UMC – Universidade de Mogi das Cruzes

Notas de Aula

Profa. MSc. Viviane Guimarães Ribeiro

Prof. ª Erika Estevam Freire Miranda

2016

Índice

Apresentação da Disciplina

Nome da Disciplina: Programação de Banco de Dados
Professora Erika Estevam Freire Miranda
E-mail:erika.umc@gmail.com

Plano de Ensino

Ementa

Sistemas de banco de dados. Arquitetura de um SGBD – Sistema de Gerenciamento de Banco de Dados. Visão de banco de dados. Métodos de acesso. Controle de Concorrência. Integridade dos dados. Transações. Plataformas de banco de dados.

Objetivo da Disciplina

Conhecer as potencialidades dos gerenciadores de banco de dados e saber aplicar esses recursos na implementação de sistemas aplicativos de processamento de dados.

Tópicos do Semestre

1. Integridade dos dados
2. Sequence
3. Insert com Query
4. Indices
5. Visões
6. Transações
7. Procedures
    a. Procedures em SQL
    b. Procedures em PLPGSQL
8. Triggers

Metodologia

Aulas expositivas e práticas em laboratório.

Avaliação

M1 = P1 (70%) + AP (20%) + Listas (10%)
ND = P2 (70%) + AP (20%) + Listas(10%)
M2 = ND (70%) + PI (30%)
Média Semestral = (M1 + M2 * 2) / 3

Onde: **P1** e **P2** são provas individuais, escritas e sem consulta, previamente agendadas.
AP são as atividades práticas realizadas em sala (sem a necessidade de agendamento prévio) ou em casa.
PI prova interdisciplinar agendada pela gestão do curso.
Listas exercícios individuais realizados em casa entregues em datas pré-estabelecidas.
Projeto Interdisciplinar refere-se a um projeto desenvolvido na disciplina de Projeto Interdisciplinar III, envolvendo o conteúdo de Banco de Dados e Ambientes Visuais.

Datas Importantes

P1
21/09/2016

P2 –
30/11/2016
Recuperação – 14/12/2016

OBS1: Não será aplicada prova substitutiva.
OBS2: A disciplina tem caráter acumulativo, portanto em todas as provas será cobrado todo o conteúdo estudado até o momento no semestre.

Bibliografia Básica

- DATE, C. J. Introdução a sistemas de bancos de dados. 7ª edição. Rio de Janeiro: Campus, 2000 803 p. ISBN 85-352-0560-8. Número de Chamada: 005.74 D232i 7. ed.
- SILBERSCHATZ, Abraham; KORTH, Henry F; SUDARSHAN, S. Sistema de banco de dados. 3ª edição. São Paulo: Makron Books, 1999-2005 778 p ISBN 85-346-1073-8 Número de Chamada: 005.74 S582s 3. ed.
- HEUSER, Carlos Alberto. Projeto de Banco de Dados, 6ª edição, Bookman, 2011. <http://online.minhabiblioteca.com.br/books/9788577804528>.
- RAMAKRISHNAN, Raghu; GEHRKE, Johannes. Sistemas de gerenciamento de banco de dados. MCGraw-Hill 2008. <http://online.minhabiblioteca.com.br/books/9788563308771>.

Bibliografia Complementar

- GILLENSON, Mark L. Fundamentos de sistemas de gerência de banco de dados. Rio de Janeiro: LTC, 2006 304 p. ISBN 8521614977 Número de Chamada: 005.74 G476f
- SETZER, Valdemar W. Bancos de dados: conceitos, modelos, gerenciadores, projeto lógico, projeto físico. 3ª edição rev. São Paulo: E. Blücher, 1999 289 p. (Computação) ISBN 85-212-0123-0 Número de Chamada: 005.74 S495b 3. ed.
- HAY, David C. Princípios de modelagem de dados. São Paulo: Makron Books, 1999. 271 p. ISBN 85-346-0870-9 Número de Chamada: 005.74 H412p
- DAMAS, Luís. SQL: structured query language. 6ª edição atual. e ampl. Rio de Janeiro: LTC, 384 p.1 CD-ROM ISBN 9788521615583 Número de Chamada: 005.7565 D155s 6. ed.
- MACHADO, Felipe Nery Rodrigues; ABREU, Mauricio Pereira de. Projeto de banco de dados: uma visão prática. 11ª edição. São Paulo: Érica, 2004 300 p. ISBN 8571943125. Número de Chamada: 005.74 M149p 11. ed

1. Revisão de Conceitos

No semestre passado vimos alguns comandos importantes do SQL para manipular e definir a estrutura de nosso Banco de Dados.

Nesta primeira aula iremos fazer uma pequena revisão sobre os conceitos de maior importância e realizar uma atividade prática de revisão.

1. Operações para definição e manutenção da estrutura das tabelas.

Criar Tabela

```
Create table Disciplina(
        cod             integer,
        nome            varchar(100) not null,
        Constraint pk_disciplina Primary Key (cod)
);

Create table Aluno(
        rgm             integer,
        nome            varchar(100) not null,
        endereco        varchar(100) not null,
        Constraint pk_aluno Primary Key (rgm)
);

Create table Cursa(
        aluno           integer,
        disciplina      integer,
        ano             numeric(4) not null,
        Constraint pk_cursa Primary Key (aluno,disciplina),
        Constraint fk_cursa_aluno Foreign Key (aluno)
                References Aluno(rgm),
        Constraint fk_cursa_disciplina Foreign Key (disciplina)
                References Disciplina(cod)
);
```

Atualizar Tabela

```
Alter table Aluno add column telefone varchar(10);
```

Apagar Tabela

```
Drop table Aluno cascade;
```

2. Operações para manipulação dos dados das tabelas.

Inserir Dados

```
Insert into Aluno (rgm, nome, endereco)
        values (123, „Fulano", „Rua das Flores, 12");
```

Atualizar Dados

```
Update Aluno set telefone=„(11)5747-8732" where rgm=123;
```

Apagar Dados

```
Delete Aluno where nome ilike „Ana%";
```

3. Operações para manipulação dos dados das tabelas (consultas).

Selecionar Dados

```
Select rgm, nome
from Aluno
where disc = (Select cod from Disciplina where nome="Matematica");

Select a.nome,d.nome
from Aluno as a, Disciplina as d
where a.disc    =  d.cod and
        d.nome =„Matematica"
order by a.nome;

Select cod as código_disciplina
from Disciplina;

Select *
from Aluno
where telefone is null;

Select *
from Aluno
where telefone is not null;
```

