Funções em PostgreSQL

1

Introdução

- Bancos de Dados possuem uma arquitetura cliente/servidor que faz com que os dados manipulados pelas aplicações tenham que trafegar pela rede;
- Para processamentos simples a quantidade de dados trafegada é pequena, o que não influencia no desempenho da aplicação;
- Porém, para processamentos complexos o volume de dados trafegados pode ser um problema para o desempenho da aplicação;
- Ainda, dependendo da capacidade de processamento do cliente, a manipulação do volume de dados retornado pode ser muito custosa/demorada.

2

2

Introdução

- Solução: realizar processamento dos dados no servidor, dentro do banco de dados;
- Dessa forma, o banco de dados entrega aos clientes apenas o resultado do processamento;
- Vantagens:
  - Desempenho superior devido a “proximidade” aos dados;
  - Menor tráfego de rede;
  - Evita reimplementação de código idêntico em diferentes aplicações.

3

3

Introdução

- Porém, a linguagem de consulta SQL é muito limitada para ser usada na criação de funções;
- Ela não contém os comandos básicos para a implementação de lógica de programação imperativa por ser uma linguagem declarativa;
- A solução é programar as funções usando outras linguagens de programação disponíveis no PostgreSQL.

4

4

Linguagem suportadas

- Uma das vantagens do PostgreSQL é a disponibilização de diversas linguagens para programação de funções diretamente no banco de dados.
- Linguagens suportadas: PHP, Python, Java, Ruby, Perl, C, ...
- A linguagem padrão do PostgreSQL é a PL/pgSQL.

5

5

Instalação da linguagem

- Como instalar a linguagem no banco de dados?
- Após conectar a database desejada, digite:

```
CREATE EXTENSION <nome da linguagem>
```

- Exemplo:

```
CREATE EXTENSION pljava;
CREATE EXTENSION plpythonu;
```

- A linguagem PL/pgSQL já está instalada por padrão.

6

6

Estrutura básica de uma função

- Sintaxe de declaração de funções em PostgreSQL:

```
CREATE OR REPLACE FUNCTION nome_da_funcao([parametros]) [RETURNS tipo] AS
$$
[ <<rotulo>> ]
[ DECLARE
      declaração de variáveis ]
BEGIN
      bloco de código
END [ rotulo ];
$$
LANGUAGE plpgsql;
```

7

7

Criação de Função

- Criando sua primeira função:

```
CREATE OR REPLACE FUNCTION primeira_funcao() RETURNS int AS
$$
BEGIN
      RETURN 1;
END;
$$
LANGUAGE plpgsql;
```

8

8

Exclusão de função

- Para excluir uma função:

```
DROP FUNCTION primeira_funcao();
```

9

9

Declaração de parâmetros

- Os parâmetros podem ser de qualquer tipo de dados suportado pelo PostgreSQL;
- São declarados da seguinte forma:

```
(parametro1 tipo, parametro2 tipo, parametro3 tipo, ...)
```

10

10

Declaração de parâmetros - Exemplo

```
CREATE FUNCTION multiplica(valor_a real, valor_b real) RETURNS real AS
$$
BEGIN
      RETURN valor_a * valor_b;
END;
$$
LANGUAGE plpgsql;


DROP FUNCTION multiplica(valor_a real, valor_b real);
```

11

11

Declaração de parâmetros – Tipos polimórficos

- São parâmetro de podem assumir qualquer tipo de dado suportado pela postgreSQL;
- PL/pgSQL deduz o tipo a partir dos valores recebidos;
- O PostgreSQL suporta os seguintes tipos polimórficos como parâmetro:
  - anyelement
  - anyarray
- Ao declarar parâmetros desses tipos, é criado um parâmetro especial $0 que representa o tipo de dado dos parâmetros polimórficos
- Todos os parâmetros polimórficos DEVEM ter o mesmo tipo
  - Mesmo se declarar atributos anyelement e anyarray

12

12

Tipos polimórficos - Exemplo

```
CREATE FUNCTION exibeDado(dado anyelement) RETURNS void AS
$$
BEGIN
    RAISE NOTICE 'O dado recebido é = %', dado;
    RETURN;
END;
$$
LANGUAGE plpgsql;
```

