UNIVERSIDADE DO OESTE DE SANTA CATARINA – UNOESC
CAMPUS DE SÃO MIGUEL DO OESTE

APOSTILA DE PLPGSQL

Autoria: Prof. Roberson Junior Fernandes Alves
E-mail: robersonjfa@gmail.com ou roberson.alves@unoesc.edu.br

São Miguel do Oeste(SC), 2016.

1) INTRODUÇÃO

PL/pgSQL (Procedural Language / PostgreSQL) é uma linguagem que combina o poder expressivo da SQL com as características típicas de uma linguagem de programação, disponibilizando estruturas de controle tais como testes condicionais, loops e manipulação de exceções. Ao escrever uma função PL/pgSQL é possível incluir qualquer um dos comandos SQL, juntamente com os recursos procedurais.

Além disso, uma função escrita em PL/pgSQL pode ser executada por meio de um gatilho (trigger). Um gatilho é um procedimento que é acionado toda vez que um certo evento (inserção, deleção, alteração) ocorre em uma tabela. Por exemplo, pode-se desejar executar uma certa função toda vez que uma nova linha é adicionada a uma determinada tabela.

Algumas características do PL/pgSQL:

- Adiciona estrutura e controle ao SQL;
- É parecida com outras linguagens da família C;
- Roda dentro do servidor, evitando a viagem e conversões entre cliente e servidor;
- Um dos seus principais usos é em gatilhos;
- Acesso à todas as funcionalidades do SQL.

2) PROCEDIMENTOS ARMAZENADOS(STORED PROCEDURES)

Também conhecidos como Stored Procedures. São um poderoso recurso de banco de dados. Nada mais que programas desenvolvidos em determinada linguagem de script e armazenados no servidor, local onde serão processados. Eles também são conhecidos como funções, este o motivo pelo qual quando nos referenciamos a uma stored procedure no PostgreSQL devemos utilizar o nome de **Function**.

O PostgreSQL conta com três formas diferentes de criar funções:

- **Funções em Linguagem SQL**: São funções que utilizam a sintaxe SQL e se caracterizam por não terem estruturas de condição (if, else, case), estruturas de repetição (while, do while, for), não permitirem a criação de variáveis e utilizam sempre algum dos seguintes comandos SQL: SELECT, INSERT, DELETE ou UPDATE;
- **Funções de Linguagens Procedurais**: ao contrário das funções SQL, aqui é permitido o uso de estruturas de condição e repetição e o uso de variáveis. As funções em linguagens procedurais caracterizam-se também por não possuírem apenas uma possibilidade de linguagem, mas várias. Normalmente a mais utilizada é conhecida como PL/PgSQL, linguagem fortemente semelhante ao conhecido PL/SQL utilizado no Oracle. Mas existem outras como por exemplo o PL/Perl, PL/Python e PL/Tcl, que possuem sintaxe igual ou semelhante as linguagens que lhes deram origem. PL/Perl é reconhecido pelas possibilidades que possui na manipulação de caracteres, enquanto o PL/Python tem grande desempenho na utilização de agregados (aggregates) e gatilhos (triggers).
- **Funções em Linguagens Externas ou de Rotinas Complexas**: São funções normalmente escritas em C++ que trazem consigo a vantagem de utilizarem uma linguagem com diversos recursos, na qual pode-se implementar algoritmos com grande complexidade. Tais funções são empacotadas e registradas no SGBD para seu uso futuro. Existem ainda outras linguagens como PL/Ruby, PL/sh e PL/Java, PL/PHP no entanto, estas são definidas por projetos independentes.

Neste documento dar-se-á ênfase as funções utilizando SQL e PL/PGSQL.