4. Funções e cláusulas especiais.

```
Select COUNT(codigo) from cliente;

Select SUM(salario) from cliente;

Select MAX(salario) from cliente;

Select MIN(salario) from cliente;

Select AVG(salario) from cliente;

Select SUM(salario) from clientes group by salario;

Select SUM(salario) from clientes GROUP BY salário HAVING SUM(salario) < 2000;

Select DISTINCT nome from clientes;

Select LOWER(nome) from clientes;

Select UPPER(nome) from clientes;

Select DISTINCT nome from clientes where date_part ('year' , dt_compra ) = 2013;
```

2. Integridade de Dados

As regras de integridade fornecem a garantia de que mudanças feitas no banco de dados por usuários autorizados não resultem em perda da consistência de dados. Assim, as regras de integridade protegem o banco de dados de danos acidentais.

2.1. Restrições de Domínio

Restrições de domínio são a forma mais elementar de restrições de integridade.
Estas testam valores inseridos no Banco de Dados, e testam (efetuam) consultas para assegurar que as comparações façam sentido.
Considere os seguintes atributos:

- Nome_cliente
- Nome_empregado
- Saldo
- Nome_agência

Caso 1 - É razoável imaginarmos que Nome_cliente e Nome_empregado estejam em um mesmo domínio.
Caso 2 - É razoável imaginarmos que Saldo e Nome_agência estejam em domínios distintos.

A cláusula check da SQL-92 permite ao projetista do esquema determinar um predicado que deva ser satisfeito por qualquer valor designado a uma variável cuja tipo seja o domínio.

Exemplo1:

```
CREATE TABLE passagem(
  num            INTEGER,
  valor          NUMERIC(5,2)   NOT NULL CHECK (valor >100.00 AND valor < 200.00),
  poltrona       INTEGER        NOT NULL,
  data_compra    TIMESTAMP      DEFAULT CURRENT_TIMESTAMP,
  tipo_cartao    INTEGER,
  num_cartao     VARCHAR(30)    NOT  NULL,
  cod_cli        INTEGER        NOT NULL,
  num_voo        INTEGER        NOT NULL,
  CONSTRAINT PK_PASSAGEM PRIMARY KEY (NUM)
);
```

Exemplo2:

```
CREATE TABLE piloto(
  num_inscricao INTEGER,
  nome          VARCHAR(40)    NOT NULL UNIQUE,
  data_adm      DATE           NOT NULL,
  CONSTRAINT PL_PILOTO PRIMARY KEY (NUM_INSCRICAO)
);
```

Exemplo3:

```
CREATE TABLE piloto(
  num_inscricao INTEGER,
  nome          VARCHAR(40)    NOT NULL UNIQUE,
  data_adm      DATE           NOT NULL,
  CONSTRAINT PL_PILOTO PRIMARY KEY (NUM_INSCRICAO),
  UNIQUE (NOME,DATA_ADM)
);
```

Exemplo4:

```
CREATE TABLE passagem(
  num            INTEGER,
  valor          NUMERIC(5,2)   NOT NULL CHECK (valor>100.00 AND valor<200.00),
  poltrona       INTEGER        NOT NULL,
  data_compra    TIMESTAMP      DEFAULT CURRENT_TIMESTAMP,
  tipo_cartao    INTEGER,
  num_cartao     VARCHAR(30)    NOT NULL,
  cod_cli        INTEGER        NOT NULL,
  num_voo        INTEGER        NOT NULL,
  CONSTRAINT PK_PASSAGEM PRIMARY KEY (NUM),
  CONSTRAINT COMPRA CHECK(NOT(TIPO_CARTAO = 1 AND VALOR > '150.00'))
);
```

2.2. Restrições de Integridade Referencial

Se uma relação R2 inclui uma chave externa (estrangeira) FK equivalendo à chave primária PK de uma relação R1 então todo valor de FK em R2 deve:

- Ou ser igual ao valor de PK de alguma tupla de R1;
- Ou ser totalmente nulo.

Exemplo1: Criação de chave estrangeira na tabela passagem.

```
CREATE TABLE passagem(
  num            INTEGER,
  valor          NUMERIC(5,2)   NOT NULL CHECK (valor>100.00 AND valor<200.00),
  poltrona       INTEGER        NOT NULL,
  data_compra    TIMESTAMP      DEFAULT CURRENT_TIMESTAMP,
  tipo_cartao    INTEGER,
  num_cartao     VARCHAR(30)    NOT NULL,
  cod_cli        INTEGER        NOT NULL,
  num_voo        INTEGER        NOT NULL,
  CONSTRAINT PK_PASSAGEM PRIMARY KEY (NUM),
  CONSTRAINT COMPRA CHECK(NOT(TIPO_CARTAO = 1 AND VALOR > „150.00")),
  CONSTRAINT cli_pass FOREIGN KEY (cod_cli)
        REFERENCES cliente(cod_cli) ON DELETE CASCADE ON UPDATE CASCADE,
  CONSTRAINT voo_pass FOREIGN KEY (num_voo)
        REFERENCES voo(num_voo) ON DELETE NO ACTION ON UPDATE SET NULL
);
```

Devido às cláusulas on delete cascade, se a remoção de uma tupla (linha) em passagem (na tabela) resultar em violação da restrição de integridade referencial, a remoção é feita em "cascata" na relação (tabela) cliente, removendo as tuplas (linhas) que se referem ao cliente que foi removido.

Atualizações em "cascata" são semelhantes.

3. Sequence

Sequence nada mais é do que um contador criado pelo usuário que pode ser associado ao valor default de uma coluna.

