionarios_auditoria`.

Para um exemplo um pouco mais complexo criaremos uma trigger contendo as três operações DML's contidas numa mesma trigger function. Para isso, realizaremos inicialmente uma alteração na nossa `tabelafuncionarios_auditoria`, onde adicionaremos uma nova coluna `operação_realizada` para armazenar o nome da operação realizada.

Após a exclusão da tabela, a criaremos novamente, mas contendo as seguintes colunas da **Listagem 8**.

```
1  CREATE TABLE funcionarios_auditoria
2  (
3  log_id INT NOT NULL,
4  data_criacao TEXT NOT NULL,
5  operacao_realizada CHARACTER VARYING
6  );
```

Listagem 8. Recriando a tabela `funcionarios_auditoria`

Neste momento podemos utilizar o comando apresentado na **Listagem 9**para criar ou recriar a nossa trigger function.

```
1  CREATE OR REPLACE FUNCTION funcionario_log_function()
2  RETURNS trigger AS $BODY$
3  BEGIN
4  -- Aqui temos um bloco IF que confirmará o tipo de operação.
5  IF (TG_OP = 'INSERT') THEN
6  INSERT INTO funcionarios_auditoria (log_id, data_criacao, operacao_rea
7  VALUES (new.codigo_func, current_timestamp, ' Operação de inserção.
8  A linha de código ' || NEW.codigo_func || 'foi inserido');
9  RETURN NEW;
10 -- Aqui temos um bloco IF que confirmará o tipo de operação UPDATE.
```

```
16 RETURN NEW;
17 -- Aqui temos um bloco IF que confirmará o tipo de operação DELETE
18 ELSIF (TG_OP = 'DELETE') THEN
19 INSERT INTO funcionarios_auditoria (log_id, data_criacao, operacao_rea
20 VALUES (OLD.codigo_func, current_timestamp, 'Operação DELETE. A linha
21 || OLD.codigo_func || ' foi excluída ');
22 RETURN OLD;
23 END IF;
24 RETURN NULL;
25 END;
26 $BODY
```

Listagem 9. Recriando a trigger function

Agora criaremos a trigger que será vinculada a tabela de funcionários, como podemos ver na **Listagem 10**.

```
1 CREATE TRIGGER trigger_log_todas_as_operacoes
2 AFTER INSERT OR UPDATE OR DELETE ON funcionarios
3 FOR EACH ROW
4 EXECUTE PROCEDURE funcionario_log_function();
```

Listagem 10. Criação da tabela de funcionários

Para termos os resultados armazenados na nossa tabela utilizando as operações DML, utilizaremos as instruções de `INSERT`, `UPDATE` e `DELETE`, de acordo com a

```
3  'Tua desespero', 4500.00);
4  UPDATE funcionarios set nome = 'Caroline Dionisio' WHERE codigo_func =
5  DELETE FROM funcionarios WHERE codigo_func= 5;
```

Listagem 11. Utilizando as operações DML

Veja que a inserção dos registros na tabela de `funcionarios_auditoria` com as informações do código dos registros, data de atualização e o tipo de operação, foi realizada. Para que possamos ver os resultados basta utilizarmos a instrução `SELECT`, como apresentada a seguir:

```
1  SELECT log_id, data_criacao FROM funcionarios_auditoria;
```

Trabalhando com as Views e as triggers

Neste momento veremos um pouco sobre a utilização das Views em conjunto com as triggers. Para isso criaremos um novo exemplo com a tabela chamada `funcionario_view` e uma View chamada `view_funcionarios`. A tabela `funcionario_view` está na **Listagem 12.**

```
1  CREATE TABLE funcionario_view
```

Listagem 12. Criação da tabela `funcionario_view`

Com a tabela criada, vamos adicionar alguns dados de testes, como podemos ver na **Listagem 13**.

```
1 INSERT INTO funcionario_view VALUES (1, 'Edson', 'edson.dionisio@gmail
2 INSERT INTO funcionario_view VALUES (2, 'Marília', 'mkessia@teste.com'
3 INSERT INTO funcionario_view VALUES (3, 'Caroline', 'carol@teste.com')
4 INSERT INTO funcionario_view VALUES (4, 'Gustavo', 'gustavo@teste.com'
5 INSERT INTO funcionario_view VALUES (5, 'Maria', 'maria@teste.com');
```

Listagem 13. Inserção de registros de teste

Após a inserção dos dados de testes criamos a View com base nos dados inseridos da seguinte forma:

```
CREATE VIEW view_funcionarios AS SELECT * FROM funcionario_view;
```

Essa foi a forma mais simples de apresentarmos uma View, mas podemos fazer de forma diferente. Vamos criar uma View que será atualizada sempre que uma trigger for disparada, como podemos ver na **Listagem 14**.

```
3  BEGIN
4  IF (TG_OP = 'INSERT') THEN
5  INSERT INTO funcionario_view VALUES (NEW.codigo_func, NEW.nome, NEW.em
6  RETURN NEW;
7  END IF;
8  RETURN NULL;
9  END;
```

