PostgreSQL o SGBD das 1001 linguagens

Diogo Biazus - diogob@gmail.com

Um dos grandes diferenciais do PostgreSQL em relação à maioria dos outros sistemas de bancos de dados é a presença de diversas linguagens procedurais para programação de funções diretamente no banco de dados. Esse recurso pode ser interessante para manter a uniformidade de linguagens no desenvolvimento de aplicações ou para aproveitar conhecimento já existente sem a necessidade de aprender uma nova sintaxe apenas para trabalhar com os dados no servidor.
As linguagens disponíveis são variadas: PHP, Python, Java, Ruby, só para citar as mais conhecidas. Além delas e de outras não tão populares também existe a PL/pgSQL, desenvolvida junto com o próprio PostgreSQL. Você deve estar se perguntando se alguma delas leva vantagem sobre as outras, saiba que elas são quase equivalentes exceto por uma pequena vantagem da PL/pgSQL na praticidade da utilização de comandos SQLs integrados à linguagem e na criação de cache dessas consultas.
Vamos explorar um pouco dessas linguagems então e ver melhor como funciona esse sistema de múltiplas Pls. No entento, antes de desenvolver o artigo vou abordas rapidamente alguns conceitos básicos.

Procedimentos armazenados

Quando trabalhamos em uma aplicação cliente/servidor, torna-se interessante realizar parte do processamento dos dados no servidor. Muitas vezes queremos aplicar funções complexas sobre um conjunto de dados sem precisar trazê-los para uma máquina cliente, fazendo com que o servidor entregue apenas o resultado dessas operações. Para definir esse conjunto de operações uma linguagem declarativa como o SQL é muito limitada, o ideal é usar alguma linguagem procedural que possa ser executada no servidor, e que as funções criadas com essa linguagem sejam armazenadas junto com os dados.
O conceito explicado acima é o de procedimentos armazenados, que surgiu nos sistemas relacionais para suprir a necessidade de processar dados no servidor utilizando o próprio SGBD como ambiente de execução. Essa abordagem tem vários beneficios em alguns cenários, em primeiro lugar essas funções geralmente tem um desempenho superior por estarem mais “próximas” dos dados, evitando também o excesso de tráfego de rede. Outro problema solucionado por esse recurso é o de vários programas que acessam o mesmo conjunto de dados e muitas vezes reimplementam lógicas identicas de tratamento de dados, o que além de um desperdício de recursos pode gerar discrepâncias nos dados.

Linguagens procedurais em SGBDs

Para escrever essas funções de modo imperativo dentro dos bancos de dados surgiram linguagens procedurais cujos interpretadores eram embutidos diretamente no SGBD. As linguagens mais populares são a PL/SQL, inpirada na sintaxe da linguagem ADA e usada no Oracle desde a versão 7.0, e o Transact-SQL usado no Sybase e MS SQL Server.
No PostgreSQL não existe apenas uma linguagem desse tipo, mas um conjunto delas, as oficialmente distribuidas com o servidor são: PL/pgSQL, PL/TCL, PL/Python e PL/Perl. Ainda existem outras que podem ser instaladas e são providas por projetos paralelos como: PL/Java, PL/Ruby, PL/R, entre outras. Essa flexibilidade é obtida separando o mecanismo do banco de dados do executor das funções, o processo do PostgreSQL desconhece completamente o funcionamento interno de funções escritas em alguma PL, ele apenas delega a sua execução para um trecho de código em C (compilado em uma biblioteca dinâmica) responsável por isso.

Instalando Linguagens procedurais

Na instalação padrão do PostgreSQL nenhuma linguagem procedural está disponível, para habilitar uma devemos instala-la no servidor usando o comando createlang. Observe que no instalador para Windows existe a possibilidade de deixar algumas Pls pré-instaladas no template1.

A sintaxe do createlang é a seguinte: createlang linguagem banco_destino

Para que a PL esteja disponível em todos os novos bancos basta instalar no template1, que serve como modelo para bases futuras. Por exemplo, para tornar a PL/pgSQL uma linguagem disponível em todas as bases criadas basta usar o seguinte comando: createlang plpgsql template1. Para bases já existentes não adianta criar a PL no template1, nesse caso temos que criá-la diretamente na base destino.

O comando createlang apenas executa alguns comandos SQL, o que poderia ser feito manualmente e está descrito em detalhes na documentação oficial, porém não é necessário utilizar esse procedimento.