Uma Sequence possui três funções de acesso:

- Nextval(„nome_sequence");
- Currval(„nome_sequence");
- Setval(„nome_sequence", novo_valor);

Exemplo 1: Criação de sequences:

```
CREATE SEQUENCE contador_cli;

SELECT nextval('contador_cli');
SELECT currval('contador_cli');
SELECT setval('contador_cli', 100);
```

Exemplo 2: Utilizando uma sequence em um campo:

```
CREATE TABLE cliente(
  cod_cli    INTEGER  DEFAULT  nextval(„contador_cli"),
  ...
);
```

Exemplo 3: Utilizando o comando insert:

```
CREATE TABLE Produtos(
        codigo          numeric(4) not null,
        nome            varchar(100) not null
);

CREATE SEQUENCE minha_sequence;

INSERT INTO Produtos (codigo, nome)
        VALUES  (nextval('minha_sequence'),  'caneta');
```

Partes de uma Sequence:

INCREMENT BY → Podemos definir o incremento da nossa sequence.

```
-- incrementa de 2 em 2
create sequence minha_sequence Increment By 2;
```

MINVALUE → Podemos definir o valor mínimo que queremos na sequence.

```
-- menor valor será 1
create sequence minha_sequence minvalue 1;
```

MAXVALUE → Podemos definir o valor máximo que queremos na sequence.

```
-- maior valor 999
create sequence minha_sequence maxvalue 999;
```

START → Podemos definir o valor no qual a sequence irá iniciar.

```
-- inicia com 10
create sequence minha_sequence start with 10;
```

Exemplo 4: Apagando uma sequence:

```
DROP SEQUENCE contador_cli;
```

Exemplo 5: Alterando uma sequence:

```
ALTER SEQUENCE contador_cli INCREMENT BY INCREMENTO;
ALTER SEQUENCE contador_cli MINVALUE VALOR;
ALTER SEQUENCE contador_cli MAXVALUE VALOR;
ALTER SEQUENCE contador_cli RESTART WITH INICIO;
```

Exemplo 6: Utilizando um campo serial:

```
CREATE TABLE passagem(
  num    SERIAL,
  ...
);
```

Deste modo, o PostgreSQL criará uma sequence que terá como nome nomeTabela_nomeCampo_seq, iniciando de 1 e com incremento de 1.

4. Insert com Query

Para inserir dados em uma relação podemos especificar uma tupla a ser inserida ou escrevermos uma consulta cujo resultado é um conjunto de tuplas a inserir.

Exemplo 1: Cópia de dados:

```
CREATE TABLE cliente(
  cpf           numeric(11,0),
  nome          character varying(100) NOT NULL,
  rua           character varying(100),
  numero        character varying(10),
  cidade        character varying(50),
  tel           character varying(15),
  email         character varying(20),
  CONSTRAINT pk_cliente PRIMARY KEY (cpf),
  CONSTRAINT uk_email UNIQUE (email)
);
```

```
INSERT INTO cliente VALUES (12345678910,"Maria","Rua A","10","Mogi","4747-8765",„maria@mail");
INSERT INTO cliente VALUES (98745678910,"José","Rua B","11","Suzano","8765-8765","jose@mail");
INSERT INTO cliente VALUES (12312378910,"Clara","Rua C","12","Guararema","3102-8765",„clara@mail");
INSERT INTO cliente VALUES (83645678910,"Ricardo","Rua D","13","Mogi","4747-6298",„ricardo@mail");
INSERT INTO cliente VALUES (12345639710,"Lucia","Rua E","14","Mogi","4747-1739",„lucia@mail");
```

```
CREATE TABLE cliente2(
  cpf           numeric(11,0),
  nome          character varying(100) NOT NULL,
  tel           character varying(15),
  email         character varying(20),
  CONSTRAINT pk_cliente2 PRIMARY KEY (cpf),
  CONSTRAINT uk_email2 UNIQUE (email)
);

SELECT * FROM cliente;
SELECT * FROM cliente2;

INSERT INTO cliente2 (SELECT cpf,nome,tel,email FROM cliente);

SELECT * FROM cliente2;
```

Exemplo 2: Valores de chave estrangeira:

```
CREATE SEQUENCE seq_depto;
CREATE SEQUENCE seq_func;

CREATE TABLE depto(
  codigo        integer,
  nome          varchar(100) NOT NULL,
  CONSTRAINT pk_depto PRIMARY KEY (codigo)
);
```

```
CREATE TABLE func(
  codigo        integer,
  nome          varchar(100) NOT NULL,
  depto         integer,
  CONSTRAINT pk_func PRIMARY KEY (codigo),
  CONSTRAINT fk_func_depto FOREIGN KEY (depto) REFERENCES depto(codigo)
);

INSERT INTO depto VALUES (nextval('seq_depto'),'Compras');
INSERT INTO depto VALUES (nextval('seq_depto'),'RH');
INSERT INTO depto VALUES (nextval('seq_depto'),'CPD');
INSERT INTO depto VALUES (nextval('seq_depto'),'Jurídico');

SELECT * FROM depto;

INSERT INTO func VALUES (nextval('seq_func'),'Maria',(SELECT codigo FROM depto
WHERE nome = 'Compras'));
INSERT INTO func VALUES (nextval('seq_func'),'João',(SELECT codigo FROM depto
WHERE nome = 'Jurídico'));
INSERT INTO func VALUES (nextval('seq_func'),'José',(SELECT codigo FROM depto
WHERE nome = 'RH'));
INSERT INTO func VALUES (nextval('seq_func'),'Claudia',(SELECT codigo FROM
depto WHERE nome = 'CPD'));
INSERT INTO func VALUES (nextval('seq_func'),'Luana',(SELECT codigo FROM depto
WHERE nome = 'CPD'));
INSERT INTO func VALUES (nextval('seq_func'),'Valéria',(SELECT codigo FROM
depto WHERE nome = 'Compras'));

SELECT * FROM func;
```

Nos exemplos apresentados o comando select é realizado primeiro, resultando em um conjunto de tuplas que é então inserido na relação cliente.

5. Índices

Índices são estruturas que podem melhorar a performance do banco de dados. Tais estruturas são utilizadas em consultas que envolvem critérios.

Quando se cria um índice, este índice está relacionado a uma determinada coluna, portanto, este índice não pode auxiliar no acesso de informações em outras colunas porque os índices são classificados de acordo com a coluna correspondente. Eis o motivo para qual deve-se saber muito bem a conceituação do assunto para criar índices que tragam realmente melhora de desempenho. É claro que você pode criar vários índices dentro de uma mesma tabela, mas um índice que seja raramente usado é um desperdício de espaço em disc. Isso sem falar de trabalho adicional para o sistema de banco de dados como um todo, pois para cada atualização em um registro será necessário também atualizar todos os índices da tabela.

