UNIVERSIDADE FEDERAL DO CEARÁ
Campus Russas

Disciplina: Fundamentos de Banco de Dados

15. Funções, Gatilhos e outros conceitos

Professora: Marília S. Mendes
E-mail: marilia.mendes@ufc.br

Organização da aula

- **Herança (*INHERITS*)**
- Regras (*Rules*)
- Funções e Procedimentos (*Stored Procedures*)
- Gatilhos (*Triggers*)
- Controle de acesso

Herança

- Herança permite a criação de uma nova tabela relacionada a uma já existente.

- Com herança, a tabela-filha recebe todas as colunas da tabela-pai, adiciona colunas extras se assim desejar.

Herança - Exemplo

```
CREATE TABLE pai (col1 INTEGER);
```

Visualize a tabela pai no Postgres.

```
CREATE TABLE filho (col2 INTEGER) INHERITS (pai);
```

Visualize a tabela filho no Postgres.

Herança – Exemplo: inserindo valores

```
INSERT INTO pai VALUES (1);
INSERT INTO filho VALUES (2,3);
```

Visualizem as tabelas! O que aconteceu em ambas?

Herança em camadas

```
CREATE TABLE neto (col3 INTEGER) INHERITS (filho);

INSERT INTO neto VALUES (4, 5, 6);
```

O que aconteceu com pai e filho?

Herança - exemplo

- Crie uma tabela que herde todos os atributos de EMPREGADO, porém, que além disso, apresente o atributo NACIONALIDADE.

```
CREATE TABLE EMP_NASC (NACIONALIDADE
VARCHAR (40)) INHERITS (EMPREGADO)
```

Herança – cuidado!

- Existem algumas deficiências com relação à restrição de integridade no uso das heranças.

Por exemplo, a tabela empregado foi criada com uma chave primária ssn, o que impede que eu possua dois empregados com uma mesma matrícula, entretanto essa restrição não é válida para a tabela herdada. Dessa forma poderiam existir dois gerentes com a mesma matrícula.

Organização da aula

- Herança (*INHERITS*)
- **Regras (*Rules*)**
- Procedimentos armazenados (*Stored Procedures*)
- Gatilhos (*Triggers*)
- Controle de acesso

Regras (Rules)

- Servem para definir uma ação alternativa a ser realizada nos eventos em tabelas ou visões.
- Uma regra faz com que sejam executados comandos adicionais quando é executado um determinado comando em uma determinada tabela.
- Na verdade, uma regra é um mecanismo que transforma comandos.
- Tal transformação ocorre antes de se executar o comando.
- Os eventos que podem disparar uma RULE são: SELECT, INSERT, UPDATE e DELETE.

Regras (Rules) - Exemplo

```
CREATE TABLE Emp_colaboradores (
  mat_id int PRIMARY KEY,
  nome varchar (15) NOT NULL,
  snome varchar(30) NOT NULL,
  cargo varchar(15),
  setor varchar(15),
  salario real,
  dt_admis date
);
```

Regras (Rules) - Exemplo

```
INSERT INTO Emp_colaboradores VALUES
(10,'Ana', 'Azevedo','gerente', 'Compras', 2000, '12-11-2011'),

(20,'Jose', 'Farias','gerente', 'Vendas', 2000, '12-11-2011'),

(30,'Mateus', 'Lopes','balconista', 'Vendas', 1000, '12-11-2011')
```

Regras (Rules) - Exemplo

```
CREATE TABLE ex_Emp_colaboradores(
  mat_id int not null,
  nome varchar (15) NOT NULL,
  snome varchar  (30) NOT NULL,
  dt_demis date
);
```

Regras (Rules) – Exemplo: criando uma regra

```
CREATE OR REPLACE RULE rl_ex_Emp_colaboradores
AS
  ON DELETE TO Emp_colaboradores
  DO INSERT INTO ex_Emp_colaboradores VALUES
  (OLD.mat_id,OLD.nome,OLD.snome,now());
```

Regras (Rules) – Exemplo:
Deletando um funcionário

```
DELETE FROM Emp_colaboradores
WHERE mat_id = 20;
```

Foi deletado na tabela Emp_Colaboradores?

Agora acessem a tabela Ex-Emp_Colaboradores.

Regras (Rules) – Exemplo: criando uma regra

```
CREATE OR REPLACE RULE rl_ex_Emp_colaboradores
AS
  ON DELETE TO Emp_colaboradores
  DO INSERT INTO ex_Emp_colaboradores VALUES
  (OLD.mat_id,OLD.nome,OLD.snome,now());
```

NEW e OLD

O nome especial de tabela NEW pode ser utilizado para referenciar os novos valores inseridos (ON INSERT) ou atualizados (ON UPDATE) em uma tabela. Já nome especial OLD contém os valores apagados através de um DELETE ou UPDATE.

Regras (Rules)

```
CREATE [ OR REPLACE ] RULE name AS ON event
    TO table_name [ WHERE condition ]
    DO [ ALSO | INSTEAD ] { NOTHING | command | ( command ; command ... ) }
```

- **Parâmetros**
- *Nome*
  - O nome da regra a ser criada, devendo ser distinto do nome de qualquer outra regra para a mesma tabela. Caso existam várias regras para a mesma tabela e mesmo tipo de evento, estas regras serão aplicadas na ordem alfabética dos nomes.
- *Evento*
  - Evento é um entre SELECT, INSERT, UPDATE e DELETE.