Para criar uma função utilizando SQL no PostgreSQL utiliza-se o comando CREATE FUNCTION, da seguinte forma:

```
CREATE [ OR REPLACE ] FUNCTION nome ( [ tipo_do_parametro1 [, ...] ] )
RETURNS tipo_retornado AS '
      ! ! Implementação_da_função;
'
LANGUAGE 'sql';
```

Na qual CREATE FUNCTION é o comando que define a criação de uma função, [OR REPLACE] informa que se acaso existir uma função com este nome, a atual função deverá sobrescrever a antiga. RETURNS tipo_retornado informa o tipo de dado que será retornado ao término da função. Tais tipos de retornos são os convencionais como o INTEGER, FLOAT, VARCHAR, etc. As funções em SQL também permitem o retorno de múltiplos valores e para isso informa-se como retorno SETOF. Implementação_da_função, como o nome mesmo diz, traz as linhas de programação para a implementação da stored procedure. LANGUAGE está avisando qual linguagem em que está sendo implementada a função. Quando passamos parâmetros à função, não utilizamos nome nas variáveis que estão dentro dos parênteses da assinatura da função. Utilizamos apenas separados por vírgulas, o tipo da variável de parâmetro. Para acessarmos o valor de tais parâmetros, devemos utilizar o '$' mais o número da posição que ocupa nos parâmetros, seguindo a ordem da esquerda para a direita:

```
CREATE FUNCTION soma(INTEGER, INTEGER)
RETURNS INTEGER AS '
      SELECT $1 + $2;
'
LANGUAGE 'sql';
```

Outro detalhe importante é o fato de que as funções utilizando SQL sempre retornam valor, o que faz com que seja sempre necessário que a última linha de comando da função utilize o comando SELECT.

```
CREATE FUNCTION cubo(INTEGER)
RETURNS FLOAT AS '
      SELECT $1 ^ 3;
'
LANGUAGE 'sql';
```

Também é possível criar funções que tragam interação entre uma determinada consulta e parâmetros utilizados na função. Na função abaixo, obtém-se o total da soma dos itens de uma nota fiscal, passando como parâmetro o número da nota:

```
CREATE FUNCTION totalNota(INTEGER)
RETURNS INTEGER AS '
      SELECT SUM(preco) FROM item_Nota WHERE numNota = $1;
'
LANGUAGE 'sql';
```

Como já mencionado, é possível retornar várias linhas de uma consulta em uma função, para isso utilizamos o tipo de retorno SETOF. No próximo exemplo é criada uma função em que retorna-se todos os clientes que fazem aniversário no dia. Utilizando current_date obtém-se a data atual do sistema:

```
CREATE FUNCTION aniversariantes()
RETURNS SETOF clientes AS'
      SELECT nome FROM clientes WHERE aniversario = current_date;
'
LANGUAGE 'sql';
```

Funções escritas em PL/pgSQL já adicionam uma série de outros recursos como controles de fluxo e laços de repetição não suportados por funções SQL.

A sintaxe básica da declaração de uma função PL/pgSQL é:

```
CREATE [OR REPLACE] FUNCTION nome ([tipo_param [, tipo_param, …]])
```

```
RETURNS tipo_retorno
AS
' DECLARE
      variável tipo_variável;
      ...
      BEGIN
      instrução;
      …
      RETURN valor_retorno;
      END;
'
LANGUAGE 'plpgsql';
```

Exemplo 01:

```
CREATE OR REPLACE FUNCTION media (NUMERIC, NUMERIC, NUMERIC)
RETURNS NUMERIC
AS
' DECLARE result NUMERIC;
BEGIN
result := ($1 + $2 + $3) / 3;
RETURN result;
END;
'
LANGUAGE 'plpgsql';
```

Parâmetros:

O tipo de cada parâmetro é definido na lista de parâmetros na assinatura da função. São nomeados automaticamente, de acordo com a ordem na lista: **$1, $2,** etc. Obs: os parâmetros são **constantes**. Portanto, não podem receber valores no corpo da função.