O PostgreSQL implementa quatro tipos de índices: B-Tree, R-Tree, GiST e Hash.

B-Tree → São árvores de pesquisa balanceadas desenvolvidas para trabalharem em discos magnéticos ou qualquer outro dispositivo de armazenamento de acesso direto em memória secundária. O índice B-Tree é uma implementação das árvores B de alta concorrência propostas por Lehman e Yao.

R-Tree → Também conhecidas com árvore R, utiliza o algoritmo de partição quadrático de Guttman, sendo utilizada para indexar estrutura de dados multidimensionais, cuja implantação está limitada a dados com até 8Kbytes, sendo bastante limitada para dados geográficos reais. Utilizada normalmente com dados do tipo box, circle, point e outros.

Hash → Valor de indentificação produzido através da execução de uma operação matemática, denominada função hash, em um item de dado. O valor identifica de forma exclusiva o item de dado, mas exige um espaço de armazenamento bem menor. Por isso, o computador pode localizar mais rapidamente os valores de hashing do que os itens de dado, que são mais extensos. Uma tabela de hashing associa cada valor a um item de dado exclusvo.

GiST → Generalized Index Search Trees (Árvore de Procura de Índice Generalizado).

Por padrão quando criamos um índice e não especificamos qual o tipo que queremos usar, o PostgreSQL utiliza o B-Tree, ou seja, nas situações mais comuns. Mas podem ocorrer vezes em que o PostgreSQL escolha outro tipo, para isso, ele analisa o caso e leva em consideração se a coluna indexada está envolvida numa comparação envolvendo determinados operadores.

Sintaxe:

```
CREATE INDEX <nome do índice>
        ON <nome da tabela> (nome do campo)
        USING <tipo desejado> (<nome da coluna>);
```

Exemplo 1: Criando um índice sobre o campo nome da tabela cliente:

```
CREATE INDEX cliente_nome_idx ON cliente (nome);
```

Exemplo 2: Criando um índice sobre o campo cidade da tabela cliente:

```
CREATE UNIQUE INDEX cliente_cidade_idx ON cliente (cidade);
```

OBS: A cláusula opcional UNIQUE deve ser utilizada quando houver a necessidade de definir que o campo utilizado para a indexação não deve conter valores repetidos. B-Tree é o único tipo de índice que aceita restrição de unicidade.

Exemplo 3: Criando um índice funcional sobre o campo valor da tabela cliente:

```
CREATE INDEX nome_lower_idx ON cliente (lower(nome));
CREATE INDEX nome_upper_idx ON cliente (upper(nome));
```

Exemplo 4: Destruindo um índice:

```
DROP INDEX cliente_nome_idx;
DROP INDEX nome_lower_idx;
```

6. Visões - View

Uma view é uma apresentação dos dados de uma ou mais tabelas. Também são chamadas de pseudo-tabelas ou consulta armazenada. Através de uma view é possível criar tabelas virtuais a partir de uma ou mais tabelas reais. As tabelas virtuais se parecem com as reais, mas não são as tabelas reais em si, mas apenas uma composição em forma de consulta predefinida a partir de uma tabela real. As tabelas reais possuem os dados cadastrados e por esta razão ocupam espaço em disco, já as virtuais possuem apenas as referências de acesso à consulta das tabelas reais, e por assim dizer, não ocupam espaço em disco. Sendo assim, todas as operações realizadas nas visões afetam as tabelas reais.

As views agilizam as operações de consulta, uma vez que concentram em cada tabela virtual os campos que realmente interessam.

Utilizar uma view é útil quando há a necessidade de fazer determinadas consultas com freqüência, ou seja, é uma forma eficiente de deixar as consultas que serão usadas como relatórios. Além disso, as views são empregadas para restringir o acesso às colunas da tabela.

Vantagens de uso:

- Visões fornecem um nível de segurança adicional para a tabela, restringindo o acesso a um conjunto de colunas pré-determinadas;
- Esconder a complexidade dos dados (união de várias tabelas);
- Simplificar comandos do usuário (união de várias tabelas);
- Apresentar os dados com uma perspectiva diferente da tabela base (renomeando colunas);
- Armazenas consultas complexas.

Exemplo1: View sobre a junção cliente e passagem

```
CREATE VIEW cliente_passagem AS
        SELECT nome, passagem.num, poltrona
            FROM cliente, passagem
            WHERE cliente.cod_cli = passagem.cod_cli
            ORDER BY nome, num;

SELECT * FROM cliente_passagem;
```

Exemplo2: Destruindo uma view

```
DROP VIEW cliente_passagem;
```

7. Transações

O termo transação refere-se a uma coleção de operações que formam uma única unidade de trabalho lógica. Por exemplo, a transferência de dinheiro de uma conta para outra é uma transação consistindo de duas atualizações, uma para cada conta.

Uma transação é uma unidade de execução do programa que acessa e possivelmente atualiza vários itens de dados. Para garantir a integridade dos dados, é necessário que o SGBD mantenha as seguintes propriedades das transações: atomicidade, consistência, isolamento e durabilidade.

• Atomicidade: uma transação é uma unidade atômica de processamento; ou ela será executada em sua totalidade ou não será de modo nenhum.

• Consistência: uma transação deve ser preservadora de consistência se sua execução completa fizer o banco de dados passar de um estado consistente para outro também consistente.

• Isolamento: uma transação deve ser executada como se estivesse isolada das demais. Isto é, a execução de uma transação não deve sofrer interferência de quaisquer outras transações concorrentes.

• Durabilidade: as mudanças aplicadas ao banco de dados por uma transação efetivada devem persistir no banco de dados. Essas mudanças não devem ser perdidas em razão de uma falha.

Essas propriedades normalmente são conhecidas como propriedades ACID. Esse acrônimo é derivado da primeira letra de cada uma das quatro propriedades.

Quando se trabalham com transações, é necessário que se faça pelo menos duas ressalvas. A primeira é que em certas situações é interessante se agregar vários comandos como sendo integrantes de uma mesma transação, como, por exemplo, em uma transferência bancária que envolve a retirada de dinheiro de uma conta e o acréscimo em outra como se fosse apenas uma única operação lógica. A segunda ressalva é que em outras situações se faz necessário sacrificar ou flexibilizar as características ACID em virtude da necessidade de maior desempenho.

