
Instituto Federal de Educação, Ciência e
Tecnologia de São Paulo campus Votuporanga

# Apostila da disciplina Administração de Banco de Dados (ABD)

*Prof. Dr. Evandro Jardini*
*eajardini@ifsp.edu.br*

Esta apostila possui conteúdo extraído de diversos materiais como livros, apostilas, notas, artigos, etc elencados nas de Referências Bibliografias.

# Sumário

# Capítulo 1

# Regras de Negócio e Programação de SGBD

## 1.1 Introdução

Nesse capítulo, iremos trabalhar com recurso utilizados tanto pelos administradores quanto pelos programadores.

Ao longo desse material, serão abordados os seguintes assuntos:

- Implementação de Regras de Negócio no momento de criação das tabelas (*Check*);
- Sequências (*Sequences*);
- Visões (*View*);

- Procedimentos Armazenados (*Stored Procedures*);
- Gatilhos (*Triggers*).

## 1.2 IMPLEMENTANDO REGRAS DE NEGÓCIO

As regras de negócio (regras aos quais os valores dos dados devem obedecer) podem serem implementadas no momento da criação das tabelas por meio das restrições ***check*** e ***unique***.

### 1.2.1 Restrição *Check*

**Exemplo:** Um empréstimos só pode ser realizado se for maior do que 100,00

*create table emprestimo(*
*nome_age_emp varchar(15) not null,*
*numero_emp varchar(10) not null,*
*valor_emp numeric(10,2),*
*constraint pk_emprestimo primary key (numero_emp),*
***constraint ck_valor check (valor_emp > 100));***

**Exemplo:** Para inserir um cliente, o mesmo deve ser dos estados de São Paulo ou Minas Gerais.

create table cliente(

codigo numeric(5),
nome varchar(40),
estado char(2),
**constraint ck_estado check (estado in ('SP', 'MG'))**);

### 1.2.2 Restrição *Unique*

Para garantir a unicidade de valores de campos que não são chave primária, no caso chaves candidatas, usamos a restrição **unique**.

**Exemplo:** Na implementação da tabela *Aluno*, a chave primária deve ser *RA* e o campo *CPF* deve ser único:

*create table aluno(*
*ra integer,*
*nome varchar(40),*
*cpf varchar(12),*
*constraint pk_aluno primary key (ra),*
*constraint un_cpf unique (cpf));*

### 1.2.3 Exercício

1. Crie o modelo físico das relações *correntista = {cpf, nome, data_nasc, cidade, uf}* e *conta_corrente {num_conta, cpf_correntista (fk), saldo}*. Garanta as seguintes regras de negócio:

(a) Uma conta corrente só pode ser aberta com saldo mínimo inicial de R$ 500,00.

(b) Os correntistas devem ser maiores que 18 anos. Para isso, você deve comparar a data de nascimento com a data atual. No Postgres, para saber a idade atual, use a função ((CURRENT_DATE - data_nasc)/365>=18) ou use a função (AGE(CURRENT_DATE, data_nasc) >= '18 Y')).

## 1.3 SEQUÊNCIAS (*SEQUENCES*)

Uma sequência (*sequence*) é um objeto de banco de dados criado pelo usuário que pode ser compartilhado por vários usuários para gerar números inteiros exclusivos de acordo com regras especificadas no momento que a sequência é criada.

A sequência é gerada e incrementada (ou decrementada) por uma rotina interna do SGBD. Normalmente, as sequências são usadas para criar um valor de chave primária que deve ser exclusivo para cada linha de uma tabela.

Vale a pena salientar que os números de sequências são armazenados e gerados de modo independente das tabelas. Portanto, o mesmo objeto sequência pode ser usado por várias tabelas e inclusive por vários usuários de banco de dados caso necessário.

Geralmente, convém atribuir à sequência um nome de acordo com o uso a que se destina; no entanto, ela poderá ser utilizada em qualquer lugar, independente do nome.