Regras (Rules)

```
CREATE [ OR REPLACE ] RULE name AS ON event
    TO table_name [ WHERE condition ]
    DO [ ALSO | INSTEAD ] { NOTHING | command | ( command ; command ... ) }
```

- *Tabela*
  - O nome (opcionalmente qualificado pelo esquema) da tabela ou da visão à qual a regra se aplica.
- *Condição*
  - Qualquer expressão condicional SQL (retornando *boolean*). A expressão condicional não pode fazer referência a nenhuma tabela, exceto NEW e OLD, e não pode conter funções de agregação.

Regras (Rules)

- INSTEAD
  - INSTEAD indica que os comandos devem ser executados *no lugar do* (*instead of*) comando original.
- ALSO
  - ALSO indica que os comandos devem ser executados *em adição* ao comando original.
    - Se não for especificado nem ALSO nem INSTEAD, o padrão é ALSO.
- *Comando*
  - O comando ou comandos que compõem a ação da regra. Os comandos válidos
    são SELECT, INSERT, UPDATE, DELETE e NOTIFY.

Regras (Rules)

- Dentro da *condição* e do *comando* podem ser utilizados os nomes especiais de tabela NEW e OLD, para fazer referência aos valores na tabela referenciada.
  - NEW é válido nas regras ON INSERT e ON UPDATE, para fazer referência à nova linha sendo inserida ou atualizada.
  - OLD é válido nas regras ON UPDATE e ON DELETE, para fazer referência à linha existente sendo atualizada ou excluída.

Regras - exercício 1

- Vamos criar uma rule de auditoria que mantém o histórico de atualizações de salário dos empregados.

Passos:

1: cria a nova tabela que vai armazenar o histórico
2: cria a regra
3: testar, alterando um valor de salário

Regras - exercício 1 - resposta

1. Criando a nova tabela:

```
CREATE TABLE LOG_EMPREGADO (
   PNOME VARCHAR(30),
   UNOME VARCHAR(30),
   SSN INT NOT NULL,
   SALARIO FLOAT
);
```

Regras - exercício 1 - resposta

2. Criando a regra:

```
CREATE RULE log_emp_rule AS ON UPDATE TO
empregado
    WHERE NEW.salario <> OLD.salario
    DO INSERT INTO log_empregado
    VALUES (NEW.PNOME, NEW.UNOME, NEW.ssn,
NEW.salario);
```

Regras - exercício 1 - resposta

3. testar, alterando um valor de salário

UPDATE empregado SET salario = 30000 WHERE pnome = ‘Alicia';

Regras - exercício 2

- Apague esta regra

```
DROP RULE log_emp_rule ON empregado
```

- Crie esta mesma regra só que inserindo o usuário que modificou o salário e a data / hora de modificação.

```
create RULE log_emp_rule AS ON UPDATE TO empregado
    WHERE NEW.salario <> OLD.salario
     DO INSERT INTO log_empregado
    VALUES (NEW.PNOME, NEW.UNOME, NEW.ssn,
NEW.salario, current_user, current_date);
```

Regras - exercício 2

- Antes de criar a regra, atualiza a tabela inserindo os dois novos campos

```
alter table log_empregado
add usuario varchar(50),
add datahora date;
```

Regras - exercício 2

- Agora teste!

pnome character varying(30)	unome character varying(30)	ssn integer	salario double precision	usuario character varying(50)	datahora date
James	Borg	665555	45000		
marilia	mendes	567864	11000		
marilia	mendes	567864	35000	postgres	2016-07-

Organização da aula

- Herança (*INHERITS*)
- Regras (*Rules*)
- **Funções, Procedimentos e Gatilhos**
- Controle de acesso

Funções e procedimentos

- As Funções e Procedimentos permitem que a "*lógica do negócio*" fique armazenada no banco de dados e seja executada a partir de comandos SQL.

