Banco de Dados Aplicado

Aula 04 – Triggers

Cristiano Santos
cristiano.santos@amf.edu.br

Bancos de Dados Ativos

- Muitos **SGBDs 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 = 0.10 * Salario
```

Regra: Quando a média dos salários dos funcionários ultrapassar 2000, 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 **uma expressão lógica** sobre o banco de dados **quanto uma *consulta***.
  - Uma expressão lógica deve devolver VERDADEIRO ou FALSO
  - No caso de uma condição expressa por meio de uma consulta, o resultado da 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
- 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 (AND), disjunção (OR), negação (NOT) e precedência

Momento de execução de uma regra ativa

Quando uma regra é disparada e sua condição é avaliada como verdadeira, a ação associada à regra pode ser executada:

- **ANTES** que o evento de disparo seja executado
  ou
- **DEPOIS** da execução do evento de disparo
  ou
- **NO LUGAR** do evento de disparo (nesse caso, o evento não é executado)

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 controlada 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
- 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

5. Uma condição (opcional) pode ser especificada por meio da cláusula **WHEN**. Nesse caso, a ação só será executada depois do disparo da regra se sua condição for satisfeita.

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	Cpf	Salario	Dnr	Cpf_supervisor

DEPARTAMENTO

Dnome	Dnr	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

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

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;
```

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;
```

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;
```

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 (Dumero > 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

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

A execução da ação do *trigger* pode ocorrer:

- 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**)

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();
```

Como criar uma função para triggers no PostgreSQL

Sintaxe:

```
CREATE  TRIGGER nome [BEFORE|AFTER|INSTEAD OF] Nome do evento
ON nome da tabela
[
 -- lógica aqui
];
```

Nome do evento pode ser: *INSERT, DELETE, UPDATE*

Como criar uma função para triggers no PostgreSQL

Exemplo:

Vamos considerar um caso em que queremos manter o teste de auditoria para cada registro inserido na tabela COMPANY, que criaremos da seguinte forma:

```
CREATE TABLE COMPANY(
   ID INT PRIMARY KEY NOT NULL,
   NAME TEXT NOT NULL,
   AGE INT NOT NULL,
   ADDRESS CHAR(50),
   SALARY REAL
);
```

Como criar uma função para triggers no PostgreSQL

Exemplo:

Para manter o teste de auditoria, criaremos uma nova tabela chamada AUDIT onde as mensagens de log serão inseridas sempre que houver uma entrada na tabela COMPANY para um novo registro

```
CREATE TABLE AUDIT(
   EMP_ID INT NOT NULL,
   ENTRY_DATE TEXT NOT NULL
);
```

Como criar uma função para triggers no PostgreSQL

Exemplo:

```
CREATE TABLE AUDIT(
   EMP_ID INT NOT NULL,
   ENTRY_DATE TEXT NOT NULL
);
```

ID é o ID do registro AUDIT, e EMP_ID é o ID, que virá da tabela COMPANY, e DATE manterá o timestamp quando o registro for criado na tabela COMPANY.

Como criar uma função para triggers no PostgreSQL

Exemplo:

Então, agora, vamos criar um gatilho na tabela COMPANY da seguinte forma:

```
CREATE TRIGGER example_trigger AFTER INSERT ON
COMPANY
FOR EACH ROW EXECUTE PROCEDURE auditlogfunc();
```

Como criar uma função para triggers no PostgreSQL

Exemplo:

Onde auditlogfunc() é um procedimento PostgreSQL e tem a seguinte definição:

```
CREATE OR REPLACE FUNCTION auditlogfunc() RETURNS TRIGGER
AS $example_table$
  BEGIN
    INSERT INTO AUDIT(EMP_ID, ENTRY_DATE) VALUES (new.ID,
current_timestamp);
    RETURN NEW;
  END;
$example_table$ LANGUAGE plpgsql;
```

Como criar uma função para triggers no PostgreSQL

Exemplo:

Agora, começaremos o trabalho real. Vamos começar a inserir o registro na tabela COMPANY, o que deve resultar na criação de um registro de log de auditoria na tabela AUDIT. Então, vamos criar um registro na tabela COMPANY da seguinte forma:

```
INSERT INTO COMPANY (ID, NAME, AGE, ADDRESS, SALARY)
VALUES (1, 'Paul', 32, 'California', 20000.00 );
```

Como criar uma função para triggers no PostgreSQL

Exemplo:

Agora, começaremos o trabalho real. Vamos começar a inserir o registro na tabela COMPANY, o que deve resultar na criação de um registro de log de auditoria na tabela AUDIT. Então, vamos criar um registro na tabela COMPANY da seguinte forma:

```
INSERT INTO COMPANY (ID, NAME, AGE, ADDRESS, SALARY)
VALUES (1, 'Paul', 32, 'California', 20000.00 );
```

Como criar uma função para triggers no PostgreSQL

Exemplo:

Ao mesmo tempo, um registro será criado na tabela AUDIT. Este registro é o resultado de um gatilho, que criamos na operação INSERT na tabela COMPANY. Da mesma forma, você pode criar seus gatilhos nas operações UPDATE e DELETE com base em seus requisitos.

Listando e Deletando Triggers

Exemplo:

Você pode **listar todos os gatilhos** no banco de dados atual da tabela pg_trigger da seguinte forma:

```
SELECT * FROM pg_trigger;
```

Se você quiser listar os gatilhos em uma tabela específica, use a cláusula AND com o nome da tabela da seguinte forma:

```
SELECT tgname FROM pg_trigger, pg_class WHERE tgrelid=pg_class.oid
    AND relname='company';
```

Listando e Deletando Triggers

Exemplo:

O seguinte é o comando DROP, que pode ser usado para descartar um gatilho existente:

```
DROP TRIGGER trigger_name;
```

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 devolver NULL ou uma tupla com a mesma estrutura das tuplas da relação para a qual a *trigger* foi disparada

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 devolver 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 devolver o NEW modificado, ou então construir uma nova tupla para devolver

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 devolver NULL para sinalizar isso
- No caso contrário, a função deve devolver 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

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

```
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
- https://www.tutorialspoint.com/postgresql/postgresql_triggers.htm