Sequências são frequentemente utilizados para produzir valores únicos em colunas definidas como **chaves primárias**.

Neste caso, você pode enxergar essas sequências como campos do tipo "*auto-incremento*".

Cada sequência deve ter um nome que a identifique. O padrão para o nome pode ser "sid_nome_da_tabela".

Por exemplo: A tabela *cliente* pode ter uma sequência de nome *sid_cliente*.

### 1.3.1 Modelo Relacional para uso de *Sequências*

O modelo relacional para os exemplos de sequência é mostrado abaixo.

seq_funcionario = {id_func, cpf, nome, ender, cidade, salario}
seq_salario_registro = {id_salreg, id_func(FK), salario, data_aumento}

Têm-se as tabelas de funcionário e registros de salários. Nesta segunda tabela, ficam armazenados os aumentos recebidos pelo funcionários.

Deverá ser criada uma sequência para os campos: *funcionario.id_func, salario_registro.id*_salreg.

Pergunta para a sala: **Porque não foi criado uma sequência para *seq_salario_registro.id*_func?**

### 1.3.2 Criação das Sequências

**Exemplo:** Crie uma sequência para a chave primária de funcionários

*create sequence sid_func;*

Esta *sequence* é usada para atribuir números sequenciais à chave primária da tabela *funcionarios*.

### 1.3.3 Uso das Sequências

Após criada a sequência, seu uso se dá, principalmente, por dois comandos

#### 1.3.3.1 *NEXTVAL*

Para você usar uma sequencia, deve chamar a função nextval.

**Exemplo:** inserindo um novo funcionário:

```
insert into seq_funcionario
  values (nextval('sid_func'),1312, 'João da Silva' , 'Rua A',
  'Votuporanga' , 2500);
```

#### 1.3.3.2 *CURRVAL*

Traz o valor atual da sequence.

**Exemplo:** para saber qual é o valor atual da sequência:

```
select currval('sid_func');
```

Entretanto, para o *currval* funcionar, é necessário que um *nextval* seja executado antes.

#### 1.3.3.3 *Exemplos de inserção dados em tabelas com uso de sequências*

***Exemplo 1*** *- Inserindo na tabela funcionário (seq_func)*

*insert into seq_funcionario*
*values (nextval('sid_func'), '121212', 'Paulo Afonso', 'Rua A',*
*'Votuporanga', 9787.86);*

***Exemplo 2** - Inserindo na tabela de aumento de salário (seq_salario_registro)*

*create sequence sid_salreg;*

*insert into seq_salario_registro*
*values (nextval('sid_salreg'), (select id_func from seq_funcionario*
*where cpf = '12121212'), 9898.98, current_date)*

### 1.3.4 Exercício de Sequências

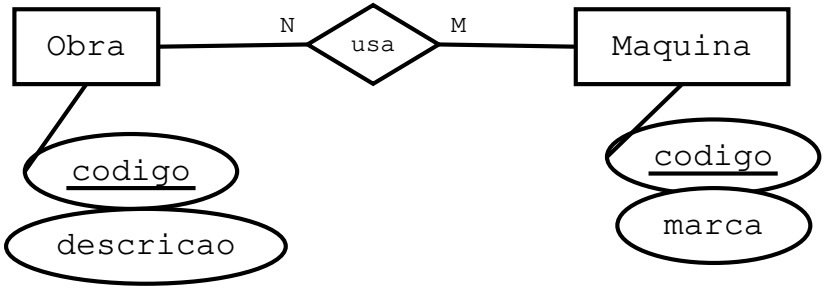


Figura 1.1: DER sequências

Considere o seguinte modelo relacional baseado no DER da figura 1.1:

***Algoritmo 1.1*** Esquema relacional para uso de Sequences

*obra={id_obra, codigo (unique), descricao}*
*maquina = {id_maquina, codigo(unique), marca}*
*usa = {id_usa, id_obra,id_maquina, data_do_uso}*

1. Crie sequências *obra, maquina e usa.*
2. Insira duas obras e duas máquinas usando as sequência criadas.
3. Atribua para cada obra as duas máquinas.

