Linguagem PL/pgSQL
Criando procedimentos/funções/Triggers dentro do SGBD

Profa.Ticiana Linhares

Linguagem PL/pgSQL

- Pode ser utilizada para criar funções e procedimentos invocados em triggers;
- Adiciona estruturas de controle a linguagem SQL;
- Suporta computações complexas;
- É de fácil uso;
- Aceita todos os tipos definido pelos usuários, funções e operadores;

Vantagens de usar PL/pgSQL

- Agrupar blocos ou um conjunto de consultas dentro do servidor de banco de dados;
- Resultados intermediários que não precisam ser conhecidos pelo usuário;
- Fase de *parser* de algumas consultas podem ser evitadas;
- Pode-se utilizar todos os tipos de dados, operadores e funções do SQL;

Manutenção da Integridade Semântica

- No modelo relacional, as integridades estruturais – chave, entidade, domínio e referencial – podem ser controladas com a criação das tabelas!
- Integridades semânticas (regras de negócio – *business rules*) podem ser controladas na aplicação (não recomendado) ou por meio de visões (com *check option*), procedimentos armazenados (*stored procedures*), gatilhos (*triggers*) e funções.

Funções em PL/pgSQL

- Aceitam como argumentos qualquer escalar ou array (suportados pelo servidor), também podendo retornar tais tipos;
- Além disso, aceitam/retornam tipos de dados compostos (row type).
  - Retorno de um registro (record);
  - tipo de linha com colunas especificadas na consulta.

Funções PL/pgSQL

- Podem retornar um "*set*" (ou tabela) de tipos de dados;
  - Saída da função é gerada executando RETURN NEXT para cada elemento desajado do result set;
  - RETURN QUERY para retornar todo o resultado da consulta.
- Retorna void, caso não seja necessário retornar algum valor.

Síntaxe de Funções

```
CREATE [OR REPLACE] FUNCTION nome_funcao
[[parametro1 [IN | OUT | IN OUT] tipo], …]
RETURN tipo AS
[ DECLARE
        declaração de variáveis, cursores,...]
BEGIN
        comandos em SQL, de controle…
END
[ nome_funcao ];
```

- Varia de SGBD para SGBD;
- Comentários por meio de -- ou /**/;
- Fim de um comando por “;”.

Exemplo de Criação de Função

Empregado(Matricula, Nome, Salario, Num_Dept)

- Considere a tabela acima;
- Crie uma função (armazenada!) que recebe como entrada o número de departamento e retorna o somatório de salários dos seus empregados.

Exemplo de Criação de Função

Empregado(Matricula, Nome, Salario, Num_Dept)

```
CREATE or REPLACE FUNCTION somasalario(numdept int) RETURNS real AS $$
DECLARE
   somatorio real;
BEGIN
  select  SUM(salario) into somatorio
  from empregado
  where num_dept=numdept;

  RETURN somatorio;
END;
$$ LANGUAGE plpgsql;
```

```
select somasalario(1);
```

Estrutura de Controle - Condição

- IF ... THEN END IF;
- IF ... THEN ... ELSE END IF;
- IF ... THEN ... ELSE IF END IF; END IF;
- IF ... THEN ... ELSIF ... THEN ... ELSE END IF;
- IF ... THEN ... ELSEIF ... THEN ... ELSE END IF;

```
IF number = 0 THEN
            result := 'zero';
ELSIF number > 0 THEN
            result := 'positive';
ELSIF number < 0 THEN
            result := 'negative';
ELSE
            --the only other possibility is that number is null
            result := 'NULL';
END IF;
```

Estrutura de Controle - Laço

```
[ <<label>> ]
LOOP
    comandos
END LOOP [ label ]
```

```
EXIT [ label ] [ WHEN boolean-expression ];
```

```
LOOP – algumas computacoes
        IF count > 0 THEN
                EXIT; -- exit loop
        END IF;
END LOOP;
LOOP
        -- algumas computacoes
        EXIT WHEN count > 0;
END LOOP;
BEGIN
        -- algumas computacoes
        IF valor > 100000 THEN
                EXIT; -- sai do bloco de begin
        END IF;
END;
```

Semelhante ao break das linguagens de programação;

Estrutura de Controle - Laço

```
[ <<label>> ]
LOOP
    comandos
END LOOP [ label ]
```

```
EXIT [ label ] [ WHEN boolean-expression ];
```

```
LOOP – algumas computacoes
        IF count > 0 THEN
                EXIT; -- exit loop
        END IF;
END LOOP;
LOOP
        -- algumas computacoes
        EXIT WHEN count > 0;
END LOOP;
BEGIN
        -- algumas computacoes
        IF valor > 100000  THEN
                EXIT; -- sai do bloco de begin
        END IF;
END;
```

Semelhante ao break das linguagens de programação;

Cursor

- Ao invés de executar toda a consulta de uma vez, a consulta é encapsulada e a leitura dos seus resultados (algumas tuplas por vez) é feita.
- Uma razão para isso é evitar a quantidade de memória em excesso gasta quando o resultado contém um grande número de tuplas;
- O laço **for** usa um cursor internamente para evitar problemas de memória;

```
DECLARE
        curs1 refcursor;
        curs2 CURSOR FOR SELECT * FROM Empregado;
        curs3 CURSOR (key integer) IS SELECT * FROM Empregado WHERE
matricula= key;
```

Unbound. Consulta definida posteriormente.

