Banco de Dados Stored Procedures

PostgreSQL não temos necessariamente a função Stored Procedure (SP) definida, como podemos encontrar em diversos outros tipos de SGBD's. Elas são, na realidade, pequenos trechos de código armazenados do lado do servidor de uma base de dados. Ao contrário do que acontece em outras bases de dados, as SPs no Postgres são definidas como FUNCTIONS, assim como as triggers, tornando esse recurso um pouco mais complicado, dependendo inclusive do seu tipo de retorno. Essas funções possuem características importantes e diferentes, mas criadas de igual forma. Trabalhar com a criação destes pequenos trechos de código é, de certa forma, uma boa prática, pois podemos deixar códigos bastante complexos atuando do lado do servidor que poderão ser utilizados por várias aplicações, evitando assim a necessidade de replicá-los em cada uma dessas aplicações.

As Stored Procedures são definidas em três tipos distintos:

- **Linguagens Não-procedurais** – são linguagens que não requerem a escrita de uma lógica de programação tradicional. Neste caso, os usuários se concentram em definir a entrada e a saída das informações, ao invés das etapas do programa necessário em uma linguagem de programação procedural, como é o caso do C++ ou Java. Elas utilizam o SQL como uma linguagem, mas não possuem estruturas comuns às linguagens de programação, como é o caso da utilização das estruturas de repetição;
- **Procedurais** – as linguagens procedurais são linguagens de programação que especificam uma série de etapas e procedimentos bem estruturados dentro de seu contexto de programação. Elas possuem uma ordem sistemática de declarações, funções e comandos para a conclusão das tarefas. A mais conhecida dentre elas é a PL/pgSQL.
- **Linguagens externas** – são funções escritas normalmente na linguagem de programação C++, que traz vantagens ao se utilizar a linguagem com diversos recursos, onde podemos implementar algoritmos com maiores complexidades. Estas funções podem ser registradas e empacotadas no SGBD para utilizações futuras.

Sintaxe básica da criação de FUNCTIONS/STORED PROCEDURES.

```
CREATE [ OR REPLACE ] FUNCTION
      name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ] [, ...] ] )
      [ RETURNS rettype
        | RETURNS TABLE ( column_name column_type [, ...] ) ]
    { LANGUAGE lang_name
      | WINDOW
      | IMMUTABLE | STABLE | VOLATILE | [ NOT ] LEAKPROOF
      | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
      | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
      | COST execution_cost
      | ROWS result_rows
      | SET configuration_parameter { TO value | = value | FROM CURRENT }
      | AS 'definition'
      | AS 'obj_file', 'link_symbol'
    } ...
      [ WITH ( attribute [, ...] ) ]
```

- **Argmode** – este parâmetro define o modo do argumento, que pode ser IN, OUT, INOUT ou VARIADIC. Por padrão, o parâmetro utilizado é o IN;
- **ArgName** – aqui definimos o nome de um argumento;
- **Argtype** – definimos o tipo de dado dos argumentos da função, que podem ser base, compostos, ou de domínio, ou mesmo fazer referência ao tipo de uma coluna da tabela;
- **default_expr** – aqui podemos utilizar um valor padrão caso o parâmetro não tenha sido especificado. A expressão tem de ser coercible para o tipo de argumento do parâmetro, ou seja, os valores correspondentes utilizem a mesma representação interna;
- **rettype** – é o tipo de dados de retorno. Quando existem parâmetros OUT ou INOUT, a cláusula RETURNS pode ser omitida. O modificador SETOF indica que a função irá retornar um conjunto de itens, ao invés de um único item;
- **column_name** – é o nome de uma coluna de saída na sintaxe do RETURNS TABLE;
- **column_type** - Tipo de dados de uma coluna de saída na sintaxe RETURNS TABLE;
- **lang_name** – é o nome da linguagem na qual a função é implementada, podendo esta ser SQL, C, ou o nome de uma linguagem procedural definida pelo usuário;
- **WINDOW** – Este atributo indica que a função é uma função de janela ao invés de ser uma função simples. Este é útil para funções escritas na linguagem C. e não pode ser alterado ao substituirmos uma definição de função existente;
- **IMMUTABLE, STABLE, VOLATILE** - Estes atributos informam ao otimizador de consulta sobre o comportamento da função. Caso nenhum tenha sido definido, o VOLATILE é a opção padrão, ou