7.1. Estados de uma Transação

Na ausência de falhas, todas as transações são completadas com sucesso. Porém, uma transação nem sempre pode completar sua execução com sucesso. Caso isso ocorra, essa transação é considerada abortada.

Se tivermos que garantir a propriedade de atomicidade, uma transação abortada não pode ter efeito sobre o estado do banco de dados. Assim, qualquer mudança que a transação abortada tenha feito no banco de dados deve ser desfeita. Quando as mudanças causadas por uma transação abortada tiverem sido desfeitas, dizemos que a transação foi revertida (rolled back).

Uma transação que completa sua execução com sucesso é considerada confirmada (committed). Uma transação confirmada que realizou atualizações transforma o banco de dados em um novo estado consistente, que precisa persistir mesmo que haja uma falha no sistema. Quando uma transação tiver sido confirmada, não podemos desfazer seus efeitos abortando-a. A única forma de desfazer os efeitos de uma transação confirmada é executar uma transação de compensação.

Em resumo, uma transação precisa estar em um dos seguintes estados:

- Ativa: é o seu estado inicial. A transação permanece nesse estado enquanto está sendo executada.
- Parcialmente confirmada: é o estado depois que a instrução final foi executada.
- Falha: é o estado depois da descoberta de que a execução normal não pode mais prosseguir.
- Abortada: estado depois que a transação foi revertida e o banco de dados foi restaurado ao seu estado anterior ao início da transação.
- Confirmada: estado após o término bem-sucedido.

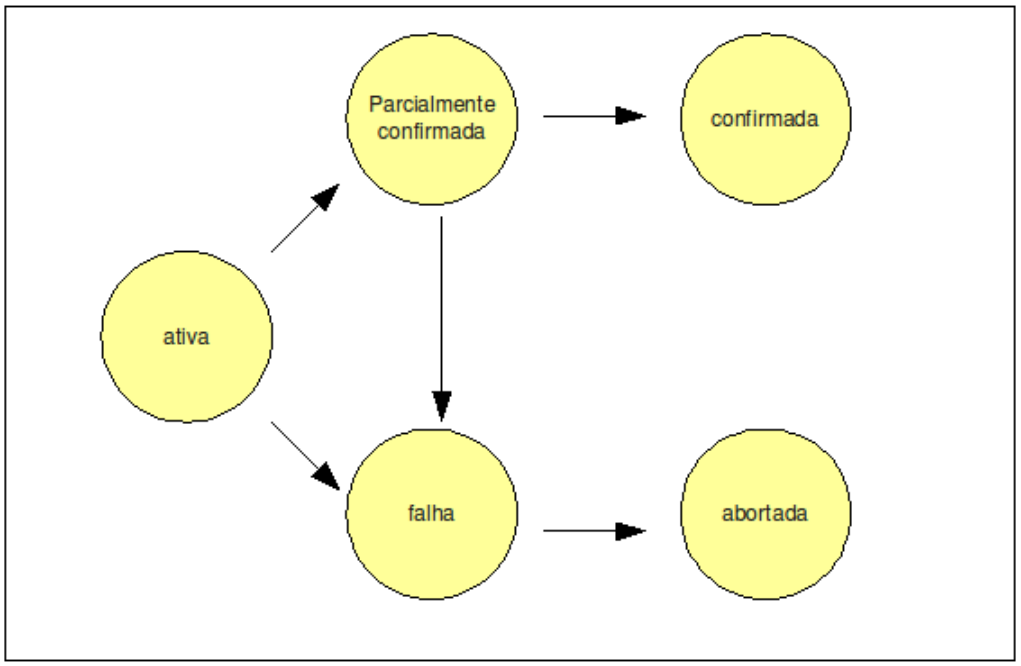

Figura 1 - Estados de uma transação

7.2. Transações no PostgreSQL

Diferentemente dos SGBD´s tradicionais, que usam bloqueios para controlar a simultaneidade, o PostgreSQL mantém a consistência dos dados utilizando o modelo multi-versão (Multiversion Concurrency Control, MVCC).

Isto significa que ao consultar o banco de dados, cada transação enxerga um estado do banco de dados, ou seja, como este era há um tempo atrás, sem levar em consideração o estado corrente dos dados subjacentes. Este modelo protege a transação para não enxergar dados inconsistentes, o que poderia ser causado por atualizações feitas por transações simultâneas nas mesmas linhas de dados, fornecendo um isolamento da transação para cada sessão do banco de dados.

A principal vantagem de utilizar o modelo de controle de simultaneidade MVCC em vez de bloqueios é que no MVCC os bloqueios obtidos para consultar dados (leitura) não conflitam com os bloqueios obtidos para escrever dados e, portanto, a leitura nunca bloqueia a escrita, e a escrita nunca bloqueia a leitura.

No PostgreSQL, assim como em outros SGBDs, são utilizados alguns comandos para lidar com transações. Dentre esses comandos, podemos citar: BEGIN, COMMIT e ROLLBACK. O comando BEGIN inicia um bloco de comandos SQL que fazem parte de uma transação. O comando COMMIT indica que todos os elementos da transação foram executados com sucesso e podem agora serem persistidos e acessados por todas as demais transações concorrentes ou subseqüentes. Já o comando ROLLBACK indica que a transação será abandonada e todas as mudanças feitas nos dados pelas instruções em SQL serão canceladas. O banco de dados e apresentará aos seus usuários como se nenhuma mudança tivesse ocorrido desde a instrução BEGIN.

