

[MAC0313]

Introdução aos Sistemas de Bancos de Dados

Aula 21

Linguagem SQL (Parte 6):

Regras Ativas (*Triggers*)

Kelly Rosa Braghetto

DCC-IME-USP

25 de outubro de 2016

# Bancos de Dados Ativos

- ▶ Muitos SGBDRs possuem funcionalidades relacionadas à criação e execução de **regras ativas** (que, nesse contexto, costumam ser chamadas de *gatilhos* – ou *triggers*)
  - ▶ Elas estão incluídas no padrão SQL:1999 (SQL3)
- ▶ Regras ativas reagem de forma autônoma a eventos que ocorrem num BD
- ▶ Eventos são causados por transações executadas sobre os dados do BD

# Regras semânticas das aplicações

O processamento das regras garante um **comportamento reativo** do BD, que difere do **comportamento passivo** dos SGBDs convencionais.

- ▶ Nos BDs ativos, parte da semântica que geralmente é codificada dentro das aplicações de software pode ser expressa por meio das regras ativas.
- ▶ As regras semânticas podem ser codificadas uma só vez no BD, e ficar automaticamente compartilhadas entre todos os usuários e aplicações que acessam o BD.

# Regras ativas

## Semântica:

- ▶ Regras ativas se baseiam no paradigma **Evento-Condição-Ação** (ECA)
- ▶ Semântica do ECA é simples e intuitiva:

**quando** o evento ocorre,  
**se** a condição é satisfeita,  
**então** execute a ação

- ▶ Essa semântica básica é seguida pela maioria dos sistemas de regras ativas

# Regra ativa

## Exemplo

```
CREATE RULE ControleSalario ON FUNCIONARIO
WHEN  INSERTED, DELETED, UPDATED (Salario)
IF    (SELECT AVG(Salario) FROM FUNCIONARIO) > 2000
THEN  UPDATE FUNCIONARIO
      SET Salario = .9 * Salario
```

Regra: Quando a média dos salários dos funcionários ultrapassar 100, então o salário de todos os funcionários deve ser reduzido em 10%.

Obs.: Esse exemplo não está na sintaxe da SQL, ele é apenas um pseudo-código.

# Regras ativas

Dizemos-que uma regra ativa é:

- ▶ **disparada** quando o seu evento de interesse ocorre,
- ▶ **considerada** quando sua condição é avaliada e
- ▶ **executada** quando sua ação é feita.

# Componentes de uma regra ativa

## Evento-Condição-Ação

- ▶ **Eventos** – são primitivas para *mudanças de estado* em BDs – ou seja, inserções, alterações e remoções de tuplas. Alguns sistemas podem também monitorar:
  - ▶ consultas
  - ▶ eventos relacionados ao *tempo* (e.g., às 17h, toda sexta-feira)
  - ▶ eventos externos, gerados explicitamente por aplicações de software que interagem com o BD

# Componentes de uma regra ativa

## Evento-Condição-Ação

- ▶ **Condição** – pode ser tanto um *predicado* sobre o banco de dados quanto uma *consulta*.
  - ▶ Um predicado (= condição) deve retornar VERDADEIRO ou FALSO
  - ▶ No caso de uma condição expressa por meio de uma consulta, a consulta é interpretada como VERDADEIRO se ela contém ao menos uma tupla na resposta, e como FALSO, no caso contrário.



# Componentes de uma regra ativa

## Evento-Condição-Ação

- ▶ **Ação** – é um procedimento qualquer de manipulação de dados. Ela pode, inclusive:
  - ▶ ter comandos transacionais (como *ROLLBACK*)
  - ▶ ter comandos de manipulação de regras (como a ativação e desativação de regras ativas ou de grupos de regras ativas)
  - ▶ ativar procedimentos externos ao BD

