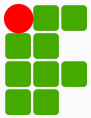
Banco de Dados I

11 - Funções e Gatilhos

Marcos Roberto Ribeiro



Instituto Federal Minas Gerais - Campus Bambuí

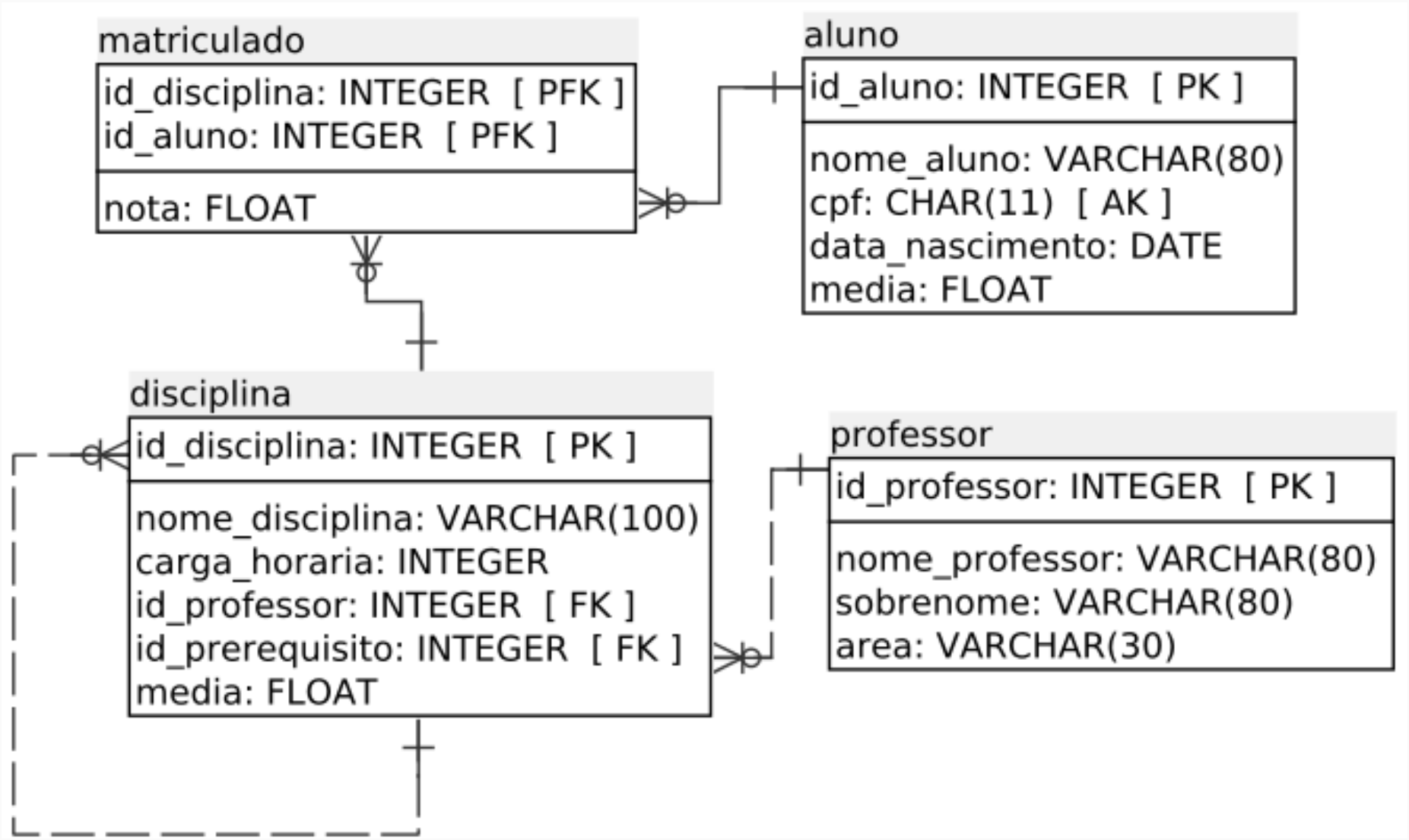
2018

Introdução

- A maioria dos SGBD permitem a programação interna, através de funções ou procedimentos cujo código é armazenado juntamente com o banco de dados;
- Estes códigos são comumente chamados de procedimentos armazenados (*stored procedures*) e podem ser escritos em diversas linguagens suportadas pelo SGBD;
- A utilização dos procedimentos armazenados pode trazer vantagens como:
  - Reduzir o tráfego de dados entre aplicação e SGBD;
  - Melhorar a performance das aplicações;
  - Redução de código em aplicações (com linguagens diferentes).
- No PostgreSQL os procedimentos armazenados são mais conhecidos como funções e possuem os seguintes tipos:
  - Funções na linguagem SQL;
  - Funções em linguagens procedurais;
  - Funções internas;
  - Funções na linguagem C.

Banco de Dados Acadêmico 2

Em nossos exemplos vamos utilizar o banco de dados acadêmico 2:



Funções na Linguagem SQL

- AS funções na linguagem SQL podem executar diversas instruções SQL;
- A última instrução, normalmente, é utilizada para retornar o resultado da função;
- As funções podem não retornar nenhum resultado (**void**), retornar um único valor ou um conjunto (**SETOF**) de valores.

Funções que Não Retornam Dados

- As funções que não retornam dados são usadas para modificação de dados. Por exemplo:

```
CREATE FUNCTION atualiza_media_alunos()
RETURNS void AS $$
  WITH m AS (
    SELECT id_aluno, AVG(nota) AS media
    FROM matriculado
    WHERE nota <> -1
    GROUP BY id_aluno)
  UPDATE aluno AS a
  SET media = m.media
  FROM m
  WHERE a.id_aluno = m.id_aluno;
$$ LANGUAGE SQL;
```

Executando Funções