Para executar os exemplos presentes nesse artigo você irá precisar instalar a PL/pgSQL. Para isso podemos usar uma base de teste.

```
createdb teste
createlang plpgsql teste
```

Criando funções

Antes de começarmos a criar funções na base de dados precisamos compreender alguns aspectos importantes sobre a implementação das funções no PostgreSQL. Todos os procedimentos são criados com o comando CREATE FUNCTION e são removidos com o comando DROP FUNCTION.

```
CREATE FUNCTION primeira_funcao() RETURNS VOID AS
'
BEGIN
        RETURN;
END;
' LANGUAGE 'plpgsql';

DROP FUNCTION primeira_funcao();
```

Como o PostgreSQL trabalha com sobrecarga de funções sempre é necessário indicar os parâmetros nas operações de criação e exclusão para identificar corretamente a função alvo. A sobrecarga – presente em algumas linguagens de programação - permite que duas ou mais funções tennham o mesmo nome desde que os argumentos sejam distintos. Podemos támbem substituir um procedimento existente usando a cláusula OR REPLACE como no exemplo abaixo:

```
CREATE OR REPLACE FUNCTION primeira_funcao() RETURNS VOID AS
'
BEGIN
        RETURN;
END;
' LANGUAGE 'plpgsql';
```

Não é possível substituir uma função alterando seu tipo de retorno dessa maneira, além disso cuide

para indicar os parâmetros corretamente pois se eles forem diferentes do alvo um novo procedimento será criado e o antigo continuará a existir na base de dados.

PL/pgSQL

A linguagem PL/pgSQL é muito próxima da PL/SQL presente no Oracle, a sintaxe lembra pascal e pode-se executar comandos SQL diretamente no corpo dos procedimentos. Isso faz dela uma escolha natural em situações onde a manipulação dos dados no banco é a principal tarefa do procedimento. Além de ser muito prática para usar comandos SQL, a PL/pgSQL também é extremamente segura, pois seu ambiente de execução é restrito ao SGBD. Ou seja, ela não pode interagir diretamente com o sistema subjacente sendo totalmente segura para permitir que qualquer programador crie suas funções em um ambiente controlado.

Otimizando o uso de cache

Para otimizar o uso de cache em retorno de funções podemos usar indicadores no comando de criação para dizer ao PostgreSQL como a rotina se comporta. Temos três opções: IMMUTABLE, STABLE e VOLATILE. A IMMUTABLE indica que sempre que os mesmos parâmetros forem passados o retorno da função será o mesmo, ou seja, ela é independente dos dados do banco. STABLE significa que mesmo com valores de parâmetros identicos a função pode ter retornos distintos, mas ela se mantém estável na mesma varredura de tabela. E VOLATILE não permite nenhum tipo de otimização, pois ela indica que a função sempre varia, indepedente do momento em que é chamada, portanto não se pode criar nenhum tipo de cache.

Estrutura básica e variáveis

A programação sempre é feita em blocos de código cuja estrutura básica é:

```
[DECLARE
        nome_da_variável      tipo;
        ...
]
BEGIN
        comandos;
END;
```

Os colchetes não fazem parte da sintaxe, eles servem para indicar que a parte de declarações é opcional, podendo ser omitida. Caso seja necessário utilizar alguma variável para armazenamento temporário de valores ela deve ser declarada na seção DECLARE do bloco. Vamos ver um exemplo bem simples onde apenas imprimimos a mensagem “Minha primeira rotina em PL/pgSQL”:

```
CREATE OR REPLACE FUNCTION primeira_funcao() RETURNS VOID AS
$body$
BEGIN
        RAISE NOTICE 'Minha primeira rotina em PL/pgSQL';
        RETURN;
END;
$body$
LANGUAGE 'plpgsql';
```

Alteramos um pouco a sintaxe do CREATE FUNCTION no último exemplo, usamos os delimitadores $body$ para indicar o início e o fim da função, isso facilita o uso de aspas simples

dentro da definição sem precisar duplicar as aspas ou usar outro caracter de escape. O $body$ só é suportado a partir do PostgreSQL 8.0.
O tipo de retorno VOID é usado para funções que não têm um valor de retorno definido.
Na declaração de variáveis podemos usar dois elementos comuns no SQL padrão, a cláusula NOT NULL e o valor DEFAULT. Além disso podemos utilizar o operador := para atribuir valores às variáveis durante a execução da rotina. Vejamos como utilizar esses elementos:

```
CREATE OR REPLACE FUNCTION atribuicao() RETURNS VOID AS
$body$
DECLARE
	numero int4 NOT NULL DEFAULT 10;
BEGIN
	RAISE NOTICE 'Valor de numero antes da atribuição: %', numero;
	numero := 15;
	RAISE NOTICE 'Valor de numero após a atribuição: %', numero;
	RETURN;
END;
$body$ LANGUAGE 'plpgsql';
```