13

13

Retorno da função

- O retorno pode ser de qualquer tipo dos dados suportado pelo PostgreSQL;
- O retorno também pode ser de um dos seguintes tipos especiais:
  - anyelement
  - anyarray
  - TABLE
  - SETOF

14

14

Retorno da função - Exemplo

- Para retorno polimórfico, ao menos um parâmetro polimórfico deve ser declarado
  - Tipo do retorno será o mesmo tipo do parâmetro

```
CREATE FUNCTION add(v1 anyelement, v2 anyelement, v3 anyelement)
RETURNS anyelement AS
$$
DECLARE result ALIAS FOR $0;
BEGIN
      result := v1 + v2 + v3;
      RETURN result;
END;
$$
LANGUAGE plpgsql;
```

15

15

Retorno via parâmetro OUT

- É possível remover da declaração da função o “RETURNS tipo”;
- Entretanto, a função não pode ficar sem retorno;
- Nesse caso, ao menos um dos parâmetro ser OUT.

```
CREATE FUNCTION soma_mult(x int, y int, OUT soma int, OUT mult int) AS
$$
BEGIN
      soma := x + y;
      mult := x * y;
END;
$$
LANGUAGE plpgsql;
```

16

16

Declaração de Variáveis

- Variáveis podem ser de qualquer tipo de dado suportado pelo PostgreSQL;
- Também podem ser dos tipos especiais:
  - %ROWTYPE
  - %TYPE
  - RECORD
- São declarados como:

nome [CONSTANT] tipo [NOT NULL] [{ DEFAULT | := } expressão];

17

17

Declaração de Variáveis - Exemplos

- Exemplos:

```
helpstr varchar(300);
mediaCompras numeric(9,2);
qtdade real NOT NULL DEFAULT 32.3;
url varchar := 'http://mysite.com';
user_id CONSTANT integer := 10;
weight integer[] DEFAULT '{10,9,8,7,6,5,4,3,2,1}';
pom varchar DEFAULT $1;
```

18

18

Comentários

- Comentário que ocupa apenas uma linha:

```
-- exemplo de comentário
```

- Comentário de mais de uma linha:

```
/* inicio do comentário
   continuação
   fim do comentário */
```

19

19

Estruturas de Controle

- PL/pgSQL oferece todas as estruturas de controle presentes em linguagens de programação estruturada;
- Controle condicional
  - IF ... THEN ... ELSIF ... THEN ... ELSE ... ... END IF;
- Laços de repetição
  - FOR ... IN ... LOOP ... ... END LOOP;
  - WHILE ... LOOP ... ... END LOOP;
  - LOOP ... ... END LOOP;

20

20

Controle condicional *if* – Exemplo

```
CREATE OR REPLACE FUNCTION maior(a int, b int) RETURNS int AS
$$
BEGIN
    IF a>b THEN
        RETURN a;
    ELSIF b>a THEN
        RETURN b;
    ELSE
        RETURN 0;
    END IF;
END;
$$
LANGUAGE plpgsql;
```

21

Laço de repetição *for*

- Sintaxe:

```
[ <<rotulo>> ]
FOR nome IN [ REVERSE ] expressão .. expressão[BY expressão] LOOP
    operações
END LOOP [ rotulo ];
```

- A variável “nome” é automaticamente declarada
  - Existe apenas dentro do loop, enquanto ele executa

22

Laço de repetição *for* – Exemplo

```
FOR count IN 1..10 LOOP
      -- count assumirá os valores 1,2,3,4,5,6,7,8,9,10
END LOOP;

FOR count IN REVERSE 10..1 LOOP
      -- count assumirá os valores 10,9,8,7,6,5,4,3,2,1
END LOOP;

FOR count IN REVERSE 10..1 BY 2 LOOP
      -- count assumirá os valores 10,8,6,4,2
END LOOP;
```

23

23

Laço de repetição *while*

- Sintaxe:

```
[ <<rotulo>> ]
WHILE expressão_booleana LOOP
      operações
END LOOP [ rotulo ];
```

24

24

Laço de repetição *loop*

- Sintaxe:

```
[ <<rotulo>> ]
LOOP
    operações
END LOOP [ rotulo ];
```

- LOOP não possui condições de parada
- Sua execução termina apenas quando o comando EXIT é executado