Declaração de variáveis:

As variáveis são declaradas na seção DECLARE. O nome pode incluir letras, underscores e números (estes não no início). O nome é case **Insensitive**. Variáveis podem ser inicializadas na declaração:

```
DECLARE salario NUMERIC := 1000;
```

Obs.: variáveis não podem ser inicializadas na declaração usando-se parâmetros ou outras variáveis:

```
DECLARE
      salario NUMERIC := 1000;
      desconto NUMERIC := salario * 0.15; ERRO!
```

Obs.: CONSTANTES são declaradas assim:

```
DECLARE pi CONSTANT REAL := 3.141593;
```

Apelidos:

Normalmente, é bastante útil criar apelidos para os parâmetros, para que possam ser usados no corpo da função. Isto é feito através da cláusula ALIAS FOR:

```
DECLARE
      saldo ALIAS FOR $1;
      saque ALIAS FOR $2;
```

Tipos Especiais:

Há 3 tipos (pseudo-tipos) de dados que se adaptam aos tipos originais de um atributo ou de uma tupla de uma tabela, ou ainda ao conjunto de atributos de um result set.

O tipo **%TYPE** permite que se defina uma variável com o mesmo tipo de outra variável ou com o mesmo tipo de um atributo específico de uma tabela.

```
DECLARE endereço1 CHAR(25);
endereço2 endereço1%TYPE;
desconto Produto.preço%TYPE;
```

O tipo **%ROWTYPE** permite que se defina uma variável do tipo registro, possuindo os mesmos campos de uma determinada tabela.

```
DECLARE um_cliente Cliente%ROWTYPE;
```

O tipo **RECORD** permite que se defina uma variável do tipo registro, cuja estrutura será determinada em tempo de execução, adaptando-se aos dados que se deseja armazenar na mesma. Veremos exemplo na seção Estruturas de Repetição.

Atribuições:
Atribuições possibilitam que se estabeleça um novo valor para uma variável. O operador utilizado é **:=** .

saldo := saldo + saque;

Se a expressão à direita do operador não resultar um valor do mesmo tipo da variável, o seu tipo será convertido para o mesmo tipo da variável. Se isto não for possível, será gerado um erro.

Armazenando em variáveis o resultado de uma consulta:
Um outro tipo de atribuição possível é o armazenamento do resultado de uma consulta em uma variável. Utiliza-se o comando **SELECT INTO**.

```
DECLARE data Cliente.dta_nasc%TYPE;
um_cliente Cliente%ROWTYPE;
sal Cliente.salario%TYPE;
BEGIN
      SELECT INTO data dta_nasc FROM Cliente WHERE cod_cli = 5;
      SELECT INTO um_cliente * FROM Cliente WHERE cod_cli = 10;
      sal := um_cliente.salario;
```

Obs: o resultado da consulta deverá ser um único valor ou uma única tupla. Caso contrário, será gerado um erro.

Mensagens:

RAISE NOTICE gera mensagem na tela.

```
RAISE NOTICE ' 'estoque abaixo do limite…' ';
RAISE NOTICE ' 'valor da média = %' ' , result;
```

Mensagem com interrupção:
RAISE EXCEPTION gera mensagem na tela e interrompe a execução do respectivo bloco.

```
RAISE EXCEPTION ' 'valor inválido!' ';
```

Estruturas Condicionais:
Sintaxe:

```
IF (condição) THEN
      instruções;
      ...
ELSE
      instruções;
      ...
END IF;
```

Exemplo 2:

```
CREATE FUNCTION fatorial(INTEGER) RETURNS INTEGER AS
' DECLARE
      arg INTEGER;
BEGIN
      arg := $1;
      IF arg IS NULL OR arg < 0 THEN
            RAISE NOTICE ' 'Valor Inválido!' ';
            RETURN NULL;
      ELSE
            IF arg = 1 THEN
                  RETURN 1;
            ELSE
                  DECLARE next_value INTEGER;
            BEGIN
                  next_value := fatorial(arg - 1) * arg;
                  RETURN next_value;
            END;
```

```
            END IF;
      END IF;
END;
' LANGUAGE 'plpgsql';
```