Exemplo:

```
CREATE TABLE genero(
        id              numeric(5),
        nome            varchar(50),
        constraint pk_genero primary key (id)
);
```

```
CREATE TABLE filme(
        id              numeric(5),
        nome            varchar(100),
        duracao         varchar(20),
        sinopse         varchar(200),
        data            date,
        genero          numeric(5),
        constraint pk_filme       primary key (id),
        constraint fk_filme_genero foreign key (genero) references genero(id)
);

CREATE SEQUENCE id_genero;
CREATE SEQUENCE id_filme;

INSERT INTO genero VALUES (nextval('id_genero'), 'Ação');
INSERT INTO filme VALUES (nextval('id_filme'), 'Duro de Matar', '120 minutos', 'Policial em prédio decide encarar....', '2009-03-03',(select id from genero where nome= 'Ação'));

SELECT * FROM filme;

BEGIN WORK;
UPDATE filme SET nome='Duro de Matar 5' WHERE id=1;
ROLLBACK WORK;

SELECT * FROM filme;

BEGIN WORK;
UPDATE filme SET nome='Duro de Matar 5' WHERE id=1;
COMMIT WORK;

SELECT * FROM filme;
```

Na versão 8 do PostgreSQL apareceram os SAVEPOINTS (pontos de salvamento), que guardam as informações até eles. Isso salva as operações existentes antes do SAVEPOINT e basta um ROLLBACK TO para continuar com as demais operações.

```
BEGIN WORK;
INSERT INTO genero VALUES (nextval('id_genero'), 'Comédia');
SAVEPOINT meu_ponto_salvamento;
INSERT INTO filme VALUES (nextval('id_filme'), 'Duplex', '120 minutos', 'Casa compra um duplex....', '2008-10-03',(select id from genero where nome= 'Comédia'));
ROLLBACK TO meu_ponto_salvamento;
INSERT INTO filme VALUES (nextval('id_filme'), 'Cada um tem a gêmea que merece', '120 minutos', 'A irmão gêmea....', '2012-02-03',(select id from genero where nome= 'Comédia'));
COMMIT;

SELECT * FROM genero;
SELECT * FROM filme;
```

8. Stored Procedures em SQL (Functions)

Stored Procedure nada mais é que um programa desenvolvido em determinada linguagem de script e armazenado no servidor, local onde é processado. Também são conhecidos como funções, motivo este pelo qual nos referenciamos a uma stored procedure no PostgreSQL pelo 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 possuírem 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.
- **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.

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';
--OBS: em LANGUAGE, as aspas podem ou não ser exigidas dependendo da versão do PostgreSQL instalado.
```

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 em que linguagem está sendo implementada a função.

Vamos utilizar a tabela abaixo para exemplificar o funcionamento de uma função escrita na linguagem SQL.

```
CREATE TABLE pessoas (
        id              numeric(10),
        nome            varchar(255),
        sobrenome       varchar(255),
        constraint pk_pessoa primary key (id)
);

CREATE SEQUENCE idpessoas;

INSERT INTO pessoas (id,nome, sobrenome)
        VALUES (nextval('idpessoas'),'Jule', 'Silva');
INSERT INTO pessoas (id,nome, sobrenome)
        VALUES (nextval('idpessoas'),'Carlinhos', 'Pereira');
```

```
INSERT INTO pessoas (id,nome, sobrenome)
        VALUES (nextval('idpessoas'),'Juaum', 'Canhaum');
INSERT INTO pessoas (id,nome, sobrenome)
        VALUES (nextval('idpessoas'),'Pepo', 'Silva');
```

Vamos montar uma função que retorne nossos parentes. Supondo que temos parentes nas famílias Silva e Pereira, a função poderia ser assim:

```
CREATE FUNCTION familia ()
RETURNS SETOF pessoas AS
'
        SELECT *
        FROM pessoas
        WHERE sobrenome = ''Silva'' OR sobrenome = ''Pereira''
'
LANGUAGE 'sql';
```

Para visualizar o resultado dessa função podemos usar:

```
SELECT * FROM familia();
SELECT familia();
SELECT nome, sobrenome FROM familia();
```

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';
```

Quando desejar excluir uma função do sistema utilize o comando:

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

Para excluir uma função é necessário passar toda a sua assinatura:

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

Ainda existe o fato de que no momento da exclusão você pode excluir a função passando mais um parâmetro, como no exemplo a seguir:

```
DROP FUNCTION totalNota(INTEGER) RESTRICT;
ou
DROP FUNCTION totalNota(INTEGER) CASCADE;
```

Passando o RESTRICT como parâmetro, a exclusão da função será recusada caso existam dependências de objetos em torno da função (como por exemplo, triggers e operadores). Com o CASCADE esses objetos serão excluídos juntamente com a função.

Outro detalhe importante a ser destacado, é a maneira como utilizamos as funções. Quando a função é criada sem o uso de retorno SETOF, basta utilizarmos a seguinte sintaxe:

```
SELECT nome_da_funcao();
```

O mesmo modo deve ser usado quando as funções são criadas utilizando SETOF com tipos já definidos pelo PostgreSQL, como por exemplo, INTEGER, TIMESTAMP ou outro. Mas quando as funções possuem o seu retorno referenciado em uma tabela ou uma view, ou seja, quando a função retorna um resultset, devemos utilizar a função da seguinte maneira:

```
SELECT * FROM nome_da_funcao();
```

Quando criamos uma função tanto em SQL como em PL/PgSQL e posteriormente necessitamos alterar a sua seqüência de instruções, basta utilizarmos o comando CREATE OR REPLACE, desde que a alteração da função não altere o tipo de retorno. Se isto ocorrer, a função terá que ser excluído primeiro para depois ser adicionada à nova versão.

Se a função for removida e recriada, a nova função não é mais a mesma entidade que era antes, ficarão inválidas as regras, visões, gatilhos, etc. existentes que fazem referência à antiga função. Use o comando CREATE OR REPLACE FUNCTION para mudar a definição de uma função, sem invalidar os objetos que fazem referência à função.

O PostgreSQL, lembrando o que acontece com linguagens como Java, permite a sobrecarga de funções, ou seja, o mesmo nome pode ser utilizado em funções diferentes, desde que os argumentos sejam de tipos distintos, portanto há possibilidade de que existam duas funções soma, uma retornando a soma de valores do tipo integer e outra do tipo float.

9. Funções em Linguagens Procedurais- PL/PgSQL

A PL/PgSQL é uma linguagem estrutural estendida da SQL que tem por objetivo auxiliar as tarefas de programação no PostgreSQL. Ela incorpora à SQL características procedurais, como os benefícios e facilidades de controle de fluxo de programas que as melhores linguagens possuem. Por exemplo, loops estruturados (for, while) e controle de decisão (if then else).

Dessa forma, programar em PL/PgSQL significa ter a disposição um ambiente procedural totalmente desenvolvido para aplicações de bancos de dados, beneficiando-se do controle transacional inerente das aplicações deste tipo.