Listagem 14. Criando uma trigger de atualização de View

Em seguida, criamos uma trigger e vinculamos a View `view_funcionarios` com o código da **Listagem 15**.

```
1  CREATE TRIGGER dispara_trigger_func
2  INSTEAD OF INSERT ON view_funcionarios
3  FOR EACH ROW
4  EXECUTE PROCEDURE atualiza_view_trigger();
```

Listagem 15. Criação da trigger dispara_trigger_func

Para que possamos ver os resultados contidos na View `view_funcionarios` utilizamos a instrução SELECT da seguinte forma:

```
1  SELECT * FROM view_funcionarios;
```


```
1 | INSERT INTO view_funcionarios VALUES (6, 'Joao inserido através da viel
```

Mas o registro foi inserido na tabela funcionário `_view` , certo? Para vermos o resultado, utilizaremos o comando `SELECT>` e veremos que nosso registro foi inserido corretamente, como segue:

```
1 | SELECT * FROM funcionario_view;
```

Para que possamos ver todas as triggers que temos adicionadas no nosso banco de dados podemos utilizar o seguinte comando:

```
1 | SELECT * FROM pg_trigger;
```

Caso a intenção seja ver todas as triggers para uma tabela específica, então a query deverá ser de acordo com a apresentada a seguir:

O `relname` é funcionários, pois estamos utilizando esta base de dados.

Para finalizarmos, caso tenhamos interesse em excluir as triggers, podemos usar o seguinte comando:

```
1 | DROP TRIGGER trigger_log_todas_as_operacoes ON funcionarios;
```

Trabalhando com Rules no PostgreSQL

Podemos usar o sistema de regras do PostgreSQL, por exemplo, para reconfigurar um comando para `null` caso o valor de uma coluna não apareça na tabela, mas não podemos usar esse sistema na implementação de tipos de restrições, como a utilização de chaves estrangeiras.

Assim como com as triggers, podemos atualizar Views com as **rules** do PostgreSQL, o que será o nosso foco neste momento.

Quando temos implementações que podem ser realizadas tanto por rules quanto por triggers, a garantia de qual será melhor utilizada depende da utilização do banco de dados. No caso das triggers, estas são disparadas uma vez para cada linha afetada, o que difere de uma rule, pois esta modifica a consulta ou gera uma consulta adicional. Neste caso, se tivermos a intenção de atingir várias linhas, a utilização de uma rule será muito mais rápida do que a trigger.

dados presentes na **Listagem 13** para realizarmos nossos testes. Não se esqueça de criar a View `funcionarios_view` novamente.

Com toda essa etapa realizada, criaremos a nossa primeira rule para a inserção de registros com o nome de v `iew_funcionarios_rule` e o seu código está na **Listagem 16.**

```
1 CREATE RULE view_funcionarios_rule AS ON INSERT
2 TO view_funcionarios
3 DO INSTEAD (INSERT INTO funcionarios_view VALUES (NEW.codigo_func, NEW
```

Listagem 16. Criação da rule `view_funcionarios_rule`

Para conhecermos as rules criadas, podemos realizar uma consulta na tabela catalogo, chamada `pg_rewrite` , para vermos se nossa rule foi devidamente gerada utilizamos a seguinte instrução:

```
1 SELECT rulename FROM pg_rewrite WHERE rulename='view_funcionarios_rule
```

Feito isso, podemos realizar a inserção de um novo registro na View

Em seguida podemos realizar uma consulta na view para ver se os dados foram inseridos corretamente. A diferença aqui é que, ao invés de utilizarmos triggers, utilizamos rules para realizar a operação de inserção.

Com isso finalizamos este artigo, onde tivemos uma abordagem mais prática sobre utilização das triggers e um pouco sobre as rules, que tornam as implementações mais rápidas em comparação as triggers, em alguns casos.

Saiu na DevMedia!

- **Ajax com jQuery: Como evitar múltiplas requisições ao servidor:**
  Aprenda neste conteúdo como bloquear um botão utilizando jQuery, algo muito útil para evitar múltiplos cliques e, consequentemente, várias requisições ao servidor.
- **Como fazer um CRUD 1:N no Laravel:**
  Neste curso veremos como desenvolver um CRUD em Laravel com duas entidades que se relacionam de forma 1N.
- **ASP.NET MVC ViewBag: Como enviar dados do controller para a view:**
  Aprenda a utilizar o ViewBag, um mecanismo para efetuar o tráfego de dados do controller para a view no ASP.NET MVC.

Saiba mais sobre Triggers e SQL Server ;)

camada de segurança no banco de dados e aprenda a garantir a segurança necessária para executar o comando update.

- **Trabalhando com Triggers DML no Oracle:**
  Neste artigo trabalharemos com o conceito de Triggers no Oracle, onde veremos o porquê e quando devemos utilizá-las. Para isso utilizaremos exemplos simples de triggers compostas.
- **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.
- **Triggers no SQL Server: teoria e prática aplicada em uma situação real:**
  Veja neste artigo como trabalhar com triggers no SQL Server, entendendo seu funcionamento e como criá-los, com base em um exemplo que reflete uma situação real.

Por **Edson**
Em 2015

Você está em

Guia de PostgreSQL » Stored procedures, Functions e Triggers

Tecnologias

Exercicios

Cursos

Artigos

Revistas

Fale conosco

Trabalhe conosco

Assinatura para empresas

Assine agora