## 1.4 OPERAÇÕES COM CONJUNTOS EM SQL

A Álgebra Relacional é uma linguagem de consulta procedural (o usuário descreve os passos a serem executados) e formal a qual a técnica utilizada é fundamental para a extração de dados de um banco de dados, além de ser um conjunto de operações, os quais utilizam como recurso de entrada uma ou mais relações, produzindo então, uma nova relação [Borello e Kneipp].

As principais operações da Álgebra Relacional são Seleção, Projeção, União, Diferença, Intesecção, Produto Cartesiano, Junção e Divisão.

As operações da linguagem SQL são baseadas nas operação da Álgebra Relacional. Foi visto na disciplina anterior as operações Projeção (comando *Select*) , Seleção (cláusula *Where*), Junção (operação *Inner Join*) e Produto Cartesiano (cláusula *FROM* sem a cláusula *Where*).

Agora, iremos aprender em SQL as operações União, Interseção e Diferença. Para isso, usaremos alguns exemplos e a base de dados disponíveis em [Passos]. Mais detalhes entre Álgebra Relacional e SQL pode ser encontrado em [Guimarães].

**Importante!**
Assim com as operações da Álgebra Relacional, as operações sobre conjuntos com os comandos SQL exigem Compatibilidade de Domínio, ou seja, campo texto embaixo de campo texto, campo numérico embaixo de campo numérico.

### 1.4.1 Operação União (*Union* e *Unial All*)

**A** união de duas tabelas é formada pela adição dos registros de uma tabela aos registros de uma segunda tabela, para produzir uma terceira. Assim, o operador *union* serve para juntar ou unir dois comandos *selects*, um abaixo do outro. As linhas repetidas são ignoradas.

**Exemplo 1**: Monte um relatório com os nomes dos instrutores e alunos cadastrados no banco de dados. Garanta que os nomes repetidos sejam eliminados.

```
SELECT inst_nome as Nome FROM instrutor
UNION
SELECT alu_nome as Nome FROM aluno;
```

**Exemplo 2**: Refaça o mesmo relatório, porém, agora, não eliminando os nomes repetidos.

```
SELECT inst_nome as Nome FROM instrutor
UNION ALL
SELECT alu_nome as Nome FROM aluno;
```

### 1.4.2 Operação Interseção (comando *Intersect*)

Esta operação atua sobre duas tabelas compatíveis em domínio e produz uma terceira contendo os registros que aparecem simultaneamente em ambas tabelas. O operador *In* redunda no mesmo resultado do operador *Union All*. Entretanto, aquele não necessita da compatibilidade de domínio.

**Exemplo 1**: Desenvolva uma consulta que preencha uma página html com os nomes homônimos de professores e alunos.

```
select inst_nome as nome from instrutor
INTERSECT
select alu_nome as nome from aluno;
```

**Exemplo 2**: A *Grid* de um *Form* de uma aplicação bancária *desktop* deve ser preenchida com os dados de uma consulta que traga os códigos do cliente que possuem conta (tabela *Depositante*) **e** também empréstimo (tabela *Devedor*). Use o operador *Intersect*.

*select cod_cli_dep from depositante*
*INTERSECT*
*select cod_cli_dev from devedor;*

### 1.4.3 Operação Diferença (comando *Except*)

A diferença de duas tabelas é uma terceira tabela contendo os registros que ocorrem na primeira tabela mas não ocorrem na segunda. O operador *Not In* redunda no mesmo resultado do operador *Except*. Entretanto, aquele não necessita da compatibilidade de domínio.

**Exemplo 1**: Monte um relatório que traga o código do cliente que possui conta (*depositante*) mas que não possui empréstimo (*devedor*).