Alguns exemplos

```
CREATE OR REPLACE FUNCTION primeira_funcao()
RETURNS VOID AS
$body$
        BEGIN
                RAISE NOTICE 'Minha primeira rotina em PL/pgSQL';
                RETURN;
        END;
$body$

LANGUAGE 'plpgsql';

select primeira_funcao();


CREATE OR REPLACE FUNCTION soma(integer,integer)
RETURNS integer AS
$body$
        declare soma integer;
        BEGIN
                RAISE NOTICE 'Nesta função os parâmetros não são nomeados';
                soma:=$1+$2;
                RETURN(soma);
        END;
$body$

LANGUAGE 'plpgsql';

select * from soma(5,1);

CREATE OR REPLACE FUNCTION subtracao(integer, integer)
RETURNS integer AS
$body$
        declare subtracao integer;
        BEGIN
                RAISE NOTICE 'Função com estrutura de decisão';
                if $1 > $2 then
                        subtracao:=$1-$2;
                else
                        subtracao:=$2-$1;
                end if;

                RETURN(subtracao);
        END;
$body$

LANGUAGE 'plpgsql';
```

```
select * from subtracao(5,1);
```

```
CREATE OR REPLACE FUNCTION subtracao2 (n1 integer,n2 integer)
RETURNS integer AS
$body$
        declare subtracao integer;
        BEGIN
                RAISE NOTICE 'Função com estrutura de decisão e nomeação dos
        parâmetros';
                if n1 > n2 then
                        subtracao:=n1-n2;
                else
                        subtracao:=n2-n1;
                end if;

                RETURN(subtracao);
        END;
$body$

LANGUAGE 'plpgsql';
```

select * from subtracao2 (5,1);

```
Create function mostra_valor()
returns integer as
$body$
  Declare valor integer := 30;
  BEGIN
        --Valor = 30
        Raise notice 'O valor da variável aqui é %', valor;

        --Sub bloco
        Declare valor integer := 50;
        Begin
                --Valor = 50
                Raise notice 'O valor da variável aqui é %', valor;          End;

        --Valor = 30
        Raise notice 'O valor da variável aqui é %', valor;
        return valor;
  END;
$body$

LANGUAGE 'plpgsql';
```

select mostra_valor();

10. Trigger

Triggers são procedimentos armazenados que são acionados por algum evento e em determinado momento. Na maioria dos bancos de dados estes eventos podem ser inserções (INSERT), atualizações (UPDATE) e exclusões (DELETE), e os momentos podem ser dois: antes da execução do evento (BEFORE) ou depois (AFTER). E isso também vale para o PostgreSQL.

Um diferencial das triggers deste banco de dados para outros é que no PostgreSQL as triggers são sempre associadas a funções de triggers e, nos demais, criamos o corpo da trigger na própria declaração desta.

No PostgreSQL a trigger apenas dispara uma função do tipo "função de trigger", isto é, ela apenas pode chamar uma função que retorna um tipo trigger. As funções de triggers devem ser escritas em C (linguagem C) ou alguma linguagem procedural disponível no banco de dados (Ex. PL/PgSQL).

A sintaxe para a criação de uma trigger é apresentada abaixo:

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

Argumentos:

- ✓ **nome** é o nome da trigger.
- ✓ **before | after** determina se a função será chamada antes ou depois do evento.
- ✓ **evento** indica em que momento a trigger será disparada. A trigger pode ser dispara antes ou depois de um evento de DELETE, UPDATE ou INSERT.
- ✓ **tabela** indica a qual tabela a trigger estará associada.
- ✓ **row | statement** especifica se a trigger deve ser disparada uma vez para cada linha afetada pelo evento ou apenas uma vez por comando SQL. Se não for especificado nenhum dos dois, o padrão é FOR EACH STATEMENT.
- ✓ **nome_da_função** especifica a função de trigger.
- ✓ **argumentos** é uma lista opcional de argumentos, separados por vírgula, fornecidos à função junto com os dados usuais dos gatilhos, como os conteúdos novos e antigos das tuplas, quando o gatilho é executado. Os argumentos são constantes literais na forma de cadeias de caracteres. Nomes simples e constantes numéricas podem ser escritos também, mas serão convertidos em cadeias de caracteres.

A sintaxe do comando para alterar uma trigger é apresentada abaixo:

```
ALTER TRIGGER nome ON tabela RENAME TO novo_nome
```

Argumentos:

- ✓ **nome** é nome do gatilho existente a ser alterado.
- ✓ **tabela** é o nome da tabela onde o gatilho atua.
- ✓ **novo_nome** é o novo nome do gatilho.

Para excluir uma trigger basta executar o comando abaixo:

```
DROP TRIGGER nome ON tabela [ CASCADE | RESTRICT ]
```

Argumentos:

- ✓ **nome** é o nome do gatilho a ser removido.
- ✓ **tabela** é o nome da tabela para a qual o gatilho está definido.

- ✓ **[CASCADE | RESTRICT]** indica se ao remover a trigger vamos remover também todos os objetos que dependem dela (CASCADE) ou recusaremos sua exclusão (RESTRICT).

A função de trigger é criada através do comando create function, sendo declarado como uma função que não recebe argumentos e retorna o tipo trigger. A função deve ser declarada sem argumentos, mesmo que espere receber argumentos especificados no comando create trigger. Os argumentos do gatilho são passados através de TG_ARGV, conforme descrito abaixo.

Quando uma função escrita em PL/PgSQL é chamada como um gatilho, diversas variáveis especiais são criadas automaticamente no bloco de nível superior. São estas:

- ✓ NEW → Tipo de dado RECORD. Variável armazenando a nova linha do banco de dados para as operações de INSERT/UPDATE nos gatilhos no nível de linha. Esta variável é nula nos gatilhos no nível de declaração.
- ✓ OLD → Tipo de dado RECORD. Variável armazenando a linha antiga do banco de dados para as operações de UPDATE/DELETE nos gatilhos no nível de linha. Esta variável é nula nos gatilhos no nível de declaração.
- ✓ TG_NAME → Tipo de dado name. Variável contendo o nome do gatilho realmente disparado.
- ✓ TG_WHEN → Tipo de dado text. Uma cadeia de caracteres contendo BEFORE ou AFTER dependendo da definição do gatilho.
- ✓ TG_LEVEL → Tipo de dado text. Uma cadeia de caracteres contendo ROW ou STATEMENT dependendo da definição do gatilho.
- ✓ TG_OP → Tipo de dado text. Uma cadeia de caracteres contendo INSERT, UPDATE ou DELETE informando para qual operação o gatilho foi disparado.
- ✓ TG_RELID → Tipo de dado oid. O ID de objeto da tabela que causou o disparo do gatilho.