Estruturas de Repetição:

```
LOOP
...
EXIT WHEN (condição); -- ou: IF (condição) THEN EXIT;
-- END IF;
...
END LOOP;

WHILE (condição) LOOP
…
END LOOP;

FOR v_controle IN valor_inicio .. valor_fim LOOP
…
END LOOP;

FOR v_controle IN SELECT… LOOP
…
END LOOP;
```

Obs: nesse último tipo de estrutura, o looping será executado uma vez para cada tupla retornada no result set da consulta.

PERFORM e EXECUTE:

Um comando SQL pode ser executado diretamente dentro do corpo de uma função. Se quisermos executar uma consulta ou uma função sem precisar guardar o seu resultado (o valor retornado, no caso da função), isto poderá ser feito por meio dos comandos PERFORM ou EXECUTE. A diferença entre eles é que com PERFORM o plano de execução da consulta é gerado e armazenado. Ou seja, PERFORM deve ser utilizado quando já se sabe a priori qual será a consulta, enquanto EXECUTE deve ser utilizado quando a consulta é montada em tempo de execução.

Exemplo 3:

```
CREATE OR REPLACE FUNCTION conta_tempo (VARCHAR)
RETURNS INTERVAL
AS
' DECLARE
      tempo_ini TIMESTAMP;
      tempo_fim TIMESTAMP;
BEGIN
      tempo_ini := timeofday();
      EXECUTE $1;
      tempo_fim := timeofday();
      RETURN (tempo_fim - tempo_ini);
END;
'
LANGUAGE 'plpgsql';
```

Quando desejar excluir uma função do sistema utilize o comando:

```
DROP FUNCTION nome_da_funcao();
```

3) GATILHOS OU DISPARADORES (TRIGGERS)

Um gatilho ou disparador (trigger) é um tipo especial de procedimento armazenado (stored procedure), que é executado sempre que há uma tentativa de modificar os dados de uma tabela que é protegida por ele.

Quando há uma tentativa de inserir, atualizar ou excluir os dados em uma tabela, e um TRIGGER tiver sido definido na tabela para essa ação específica, ele será executado automaticamente, não podendo nunca ser ignorado. O gatilho pode ser especificado para disparar antes de tentar realizar a operação na linha (antes das restrições serem

verificadas e o comando INSERT, UPDATE ou DELETE ser tentado), ou após a operação estar completa (após as restrições serem verificadas e o comando INSERT, UPDATE ou DELETE ter completado). Se o gatilho for disparado antes do evento, o gatilho pode fazer com que a operação não seja realizada para a linha corrente, ou pode modificar a linha sendo inserida (para as operações de INSERT e UPDATE somente). Se o gatilho for disparado após o evento, todas as mudanças, incluindo a última inserção, atualização ou exclusão, estarão "visíveis" para o gatilho.

Um gatilho que está marcado FOR EACH ROW é chamado uma vez para cada linha que a operação modifica. Por exemplo, um comando DELETE afetando 10 linhas faz com que todos os gatilhos ON DELETE da relação de destino sejam chamados 10 vezes, uma vez para cada linha excluída.

Diferentemente, um gatilho que está marcado FOR EACH STATEMENT somente executa uma vez para uma determinada operação, não importando quantas linhas sejam modificadas; em particular, uma operação que não modifica nenhuma linha ainda assim resulta na execução de todos os gatilhos FOR EACH STATEMENT aplicáveis. Se existirem vários gatilhos do mesmo tipo definidos para o mesmo evento, estes serão disparados na ordem alfabética de seus nomes. Ao contrário dos procedimentos armazenados do sistema, os disparadores não podem ser chamados diretamente e não passam nem aceitam parâmetros.

O SELECT não modifica nenhuma linha e, portanto, não é possível criar gatilhos para SELECT. Neste as views (visões) são mais apropriadas.