- A execução das funções pode ser feita com uma instrução **SELECT**. Por exemplo:

```
SELECT atualiza_media_alunos();
```

- Modifique as médias dos alunos e execute a função para atualizá-las, observe se a atualização aconteceu corretamente.

Passagem de Parâmetros

- As funções podem receber parâmetros que são referenciados no corpo da função como **$n**, onde **n** é a ordem do parâmetro. Exemplo:

```
CREATE FUNCTION soma(INT, INT)
RETURNS INT AS $$
  SELECT $1 + $2;
$$ LANGUAGE SQL;

SELECT soma(14, 42);
```

Função Útil I

- Uma função mais útil poderia ser somar um valor a nota de um aluno em uma disciplina:

```
CREATE FUNCTION soma_nota(INT, INT, FLOAT)
RETURNS void AS $$
  UPDATE matriculado
  SET nota = nota + $3
  WHERE id_disciplina = $1
    AND id_aluno = $2;
$$ LANGUAGE SQL;
```

- Quais são os parâmetros desta função?
- O que acontece se o aluno e disciplina informados não existirem?
- Como melhorar esta função?

Função Útil II

```
DROP FUNCTION soma_nota(INT, INT, FLOAT);

CREATE FUNCTION soma_nota(INT, INT, FLOAT)
RETURNS float AS $$
  UPDATE matriculado
  SET nota = nota + $3
  WHERE id_disciplina = $1
    AND id_aluno = $2;

  SELECT nota
  FROM matriculado
  WHERE id_disciplina = $1
    AND id_aluno = $2;
$$ LANGUAGE SQL;
```

Função Útil III

- Por que foi preciso o **DROP**?

Funções de Tipos Compostos

- As funções que vimos até agora, recebem tipos de dados básicos e retornam tipos básicos. Agora um exemplo de função recebe um tipo composto:

```
CREATE FUNCTION melhora_media(aluno)
RETURNS FLOAT AS $$
  SELECT CASE
         WHEN $1.media < 60 THEN 60
         ELSE $1.media
         END AS media_sonho;
$$ LANGUAGE SQL;

SELECT a.*, melhora_media(a.*)
FROM aluno AS a;

SELECT 'Tereza', melhora_media(ROW(0,'',NULL,NULL,50));
```

- Que tipo composto a função recebeu? Para que serve **ROW**?