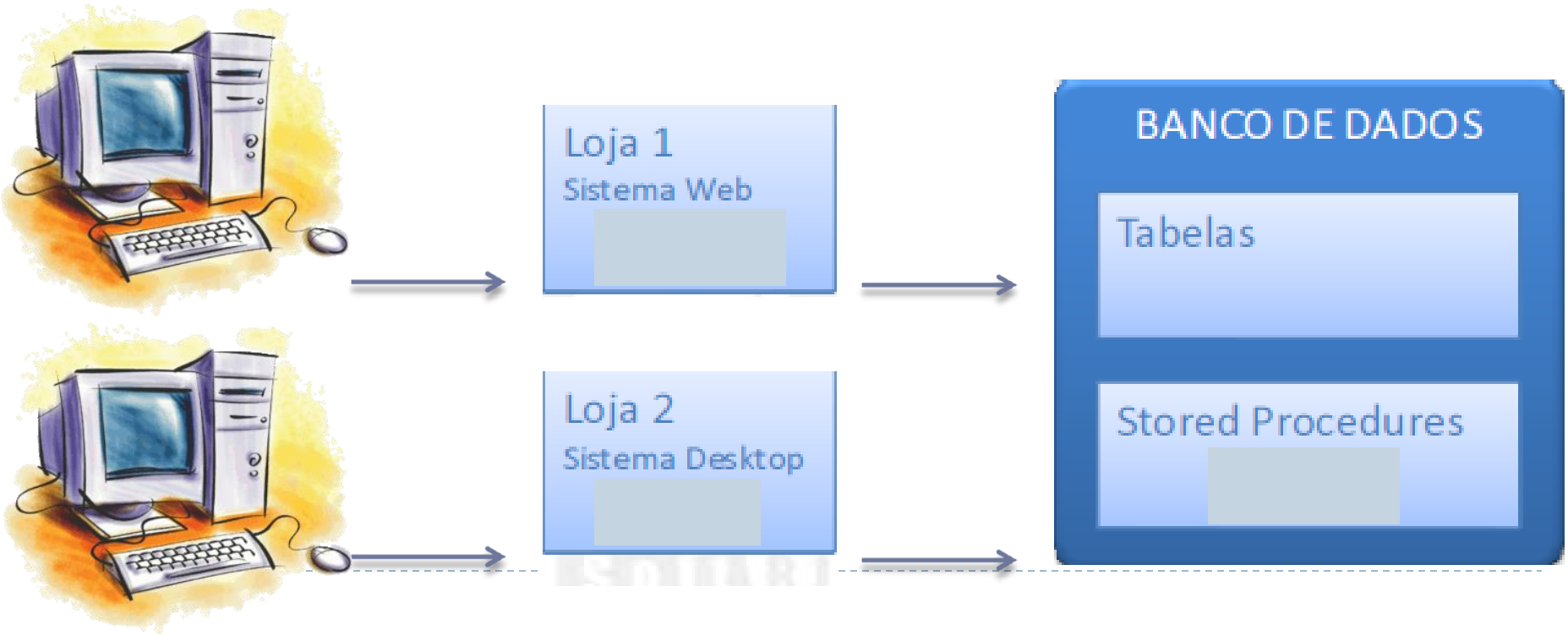
- Por exemplo: Domínio Universidade
  - Regras sobre quantos cursos um aluno pode realizar em determinado semestre, o número mínimo de cursos que um instrutor de tempo integral deve ministrar por ano, o número máximo de disciplinas em que um aluno pode estar matriculado, e assim por diante.

Procedimentos armazenados

- Vantagens
  - Centralização
  - Segurança
  - Performance / Velocidade
  - Suporte a transações

Exemplo de Stored Procedure

- Limpeza de registro vencidos
  - Verifica em uma tabela de pedidos quais foram abertos a mais de uma semana e ainda não foram confirmados, excluindo-os do sistema



Gerenciando Stored Procedures

- Criando uma Stored Procedure:

```
CREATE PROCEDURE Nome
```

- Invocando uma Stored Procedure:

```
CALL Nome
EXECUTE Nome
```

- Excluindo uma Stored Procedure:

```
DROP PROCEDURE Nome
```

Procedimentos armazenados

- Formato geral:

```
CREATE PROCEDURE <nome do procedimento>
(<parametros>)
<declaracoes de local>
<corpo do procedimento> ;
```

```
CREATE FUNCTION <nome da funcao>
(<parametros>)
RETURNS <tipo de retorno>
<declaracoes de local>
<corpo da funcao> ;
```

São Funções no PostgreeSQL

- Diferentemente de outros SGBDs que tratam os conceitos de procedimentos armazenados, gatilhos e funções como coisas distintas, o PostgreSQL trata todos eles como funções. Essas funções tem características diferentes, mas são todas criadas como funções ou functions.
  - Por exemplo, o que diferencia uma função de trigger (gatilho) das outras é o tipo de dado que ela retorna.

Procedural Languages

- PostgreSQL usa o Bison como parser, de modo que é fácil portar linguagens open source, bem como reusar código.

- Linguagens procedurais disponíveis para o PostgreSQL:
  - PL/pgSQL
  - PL/Java
  - PL/Perl
  - PL/pgPSM
  - PL/php
  - PL/Python
  - PL/R
  - PL/Ruby
  - PL/sh
  - PL/Tcl
  - PL/Lua.

PL/pgSQL - Procedural Language/ PostgreSQL

- Linguagem procedural carregável desenvolvida para o PostgreSQL.

- Objetivos
  - criar procedimentos de funções e de gatilhos;
  - adicionar estruturas de controle à linguagem SQL;
  - realizar processamentos complexos;
  - herdar todos os tipos de dado, funções e operadores definidos pelo usuário;
  - ser definida como confiável pelo servidor;
  - ser fácil de usar.

Funções (Functions)

Funções SQL permitem que nomeiem consultas e as armazenem dentro do banco de dados para acesso posterior.

```
create function soma(a real, b real) returns real as $$
begin
  return a + b;
end;
$$ language plpgsql;

select soma(23.45,87.123);
```

Funções (Functions)

- Outro exemplo:

```
CREATE FUNCTION tax(numeric)
RETURNS numeric
AS 'SELECT ($1 * 0.06);'
LANGUAGE 'sql';


SELECT tax(100);
```

Funções (Functions)

- Mais exemplos:

```
CREATE TABLE ferram (id INTEGER, nome varchar(30), valor
NUMERIC(6,2));

INSERT INTO ferram VALUES (637, 'cabo', 14.29), (638, 'chip',
0.84), (639, 'chave', 3.68);
```

Funções (Functions)