Usos e aplicabilidade dos TRIGGERS

- Impor uma integridade de dados mais complexa do que uma restrição CHECK;
- Definir mensagens de erro personalizadas;
- Manter dados desnormalizados;
- Comparar a consistência dos dados – posterior e anterior – de uma instrução UPDATE;

Os TRIGGERS são usados com enorme eficiência para impor e manter integridade referencial de baixo nível, e não para retornar resultados de consultas. A principal vantagem é que eles podem conter uma lógica de processamento complexa.

Você pode usar TRIGGERS para atualizações e exclusões em cascata através de tabelas relacionadas em um banco de dados, impor integridades mais complexas, definir mensagens de erro personalizadas, manter dados desnormalizados e fazer comparações dos momentos anteriores e posteriores a uma transação.

Quando queremos efetuar transações em cascata, por exemplo, um TRIGGER de exclusão na tabela ***PRODUTO*** de um banco de dados pode excluir os registros correspondentes em outras tabelas que possuem registros com os mesmos valores de ***PRODUCTID*** excluídos para que não haja quebra na integridade.

Conforme o ditado popular: “pai pode não ter filhos, mas filhos sem um pai não existe!”.

Você pode utilizar os TRIGGERS para impor integridade referencial da seguinte maneira:

- **Executando uma ação ou atualizações e exclusões em cascata:** A integridade referencial pode ser definida através do uso das restrições *FOREIGN KEY* e *REFERENCE*, com a instrução *CREATE TABLE*. Os TRIGGERS fazem bem o trabalho de checagem de violações e garantem que haja coerência de acordo com a sua regra de negócios. Se você exclui um cliente, de certo, você terá que excluir também todo o seu histórico de movimentações. Não seria boa coisa se somente uma parte desta transação acontecesse.
- **Criando disparadores de vários registros:** Quando mais de um registro é atualizado, inserido ou excluído, você deve implementar um TRIGGER para manipular vários registros.

CRIANDO TRIGGERS:

As TRIGGERS são criadas utilizando a instrução CREATE TRIGGER que especifica a tabela onde ela atuará, para que tipo de ação ele irá disparar suas ações seguido pela instrução de conferência para disparo da ação.

Nome:

```
CREATE TRIGGER -- cria um gatilho
```

Sinopse:

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

Descrição:

O comando CREATE TRIGGER cria um gatilho. O gatilho fica associado à tabela especificada e executa a função especificada nome_da_função quando ocorrem determinados eventos.

Parâmetros:

nome

O nome a ser dado ao novo gatilho. Deve ser distinto do nome de qualquer outro gatilho para a mesma tabela.

BEFORE

AFTER

Determina se a função será chamada antes ou depois do evento. Um entre INSERT, UPDATE ou DELETE; especifica o evento que dispara o gatilho. Podem ser especificados vários eventos utilizando OR.

Tabela

O nome (opcionalmente qualificado pelo esquema) da tabela que o gatilho se destina.

FOR EACH ROW

FOR EACH STATEMENT

Especifica se o procedimento do gatilho deve ser disparado uma vez para cada linha afetada pelo evento do gatilho, ou apenas uma vez por comando SQL. Se não for especificado nenhum dos dois, o padrão é FOR EACH STATEMENT.

nome_da_função

Uma função fornecida pelo usuário, declarada como não recebendo nenhum argumento e retornando o tipo trigger, que é executada quando o gatilho dispara.

argumentos

Uma lista opcional de argumentos, separados por vírgula, a serem fornecidos para a função quando o gatilho for executado. Os argumentos são constantes cadeia de caracteres literais. Também podem ser escritos nomes simples e constantes numéricas, mas serão todos convertidos em cadeias de caracteres. Por favor, verifique na descrição da linguagem de implementação da função de gatilho como os argumentos dos gatilhos são acessados dentro da função; pode ser diferente dos argumentos das funções normais.