Funções de Tipos Compostos

- Agora uma função que retorna um tipo composto:

```
CREATE FUNCTION novo_aluno(VARCHAR)
RETURNS aluno AS $$
  INSERT INTO aluno(nome_aluno)
  VALUES ($1);

  SELECT * FROM aluno
  WHERE nome_aluno = $1
$$ LANGUAGE SQL;

SELECT * FROM novo_aluno('Teodoro');
```

- O que esta função faz?

Funções que Retornam Conjunto

- Todas as funções vistas até o momento retornam um único valor, mesmo que a consulta obtenha várias linhas, apenas a primeira é retornada;
- As funções podem retornar conjuntos de dados usando tipo **SETOF algum tipo**. Exemplo:

```
CREATE FUNCTION disciplinas_aluno(INT)
RETURNS SETOF matriculado AS $$
  SELECT * FROM matriculado
  WHERE id_aluno = $1
$$ LANGUAGE SQL;

SELECT * FROM disciplinas_aluno(1);
```

Sobrecarga de Função

- Os nomes de funções podem ser sobrecarregados, desde que tenham parâmetros diferentes. Exemplo:

```
CREATE FUNCTION novo_aluno(VARCHAR)
RETURNS aluno AS $$
  INSERT INTO aluno(nome_aluno) VALUES ($1);
  SELECT * FROM aluno WHERE nome_aluno = $1
$$ LANGUAGE SQL;

CREATE FUNCTION novo_aluno(VARCHAR, CHAR)
RETURNS aluno AS $$
  INSERT INTO aluno(nome_aluno, cpf) VALUES ($1, $2);

  SELECT * FROM aluno WHERE nome_aluno = $1
$$ LANGUAGE SQL;
```

Linguagens Procedurais

- A grande maioria dos SGBD permitem a criação de funções através de *linguagens procedurais*, estas linguagens são muito próximas das linguagens de programação para desenvolvimento de sistemas aplicativos;
- o PostgreSQL possui as seguintes linguagens procedurais por padrão:
  - PL/pgSQL
  - PL/Tcl
  - PL/Perl
  - PL/Python
- Em nosso curso abordaremos apenas a linguagem PL/pgSQL;

Funções com a Linguagem PL/pgSQL

- Basicamente as funções na linguagem PL/pgSQL possuem a seguinte estrutura:

```
CREATE [OR REPLACE] FUNCTION <nome da função>(<parâmetros>)
 ↪  RETURNS <tipo retornado> AS $$
[ DECLARE
  <declarações> ]
BEGIN
  <comandos>
END;
$$ LANGUAGE plpgsql;
```

Exemplo Simples

```
CREATE FUNCTION ola_mundo() RETURNS VARCHAR AS $$
BEGIN
  RETURN 'Olá, mundo';
END;
$$ LANGUAGE plpgsql;

SELECT ola_mundo();
```

Declarações

- As variáveis da linguagem PL/pgSQL podem possuir qualquer tipo de dado da linguagem SQL;
- A declaração de variáveis segue a seguinte sintaxe:

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

- Exemplo:

```
id_usuario  integer;
quantidade  numeric(5);
url         varchar;
minha_linha nome_da_tabela%ROWTYPE;
meu_campo   nome_da_tabela.nome_da_coluna%TYPE;
uma_linha   RECORD;
```

Exemplo Simples

```
CREATE OR REPLACE FUNCTION ola_mundo(nome VARCHAR) RETURNS
↪  VARCHAR AS $$
DECLARE
  mensagem VARCHAR := 'Olá, mundo. ';
BEGIN
  mensagem := mensagem || 'Olá, ' || nome;
  RETURN mensagem;
END;
$$ LANGUAGE plpgsql;

SELECT ola_mundo('José');
```

Cópia de Tipo

- A expressão %TYPE fornece o tipo de dado da variável ou da coluna da tabela. Pode ser utilizada para declarar variáveis que armazenam valores do banco de dados. Por exemplo, supondo que exista uma coluna chamada **id_aluno** na tabela aluno, para declarar uma variável com o mesmo tipo de dado de **aluno.id_aluno** deve ser escrito:

```
id aluno.id_aluno%TYPE;
```

- Utilizando %TYPE não é necessário conhecer o tipo de dado da estrutura sendo referenciada e, ainda mais importante, se o tipo de dado do item referenciado mudar no futuro, não será necessário mudar a definição na função.

