Artigo

Trabalhando com Stored Procedures no PostgreSQL

Neste artigo trabalharemos com a utilização e criação de Stored Procedures com o banco de dados PostgreSQL, hoje na sua versão mais estável, a 9.4.

Um dos recursos mais utilizados pelos desenvolvedores em Banco de dados é a Stored Procedure, pois mantém concentrada a lógica necessária para determinadas funções, tendo assim uma maior agilidade no retorno de informações importantes.




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

recursos, onde podemos implementar algoritmos com maiores complexidades. Estas funções podem ser registradas e empacotadas no SGBD para utilizações futuras.

Na **Listagem 1** apresentamos a sintaxe básica de uma SP.

Listagem 1. Sintaxe básica da criação de FUNCTIONS/STORED PROCEDURES.

```
CREATE [ OR REPLACE ] FUNCTION
      name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } defau
      [ 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 CURREN
      | AS 'definition'
      | AS 'obj_file', 'link_symbol'
    } ...
      [ WITH ( attribute [, ...] ) ]
```

A sintaxe começa com a criação de uma nova função com a expressão CREATE FUNCTION, seguido do name para definir o nome com o qual será criada. Em seguida alguns parâmetros são apresentados:

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

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.

Para exemplificarmos os tipos de retorno possíveis para Stored Procedures no PostgreSQL, veremos a seguir alguns exemplos práticos. No primeiro exemplo temos uma Stored Procedure contendo o termo “void”, como podemos ver de acordo com a **Listagem 2**.

Listagem 2. Criando uma Stored Procedure que não retorna valor.

```
9       END;
10      $ LANGUAGE 'plpgsql';
```

Para criarmos nossa procedure utilizamos o termo “void”, que define o retorno sem a apresentação de mensagens. 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:

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

Após a inserção do novo registro podemos fazer um SELECT para verificar se a informação foi adicionada corretamente usando a seguinte instrução:

```
1  SELECT codigo, nome, email, telefone, cidade, estado
2      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 da **Listagem 3**.

Listagem 3. Retornando resultados com parâmetros de saída.

```
7       divisao := x / y;
8   END;
9
10  $ 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, presente na **Listagem 4**.

Listagem 4. Retornando funções como tabelas.

```
1   CREATE FUNCTION extended_sales(p_itemno int)
2   RETURNS TABLE(quantity int, total numeric) AS $
3   BEGIN
4       RETURN QUERY SELECT s.quantity, s.quantity * s.price FROM sales AS
5                   WHERE s.itemno = p_itemno;
6   END;
7
8   $ LANGUAGE plpgsql;
```

Os campos de uma tabela podem ser gerados com novas informações em uma tabela a parte, sem que haja a necessidade de criarmos novos campos ou usar outras funções para ter o mesmo resultado.

Criando funções utilizando o SECURITY DEFINER

usuários mal-intencionados criem objetos que escondam objetos usados pela função.

Para realizarmos os ajustes do search_path, podemos fazê-lo de forma permanente ou temporária. Caso este seja realizado de forma temporária, definimos o search_path da seguinte forma:

```
1 set search_path="schema";
```

Ou podemos alterá-lo em definitivo para o banco de dados ou para um usuário especifico com a utilização da seguinte sintaxe:

```
1 ALTER ROLE regraNegocio RESET search_path;
2   ALTER DATABASE funcionarios RESET search_path;
```

Em seguida, podemos definir o search_path com base nas nossas necessidades. O esquema mais importante neste quesito é o da tabela temporária, o qual pode ser acessado por qualquer usuário, e, por conseguinte, é a mais procurado para a realização de ataques. Uma forma de garantirmos um pouco de segurança nesse ponto é forçando a execução deste esquema. Dito isso, escreveremos nossa função de exemplo para termos a pg_temp como sendo a última entrada para o search_path, como apresentado na **Listagem 5**.