- Mais exemplos:

```
SELECT
id, nome, valor, tax(valor),
valor+tax(valor) AS total
FROM ferram ORDER BY id;
```

Funções (Functions)

- Mais exemplos:

```
CREATE FUNCTION frete(numeric)
RETURNS numeric
AS 'SELECT CASE
WHEN $1 < 3 THEN CAST(3.00 AS numeric(8,2))
WHEN $1 >= 3 AND $1 < 10 THEN CAST(5.00 AS
      numeric(8,2))
WHEN $1 >= 10 THEN CAST(6.00 AS numeric(8,2))
END;'
LANGUAGE 'sql';
```

Funções (Functions)

- Mais exemplos:

```
SELECT
id, nome, valor, tax(valor),
valor+tax(valor) AS subtotal,
frete(valor),
valor + tax(valor)+frete(valor) AS total

FROM ferram
ORDER BY id;
```

Declarando e chamando funções e procedimentos SQL

Dado o nome do departamento, escrever uma função que retorne a contagem do número de instrutores do departamento.

```
Create funcion cont_dept (nome_dept varchar(20))
Returns integer
Begin
Declare cont_d integer;
    Select count(*) into cont_d
    From instrutor
    Where instrutor.nome_dept=nome_dept
Return cont_d;
end
```

```
Select nome_dept, orçamento
From departamento
Where cont_dept(nome_dept)>12;
```

Exercício 1 – Database Empresa

- Escrever uma função que, dado o nome do departamento, retorne a contagem do número de empregados deste departamento.

```
create function conta_dept(nome_dept
varchar(20))
returns integer AS $$
BEGIN

return count(*)
from departamento, empregado
where dnome=nome_dept and
dnumero=dno;

end;

$$ LANGUAGE 'plpgsql'
```

Para chamar:

Select dnome from departamento where conta_dept(dnome)>3

Exercício 2 – Database Empresa

- Escrever uma função que, dado o nome do departamento, retorne a soma dos salários dos empregados dele.

```
create function somaSal_dept(nome_dept
varchar(20))
returns integer AS $$
BEGIN

return sum(salario)
from departamento, empregado
where dnome=nome_dept and
dnumero=dno;

end;

$$ LANGUAGE 'plpgsql'
```

Para chamar:

Select SomaSal_dept('Pesquisa')

Exercício 3 – Database Empresa

- Escrever uma função que, dado o nome do empregado, retorne a contagem do número de dependentes dele.

```
create function
conta_dependentes(nomemp varchar(20))
returns integer AS $$
BEGIN

return count(essn) from empregado,
dependente
where essn=ssn and pnome=nomemp;

end;

$$ LANGUAGE 'plpgsql'
```

Para chamar:

Select conta_dependentes('John')

Funções de tabela

O padrão SQL admite funções que podem retornar tabelas como resultados

Exercício 4 – Database Empresa

- Elabore uma função que retorne uma tabela com todos os empregados de um determinado departamento.

```
create function ListaEmp(nome_dept
varchar(20))
returns table (empregados varchar (30))
 AS $$
BEGIN
return query
Select pnome as empregados from
empregado, departamento
Where dnumero=dno and
dnome=nome_dept;

end;

$$ LANGUAGE 'plpgsql'
```

Para chamar:

Select empregados from ListaEmp('Pesquisa')

Sintaxe fora do padrão para procedimentos e funções

- Embora o padrão SQL defina a sintaxe para procedimentos e funções, a maioria dos bancos de dados não segue o padrão diretamente, e existe uma variação considerável na sintaxe admitida.

- Um dos motivos para essa situação é que esses bancos de dados normalmente introduziram suporte para procedimentos e funções antes que a sintaxe fosse padronizada, e continuam a aceitar sua sintaxe original.

[SILBERSCHATZ, 2012]

Vantagens de usar os procedimentos armazenados do PostgreSQL

- Todas as instruções SQL estão envolvidas dentro de uma função armazenada no servidor de banco de dados, portanto, o aplicativo só deve emitir uma chamada de função para obter o resultado em vez de enviar várias instruções SQL e aguardar o resultado entre cada chamada.
- Aumenta o desempenho do aplicativo porque as funções definidas pelo usuário foram pré-compiladas e armazenadas no servidor de banco de dados PostgreSQL.
- Reuso em muitas aplicações. Depois de desenvolver uma função, você pode reutilizá-la em qualquer aplicativo.

Desvantagens de usar os procedimentos armazenados do PostgreSQL

- Maior tempo no desenvolvimento de software porque requer habilidades especializadas que muitos desenvolvedores não possuem.
- Torna difícil gerenciar versões e difícil de depurar.
- Pode não ser portátil para outros sistemas de gerenciamento de banco de dados, por exemplo, MySQL ou Microsoft SQL Server.