Exemplo Simples

```
CREATE OR REPLACE FUNCTION aluno_nome(id aluno.id_aluno%TYPE)
 ↪  RETURNS aluno.nome_aluno%TYPE AS $$
DECLARE
  nome aluno.nome_aluno%TYPE;
BEGIN
  SELECT nome_aluno INTO nome FROM aluno WHERE id_aluno = id;
  RETURN nome;
END;
$$ LANGUAGE plpgsql;

SELECT aluno_nome(1);
```

- A instrução **SELECT campos INTO variáveis ...** guarda na variável o valor do campo retornado pela consulta.

Tipo de Linha

- Uma variável de tipo **tabela%ROWTYPE** pode armazenar toda uma linha de resultado de um **SELECT**;
- Os campos são acessados utilizando a notação de ponto.

Exemplo

```
CREATE OR REPLACE FUNCTION prof_nome_completo(id
 ↪  professor.id_professor%TYPE) RETURNS VARCHAR AS $$
DECLARE
  linha professor%ROWTYPE;
BEGIN
  SELECT * INTO linha FROM professor WHERE id_professor = id;
  RETURN linha.nome_professor || ' ' || linha.sobrenome;
END;
$$ LANGUAGE plpgsql;

SELECT prof_nome_completo(10);
```

Estruturas Condicionais

- A **PL/pgSQL** possui as seguintes variações de **IF**:

```
IF ... THEN
IF ... THEN ... ELSE
IF ... THEN ... ELSE IF
IF ... THEN ... ELSIF ... THEN ... ELSE
```

Exemplos

```
CREATE FUNCTION maior(n1 INT, n2 INT) RETURNS INT AS $$
DECLARE
  maior INT;
BEGIN
  IF (n1 > n2) THEN
    maior := n1;
  ELSE
    maior := n2;
  END IF;
  RETURN maior;
END;
$$ LANGUAGE plpgsql;

SELECT maior(3,8);
```

Exemplos

```
CREATE FUNCTION maior_tres(n1 INT, n2 INT, n3 INT) RETURNS INT AS
 ↪  $$
DECLARE
  maior INT;
BEGIN
  IF (n1 > n2) AND (n1 > n3) THEN
    maior := n1;
  ELSIF (n2 > n3) THEN
    maior := n2;
  ELSE
    maior := n3;
  END IF;
  RETURN maior;
END;
$$ LANGUAGE plpgsql;

SELECT maior_tres(3,5,8);
```

CASE I

```
CREATE FUNCTION mes_extenso(mes INT) RETURNS TEXT AS $$
DECLARE
  nome TEXT;
BEGIN
  CASE mes
  WHEN 1 THEN nome = 'janeiro';
  WHEN 2 THEN nome = 'fevereiro';
  WHEN 3 THEN nome = 'março';
  WHEN 4 THEN nome = 'abril';
  WHEN 5 THEN nome = 'maio';
  WHEN 6 THEN nome = 'junho';
  WHEN 7 THEN nome = 'julho';
  WHEN 8 THEN nome = 'agosto';
  WHEN 9 THEN nome = 'setembro';
```

CASE II

```
  WHEN 10 THEN nome = 'outubro';
  WHEN 11 THEN nome = 'novembro';
  WHEN 12 THEN nome = 'dezembro';
  ELSE nome = 'INVÁLIDO';
  END CASE;
  RETURN nome;
END;
$$ LANGUAGE plpgsql;

SELECT mes_extenso(3);
SELECT mes_extenso(13);
```

Laços de Repetições

- A linguagem **PL/pgSQL** possui três tipos de laços:
  - **LOOP**
  - **WHILE**
  - **FOR**

Exemplo com LOOP

```
CREATE FUNCTION somatorio(n1 INT, n2 INT) RETURNS INT AS $$
DECLARE
  soma INT := 0;
BEGIN
  LOOP
    soma := soma + n1;
    n1 := n1 + 1;
    EXIT WHEN n1 > n2;
  END LOOP;
  RETURN soma;
END;
$$ LANGUAGE plpgsql;

SELECT somatorio(21,54);
```