Fetch Cursor

```
FETCH [ direction { FROM | IN } ] cursor INTO target;
```

```
FETCH curs1 INTO x;
FETCH curs2 INTO x, y, z;
FETCH LAST FROM curs3 INTO x, y;
```

Close Cursor

```
CLOSE curs1;
```

Criando rotinas em PL/pgSQL

```
CREATE OR REPLACE FUNCTION  cursor_empregado() RETURNS void AS
$$
DECLARE
        cursor1 CURSOR  IS SELECT * FROM EMPREGADO;
        emp  empregado%ROWTYPE;
BEGIN
        open cursor1;
        fetch cursor1 into emp;
        loop
                exit when not found;
                RAISE NOTICE 'Empregado de matricula: %', en
                fetch cursor1 into emp;
        end loop;
        close cursor1;
END;
$BODY$
LANGUAGE 'plpgsql';
```

Abre o cursor

Declara a variável cursor

Coloca a próxima linha do cursor na variável

Fecha cursor

Exemplo de Criação de Procedimento

Empregado(Matricula, Nome, Salario, Num_Dept)
Trabalha(Mat_Empr, Num_Proj, Horas)

- Considere as duas tabelas acima;
- Crie um procedimento (armazenada!) que produz um relatório de negócio contendo para cada empregado o número de horas trabalhadas em projetos.

Exemplo de Criação de Procedimento

Empregado(Matricula, Nome, Salario, Num_Dept) Trabalha(Mat_Empr, Num_Proj, Horas)

```
CREATE or REPLACE FUNCTION relatorio() RETURNS void AS $$
DECLARE
  mat int;
  qtdhoras int;
  meu_cursor cursor is select matricula, SUM(horas)
                       from Empregado E, Trabalha T
                       where E.matricula=T.mat_empr
                       group by matricula;
BEGIN
  OPEN meu_cursor;
  RAISE INFO 'Empregado – TOTAL DE HORAS';
  LOOP
    fetch meu_cursor into mat,qtdhoras;
    If not found then exit;
    end if;
    RAISE INFO '% - %', mat, qtdhoras;
  END LOOP;
  CLOSE meu_cursor;
  RETURN somatorio;
END;
$$ LANGUAGE plpgsql;
```

Triggers

- Invocado (disparado) quando ocorre uma tentativa de modificação nos dados da tabela (relação) à qual ele está associado (vinculado).
- Podem ser invocados por cada linha afetada ou por cada comando INSERT, UPDATE ou DELETE.
- Podem ser definidos para executar antes ou depois de qualquer modificação no banco de dados.

Triggers

- Os triggers são usados para garantir regras de negócio e integridade dos dados, ou para ações complexas, tais como automaticamente atualizar um resumo dos dados.
- Caso a execução de um trigger possa violar alguma restrição, então o comando não será executado/ou será abortado.
- São automaticamente disparados pelo Postgres.

Síntaxe de Triggers

```
CREATE [OR REPLACE] FUNCTION nome_funcao
[[parametro1 [IN | OUT | IN OUT] tipo], …]
RETURNS TRIGGER AS
[ DECLARE
        declaração de variáveis, cursores,...]
BEGIN
        comandos em SQL, de controle…
END
[ nome_funcao ];
```

```
CREATE TRIGGER "NOME_TRIGGER"
  BEFORE INSERT/UPDATE/DELETE
  ON “nome_tabela"
  FOR EACH ROW
  EXECUTE PROCEDURE nome_funcao;
```

Exemplo de Trigger

- Regra de negócio: Empregado devem ter salário igual ou superior a R$700,00.

```
CREATE OR REPLACE FUNCTION func_ins () RETURNS trigger AS
$$
BEGIN
        if new.salario>= 700 THEN
                RETURN new;
        else RETURN null;
        End if;
END;
$$
LANGUAGE 'plpgsql' ;
```


```
CREATE TRIGGER "ins_empregado"
BEFORE INSERT
ON “empregado"
FOR EACH ROW
EXECUTE PROCEDURE func_ins();
```

Dicas

```
declare
        dataFrequencia  char(20);
        timestampFrequencia     timestamp;
        dataFrequencia  = ano || '-' || mes || '-' || dia;
        timestampFrequencia= CAST (dataFrequencia AS timestamp);

now();
date_part(year, freq_data);
CAST ($2 AS character);
CAST (date_part('month', freq_data) as integer);
```