seja,pode fazer qualquer coisa, incluindo a modificação do banco de dados. Já STABLE não pode modificar o banco de dados e retorna os mesmos argumentos para todas as linhas dentro de uma única instrução, enquanto que IMMUTABLE também não pode modificar o banco de dados e retorna os mesmos mesmos argumentos sempre;

- **CALLED ON NULL INPUT, RETURNS NULL ON NULL INPUT, STRICT – o CALLED ON NULL INPUT** é utilizado por padrão para indicar que a função é chamada normalmente quando algum de seus argumentos for nulo. Os outros dois tipos indicam que a função sempre retornará nulo sempre que qualquer um dos seus argumentos for nulo.
- **[EXTERNAL] SECURITY INVOKER, [EXTERNAL] SECURITY DEFINER – o SECURITY INVOKER** indica que a função deve ser executada com os privilégios do usuário que o chama, sendo este definido por padrão. Já o SECURITY DEFINER especifica que a função deve ser executada com os privilégios do usuário que o criou;
- **result_rows** – Retorna um número de linhas como resultado da função.

A principal vantagem de usar as Stored Procedures é a redução do número de solicitações feitas pelo servidor da aplicação ao banco de dados, de forma a termos as instruções SQL emitidas através de uma única chamada de uma função para a obtenção do resultado esperado, consequentemente temos um aumento de desempenho da aplicação, além da possível reutilização do mesmo código em diversas aplicações.

Exemplos:

Criando uma Stored Procedure que não retorna valor.

```
CREATE OR REPLACE FUNCTION InsereFuncionario(codigo INTEGER,
nome VARCHAR(100), email VARCHAR(150) , telefone VARCHAR(15) ,
cidade VARCHAR(50) , estado VARCHAR(2))

      RETURNS void AS $
      BEGIN
        INSERT INTO "EmpresaDevmedia".tb_funcionarios
         VALUES (codigo, nome, email, telephone, cidade, estado);
      END;
      $ LANGUAGE 'plpgsql';
```

Com nossa SP criada iremos realizar um teste de inserção para que possamos vê-la em funcionamento, o que podemos fazer usando a seguinte instrução:

```
SELECT insereFuncionario(5, 'Edson José Dionisio',
  'edson.dionisio@gmail.com', '(81)997402800',  'Paulista', 'PE');
```

```
SELECT codigo, nome, email, telefone, cidade, estado
    FROM "EmpresaDevmedia".tb_funcionarios;
```

Quando declaramos uma função PL/pgSQL composta por parâmetros de saída, estes serão passados com nomes e apelidos de forma opcional, como por exemplo, $Retorno, exatamente da mesma maneira como os parâmetros normais de entrada. Um parâmetro de saída é uma variável que começa com NULL e deve ser atribuído durante a execução da função. O valor final do parâmetro é o que é retornado, como podemos ver num exemplo de operações matemáticas

```
CREATE FUNCTION calculosMatematicos(x int, y int, OUT soma int,
 OUT subtracao int, OUT multiplicacao int, OUT divisao int) AS $
BEGIN
    soma := x + y;
    subtracao := x - y;
    multiplicao := x * y;
    divisao := x / y;
END;

$ LANGUAGE plpgsql;
```

Neste caso utilizamos o comando OUT, que dá a saída dos resultados obtidos a partir dos valores de entrada x e y. Outra forma de obtermos resultados nas SP's é retornando uma função como uma TABLE

```
CREATE FUNCTION extended_sales(p_itemno int)
RETURNS TABLE(quantity int, total numeric) AS $
BEGIN
    RETURN QUERY SELECT s.quantity, s.quantity * s.price FROM sales AS s
                 WHERE s.itemno = p_itemno;
END;

$ LANGUAGE plpgsql;
```

Trigger

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.