Exemplo com WHILE

```
CREATE OR REPLACE FUNCTION somatorio(n1 INT, n2 INT) RETURNS
 ↪  INT AS $$
DECLARE
  soma INT := 0;
BEGIN
  WHILE n1 <= n2 LOOP
    soma := soma + n1;
    n1 := n1 + 1;
  END LOOP;
  RETURN soma;
END;
$$ LANGUAGE plpgsql;

SELECT somatorio(21,54);
```

Exemplo com FOR

```
CREATE OR REPLACE FUNCTION somatorio(n1 INT, n2 INT) RETURNS
 ↪  INT AS $$
DECLARE
  soma INT := 0;
BEGIN
  FOR i IN n1..n2 LOOP
    soma := soma + i;
  END LOOP;
  RETURN soma;
END;
$$ LANGUAGE plpgsql;

SELECT somatorio(21,54);
```

Laço sobre Resultados de Consultas

O **FOR** pode ser usado também para interagir com o resultado de consultas.

Exemplo

```
CREATE OR REPLACE FUNCTION num_alunos_media() RETURNS INT AS $$
DECLARE
  linha aluno%ROWTYPE;
  n INT := 0;
BEGIN
  FOR linha IN SELECT * FROM aluno LOOP
    IF linha.media > 70 THEN
      n := n + 1;
    END IF;
  END LOOP;
  RETURN n;
END;
$$ LANGUAGE plpgsql;

SELECT num_alunos_media();
```

Erros e Mensagens

- A instrução **RAISE** é utilizada para gerar mensagens informativas e causar erros:

```
RAISE <nível> 'formato' [, variável [, ...]];
```

- Os níveis possíveis são **DEBUG**, **LOG**, **INFO**, **NOTICE**, **WARNING**, e **EXCEPTION**.
- O nível EXCEPTION causa um erro (que normalmente interrompe a transação corrente); os outros níveis apenas geram mensagens com diferentes níveis de prioridade (informação simples, log, e outras).
- Dentro da cadeia de caracteres de formatação, o caractere **%** é substituído pelo valor de uma variável ou expressão, na ordem pela qual são passadas.

Exemplo com RAISE

```
CREATE OR REPLACE FUNCTION repete(texto TEXT, n INT) RETURNS
↪  VOID AS $$
BEGIN
  FOR i IN 1..n LOOP
    RAISE NOTICE 'Iteração % -> %', i, texto;
  END LOOP;
END;
$$ LANGUAGE plpgsql;

SELECT repete('teste',10);
```

- A informação impressa não é o retorno da função, são apenas mensagens geradas pelo **RAISE**
- Esta função não retorna nenhum dado por ser do tipo **VOID**

SELECT INTO

- Já utilizamos a instrução **SELECT ... INTO**, mas agora vamos destacar alguns detalhes importantes;
- Se a consulta não retornar nenhuma linha, são atribuídos valores nulos aos destinos. Se a consulta retornar várias linhas, a primeira linha é atribuída aos destinos e as demais são desprezadas;
- A variável especial **FOUND** pode ser verificada imediatamente após a instrução **SELECT INTO** para determinar se foi retornada pelo menos uma linha pela consulta.

Exemplo

```
CREATE OR REPLACE FUNCTION prof_nome_completo2(id
 ↪  professor.id_professor%TYPE) RETURNS VARCHAR AS $$
DECLARE
  linha professor%ROWTYPE;
BEGIN
  SELECT * INTO linha FROM professor WHERE id_professor = id;
  IF NOT FOUND THEN
    RAISE NOTICE 'Professor não encontrado!';
  END IF;
  RETURN linha.nome_professor || ' ' || linha.sobrenome;
END;
$$ LANGUAGE plpgsql;

SELECT prof_nome_completo2(10);
SELECT prof_nome_completo2(11);
```

Retornando Conjuntos

- Funções **PL/pgSQL** que retorna conjuntos, devem utilizar os comandos **RETURN NEXT** ou **RETURN QUERY**;
- Os elementos são retornados com **RETURN NEXT** ou **RETURN QUERY**;
- Um **RETURN** sem parâmetros é usado no final da função para informar que não existem elementos para serem retornados.