Observação: Para poder criar um gatilho em uma tabela, o usuário deve possuir o privilégio TRIGGER na tabela.

PRATICANDO COM PL/PGSQL

1) Verificando se existe suporte a PL/pgSQL no banco de dados em uso.

   a) Via sql: SELECT true::BOOLEAN FROM pg_language WHERE lanname='plpgsql';

2) Instalando a PL/pgSQL no meu banco de dados(usuário precisa ser superuser).

   a) Por linha de comando: createlang plpgsql nome_banco_de_dados
   b) Via sql(conectado ao banco de dados): create language plpgsql;

3) Função que retorna o maior valor entre dois valores inteiros passados.

```
CREATE OR REPLACE FUNCTION min(integer, integer) RETURNS integer AS $body$
   SELECT CASE WHEN $1 < $2 THEN $1 ELSE $2 END
```

```
$body$ LANGUAGE SQL STRICT;
```

4) Retorna uma lista de valores utilizando RETURN NEXT.

```
CREATE OR REPLACE FUNCTION getAlunos() RETURNS SETOF aluno AS
$body$
DECLARE
   a aluno%rowtype;
BEGIN
   FOR a IN SELECT * FROM aluno
   WHERE codalu > 0
   LOOP
      -- can do some processing here
     IF (MOD(A.CODALU,2) = 0) THEN
         RETURN NEXT a; -- return current row of SELECT

      END IF;
   END LOOP;
   RETURN;
END
$body$
LANGUAGE plpgsql;
```

- USANDO A FUNÇÃO

```
SELECT * FROM getAlunos();
```

5) Um exemplo de uma função em PL/pgSQL para remover acentos.

```
--chamaremos a função de ft_racento(função remove acentos)
CREATE OR REPLACE FUNCTION ft_racento(termo VARCHAR) RETURNS VARCHAR AS
$body$
DECLARE
        com_acento VARCHAR(100):='ÁÀÂÃÄÉÈÊËÌÍÎÏÓÒÔÕÖÚÙÛÜÇáàâãäéèêëíìîïóòôõöúùûüç°ºª';
        sem_acento VARCHAR(100):='AAAAAEEEEIIIIOOOOOUUUUCaaaaaeeeeiiiioooooouuuucooa';
BEGIN
        return TRANSLATE(termo, com_acento, sem_acento);
END;
$body$
LANGUAGE plpgsql;

-- chamando a função para um exemplo
SELECT ft_racento('JOÃO');

clientDataSet1.commandText := 'SELECT * FROM pessoa
                                    WHERE ft_racento(nompes) ilike ft_racento(:termo);';
clientDataSet1.params[0].value := 'joão' ;
clientDataSet1.open;
```

6) Exemplo tratando exception no momento de uma inserção ou update.

```
CREATE TABLE db (a INT PRIMARY KEY, b TEXT);

CREATE FUNCTION merge_db(key INT, data TEXT) RETURNS VOID AS
$body$
BEGIN
   LOOP
      -- first try to update the key
      UPDATE db SET b = data WHERE a = key;
      IF found THEN
         RETURN;
      END IF;
      -- not there, so try to insert the key
      -- if someone else inserts the same key concurrently,
      -- we could get a unique-key failure
      BEGIN
         INSERT INTO db(a,b) VALUES (key, data);
```

```
            RETURN;
        EXCEPTION WHEN unique_violation THEN
            -- do nothing, and loop to try the UPDATE again
        END;
    END LOOP;
END;
$body$
LANGUAGE plpgsql;

-- USANDO A FUNÇÃO
SELECT merge_db(1, 'david');
SELECT merge_db(1, 'dennis');
```