Triggers (gatilhos)

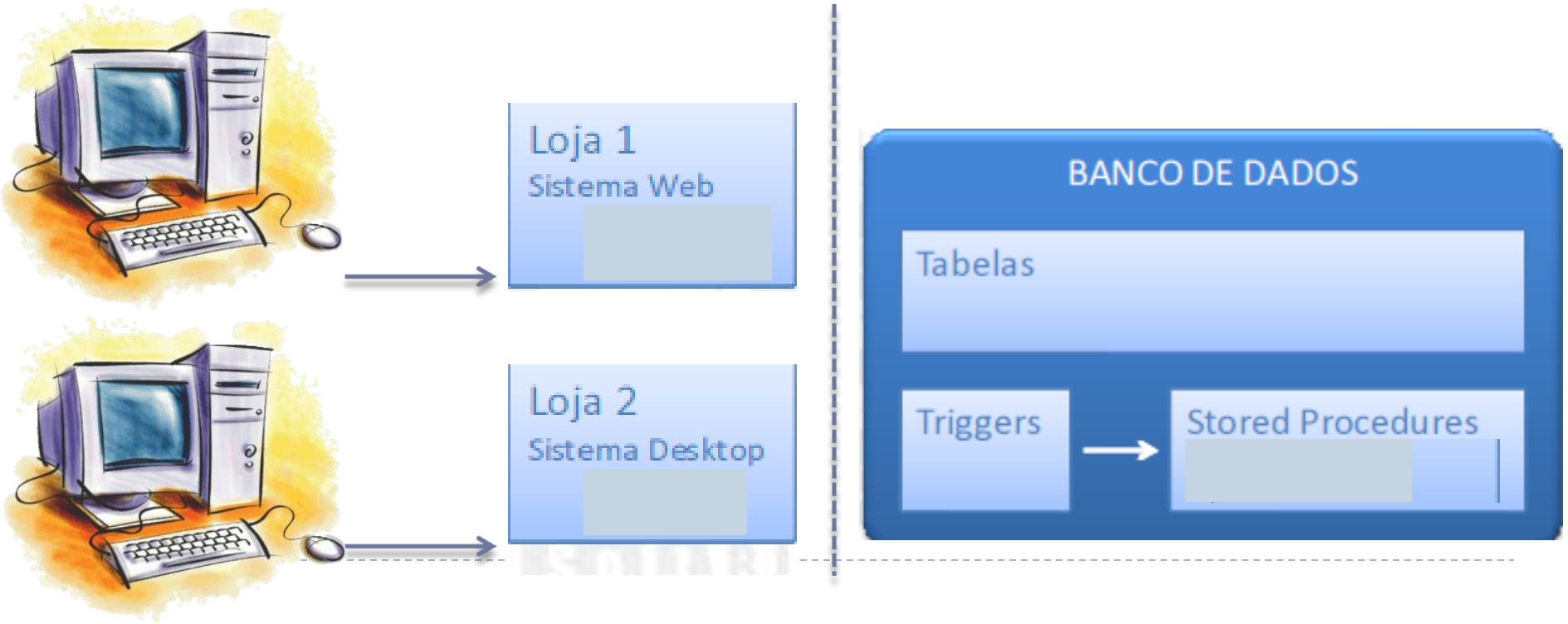
- Uma Trigger (gatilho) é um comando que o sistema executa automaticamente como um efeito colateral de uma modificação no banco de dados.
- Para criar um mecanismo de trigger, precisamos atender a dois requisitos
  - Especificar quando uma ***trigger*** deve ser executada. Isso é dividido em um **evento** que faz com que a ***trigger*** seja verificada e uma **condição** que deve ser satisfeita para que a execução da ***trigger*** prossiga
  - Especificar as **ações** a serem tomadas quando a ***trigger*** for executada.

Gatilhos (Triggers)

- Eventos que disparam códigos SQL
- Vantagens
  - As mesmas das Stored Procedures
  - Execução de código SQL baseado em eventos
- Tipos
  - Before insert
  - Before update
  - Before delete
  - After insert
  - After update
  - After delete
  - Temporais

Exemplo de Triggers

- Limpeza de registro vencidos
  - O mesmo exemplo apresentado para Stored Procedures, agora com início automático baseado em algum evento de trigger



Gerenciando Triggers

- Criando um Trigger:

```
CREATE TRIGGER Nome Tipo ON tabela
```

- Excluindo uma Trigger:

```
DROP TRIGGER Nome
```

Gatilhos (Triggers) - Exemplo

```
/* Criação da tabela original */
CREATE TABLE tb_teste (campo1 int);

/* Criação da tabela de logs */
CREATE TABLE tb_teste_log(
codigo serial primary key,
data date,
usuario varchar(15),
modificacao char(6)
);
```

Gatilhos (Triggers) - Exemplo

```
/* Criação da função vinculada ao trigger */
CREATE FUNCTION func_log() RETURNS trigger AS $$
BEGIN
  INSERT INTO tb_teste_log(data, usuario,
    modificacao) VALUES (now(), user, TG_OP);
  RETURN NEW;
END;
$$ LANGUAGE 'plpgsql';
```

Gatilhos (Triggers) - Exemplo

```
/* Criação do trigger */
CREATE TRIGGER tg_teste_log
AFTER INSERT OR UPDATE OR DELETE ON tb_teste
FOR EACH ROW EXECUTE PROCEDURE func_log();
```