## Sobre os eventos monitorados

- ▶ Geralmente, uma regra monitora uma coleção de eventos, com uma semântica implícita disjuntiva
  - ▶ a regra é disparada quando qualquer um dos seus eventos monitorados ocorrer
- ▶ Em alguns sistemas, é possível verificar na condição da regra qual foi o evento que a disparou
- ▶ Alguns sistemas suportam *linguagens de eventos* mais ricas, que permitem que eventos complexos sejam definidos a partir de eventos mais simples, por meio de operadores como os de conjunção, disjunção, negação e precedência

## Momento de consideração de uma regra ativa

Geralmente, a condição é avaliada como parte da mesma transação do evento de disparo

A consideração pode acontecer:

- ▶ **ANTES** que o evento de disparo seja executado, ou
- ▶ **DEPOIS** do evento, ou
- ▶ **NO LUGAR** do evento (nesse caso, a primitiva do evento não é executada)

# Monitoramento das mudanças de estado no BD

As regras ativas podem monitorar mudanças de estados no BD sob dois diferentes níveis de granularidade:

- ▶ **nível de linha** – consideram mudanças que afetam linhas individuais dentro das tabelas;
- ▶ **nível de comando** – consideram como eventos as instruções de manipulação dos dados (independentemente de quantas linhas elas tenham afetado).

## BDs Ativos – Aplicações “internas” ao BD

Usa-se regras ativas para implementar funcionalidades clássicas do gerenciamento de banco de dados, como:

- ▶ manutenção de integridade
- ▶ manutenção de dados derivados
- ▶ gerenciamento de replicação controla de dados

Muitas vezes, essas regras são geradas de forma automática pelo SGBD e ficam escondidas dos usuários.

Exemplos de outras aplicações internas (funcionalidades estendidas):

- ▶ manutenção de versões
- ▶ administração de segurança
- ▶ *logging* (auditoria)

## BDs Ativos – Aplicações “externas” ao BD

São as **regras de negócio**, tais como:

- ▶ regras para a gestão de estoque de produtos
- ▶ regras para a aprovação de crédito para clientes
- ▶ regras para o cálculo das médias de alunos

## Triggers na linguagem SQL

O comando para a definição de *triggers* da SQL oferece diferentes opções para o projetista. As principais são:

1. A ação pode ser executada antes (**BEFORE**), depois (**AFTER**) ou no lugar (**INSTEAD OF**) do evento que disparou a regra
2. A ação pode referenciar tanto os **valores antigos** quanto os **novos valores** das tuplas que serão incluídas, removidas ou alteradas pelo evento que disparou a ação
3. Os eventos que podem ser monitorados são: **INSERT**, **UPDATE** e **DELETE**
4. Nos eventos de alteração monitorados, podemos especificar um **atributo particular** ou um **conjunto de atributos** para monitorar

## Triggers na linguagem SQL

O comando para a definição de *triggers* do SQL oferece diferentes opções para o projetista de banco de dados. As principais são:

5. Uma condição (opcional) pode ser especificada por meio da cláusula **WHEN**, e a ação só é executada se a regra é disparada e a condição é satisfeita quando o evento de disparo ocorre
6. O projetista tem a opção de especificar que a ação é executada:
  - ▶ **Uma vez para cada tupla** modificada (**FOR EACH ROW**) em um operação do BD, **ou**
  - ▶ **Uma só vez para todas as tuplas** que são modificadas em uma operação sobre o BD (**FOR EACH STATEMENT**)



# Triggers na linguagem SQL

Modelo relacional que será usado nos exemplos a seguir:

## FUNCIONARIO

Nome	<u>Cpf</u>	Salario	Dnr	Cpf_supervisor
------	------------	---------	-----	----------------

## DEPARTAMENTO

Dnome	<u>Dnr</u>	Sal_total	Cpf_gerente
-------	------------	-----------	-------------

- ▶ O atributo `Sal_total` é um atributo derivado: é definido em função do salário dos funcionários do departamento.
- ▶ **Objetivo:** criar um conjunto de *triggers* que mantenham automaticamente a consistência de `Sal_total`.