Estruturas de controle
Para controlar o fluxo da rotina, a PL/pgSQL oferece todos os métodos tradicionais presentes em outras linguagens de programação estruturada. Utilizamos blocos IF ... THEN ... ELSIF ... THEN ... ELSE ... END IF; como controle condicional, o FOR ... LOOP END LOOP; para percorrer intervalos pré-definidos e as estruturas LOOP ... END LOOP; e WHILE ... LOOP ... END LOOP; para laços baseados em condições de saída.
Podemos var o funcionamento dessas estruturas em uma rotina de exemplo que executa o mesmo laço usando uma dessas estruturas dependendo do parâmtro que for passado:

```
CREATE OR REPLACE FUNCTION lacos(tipo_laco int4) RETURNS VOID AS
$body$
DECLARE
	contador int4 NOT NULL DEFAULT 0;
BEGIN
	IF tipo_laco = 1 THEN
		-- Loop usando WHILE
		WHILE  contador < 10 LOOP
			contador := contador  + 1;
			RAISE NOTICE 'Contador: %', contador;
		END LOOP;
	ELSIF tipo_laco = 2 THEN
		-- Loop usando LOOP
		LOOP
			contador := contador  + 1;
			RAISE NOTICE 'Contador: %', contador;
			EXIT WHEN contador > 9;
		END LOOP;
	ELSE
		-- Loop usando FOR
		FOR contador IN 1..10 LOOP
```

```
                RAISE NOTICE 'Contador: %', contador;
            END LOOP;
        END IF;
        RETURN;
END;
$body$ LANGUAGE 'plpgsql';
```

Observe que no caso do LOOP usamos o comando EXIT com a cláusula WHEN, para executar a saída do bloco mediante uma condição, essa cláusula é opcional, poderíamos escrever apenas EXIT. Outro detalhe é que a variável do FOR é implicitamente declarada dentro do bloco, então não precisamos usar uma variável pré-declarada.

Interagindo com o banco de dados

Os comandos SQL podem ser embutidos na programação PL/pgSQL. Podemos colocar variáveis dentro dos comandos SQL que elas serão substituídas em tempo de execução. Isso é válido para comandos de DML (manipulação de dados) para comandos DDL (definição de dados) será necessário usar SQLs dinâmicos, que veremos adiante.

A substituição automática de variáveis torna a interação com os dados muito simples, porém gera um efeito colateral, não podemos ter variáveis com o mesmo nome que objetos no banco de dados, pois isso iria gerar ambigüidades nos comandos SQL. Para evitar um esforço maior de criatividade no momento de nomear as variáveis, normalmente adicionamos um prefixo para evitar esse tipo de ambigüidade. Costumo usar o prefixo v para variáveis e p para parâmetros, de quebra podemos identificar facilmente os parâmetros, que não podem ter seus valores alterados. Os comandos SQL devem ser terminados por ponto e vírgula e podem ser quebrados em várias linhas como qualquer outra linha de comando.

Como exemplo veremos uma função de exclusão para remover linhas de uma tabela chamada clientes. Veremos outras versões dessa função em outros exemplos durante o artigo, e pode se tornar muito útil para controle de segurança do banco de dados.

```
CREATE OR REPLACE FUNCTION exclui_cliente(pid_cliente int4) RETURNS int4 AS
$body$
DECLARE
        vLinhas int4 DEFAULT 0;
BEGIN
        DELETE FROM clientes WHERE id_cliente = pid_cliente;
        GET DIAGNOSTICS vLinhas = ROW_COUNT;
        RETURN vLinhas;
END;
$body$ LANGUAGE 'plpgsql';
```

A função recebe o valor do campo chave que deve ser excluído, isso garante que apenas uma linha será excluída, você já pode imaginar como isso vai nos auxiliar na parte de segurança. Ela retorna o número de linhas afetadas pelo comando, obtido com o comando GET DIAGNOSTICS, que pode ser usado após qualquer comando DML.