As triggers functions podem ser definidas em linguagens compatíveis ao PostgreSQL, como **PL/pgSQL**, **PL/Python**, **PL/Java** dentre outros.

```
CREATE OR REPLACE FUNCTION trigger_function_name
RETURNS trigger AS $ExemploFuncao$
BEGIN
/* Aqui definimos nossos códigos.*/
RETURN NEW;
END;
$ExemploFuncao
```

Repare que uma trigger function é, na realidade, uma **função no PostgreSQL,** mas com a diferença de que ela não recebe argumentos, e sim uma estrutura de dados 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.

Quanto aos eventos (events), estes podem ser **INSERT**, **UPDATE**, **DELETE** ou **TRUNCATE**, os quais especificam o evento que irá disparar a trigger.

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.

```
CREATE TABLE funcionarios
(
  nome character varying(100) NOT NULL,
  email character varying(200) NOT NULL,
  telefone character(14) NOT NULL,
  profissao character varying(150) NOT NULL,
  endereco character varying(100) NOT NULL,
  salario real
)
```

```
CREATE TABLE funcionarios_funcionarios_auditoria (
    codigo_func INT NOT NULL,
    data_alteracao TEXT NOT NULL
);
```

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.

```
CREATE OR REPLACE FUNCTION funcionario_log_func()
RETURNS trigger AS $teste_trigger$
BEGIN
INSERT INTO funcionario_funcionarios_auditoria
(log_id, data_criacao)
VALUES
(new.codigo_func, current_timestamp);
RETURN NEW;
END;
$teste_trigger
```

Com a nossa função criada, definiremos agora a nossa trigger e em seguida, a associaremos a tabela de funcionários

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

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.

Antes de prosseguirmos, excluiremos a tabela **funcionarios_auditoria** utilizando o comando DROP

```
Drop table funcionarios_auditoria cascade;
```

```
CREATE TABLE funcionarios_auditoria
(
log_id INT NOT NULL,
data_criacao TEXT NOT NULL,
operacao_realizada CHARACTER VARYING
);
```

criar ou recriar a nossa trigger function.

```
CREATE OR REPLACE FUNCTION funcionario_log_function()
RETURNS trigger AS $BODY$
BEGIN
-- Aqui temos um bloco IF que confirmará o tipo de operação.
IF (TG_OP = 'INSERT') THEN
INSERT INTO funcionarios_auditoria (log_id, data_criacao, operacao_realizada)
VALUES (new.codigo_func, current_timestamp, ' Operação de inserção.
A linha de código ' || NEW.codigo_func || 'foi inserido');
RETURN NEW;
-- Aqui temos um bloco IF que confirmará o tipo de operação UPDATE.
ELSIF (TG_OP = 'UPDATE') THEN
INSERT INTO funcionarios_auditoria (log_id, data_criacao, operacao_realizada)
VALUES (NEW.codigo_func, current_timestamp, 'Operação de UPDATE.
A linha de código ' || NEW.codigo_func || ' teve os valores atualizados '
|| OLD || ' com ' || NEW.* || '.');
RETURN NEW;
-- Aqui temos um bloco IF que confirmará o tipo de operação DELETE
ELSIF (TG_OP = 'DELETE') THEN
INSERT INTO funcionarios_auditoria (log_id, data_criacao, operacao_realizada)
VALUES (OLD.codigo_func, current_timestamp, 'Operação DELETE. A linha de código '
|| OLD.codigo_func || ' foi excluída ');
RETURN OLD;
END IF;
RETURN NULL;
END;
$BODY
```

Agora criaremos a trigger que será vinculada a tabela de funcionários

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

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

```
INSERT INTO funcionarios (codigo, nome, email, telefone, profissao, endereco, salario)
VALUES (6, 'João da Silva 2', 'joaozinho@gmail.com', '(81)997445854', 'Analista de testes',
'rua desespero', 4500.00);
UPDATE funcionarios set nome = 'Caroline Dionisio' WHERE codigo_func = '3';
DELETE FROM funcionarios WHERE codigo_func= 5;
```

]

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**

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