Listagem 5. Garantindo mais segurança nas consultas com o Security Definer.

```
        FROM    LoginSenhaUsers
        WHERE   username = $1;
        RETURN aprovado;
END;
$ LANGUAGE plpgsql  SECURITY DEFINER
    -- aqui definimos um esquema seguro e em sequência, passamos o pg_
    SET search_path = admin, pg_temp;
```

Neste exemplo passamos o tipo de retorno como sendo uma informação booleana, ou seja, True em caso de sucesso, ou False se obtivermos uma falha na segurança das informações passadas.

Um fator importante referente a nossas Stored Procedures é referente as pessoas ou schemas que terão permissão para utilizá-las, a fim de garantir implementações mais seguras. O schema PUBLIC, por padrão, tem suas permissões concedidas no momento da criação das procedures. Com o propósito de limitar a utilização das procedures, devemos revogar as permissões presentes no schema PUBLIC, concedendo ao mesmo privilégios apenas para as procedures que ele poderá utilizar, como podemos ver no código da**Listagem 6**.

Listagem 6. Criando uma função para revogar privilégios.

```
BEGIN;
CREATE FUNCTION garanteAcesso(username TEXT, password TEXT) ... SECURI
REVOKE ALL ON FUNCTION garanteAcesso(username TEXT, password TEXT) FRO
GRANT EXECUTE ON FUNCTION garanteAcesso(username TEXT, password TEXT)
```

esquema PUBLIC e permitir que apenas quem tem o esquema de "administradores" possa utilizar a SP garanteAcesso.

O que fazer para desenvolver nossas Stored Procedures

Para que tenhamos uma SP bem definida e que seja de fácil entendimento, é bom que sigamos algumas boas práticas para o seu desenvolvimento, assim precisaremos tomar cuidado com vários fatores.

Inicialmente, precisamos ter cuidado com conflitos de variáveis locais e objetos de banco de dados. Para isso temos que utilizar prefixos de variáveis, como por exemplo, para uma variável local podemos utilizar o "_" e também utilizarmos os atributos qualificadores, como tabelas e colunas em todos os comandos SQL presentes nas procedures. Além disso, devemos declarar as variáveis por tipo derivado -%TYPE e ROWTYPE%.

Podemos utilizar o PL/SQL nativo em todos os lugares que for necessário, pois além de ser possível, torna-se também aconselhável a sua utilização. Precisamos evitar a utilização das SQL's dinâmicas, como por exemplo, podemos substituir os ciclos de repetição presentes nas funções por instruções SELECT com a utilização de CASES.

Nas **Listagens 7** e **8**temos a construção de duas formas: a incorreta e a correta, respectivamente.

```
    END LOOP;
```

Listagem 8. Criação de uma função melhor desenvolvida.

```
INSERT INTO tabela2
    SELECT 100, campo2 FROM tabela1 WHERE campo1 > 100;
  INSERT INTO tabela3
    SELECT campo1, campo2 FROM tabela1 WHERE campo1 <= 100;
```

Percebam que ambas as estruturas têm o mesmo objetivo, o qual não conseguimos compreender de forma coerente e fácil na **Listagem 7**. Além desses pontos, temos que a função deve conter apenas um comando RETURN e uma saída. Devemos evitar a duplicação de código nas aplicações e usar mais a função ASSERT, nos casos em que for cabível, como podemos ver no exemplo da **Listagem 9**.

Listagem 9. Utilizando a função ASSERT nas Stored Procedures.

```
BEGIN   IF NOT $1 OR $1 IS NULL THEN
     IF $2 IS NOT NULL THEN
       RAISE EXCEPTION 'A afirmação falhou...: %', $2;
     END IF;
     RAISE NOTICE 'Utilizando o Assert. Mensagem é nula';
   END IF;
  END;
  $ LANGUAGE plpgsql;

  CREATE OR REPLACE FUNCTION Assert_IsNotNull(elementoTeste, varchar)
```

termos uma lista de erros definidos pelo usuário de antemão, de forma a seguirmos as exceções adicionadas no texto. Não há a necessidade de testarmos variáveis nulas quando já temos definida na declaração que ela será NOT NULL. Devemos realizar testes em conteúdo de variáveis presentes no sistema lógico.