Exemplo

```
CREATE OR REPLACE FUNCTION alunos_acima(nota REAL) RETURNS
↪  SETOF aluno AS $$
DECLARE
  linha aluno%ROWTYPE;
BEGIN
  FOR linha IN SELECT * FROM aluno LOOP
    IF linha.media >= nota THEN
      RETURN NEXT linha;
    END IF;
  END LOOP;
  RETURN;
END;
$$ LANGUAGE plpgsql;

SELECT * FROM alunos_acima(70);
SELECT * FROM alunos_acima(0);
```

Exemplo

```
CREATE OR REPLACE FUNCTION alunos_acima2(nota REAL) RETURNS
 ↪  SETOF aluno AS $$
BEGIN
  RETURN QUERY SELECT * FROM aluno WHERE media >= nota;
  RETURN;
END;
$$ LANGUAGE plpgsql;

SELECT * FROM alunos_acima2(70);
SELECT * FROM alunos_acima2(0);
```

- Quando usar **RETURN NEXT** ou **RETURN QUERY**?

Observação

RETURN NEXT e **RETURN QUERY** armazenam todo o resultado na memória antes de retornar, o que pode causas uso exagerado de memória e disco se o resultado muito grande.

Cursores I

- Em vez de executar toda a consulta de uma vez, é possível definir cursores e ler linha por linha do resultado da consulta;
- Um dos motivos de se fazer desta maneira, é para evitar o uso excessivo de memória quando o resultado contém muitas linhas;
- Na linguagem **PL/pgSQL** os laços **FOR ... IN SELECT** já utilizam implicitamente cursores.

Declaração de Cursores

```
DECLARE
  -- Cursor desligado
  curs1 refcursor;
  -- Cursor ligado
  curs2 CURSOR IS SELECT * FROM aluno;
  curs3 CURSOR (id INT) IS SELECT * FROM aluno WHERE id_aluno
↪   = id;
```

Abertura de Cursores

- Antes do cursor poder ser utilizado para trazer linhas, este deve ser aberto;
- Somente um cursor fechado pode ser aberto.

Exemplo

```
  -- Abrindo cursor desligado
  OPEN curs1 FOR SELECT * FROM disciplina WHERE carga_horaria
↪   = 80;
  -- Abrindo cursor ligado
  OPEN curs2;
  OPEN curs3(42);
```

Manipulado Cursores

- Após a abertura do cursos podemos navegar pelo resultado da consulta usando o comando **FETCH**;
- Por padrão o **FETCH** sempre passa para o próximo registro da consulta. Mas podem ser usadas outras posições como **PRIOR** (anterior), **FIRST** (primeira) e **LAST** (última);
- Como no **SELECT INTO**, pode ser verificada a variável especial **FOUND** para ver se foi obtida uma linha, ou não.

Exemplo

```
FETCH curs1 INTO variavel_linha;
FETCH LAST curs2 INTO id, nome, ch, prof;
```

Fechamento de Cursores

- Quando o cursor não for mais usado é conveniente que o mesmo seja fechado para liberar a memória utilizada.
- O fechamento é feito com a instrução **CLOSE**;

Exemplo

```
CLOSE curs1;
```

```
CREATE OR REPLACE FUNCTION alu_notas_abaixo_media() RETURNS SETOF RECORD
 ↪   AS $$
DECLARE
  c CURSOR IS SELECT * FROM aluno;
  alu_atual aluno%ROWTYPE;
BEGIN
  OPEN c;
  LOOP
    FETCH c INTO alu_atual;
    EXIT WHEN NOT FOUND;
    RETURN QUERY
     SELECT alu_atual.id_aluno, alu_atual.nome_aluno, COUNT(*) AS
 ↪   num_notas_abaixo
     FROM matriculado AS m
     WHERE m.id_aluno <> alu_atual.id_aluno
       AND m.nota < alu_atual.media AND m.nota <> -1;
  END LOOP;
  CLOSE c;
  RETURN;
```

```
END;
$$ LANGUAGE plpgsql;
```

```
SELECT * FROM alu_notas_abaixo_media()
AS (id int, nome varchar, num_notas_abaixo bigint);
```

- O que esta função faz exatamente?
- Como fazer esta função usando um **FOR ... IN SELECT**?