*select cod_cli_dep from depositante*
*EXCEPT*
*select cod_cli_dev from devedor;*

**Exemplo 2**: Monte a consulta em SQL para um relatório que traga os nomes dos instrutores que não são **homônimos** dos alunos (usando o *Except*).

*select inst_nome as nome from instrutor*
*EXCEPT*
*select alu_nome as nome from aluno;*

### 1.4.4 Exercícios

1. Monte uma consulta SQL para trazer os código e nomes dos clientes (tabela cliente) e vendedores (vendedor). Utilize o operador *UNION*.

2. Desenvolva uma consulta SQL que traga a descrição dos produtos que estão inseridos tanto na tabela produto quanto na tabela item_pedido. Utilize o operador *INTERSECT*.

## 1.5 *VIEW* (VISÃO)

Do ponto de vista do negócio, visões são elementos estratégicos que normalmente limitam o poder de acesso a informações. Do lado técnico, uma visão é uma tabela virtual resultante de uma consulta efetuada sobre uma ou mais tabelas. A visão é baseada em uma ou mais tabelas ou outra *view*, logo uma view não contém dados próprios mas sim dados provenientes de outas tabelas. Quando se aplica o comando *SELECT* em uma visão, o que o SGBD faz é executar o *SELECT* da própria visão.

As visões podem ser usadas em:

- Substituir consultas longas e complexas por algo fácil de ser entendido e manipulado.

- Elementos de segurança, pois a partir do momento em que conseguem limitar o acesso dos usuários a determinados grupos de informações no BD.

### 1.5.1 DER e Modelo Relacional para o uso de *View*

O *script* da base de dados está dentro do arquivo *criabaseBD_Postgres.sql*.

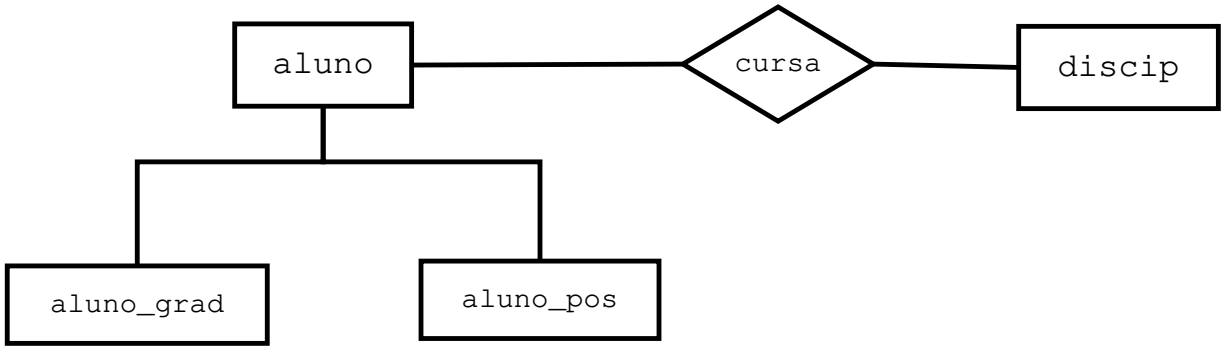


Figura 1.2: DER exemplo para o uso de *Views*

**Algoritmo 1.2** Esquema relacional para criação de *Views*

*alunov = {id, ra, nome, ender, cidade}*
*aluno_grad = {id, ano_curso}*
*aluno_pos = {id, orientador}*
*cursa = {id, discip, nota1, nota2, nota3, nota4}*
*discip = (id, codigo, descricao}*

### 1.5.2 Criando e Modificando uma Visão

**Sintaxe:** Para criar uma visão executamos o seguinte comando:

```
create [or replace] view
as
<<subconsulta>>
```

**Exemplo 1:** Desejamos criar uma visão em que aparece somente os alunos de Votuporanga:

```
create view v_aluno_votuporanga
as
select *
from alunov
where cidade = 'Votuporanga';

Consultando
select * from v_aluno_votuporanga;
```