Gatilhos (Triggers) - Exemplo

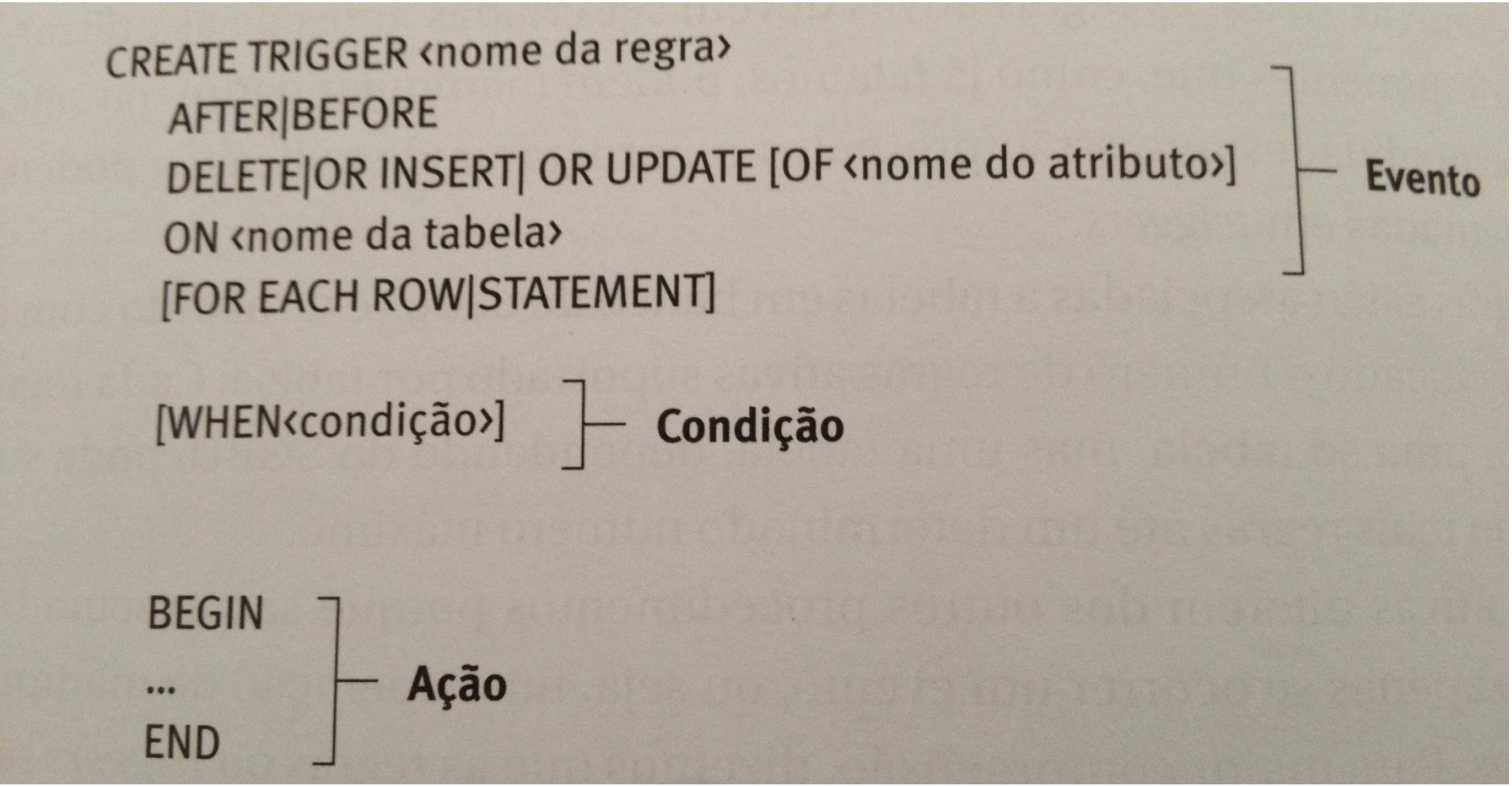
/* Inserções, Remoções e Modificações na tabela tb_teste */

```
INSERT INTO tb_teste VALUES (1),(2),(3),(4);

DELETE FROM tb_teste WHERE
  campo1 = 2 OR campo1 = 4;

UPDATE tb_teste SET campo1 = 7 WHERE
  campo1 = 3;
```

Gatilhos (Triggers) - Exemplo

```
/* Verificando a tabela de logs */
SELECT * FROM tb_teste_log;
```

Gatilhos (Triggers)



Gatilhos (Triggers) - exemplo

```
CREATE TRIGGER verificaIdade_pilotos
AFTER INSERT
ON pilotos
FOR EACH ROW
BEGIN
        IF (NEW.idade<18) THEN
            INSERT INTO log
            SET
            dataLog= curdate(),
            obs = 'Idade fora do padrao',
            tabela = 'pilotos',
            atributo = 'idade';
        END IF;
END;
```

Gatilhos (Triggers) - exemplo

```
CREATE TRIGGER verificaSalario_funcionario
BEFORE UPDATE
ON funcionarios
FOR EACH ROW
BEGIN
    IF(NEW.salario < 800) THEN
        SET
        NEW.salario = 800;
    END IF;
END;

SELECT * FROM funcionarios;
```

Regras x Gatilhos

- Regras
  - Não é possível usar uma linguagem procedural;
  - Substitui, Adiciona ou desabilita comandos;
  - Ao contrário de triggers, podem alterar a consulta executada;
  - Independem das funções definidas pelo usuário.
- Gatilhos
  - É um mecanismo mais complexo, porém muito mais poderoso;
  - Determinam funções a serem executadas antes ou depois de um tipo de evento;
  - Antes de sua criação, é preciso criar uma função cujo retorno é do tipo trigger e não pode receber parâmetros;
  - Pode executar um comando que acione outro trigger