## Triggers na linguagem SQL

### Mantendo a consistência do atributo `Sal_total`

Eventos que podem causar a mudança de `Sal_total`:

- ▶ Alterar o salário de um ou mais funcionários existentes
- ▶ Mudar um ou mais funcionários de departamento
- ▶ Inserir uma ou mais tuplas de novos funcionários
- ▶ Excluir um ou mais funcionários

## Triggers na linguagem SQL

### Exemplo de *trigger* considerada para cada linha

```
CREATE TRIGGER Salario_total1
AFTER UPDATE OF Salario ON FUNCIONARIO
REFERENCING OLD ROW AS O, NEW ROW AS N
FOR EACH ROW
WHEN ( N.Dnr IS NOT NULL )
    UPDATE DEPARTAMENTO
    SET Sal_total = Sal_total + N.salario - O.salario
    WHERE Dnumero = N.Dnr;
```

- ▶ **OLD ROW** – tupla antiga (antes de sofrer a alteração)
- ▶ **NEW ROW** – tupla nova (depois da alteração)
- ▶ Uma operação de inserção só tem a tupla NEW; uma de remoção só tem a tupla OLD
- ▶ OLD e NEW ROW só existem em *triggers* do tipo FOR EACH ROW

## Triggers na linguagem SQL

Exemplo de *trigger* considerada para cada linha (completando o exemplo)

```
CREATE TRIGGER Salario_total2
AFTER UPDATE OF Dnr ON FUNCIONARIO
REFERENCING OLD ROW AS O, NEW ROW AS N
FOR EACH ROW
BEGIN
UPDATE DEPARTAMENTO
SET Sal_total = Sal_total + N.salario
WHERE Dnumero = N.Dnr;
UPDATE DEPARTAMENTO
SET Sal_total = Sal_total - O.salario
WHERE Dnumero = O.Dnr;
END;
```

## Triggers na linguagem SQL

Exemplo de *trigger* considerada para cada linha (completando o exemplo)

```
CREATE TRIGGER Salario_total3
AFTER INSERT ON FUNCIONARIO
REFERENCING NEW ROW AS N
FOR EACH ROW
WHEN ( N.Dnr IS NOT NULL )
    UPDATE DEPARTAMENTO
    SET Sal_total = Sal_total + N.salario
    WHERE Dnumero = N.Dnr;
```

## Triggers na linguagem SQL

Exemplo de *trigger* considerada para cada linha (completando o exemplo)

```
CREATE TRIGGER Salario_total4
AFTER DELETE ON FUNCIONARIO
REFERENCING OLD ROW AS O
FOR EACH ROW
WHEN ( O.Dnr IS NOT NULL )
    UPDATE DEPARTAMENTO
    SET Sal_total = Sal_total - O.salario
    WHERE Dnumero = O.Dnr;
```

## Triggers na linguagem SQL

### Exemplo de *trigger* considerada por comando

```
CREATE TRIGGER Salario_total
AFTER UPDATE OF Salario ON FUNCIONARIO
REFERENCING OLD TABLE AS O, NEW TABLE AS N
FOR EACH STATEMENT
WHEN EXISTS ( SELECT * FROM N WHERE N.Dnr IS NOT NULL ) OR
      EXISTS ( SELECT * FROM O WHERE O.Dnr IS NOT NULL )
UPDATE DEPARTAMENTO AS D
SET D.Sal_total = D.Sal_total
+ ( SELECT SUM(N.Salario) FROM N WHERE D.Dnumero = N.Dnr )
- ( SELECT SUM(O.Salario) FROM O WHERE D.Dnumero = O.Dnr )
WHERE Dnumero IN ( ( SELECT Dnr FROM N ) UNION
                  ( SELECT Dnr FROM O ) );
```

- ▶ **OLD TABLE** – tabela com as tuplas antigas (antes de sofrerem a alteração)
- ▶ **NEW TABLE** – tabela com as tuplas novas (depois da alteração)
- ▶ Uma operação de inserção só tem a tabela NEW; uma de remoção só tem a tabela OLD