25

25

Laço de repetição *loop*

- Para sair do laço de repetição é necessário incluir a seguinte expressão dentro do *loop*:

```
EXIT [ rotulo ] [ WHEN expressão_booleana ];
```

26

26

Laço de repetição *loop* - Exemplo

```
count := 0;
LOOP
    count := count + 1;
    RAISE NOTICE 'Valor de count é: %', count;
    EXIT WHEN count > 10;
END LOOP;
```

27

Funções usando apenas SQL

- É possível criar funções usando apenas comandos SQL;

```
CREATE FUNCTION exclui_empregados() RETURNS void AS
$$
   DELETE FROM empregado WHERE salario < 0;
$$
LANGUAGE SQL;
```

- A função pode ter como retorno uma tabela;

```
CREATE FUNCTION retorna_empregados(pnome char(100)) RETURNS empregado AS
$$
   SELECT * FROM empregado WHERE nome = pnome;
$$
LANGUAGE SQL;
```

28

Funções PL/pgSQL com comandos SQL

- Comandos SQL podem ser embutidos nas funções desenvolvidas em PL/pgSQL;
  - DML: Data Manipulation Language
    - SELECT, INSERT, UPDATE, DELETE, ...
  - DDL: Data Definition Language
    - CREATE, ALTER, DROP, ...
- Clausulas SQL agem como se fossem clausulas da própria PL/pgSQL.

29

29

Funções PL/pgSQL com SQL – Exemplo

```
CREATE OR REPLACE FUNCTION cria_empregados() RETURNS void AS $$
BEGIN
        CREATE TABLE empregado (
                cod integer PRIMARY KEY,
                nome varchar,
                salario real);
        INSERT INTO empregado VALUES (1,'André', 15000);
        INSERT INTO empregado VALUES (2,'Bruno', -1);
        INSERT INTO empregado VALUES (3,'Bruna', -1);
        INSERT INTO empregado VALUES (4,'Carla', 17000);
        INSERT INTO empregado VALUES (5,'José', -1);
        INSERT INTO empregado VALUES (7,'Pedro', 25450);
        RETURN;
END;
$$ LANGUAGE plpgsql;
```

30

30

Funções PL/pgSQL com SQL – Exemplo

- Em PL/pgSQL é possível retornar o número de linhas afetadas pela execução do comando SQL:

```
CREATE FUNCTION limpa_empregados() RETURNS integer AS $$
DECLARE
        linhas_afetadas integer DEFAULT 0;
BEGIN
        DELETE FROM empregado WHERE salario < 0;
        GET DIAGNOSTICS linhas_afetadas = ROW_COUNT;
        RETURN linhas_afetadas;
END;
$$ LANGUAGE plpgsql;
```

31

31

Criando SQLs dinâmicas dentro da Função

- Outra possibilidade é criar consultas SQL dinâmicas utilizando parâmetros ou variáveis da função como parte da cláusula SQL;

```
CREATE FUNCTION exclui_empregado(pcod int) RETURNS int4 AS $$
DECLARE
        linhas_afetadas int DEFAULT 0;
BEGIN
        DELETE FROM empregado WHERE cod = pcod;
        GET DIAGNOSTICS  linhas_afetadas = ROW_COUNT;
        RETURN  linhas_afetadas;
END;
$$ LANGUAGE plpgsql;
```

32

32

Criando SQLs dinâmicas dentro da Função

- Outra possibilidade é criar consultas SQL dinâmicas utilizando parâmetros ou variáveis da função como parte da cláusula SQL;

```
CREATE FUNCTION exclui_empregado(pcod int) RETURNS int4 AS $$
DECLARE
        linhas_afetadas int DEFAULT 0;
BEGIN
        DELETE FROM empregado WHERE cod = pcod;
        GET DIAGNOSTICS  linhas_afetadas = ROW_COUNT;
        RETURN  linhas_afetadas;
END;
$$ LANGUAGE plpgsql;
```

Variáveis e parâmetros não podem ter mesmo nome de campos ou tabelas do banco de dados