Organização da aula

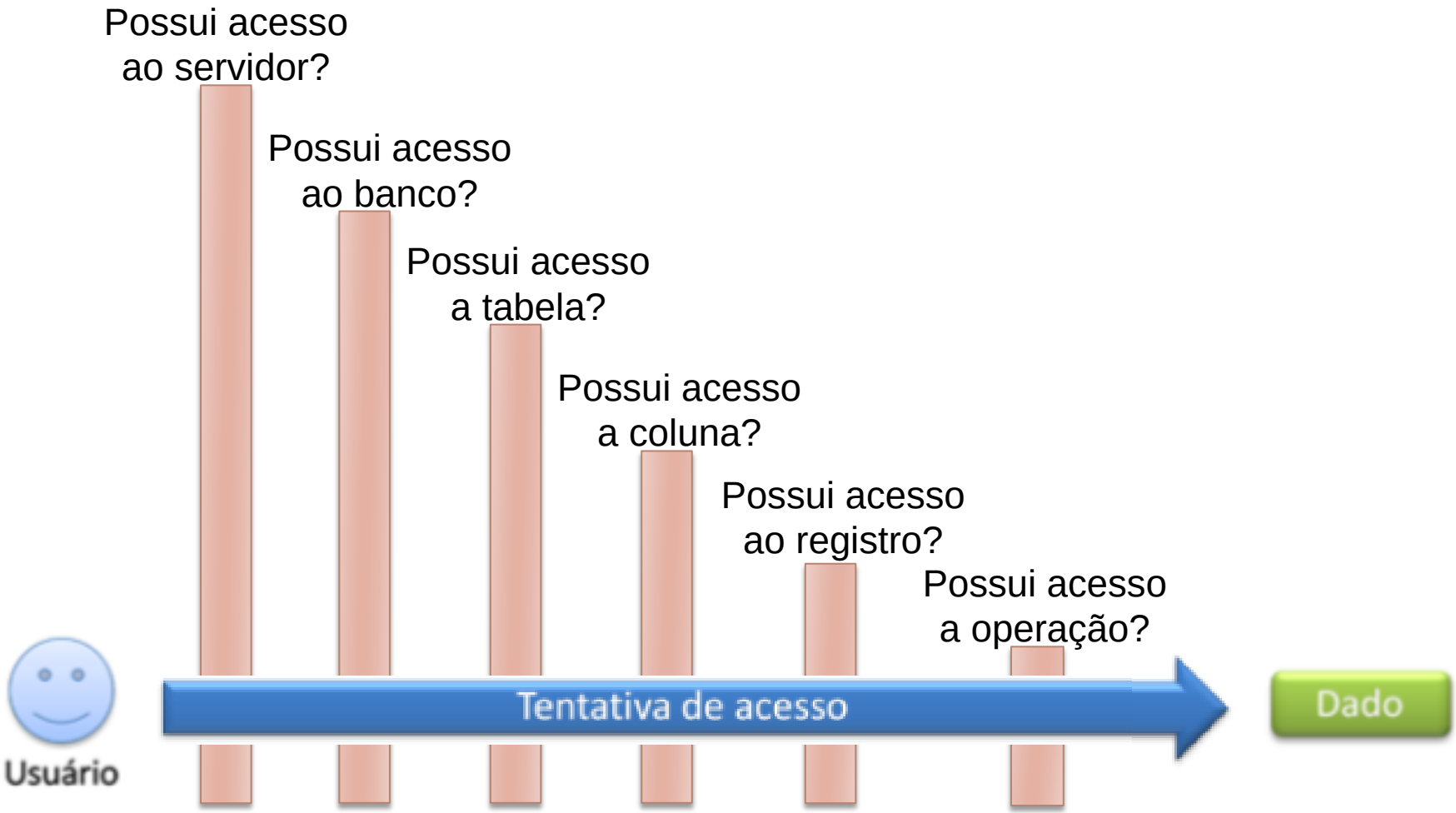
- Herança
- Regras
- Funções
- Gatilhos
- **Controle de acesso**

Controle de acesso

- DCL - **Data Control Language** - Linguagem de Controle de Dados.
- Formas de garantir que somente pessoas autorizadas possam realizar ações com os dados

- Níveis de acesso
  - Banco de Dados
  - Tabelas
  - Colunas
  - Registros
- Níveis de ações
  - Gerenciar estruturas
  - Gerenciar dados
  - Ler dados

Controle de acesso



Data Control Language

- Linguagem de Controle de Dados
  - **CREATE USER** Nome: cria um usuário
  - **DROP USER** Nome: exclui um usuário
  - **GRANT**: habilita acessos
  - **REVOKE**: revoga acessos

Gerenciando acessos

- Habilitando acesso
  - **GRANT** Ação **ON** Estrutura **TO** Usuário
- Revogando acesso
  - **REVOKE** Ação **ON** Estrutura **FROM** Usuário
- Ações
  - ALL, SELECT, INSERT, UPDATE, DELETE
- Estruturas
  - TABLES, VIEW, SEQUENCE

Gerenciando acessos - POSTGRESQL

- O PostgreSQL gerencia permissões de acesso usando o conceito de "roles", cujo uso pode ser para usuários e grupos, dependendo de como um "role" é definido.