# Um “parênteses”: declarando restrições em SQL

## Tipos de restrições existentes em SQL

- ▶ Restrições de chave – **PRIMARY KEY** e **UNIQUE**
- ▶ Restrições de integridade referencial – **REFERENCES** E **FOREIGN KEY**
- ▶ Restrição de valores nulos – **NOT NULL**
- ▶ Restrição sobre o valor de um atributo de uma relação – **CHECK**
- ▶ Restrição sobre os valores de uma tupla de uma relação – **CHECK**
- ▶ Asserção sobre uma ou mais relações do BD – **ASSERTION**
- ▶ Gatilhos, que permitem associar verificações à ocorrência de eventos no BD – **TRIGGER**



# Um “parênteses”: declarando restrições em SQL

## Exemplos de uso de algumas restrições

```
CREATE TABLE DEPARTAMENTO
( Dnome          VARCHAR(15) NOT NULL,
  Dnumero        INT NOT NULL
                  CHECK (Dnumero > 0 AND Dnumero < 21),
  Cpf_gerente    CHAR(11) NOT NULL
                  DEFAULT '888665555121',
  Dt_criacao     DATE,
  Dt_inicio_ger DATE,
  PRIMARY KEY (Dnumero),
  UNIQUE (Dnome),
  FOREIGN KEY (Cpf_gerente) REFERENCES FUNCIONARIO(Cpf),
  CHECK (Dt_criacao <= Dt_inicio_ger) );
```

## Um “parênteses”: declarando restrições em SQL

Uma asserção é uma expressão booleana em SQL que precisa ser verdadeira em qualquer estado válido do BD.

### Exemplo de uso de *Assertion*

```
CREATE ASSERTION RESTRICAO_SALARIO CHECK (  
    NOT EXISTS ( SELECT * FROM FUNCIONARIO F, FUNCIONARIO G,  
                 DEPARTAMENTO D  
                 WHERE F.Salario > G.Salario  
                       AND F.Dnr = D.Dnumero  
                       AND D.Cpf_gerente = G.Cpf ) );
```

- ▶ Especifica a restrição de que o salário de um funcionário não pode ser maior que o do gerente do seu departamento.
- ▶ O PostgreSQL não implementa ainda *Assertions*.

# Um “parênteses”: declarando restrições em SQL

## *Assertion* × *check*

```
CREATE TABLE FUNCIONARIO
( Nome          VARCHAR(30) NOT NULL,
  Cpf           VARCHAR(11) NOT NULL,
  Salario       NUMERIC NOT NULL,
  Dnr           INT,
  PRIMARY KEY (Cpf),
  FOREIGN KEY (Dnr) REFERENCES DEPARTAMENTO(Dnumero),
  CHECK (NOT EXISTS
        ( SELECT * FROM FUNCIONARIO F, FUNCIONARIO G, DEPARTAMENTO D
          WHERE F.Salario > G.Salario AND F.Dnr = D.Dnumero
              AND D.Cpf_gerente = G.Cpf ) );
```

- ▶ A restrição acima só será verificada quando acontecer uma mudança na relação FUNCIONARIO. Entretanto, uma mudança em DEPARTAMENTO também pode infringir a restrição. Portanto, o uso de *check* nesse caso não é correto.

# Um “parênteses”: declarando restrições em SQL

## Exemplo de uso de *Trigger* (como restrição)

```
CREATE TRIGGER VIOLACAO_SALARIO
BEFORE INSERT OR UPDATE OF Salario, Cpf_supervisor
  ON FUNCIONARIO
FOR EACH ROW
WHEN ( NEW.Salario > ( SELECT Salario FROM FUNCIONARIO
                        WHERE Cpf = NEW.Cpf_supervisor ) )
  INFORMA_SUPERVISOR(NEW.Cpf_supervisor, NEW.Cpf );
```

- ▶ Regra: sempre que for verificado que o salário de um funcionário é maior que o de seu supervisor, o supervisor deve ser informado.
- ▶ Vários eventos podem disparar essa regra: a inserção de um novo registro de funcionário, a mudança no salário de um funcionário, ou a mudança do supervisor de um funcionário