Para recuperar dados podemos usar o SELECT INTO, comando que coloca dados retornados por um comando SELECT em uma lista de variáveis simples ou em uma variável composta. Uma variável composta é uma variável cujo tipo representa uma tupla, podendo ser declarada como RECORD ou como %ROWTYPE. As variáveis RECORD são mais flexíveis podendo assumir qualquer estrutura que desejarmos. Já as variáveis %ROWTYPE estão atreladas a uma relação

existente no banco de dados. Por exemplo, se temos uma tabela chamada clientes e desejamos uma variável para armazenar uma linha dessa tabela podemos declará-la como sendo do tipo clientes%ROWTYPE.
No exemplo abaixo vemos como utilizar as variáveis compostas e o SELECT INTO para recuperar informações. Vamos receber como parâmetro o identificador de um cliente e retornar o seu volume de compras médio.

```
CREATE OR REPLACE FUNCTION media_compras(pid_cliente int4) RETURNS numeric AS
$body$
DECLARE
        linhaCliente            clientes%ROWTYPE;
        mediaCompras            numeric(9,2);
        totalCompras            numeric(9,2);
        periodo                 int4;
BEGIN
        SELECT * INTO linhaCliente FROM clientes WHERE id_cliente = pid_cliente;
        -- Calcula o periodo em dias que trabalhamos com o cliente subtraindo da data atual a data
de inclusão do cliente
        periodo := (current_date -  linhaCliente.data_inclusao);
        -- Coloca na variável totalCompras o somatório de todos os pedidos do cliente
        SELECT SUM(valor_total) INTO  totalCompras FROM pedidos WHERE id_cliente =
pid_cliente;
        -- Faz a divisão e retorna o resultado
         mediaCompras :=  totalCompras / periodo;
        RETURN  mediaCompras;
END;
$body$ LANGUAGE 'plpgsql';
```

Poderíamos substituir o tipo da variável linhaCliente por RECORD sem alterar a funcionalidade da rotina, a única diferença é que a estrutura de RECORD não é pré-estabelecida possibilitando que o select usado para atribuir o seu valor retorne tuplas que não existem em nenhuma relação do banco de dados.

Cursores

Cursores são áreas de memória utilizadas pelo PostgreSQL para armazenar registros enquanto eles são processados. Quando estamos programando em PL/pgSQL muitas vezes é necessário manipular esses cursores manualmente, para buscar dados do banco e interagir com eles. Podemos usar os cursores para percorrer as relacões e processar conjuntos de dados. Vejamos como fazer um procedimento nos moldes do exemplo anterior que calcule a média de compras de todos os clientes do banco de dados e armazene essa informação em um campo da base de dados.
Usaremos o comando de iteração sobre cursores FOR IN, por ser mais prático, a manipulação de cursores pode ser feita de um modo totalmente manual, porém não pretendo abordar esse tópico no artigo, para mais informações podem consultar a documentação do PostgreSQL na seção de PL/pgSQL – Cursores.

```
CREATE OR REPLACE FUNCTION media_compras() RETURNS VOID AS
```

```
$body$
DECLARE
	linhaCliente		clientes%ROWTYPE;
	mediaCompras		numeric(9,2);
	totalCompras		numeric(9,2);
	periodo			int4;
BEGIN
	FOR linhaCliente IN  SELECT * FROM clientes LOOP
		periodo := (current_date -  linhaCliente.data_inclusao);
		SELECT SUM(valor_total) INTO  totalCompras FROM pedidos WHERE id_cliente
= pid_cliente;
		mediaCompras :=  totalCompras / periodo;
		UPDATE clientes SET media_compras =  mediaCompras WHERE id_cliente =
pid_cliente;
	END LOOP;
	RETURN;
END;
$body$ LANGUAGE 'plpgsql';
```

O FOR cria um cursor com os resultados da consulta SELECT * FROM clientes e a cada iteração a variável linhaCliente recebe uma tupla resultante do comando, quando o cursor chega ao fim o laço é automaticamente encerrado e o cursor é fechado.
Notem que esse é um exemplo puramente didático, pois nesse caso normalmente é mais interessante produzir esses resultados dinâmicamente usando apenas uma consulta SQL, mas serve como exemplo de como aplicar esse tipo de recurso.