Além disso, torna-se uma boa prática a utilização de rótulos << label >> para loops e blocos, além de que, devemos usar tipos booleanos de modo mais legível aos nossos códigos. Ao trabalharmos com conversões automáticas dos tipos date e datetime, nem sempre eles atuam de forma correta, pois estes dependem de configurações do banco. Ao invés de utilizá-los, podemos usar as funções to_char e to_date.

Por fim, não devemos colocar comandos SQL para fazer nosso código funcionar sem que estes tenham alguma funcionalidade útil, além da necessidade de termos que reduzir a utilização de cursores e tabelas temporárias em nossos projetos.

Nas **Listagens 10** e **11**tem-se a exemplificação da utilização das SP's dentro dos padrões que são precisos e as que podemos encontrar erroneamente pela internet.

Listagem 10. Má utilização de código numa stored procedure.

```
1  DECLARE teste varchar;
2    BEGIN
3      teste := 'a';
4      teste := teste || 'b';
5      teste := teste || 'c';
6      RETURN teste;
```


```
1  BEGIN
2      RETURN 'a' || 'b' || 'c';
3    END;
```

Outra forma de reduzirmos a repetição dessas declarações de atribuição de variáveis é com a utilização da função COALESCE, a qual tem por objetivo retornar valores presentes no primeiro argumento não nulo encontrado, como podemos ver nas estruturas apresentadas pelas **Listagens 12** e **13**.

Listagem 12. Representação de uma má estruturação de código.

```
1  DECLARE coluna1 varchar := '';
2    BEGIN
3      IF texto1 IS NULL THEN
4        coluna1 := coluna1 || 'NULL',
5      ELSE
6        coluna1 := coluna1 || texto1;
7      END IF;
8
9      IF texto2 IS NULL THEN
10       coluna1 := coluna1 || 'NULL',
11     ELSE
12       coluna1 := coluna1 || texto2;
13     END IF;
```

Listagem 13. Representação de uma estrutura com a função COALESCE.




Como pode ser visto no exemplo apresentado pela **Listagem 13**, temos o código melhor estruturado, de forma simples e fácil de entender.

Em relação a utilização de SQL para funções simples ao invés de utilizarmos PL/pgSQL, temos algumas observações importantes que podemos ver nas **Listagens 14** e **15**.

Listagem 14. Exemplo com a utilização de PL/pgSQL com função simples.

```
1  CREATE OR REPLACE FUNCTION ultimoDiaMes(IN dataMes date)
2  RETURNS date AS $
3  BEGIN
4    RETURN CAST(date_trunc('month',current_date + interval '2 month') AS
5  END;
6  $ LANGUAGE plpgsql IMMUTABLE STRICT;
```

Listagem 15. Exemplo com a utilização da linguagem SQL com função simples.

```
1  CREATE OR REPLACE FUNCTION ultimoDiaMes(IN dataMes date)  RETURNS date
2      SELECT CAST(date_trunc('month', $1 + interval '2 month') AS date)
3    $ LANGUAGE sql;
```

Ambas as funções retornam a mesma informação, que é o último dia do mês. A diferença é que estamos utilizando uma LANGUAGE diferente para cada uma das funções.

códigos, temos a intenção de reduzir ataques que podem prejudicar nosso trabalho e colocar em risco dados importantes que estejam sendo utilizados, eis a questão de entendermos quando e por quê trabalhar com cautela é importante no ramo de DBA's.

Esperamos que tenham gostado. Até a próxima!

Por **Edson**
Em 2015

Suporte ao aluno - Tire a sua dúvida.

Poste aqui a sua dúvida, nessa seção só você e o consultor podem ver os seus comentários.