**Exemplo 2:** Monte um consulta SQL para o relatório que traga o nome do cliente e a quantidade de pedido que o mesmo realizou ordenado pelo o cliente que fez mais pedido para o que fez menos:

```
create view v_cliente_pedido
as
select nome_cliente, count(num_pedido)
from cliente cli, pedido ped
where cli.codigo_cliente = ped.codigo_cliente
```

*group by 1*
*order by 2 desc;*

### 1.5.3 Removendo uma View

Para remover uma view, utilize o comando drop view da seguinte maneira:

*DROP VIEW v_aluno_votuporanga;*

**Exemplo 1:** Crie uma visão para um relatório que informe o ***ra, nome e o ano*** dos alunos de graduação:

*create view v_aluno_grad*
*as*
*select ra, nome, ano_curso*
*from alunov alu* ***inner join*** *aluno_grad alugrad*
*on (alu.id = alugrad.id);*

**Exemplo 2:** Crie uma visão que informe os nomes dos alunos de pós-graduação e os nomes de seus respectivos orientadores.

*create view v_aluno_pos*
*as*
*select nome, orientador*
*from alunov Alu, aluno_pos alupos*
*where alu.id = alupos.id;*

**Exemplo 3:** Crie uma visão para um relatório que informe o nome dos alunos; se o aluno for de graduação, informe o ano; se for de pós, informe seu orientador.

```
create view v_rel_aluno

    as
    select nome, ano_curso, orientador
    from aluno alu left outer join aluno_grad alugrad
                on (alu.id = alugrad.id)
                    left outer join aluno_pos alupos
                on (alu.id = alupos.id) ;
```

### 1.5.4 Regras para execução das operações DML em uma *View*

As visões podem ser somente de leitura ou atualizáveis.

Não será possível modificar os dados em uma visão se ela contiver:

- Funções de grupo (sum, count, etc)
- Uma cláusula GROUP BY
- A palavra-chave DISTINCT
- Todos os campos obrigatórios (*not null*) da tabela base devem está presentes na visão.

#### 1.5.4.1 Inserindo em uma visão

Vamos criar uma *view* sobre a tabela cliente com os campos *nome_cliente, endereco e cidade*:

```
create or replace view v_dados_cliente
as
select nome_cliente, endereco, cidade
from cliente;
```

Tente fazer

```
insert into v_dados_cliente
values ('Francisco Silva', 'Rua das Araras', 'Votuporanga');
```

Perceba que houve erro, pois o código_cliente não estava presente na *view*. Apague a *view* (**DROP TABLE**) e recrie-a adicionando esse campo e tente inserir o cliente com o código 3210.

### 1.5.5 Exercícios

1. De acordo com o DER da figura 1.1, desenvolva as seguintes visões:

   (a) Uma visão que mostre a descrição da obra, a máquina utilizada e a data do uso. Ordene pela descrição da obra.

   (b) Uma visão que mostre a descrição da obra e a quantidade de máquinas utilizadas.

## 1.6 *FUNÇÕES (FUNCTIONS) OU PROCEDIMENTOS ARMAZENADOS (STORED PROCEDURES)*

Quando uma aplicação solicita a execução de uma *query* SQL comum, todo o texto da mesma é enviado pela rede do computador cliente ao servidor em que será compilado e executado.

Isso gera certa demora na resposta da *query*.

Para aumentar o desempenho em relação *queries,* os Sistemas Gerenciadores de Banco de Dados (SGBDs) - entre eles o Oracle, Postgres, SqlServer, etc, oferecem a capacidade de funções (*functions*) ou os procedimentos armazenados (*Stored Procedures*) como parte dos seus metadados,

Funções podem ser entendidos como uma sequência de comandos SQLs agrupados, que são executados dentro do SGBD.

O Postgres trabalha com Funções (*Functions*) ao invés de procedimentos. As funções são blocos PL/pgSQL nomeado que pode aceitar parâmetros (conhecidos como argumentos).