## Triggers no PostgreSQL – Limitações

- ▶ No PostgreSQL, não é possível renomear as tuplas **OLD** e **NEW** (ele não possui a cláusula **REFERENCING**)
- ▶ O PostgreSQL não permite que as tabelas **OLD TABLE** e **NEW TABLE** sejam usadas em *triggers* do tipo **FOR EACH STATEMENT** (isso ainda não foi implementado nesse SGBD)
- ▶ No PostgreSQL, a condição na cláusula **WHEN** não pode envolver subconsultas
  - ▶ Ela pode apenas fazer verificações sobre os valores das tuplas **OLD** e **NEW**
  - ▶ Não é útil para *triggers* com **FOR EACH STATEMENT**

## Triggers no PostgreSQL – Características

O *trigger* pode disparar:

- ▶ BEFORE – antes de tentar realizar a operação na linha (antes das restrições serem verificadas e o comando INSERT, UPDATE ou DELETE ser tentado), ou
- ▶ AFTER – após a operação estar completa (após as restrições serem verificadas e o INSERT, UPDATE ou DELETE ter completado)

A ação de um *trigger* no PostgreSQL é sempre a chamada de uma função (por meio da cláusula **EXECUTE PROCEDURE**)

## Triggers no PostgreSQL – Características

### Triggers INSTEAD OF e Triggers sobre Views

- ▶ *Triggers* que são especificados para executar INSTEAD OF do evento devem ser do tipo FOR EACH ROW, só podem ser definidos sobre visões e não podem “monitorar” colunas específicas da visão.
- ▶ *Triggers* do tipo BEFORE e AFTER em uma visão precisam ser declarados com FOR EACH STATEMENT

## Como criar uma função para triggers no PostgreSQL

```
CREATE OR REPLACE FUNCTION AtualizaDepartamento()  
    RETURNS TRIGGER AS $$  
BEGIN  
    UPDATE DEPARTAMENTO  
        SET Salario_total = Salario_total +  
            NEW.salario -  
            OLD.salario  
        WHERE Dnumero = NEW.Dnr  
    RETURN NEW;  
END;  
$$ LANGUAGE plpgsql;  
  
CREATE TRIGGER Salario_total1  
AFTER UPDATE OF Salario ON FUNCIONARIO  
FOR EACH ROW  
WHEN ( NEW.Dnr IS NOT NULL )  
    EXECUTE PROCEDURE AtualizaDepartamento();
```



## PostgreSQL: Funções chamadas em *Triggers*

- ▶ Uma função de *trigger* deve ser declarada como uma função que não recebe argumentos e que retorna o tipo TRIGGER
- ▶ Uma função de *trigger* pode ser usada por vários triggers
- ▶ As funções de *trigger* podem retornar NULL ou uma tupla com a mesma estrutura das tuplas da relação para a qual a *trigger* foi disparada

Veremos mais sobre a criação de funções nas próximas aulas!

## PostgreSQL: Funções chamadas em *Triggers*

Para *triggers* com BEFORE + FOR EACH ROW

- ▶ Se a função devolver **NULL**, então a operação subsequente será **cancelada** (ou seja, o comando INSERT, UPDATE e DELETE não será mais executado sobre a linha)

# PostgreSQL: Funções chamadas em *Triggers*

## Para *triggers* com BEFORE + FOR EACH ROW

- ▶ **Se a função devolver um valor diferente de NULL**, então a operação subsequente **prosseguirá** usando o valor de retorno como novo valor para a linha
  - ▶ Devolver um valor diferente do valor original de NEW modificará a linha que será inserida ou alterada
  - ▶ Para que ação do trigger prossiga normalmente, sem nenhuma alteração no valor da linha, então a função deve retornar NEW (sem modificá-lo)
  - ▶ Para modificar o valor da linha a ser armazenada, é possível alterar diretamente os valores dos atributos em NEW e então retornar o NEW modificado, ou então construir uma nova tupla para retornar

## PostgreSQL: Funções chamadas em *Triggers*

### Para *triggers* com INSTEAD OF

Triggers desse tipo só podem ser usadas em **visões** e sempre devem ser do tipo **FOR EACH ROW**