Retornando relações

Quando queremos que a nossa função não retorne um valor simples, mas sim uma relação inteira, podemos fazê-lo utilizando o comando RETURN NEXT e a cláusula SETOF no tipo de retorno. Colocando a palavra SETOF antes do tipo de retorno no CREATE FUNCTION estaremos indicando ao PostgreSQL que a função deve retornar um conjunto do tipo especificado, no nosso caso um conjunto de RECORDs. A instrução RETURN NEXT armazena um determinado retorno em um buffer e continua a executar a rotina até encontrar um RETURN. Então todo o buffer é retornado na forma de uma relação, normalmente usamos variáveis compostas do tipo record para esse tipo de retorno. Sempre que chamamos uma função desse tipo usamos a sintaxe: SELECT * FROM funcao() tabela(campo1 tipo, campo2 tipo, ...);
A função assume a posição de uma tabela do select e, se a variável de retorno for do tipo RECORD, precisamos especificar qual o formato de tupla que esperamos como resultado. Isso é feito com uma lista de campos e seus respectivos tipos após o apelido usado para a função.
Voltemos ao exemplo do cálculo de média de compras, mas agora ao invés de armazenar o valor para cada cliente vamos retornar uma relação contendo o campo mediaCompras e totalCompras.

SQL Dinâmico

Para dar mais flexibilidade a construção de SQLs em tempo de execução, existe o recurso de SQL dinâmico. Você pode construir strings, usando concatenação, contendo o SQL a ser executado e enviá-las para o servidor com o comando EXECUTE. Para recuperar valores com SQLs dinâmicos não podemos usar o SELECT INTO, para isso temos o FOR IN EXECUTE. Como exemplo vamos criar uma função baseada na exclui_cliente só que genérica, para atender a exclusão de qualquer tabela.

```
CREATE OR REPLACE FUNCTION exclui_registro(nome_tabela text, nome_chave text, id int4)
RETURNS int4 AS
$body$
DECLARE
        vLinhas int4 DEFAULT 0;
BEGIN
        EXECUTE 'DELETE FROM ' || nome_tabela || ' WHERE ' || nome_chave || ' = ' || id;
        GET DIAGNOSTICS vLinhas = ROW_COUNT;
        RETURN vLinhas;
END;
$body$ LANGUAGE 'plpgsql';
```

Usamos o operador de concatenação (||) para juntar o nome da tabela e o nome do campo chave, passados como parâmetro, ao comando DELETE. Assumindo que as tabelas do banco possuem uma única chave numérica essa função poderia ser usada para excluir dados de qualquer tabela. Para bancos com chaves compostas ainda poderíamos usar uma função semelhante, mas teríamos que usar arrays de parâmetros e laços para montar o SQL.

Usando procedimentos para aumentar a segurança

Muitas vezes criamos procedimentos para controlar a interação entre os usuários, ou aplicações cliente e o SGBD. Por exemplo, podemos querer controlar as exclusões feitas, para impedir que vários registros sejam excluídos simultâneamente em certas tabelas ou por certos usuários. Podemos criar logs de operação, notificar alguém caso alguma operação seja feita, enfim, uma grande gama de opções. Para isso usamos um recurso muito interessante, que é o SECURITY DEFINER. Essa cláusula é colocada junto com o comando de criação de função e indica que a execução da função será feita com os privilégios de quem a definiu, e não com os de quem a invocou.

Portanto podemos tirar as permissões de exclusão nas tabelas de um usuário e permitir que ele execute funções de exclusão. Conceitualmente essa idéia é próxima ao encapsulamento de dados obtido em linguagens orientadas a objeto. Vamos usar como exemplo a nossa função de exclusão, mas agora com um controle para impedir o uso malicioso dela para apagar uma tabela inteira.

```
CREATE OR REPLACE FUNCTION exclui_registro(nome_tabela text, nome_chave text, id int4)
RETURNS int4 AS
$body$
DECLARE
        vLinhas int4 DEFAULT 0;
BEGIN
        EXECUTE 'DELETE FROM ' || nome_tabela || ' WHERE ' || nome_chave || ' = ' || id;
        GET DIAGNOSTICS vLinhas = ROW_COUNT;
        IF vLinhas > 1 THEN
                RAISE EXCEPTION 'A exclusão de mais de uma linha não é permitida.';
        END IF;
        RETURN vLinhas;
END;
$body$ LANGUAGE 'plpgsql' SECURITY DEFINER;
```

O RAISE EXCEPTION gera uma exceção, além de mostrar a mensagem cria um erro que aborta a transação do usuário e impede que a exclusão seja concluida. Assim possibilitamos a exclusão de

apenas um registro por vez.