As Funções são utilizadas para executar uma ação. Elas contêm um cabeçalho, uma seção declarativa, uma seção executável e uma seção opcional de tratamento de exceções.

A função é chamada quando o seu nome é utilizado ou no comando SELECT ou na seção executável de outro bloco PL/pgSQL .

As *Functions* são compiladas e armazenados no banco de dados como objetos de esquema. Elas promovem a capacidade de manutenção e reutilização. Quando validados, elas podem ser usadas em várias aplicações.

Vantagens:

- Podem ser criadas rotinas especializadas altamente reutilizáveis, o que torna extremamente produtivo em

ambientes do tipo *cliente/servidor*.

- As rotinas rodam no servidor, liberando está carga do cliente.

**Desvantagem:**

- Ficar restrito a sintaxe de um SGBD específico.

### 1.6.1 Criação de uma *Function*

Pode usar a instrução SQL CREATE OR REPLACE FUNCTION para criar *functions* que são armazenados em um banco de dados Postgres.

Uma *FUNCTION* é similar a uma miniatura de programa: ela executa uma ação específica. Especifica-se o nome da função, seus parâmetros, suas variáveis locais e o bloco BEGIN-END que contém seu código e trata qualquer exceção.

- Os blocos PL/pgSQL começam com uma instrução BEGIN, podendo ser precedidos da declaração de variáveis locais, e terminam com uma instrução END ou END
- Nome do *function*. A opção REPLACE indica que, se a *function* existir, ela será eliminado e substituído pela nova versão criada pela instrução.
  - A opção REPLACE não elimina nenhum dos privilégios associados à função.
- *Parâmetro1* representa o nome de um parâmetro.

- *Tipodedados*1 especifica o tipo de dados do parâmetro, sem nenhuma precisão.

**Observação**: Os parâmetros podem ser considerados como variáveis locais.

Abaixo, um modelo de código de uma Função:

*CREATE [OR REPLACE] FUNCTION NomeFunção [(arg1 tipo_dado,..., argN tipo_dado)] RETURNS Void | tipo_dado*
*AS*
[ DECLARE *variável tipo_dado*]
*$$*
*BEGIN*
*RETURN null | tipo_dado;*
END;
$$
LANGUAGE plpgsql;

**Exemplo 1:** Primeira função que mostra a frase: Olá mundo

*create or replace function f_olamundo() returns text as*
*$$*
*begin*
*- - Função que mostra a frase Olá Mundo!;*

```
 return ' Olá Mundo!' ;
end;
$$
language PLPGSQL;
```

Para executar: select f_olamundo();

### 1.6.2 Acessando os Argumentos pelo Nome

Acessar o parâmetro pelo nome, faz o código da função mais legível. A seguir um exemplo de como acessar o parâmetro pelo nome:

**Exemplo 1:** Retornado as posições de 2 a 5 do primeiro valores dos parâmetros de entrada, mas agora acessando o parâmetros pelo nome:

```
CREATE OR REPLACE FUNCTION f_substringPorNome(nomePar varchar, posicaoInicialPar
integer) RETURNS varchar
    AS
    $$
    BEGIN
     RETURN substring(nomePar,posicaoInicialPar);
    END;
```

*$$*
*LANGUAGEplpgsql;*
uso: *select f_substringPorNome('Borodin', 2);*

### 1.6.3 Tipos de Dados de Variáveis

As variáveis podem ter os seguintes tipos de dados:

- *Boolean*: recebe os valores True,False ou Null
- Integer: recebe valores inteiros.
- *Num*eric: recebe valores numéricos, tanto inteiros como decimais.
- *Varchar*: recebe valores alfanuméricos.
- *Date*: recebe valores do tipo data.
- *%type*: atribui à variável que está sendo criada os mesmos tipos de dados usados pela coluna que está sendo usada. Por exemplo, seu a variável *codcli* for declarada assim *codcli cliente.codigocliente%type*, ela terá o mesmo tipo de dados do camo *codigocliente* da tabela *cliente*.
- *%rowtype*: declara uma variável composta pelos campos de um registro de uma tabela. Exemplo, *regcliente cliente%rowtype*. A variável *regcliente* terá todos os campos da tabela *cliente*.