- ▶ Se a função não fizer nenhuma modificação no BD, então pode retornar NULL para sinalizar isso
- ▶ No caso contrário, a função deve retornar um valor diferente de NULL para indicar que o *trigger* executou a operação solicitada
  - ▶ Para operações INSERT ou UPDATE, o valor de retorno pode ser o NEW
  - ▶ Para operações DELETE, o valor de retorno pode ser OLD

## PostgreSQL: Funções chamadas em *Triggers*

### Para *triggers* com AFTER ou FOR EACH STATEMENT

- ▶ O valor de retorno da função é sempre ignorado
  - ▶ Ele pode, inclusive, ser NULL (ele não cancelará a operação que já foi realizada!)

Apesar disso, qualquer um desses tipos de *trigger* pode abortar a operação toda gerando uma exceção dentro da função (com o comando RAISE EXCEPTION 'mensagem de erro')

## Exemplo de *trigger* no PostgreSQL

- ▶ Impede que o salário de um funcionário seja diminuído.

```
CREATE OR REPLACE FUNCTION CorrigeSalario()  
  RETURNS TRIGGER AS $$  
BEGIN  
  NEW.salario = OLD.salario;  
  RETURN NEW;  
END;  
$$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER EvitaRebaixamento  
BEFORE UPDATE OF Salario ON FUNCIONARIO  
FOR EACH ROW  
WHEN (NEW.Salario IS NULL or NEW.Salario < OLD.Salario)  
EXECUTE PROCEDURE CorrigeSalario();
```

## Exemplo de *trigger* no PostgreSQL

- ▶ Impede que o salário de um funcionário seja diminuído.  
Nesta outra solução, o comando todo de UPDATE é **cancelado** quando o novo salário é menor que o anterior.

```
CREATE OR REPLACE FUNCTION CancelaAlteracao()  
  RETURNS TRIGGER AS $$  
BEGIN  
  RETURN NULL;  
END;  
$$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER EvitaRebaixamento  
BEFORE UPDATE OF Salario ON FUNCIONARIO  
FOR EACH ROW  
WHEN (NEW.Salario IS NULL or NEW.Salario < OLD.Salario)  
EXECUTE PROCEDURE CancelaAlteracao();
```

## Triggers no PostgreSQL

Exemplo de regra com subconsulta no WHEN (válida no SQL3, mas não no PostgreSQL)

```
CREATE TRIGGER EliminaExcessos
AFTER INSERT OR UPDATE OF Salario ON FUNCIONARIO
FOR EACH ROW
WHEN ( NEW.salario >
      ( SELECT G.salario from DEPARTAMENTO as D,
        FUNCIONARIO as G
        WHERE D.Dnr = New.Dnr AND
              D.Cpf_gerente = G.cpf) )
DELETE FROM FUNCIONARIO where Cpf = New.Cpf;
```

- ▶ Remove o funcionário com salário maior que o do gerente do seu departamento.



# Triggers no PostgreSQL

## Regra do slide anterior num formato válido no PostgreSQL

```
CREATE OR REPLACE FUNCTION RemoveFuncionario()
  RETURNS TRIGGER AS $$
BEGIN
  IF ( NEW.salario >
      ( SELECT G.salario from DEPARTAMENTO as D,
        FUNCIONARIO as G
        WHERE D.Dnr = NEW.Dnr AND
              D.Cpf_gerente = G.cpf ) ) THEN
    DELETE FROM FUNCIONARIO WHERE Cpf = NEW.Cpf;
  END IF;
  RETURN NULL;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER EliminaExcessos
AFTER INSERT OR UPDATE OF Salario ON FUNCIONARIO
FOR EACH ROW
EXECUTE PROCEDURE RemoveFuncionario();
```

## Referências Bibliográficas

- ▶ *Sistemas de Bancos de Dados* (6ª edição), Elmasri e Navathe. Pearson, 2010. – Capítulo 26
- ▶ *A First Course in Database Systems* (1ª edição), Ullman e Widom, 1997. – Capítulo 6