7) Exemplo utilizando CURSOR(cursor that encapsulates the query, and then read the query result a few rows at a time).

```
CREATE OR REPLACE FUNCTION update_historico()
RETURNS integer AS
$body$
DECLARE
    hist    RECORD;
    counter int;
    curs1   refcursor;
BEGIN
    counter := 0;
    OPEN curs1 FOR
            SELECT * FROM historico WHERE MOD(codalu, 2) = 0;

    LOOP
            FETCH curs1 INTO hist;
            EXIT WHEN NOT FOUND;
            RAISE NOTICE 'ROW %  %.',counter, timeofday();
        UPDATE historico SET vlrnot = vlrnot * 1.25 WHERE codalu = hist.codalu and coddis =       hist.coddis;
            counter := counter + 1;
    END LOOP;
    RETURN counter;
END;
$body$ LANGUAGE plpgsql;
```

8) Mais um exemplo utilizando cursor.

```
CREATE OR REPLACE FUNCTION getAlunoNotas(refcursor)
RETURNS refcursor AS
$body$
BEGIN
  OPEN $1 FOR  SELECT A.CODALU, A.NOMALU, D.NOMDIS, H.VLRNOT
                    FROM ALUNO AS A
                    INNER JOIN HISTORICO AS H ON H.CODALU = A.CODALU
                    INNER JOIN DISCIPLINA AS D ON D.CODDIS = H.CODDIS
                    ORDER BY A.NOMALU, NOMDIS;
  RETURN $1;
END
$body$
LANGUAGE 'plpgsql';

-- fazendo a chamada
BEGIN;
            SELECT getAlunoNotas('notas');
            FETCH ALL IN notas;
END;
```

9) Um exemplo de auditoria utilizando gatilhos e funções.

```
-- criando a tabela onde serão armazenadas as informações auditadas
CREATE TABLE AUDITORIA  (
```

```
                ID SERIAL NOT NULL PRIMARY KEY,
                TABELA VARCHAR(50) NOT NULL,
                USUARIO VARCHAR(50) NOT NULL,
                DATA TIMESTAMP NOT NULL,
                OPERACAO VARCHAR(1) NOT NULL, -- I – INCLUSÃO, E – EXCLUSÃO, A - ALTERAÇÃO
                NEWREG TEXT,
                OLDREG TEXT
        );

        -- criando a função genérica de auditoria
        CREATE OR REPLACE FUNCTION ft_auditoria() RETURNS TRIGGER AS
        $body$
        BEGIN
                -- Cria uma linha na tabela AUDITORIA para refletir a operação
                -- realizada na tabela que invoca a trigger.        --
                IF (TG_OP = 'DELETE') THEN
                INSERT INTO auditoria(tabela, usuario, data, operacao,oldreg)                                   SELECT
TG_RELNAME, user, current_timestamp, 'E', OLD::text;
                RETURN OLD;
                ELSIF (TG_OP = 'UPDATE') THEN
                INSERT INTO auditoria(tabela, usuario, data, operacao,newreg,oldreg)
        SELECT TG_RELNAME, user, current_timestamp, 'A',NEW::text,OLD::text;
                RETURN NEW;
                ELSIF (TG_OP = 'INSERT') THEN
                INSERT INTO auditoria(tabela, usuario, data, operacao,newreg)
        SELECT TG_RELNAME, user, current_timestamp, 'I',NEW::text;
                RETURN NEW;
                END IF;
                RETURN NULL; -- o resultado é ignorado uma vez que este é um gatilho AFTER          END;
        $body$
        LANGUAGE plpgsql;

        -- auditando a tabela academico e a tabela uf

        CREATE TRIGGER naturalidade_audit AFTER INSERT OR UPDATE OR DELETE ON naturalidade FOR EACH
ROW EXECUTE PROCEDURE ft_auditoria();

        CREATE TRIGGER aluno_audit AFTER INSERT OR UPDATE OR DELETE ON aluno FOR EACH ROW
EXECUTE PROCEDURE ft_auditoria();

        -- agora basta executar os comandos insert, update e delete sobre estas tabelas, e as mesmas serão auditadas.
```