Gatilhos (Triggers)

- A linguagem **PL/pgSQL** pode ser utilizada para definir procedimentos de gatilho (*triggers*) semelhantes as regras (*rules*). No caso dos gatilhos é possível fazer manipulações mais poderosas no banco de dados;
- Para se criar um gatilho é necessário que já exista uma função, a qual será usada como ação disparada por um evento. Esta função deverá ter o retorno do tipo **TRIGGER**.
- Sintaxe:

```
CREATE TRIGGER <nome> { BEFORE | AFTER } { <evento> [ OR ...
↪  ] }
ON <tabela> [ FOR [ EACH ] { ROW | STATEMENT } ]
EXECUTE PROCEDURE <função( argumentos )>;
```

Variáveis Especiais

Funções para gatilhos possuem diversas variáveis especiais:

NEW	registro contendo a nova linha para **INSERT** ou **UPDATE**;
OLD	registro contendo a antiga linha para **UPDATE** ou **DELETE**;
TG_NAME	contém o nome do gatilho disparado;
TG_WHEN	**'BEFORE'** ou **'AFTER'**, depende da definição do gatilho;
TG_LEVEL	contém **'ROW'** ou **'STATEMENT'**, depende da definição do gatilho;
TG_OP	contém **'INSERT'**, **'UPDATE'**, ou **'DELETE'**, informando para qual operação o gatilho foi disparado;
TG_TABLE_NAME	nome da tabela que causou o disparo do gatilho;
TG_NARGS	número de argumentos fornecidos ao procedimento de gatilho na instrução **CREATE TRIGGER**;
TG_ARGV[]	argumentos da instrução **CREATE TRIGGER**. O contador do índice começa por 0. Índices inválidos resultam em um valor nulo.

Tipos Retornados

- Uma função de gatilho deve retornar nulo, ou um registro/linha possuindo a mesma estrutura da tabela para a qual o gatilho foi disparado;
- Os gatilhos disparados antes (**BEFORE**) podem retornar nulo, para sinalizar ao gerenciador de gatilhos para pular o restante da operação (ou seja, os gatilhos posteriores não serão disparados, e não ocorrerá o **INSERT**, **UPDATE**, ou **DELETE** para esta linha)
- Se for retornado um valor diferente de nulo, então a operação prossegue com este valor de linha. Retornar um valor de linha diferente do valor original de **NEW** altera a linha que será inserida ou atualizada.

Exemplo I

```
CREATE OR REPLACE FUNCTION func_gatilho_matriculado_media()
↪  RETURNS TRIGGER AS $$
DECLARE
  c CURSOR IS SELECT * FROM aluno;
  alu_atual aluno%ROWTYPE;
BEGIN
  UPDATE aluno AS a
  SET media = m.media
  FROM (SELECT id_aluno, AVG(nota) AS media
        FROM matriculado
        WHERE nota <> -1
        GROUP BY id_aluno) AS m
  WHERE a.id_aluno = m.id_aluno;
  RETURN NULL;
```

Exemplo II

```
END;
$$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER func_gatilho_matriculado_media
AFTER INSERT OR UPDATE OR DELETE
ON matriculado FOR EACH STATEMENT
EXECUTE PROCEDURE func_gatilho_matriculado_media();
```

- Este gatilho funcionaria para **EACH ROW**? E o funcionamento é eficiente?

Referências

(2012).
Postgresql documentation.

Elmasri, R. and Navathe, S. B. (2011).
Sistemas de banco de dados.
Pearson Addison Wesley, São Paulo, 6 edition.

Ramakrishnan, R. and Gehrke, J. (2008).
Sistemas de gerenciamento de banco de dados.
McGrawHill, São Paulo, 3 edition.