- ✓ TG_RELNAME → Tipo de dado name. O nome da tabela que causou o disparo do gatilho.
- ✓ TG_NARGS → Tipo de dado integer. O número de argumentos fornecidos ao procedimento de gatilho pela declaração CREATE TRIGGER.
- ✓ TG_ARGV[] → Tipo de dado matriz de text. Os argumentos da declaração CREATE TRIGGER. O contador do índice começa com 0 (zero). Índices inválidos (menor que 0 ou maior ou igual a TG_NARGS) resultam em um valor nulo.

Uma função de gatilho deve retornar nulo, ou um valor registro/linha possuindo a mesma estrutura da tabela para a qual o gatilho foi disparado.

Os gatilhos no nível de linha disparados BEFORE (antes) podem retorna nulo, para sinalizar ao gerenciador do gatilho para pular o restante da operação para esta linha (ou seja, os gatilhos posteriores não são disparados, e o INSERT/UPDATE/DELETE não ocorre para esta linha.

Se for retornado um valor não 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 (mas não tem efeito direto no caso do DELETE). Para alterar a linha a ser armazenada, é possível substituir valores únicos diretamente no NEW e retornar o NEW modificado, ou construir uma linha (registro) nova completa para retornar.

O valor retornado por um gatilho BEFORE ou AFTER no nível de declaração, ou por um gatilho AFTER no nível de linha, é sempre ignorado; pode muito bem ser nulo. Entretanto, qualquer um destes tipos de gatilho pode interromper toda a operação causando um erro.

Considere a tabela pessoa para os exemplos 1 e 2:

```
CREATE TABLE pessoa(
        id      numeric(11) NOT NULL,
        nome    varchar(100),
        rg      numeric(9) NOT NULL,
        cpf     numeric(13),
        CONSTRAINT pessoa_pkey PRIMARY KEY (id),
        CONSTRAINT uk_rg UNIQUE (rg),
        CONSTRAINT uk_cpf UNIQUE (cpf)
);
```

1) O primeiro exemplo cria uma tabela log1 onde serão armazenados os dados referentes a deleção de uma linha da tabela pessoa.

```
CREATE SEQUENCE seq_log1;

CREATE TABLE log1(
        id              integer  default(nextval(„seq_log1")),
        id_pessoa       numeric(11),
        nome            varchar(100),
        data            timestamp,
        usuario         text,
        constraint pk_log1 primary key (id)
);
```

Depois cria uma função func_log() que retorna um tipo trigger, que insere na tabela log1, o id e o nome da pessoa deletada e ainda coloca a data e o responsável pela deleção.

```
CREATE OR REPLACE FUNCTION func_log()
RETURNS trigger AS
"
BEGIN
        Insert into log1 (id_pessoa, nome, data, usuario)
                values (OLD.id, OLD.nome, now(), current_user);
        RETURN OLD;

END;
"
Language „plpgsql";
```

Em seguida deve-se criar a trigger para a função acima. Antes de deletar uma linha na tabela pessoa execute a função func_log().

```
CREATE TRIGGER func_log BEFORE delete ON pessoa
        FOR EACH ROW EXECUTE PROCEDURE func_log();


SELECT * FROM PESSOA;
INSERT INTO PESSOA VALUES (1,"ZE",1123,95762);
DELETE FROM PESSOA WHERE ID = 1;
```

2) O segundo exemplo cria uma tabela log2 onde serão armazenados os dados referentes a inserção de uma linha da tabela pessoa.

```
CREATE SEQUENCE seq_log2;

CREATE TABLE log2(
        id              integer   default(nextval(„seq_log2")),
        id_pessoa       numeric(11),
        nome            varchar(100),
        data            timestamp,
        usuario         text,
        constraint pk_log2 primary key (id)
);

CREATE OR REPLACE FUNCTION func_log2()
RETURNS trigger AS
"
BEGIN
        Insert into log2 (id_pessoa, nome, data, usuario)
                values (NEW.id, NEW.nome, now(), current_user);
        RETURN NEW;

END;
"
Language „plpgsql";

CREATE TRIGGER func_log2 BEFORE insert or update ON pessoa
        FOR EACH ROW EXECUTE PROCEDURE func_log2();

INSERT INTO PESSOA VALUES (3,"ZED",11323,154796);
```

3) O terceiro exemplo cria uma tabela emp e verifica se os dados estão sendo passados corretamente antes de realizar a inserção ou atualização.

```
CREATE TABLE emp(
        id              integer,
        nome_emp        text,
        salario         numeric(10,2),
        ultima_data     timestamp,
        ultimo_usuario  text,
        constraint pk_emp primary key (id)
);

CREATE OR REPLACE FUNCTION emp_carimbo()
RETURNS trigger AS
"
BEGIN
        --Verifica se o nome do empregado foi fornecido
        IF NEW.nome_emp IS NULL THEN
          RAISE EXCEPTION „„O nome do empregado não pode ser nulo"";
        END IF;

        --Verifica se o salário foi fornecido
        IF NEW.salario IS NULL THEN
          RAISE EXCEPTION „„% não pode ter um salário nulo"", NEW.nome_emp;
        END IF;
```

```
        --Verifica se o salário é negativo
        IF NEW.salario < 0 THEN
          RAISE EXCEPTION „„% não pode ter um salário negativo"", NEW.nome_emp;
        END IF;

        --Registrar quem alterou o pagamento e quando
        NEW.ultima_data := „„now"";
        NEW.ultimo_usuario := current_user;
        RETURN NEW;
END;
"

Language „plpgsql";

CREATE TRIGGER emp_carimbo BEFORE insert or update ON emp
        FOR EACH ROW EXECUTE PROCEDURE emp_carimbo();

INSERT INTO EMP (id,nome_emp,salario) VALUES (1,"BRUNO",-10);
INSERT INTO EMP (id,nome_emp,salario) VALUES (1,"BRUNO",10000);
```