33

33

Criando SQLs dinâmicas dentro da Função

- Também é possível construir cláusulas SQL concatenando parâmetros ou variáveis a ela em tempo de execução;
- Para isso deve-se utilizar o comando EXECUTE;

```
CREATE FUNCTION apagaReg(tabela text, chave text, id int)  RETURNS int AS
$$
DECLARE linhas_afetadas  int DEFAULT 0;
BEGIN
        EXECUTE 'DELETE FROM ' || tabela || ' WHERE ' || chave || ' = ' || id;
        GET DIAGNOSTICS  linhas_afetadas = ROW_COUNT;
        RETURN  linhas_afetadas;
END;
$$ LANGUAGE 'plpgsql';
```

34

34

Criando SQLs dinâmicas dentro da Função

- Também é possível construir cláusulas SQL concatenando parâmetros ou variáveis a ela em tempo de execução;
- Para isso deve-se utilizar o comando EXECUTE;

```
CREATE FUNCTION apagaReg(tabela text, chave text, id int)  RETURNS int AS
$$
DECLARE linhas_afetadas  int DEFAULT 0;
BEGIN
      EXECUTE 'DELETE FROM ' || tabela || ' WHERE ' || chave || ' = ' || id;
      GET DIAGNOSTICS  linhas_afetadas = ROW_COUNT;
      RETURN  linhas_afetadas;
END;
$$ LANGUAGE 'plpgsql';
```

Concatenação de strings

35

Manipulando dados armazenados com SELECT INTO

- É possível manipular dados armazenados por meio da sua atribuição a uma variável;
- SELECT INTO é utilizado em consultas que geram apenas um registro como resultado;
- Sintaxe do comando:

SELECT ... **INTO nome_da_variavel** FROM ... ... ;

```
CREATE FUNCTION getEmpNome(pcod integer) RETURNS varchar AS
$$
DECLARE vNome varchar DEFAULT '';
BEGIN
      SELECT nome INTO vNome FROM empregado WHERE cod = pcod;
      RETURN vNome;
END;
$$ LANGUAGE plpgsql;
```

36

Tipos especiais em consultas SQL

- PL/pgSQL tem tipos especiais para recuperar dados por consultas SQL.
  - %TYPE
  - %ROWTYPE
  - RECORD

37

Tipos especiais em consultas SQL

- %TYPE
  - Tipo da variável será o mesmo que o atributo da tabela.

```
...
DECLARE
      dado empregado.nome%TYPE;
BEGIN
      SELECT nome INTO dado FROM empregado WHERE cod = pcod;
...
```

38

Tipos especiais em consultas SQL

- %ROWTYPE
  - Armazena uma linha inteira de uma dada tabela especificada.

```
...
DECLARE
      registroEmpregado empregado%ROWTYPE;
BEGIN
      SELECT * INTO  registroEmpregado FROM empregado WHERE cod = pcod;
...
```

39

39

Tipos especiais em consultas SQL

- RECORD
  - Semelhante ao %ROWTYPE, mas aceita uma linha de qualquer tabela.

```
...
DECLARE
      registroEmpregado RECORD;
BEGIN
      SELECT * INTO  registroEmpregado FROM empregado WHERE cod = pcod;
...
```

40

40

Manipulando cursores de dados armazenados com FOR IN

- Cursores são áreas de memória utilizadas pelo banco de dados para armazenar resultados de consultas enquanto eles são processados;
- FOR IN é utilizado para processar consultas que geram mais de um resultado;
- Comando de iteração: FOR ... IN

```
[ <<rotulo>> ]
FOR alvo IN consulta LOOP
    operações
END LOOP [ rotulo ];
```

41

41

Exemplo de uso do FOR IN

```
CREATE FUNCTION processaEmpregados() RETURNS void AS $$
DECLARE vEmpregado RECORD;
BEGIN
    FOR vEmpregado IN SELECT * FROM empregado LOOP
        vEmpregado.salario := vEmpregado.salario * 1.1;
    END LOOP;
    RETURN;
END;
$$ LANGUAGE plpgsql;
```

42

42

Como executar um função?

- Execução isolada da função:

```
SELECT nome_da_funcao();
```

- Execução da função dentro de um comando SQL:

```
SELECT * FROM empregado WHERE salario >= mediaDeSalarios();
```