- "Roles" podem controlar quem deverá ter acesso a objetos do BD.

- É possível conceder um membro de um "role" para outro, permitindo assim usufruir dos privilégios que foram concedidos a esse outro "role".

Gerenciando acessos - POSTGRESQL

CREATE ROLE nome_role [[WITH] opções [...]]

- As opções podem ser:
  - SUPERUSER | NOSUPERUSER;
  - CREATEDB | NOCREATEDB;
  - CREATEROLE | NOCREATEROLE :
  - INHERIT | NOINHERIT
  - LOGIN | NOLOGIN

Gerenciando acessos - POSTGRESQL

```
/* Criação de um usuário*/
CREATE ROLE usuario1 LOGIN;

/* Criação de usuario comum com senha definida em "123" */
CREATE ROLE usuario2 LOGIN ENCRYPTED
PASSWORD '123';

/* Criação de super usuário e senha "123" */
CREATE ROLE dbmaster LOGIN SUPERUSER
ENCRYPTED PASSWORD '123';
```

Gerenciando acessos - POSTGRESQL

```
/* Criação do role container(grupo) financeiro */
CREATE ROLE financeiro;

/* Criação do role marcia dentro do grupo financeiro */
CREATE ROLE marcia LOGIN ENCRYPTED PASSWORD
'123' IN ROLE financeiro;

/* Criação do role clara dentro do grupo financeiro */
CREATE ROLE clara LOGIN ENCRYPTED PASSWORD
'123' IN ROLE financeiro
```

Gerenciando acessos - POSTGRESQL

Apagando roles

```
/* Criação de um objeto do tipo tabela */
CREATE TEMP TABLE tb_1 (valor int);

/* Atribuição de propriedade da tabela criada para usuario1 */
ALTER TABLE tb_1 OWNER TO usuario1 ;

/* Tentativa (frustrada) de remoção do role usuario1 */
DROP ROLE usuario1;
```

Gerenciando acessos - POSTGRESQL

Apagando roles

```
/* Passando as propriedades de usuario 1 para usuario2*/
REASSIGN OWNED BY usuario1 TO usuario2 ;

/* Tentativa (com sucesso) de remoção do role usuario1 */
DROP ROLE usuario1;

/* Removendo todos os objetos de propriedade de usuario2 sem remover o role */
DROP OWNED BY usuario2;
```

Administração de usuários - Grupos

Também conhecidos como "roles containers": são como grupos de usuários. São "roles" que contém outros "roles":

```
/* Roles contâiner */
CREATE ROLE comercial;
CREATE ROLE vendas;

/* Criação container com 2 roles assimilados */
CREATE ROLE diretores WITH ROLE clara,marcia;

/* Inserindo um role de login em um container */
GRANT vendas TO marcia;
```

Administração de usuários - Grupos

Exemplos

/ Removendo um role de login de um role contâinter */*

REVOKE financeiro FROM clara;

/ Usando o comando "GRANT" e o parâmetro "WITH ADMIN OPTION“ permite conceder o direito de administração do role a outro role */*

GRANT vendas TO clara WITH ADMIN OPTION;

/ Inversamente podemos revogar tal direito */*

REVOKE ADMIN OPTION FOR vendas FROM clara;

Concedendo ou Revogando acesso a objetos

- Os tipos de privilégios que podem ser concedidos são:

  - Select
  - Insert
  - Update
  - Delete
  - References
  - Trigger
  - Create
  - Connect
  - Execute
  - All Privileges

Administração de usuários
Privilégios: exemplo

```
/* Role container */
CREATE ROLE masters SUPERUSER;

/* Usuário simples */
CREATE ROLE newusr NOSUPERUSER LOGIN ENCRYPTED
PASSWORD '123';

/* Tabelas para teste */
CREATE TEMP TABLE tb_acesso1(valor int);
CREATE TEMP TABLE tb_acesso2(valor int);
CREATE TEMP TABLE tb_acesso3(valor int);
CREATE TEMP TABLE tb_acesso4(valor int);
CREATE TEMP TABLE tb_acesso5(valor int);
```

Administração de usuários
Privilégios: exemplo

```
/* Conceder direito de INSERT em 2 tabelas */
GRANT INSERT ON tb_acesso1,tb_acesso2 TO masters;

/* Conceder direito de SELECT para qualquer um */
GRANT SELECT ON tb_acesso1 TO PUBLIC;
GRANT masters TO newusr;

/* Revogando todos os privilegios de todos os usuários em 3 tabelas */
REVOKE ALL ON tb_acesso2,tb_acesso3,tb_acesso4 FROM
PUBLIC;

/* Revogando o direito de inserir na tabela tb_acesso5 para os roles
newusr e masters */
REVOKE INSERT ON tb_acesso5 FROM newusr,masters;
```

Bibliografia Utilizada nesta aula

- ELMASRI, R.; NAVATHE, S. B. Sistemas de banco de dados. 6 ed. Pearson/Addison-Wesley, 2011. ISBN: 9788579360855
- Silberschatz, A., Korth, H., Sudarshan, S. **Sistema de Banco de Dados**. 5ª Edição, Editora Campus, 2006.
- Ramakrishnan, R. **Sistemas de Gerenciamento de Banco de Dados**, 3ª Edição, McGraw-Hill, 2008.