**Exemplo 1:** Função que some os três parâmetros passados a ela:

*CREATE OR REPLACE FUNCTION f_SomaTresPar(Valor1 numeric, Valor2 integer, Valor3 Numeric) RETURNS numeric*
*AS*
*$$*
*DECLARE*
*Resultado numeric;*
*BEGIN*
*resultado = Valor1+Valor2+Valor3;*
*RETURN resultado;*
*END;*
*$$*
*LANGUAGE plpgsql;*
*Uso: select f_SomaTresPar(2.2,4,6.3);*

### 1.6.4 Estrutura de Controle de Fluxo de Dados

O comando IF permite a execução do fluxo de comandos baseados em certas condições. A sintaxe dele é:

*1 - IF <condição> THEN*
*<comandos>*
*END IF;*

*2 - IF <condição> THEN*
*<comandos>*
*ELSE*
*<comandos>*
*END IF;*

*3 - IF <condição> THEN*
*<comandos>*
*ELSIF <condição> THEN*
*<comandos>*
*ELSE*
*<comandos>*
*END IF;*

Exemplo 1: Desenvolva uma função em que o usuário informe seu sexo por meio de parâmetro. Caso for M retorne ”Masculino”, se for F retorne ”Feminino”. Senão retorne ”Indefinido”:

```
create or replace function f_definicao (sexo char) returns text
    as
    $$
    begin
      if (sexo='M' or sexo='m') then
          return 'Masculino';
```

```
  elsif (sexo='F' or sexo='f') then
      return 'Feminino';
  else
      return 'Indefinido';
  end if;
end;
$$
language plpgsql;
uso: select f_definicao('M');
```

### 1.6.5 Consultas simples com o comando SELECT ... INTO

O comando SELECT ... INTO possibilita que usemos valores recuperados das tabelas do banco de dados dentro das funções. Desta forma, muitas das rotinas que são desenvolvidas nas linguagens de programação e que acessam muitos dados podem ser convertidas para dentro do SGBD. A sintaxe desse comando é:

```
select campo1, campo2,... ,campoN into var1, var2,... , varN
[from tabela]
```

**Exemplo 1:** Projete uma função que receba dois números como parâmetro e devolva a soma deles. Realize a soma com o comando *select*.

```
CREATE OR REPLACE FUNCTION f_SomaSelect (num1 numeric, num2 numeric) RETURNS
numeric
```

```
AS
$$
DECLARE retval numeric;
BEGIN
  SELECT num1 + num2 INTO retval;
  RETURN retval;
END;
$$ LANGUAGE plpgsql;
```

**Exemplo 2:** Desenvolva uma função que receba o código do cliente como parâmetro e devolva o nome e o endereço concatenados.

```
CREATE OR REPLACE FUNCTION f_Nome_Endereco (codcliente integer) RETURNS text
AS $$
DECLARE nomecli varchar;
                enderecocli varchar;
BEGIN
   SELECT nome_cliente, endereco INTO nomecli, enderecocli
   FROM cliente
   WHERE codigo_cliente = codcliente;
RETURN nomecli || ' - ' || enderecocli ;
END; $$
```

LANGUAGE plpgsql;
uso: select f_Nome_Endereco (720)

### 1.6.6 Exercício

1. Implemente um procedimento que receba 4 parâmetros. Os dois primeiros serão números que sofrerão uma das quatro operações básicas da matemática adição, subtração, multiplicação e divisão; o terceiro parâmetro será uma variável que armazenará o resultado da operação e por fim, o quarto parâmetro indicará qual será a operação realizada. Após implementar, teste o procedimento e veja se está funcionando corretamente.

2. Projete uma função que informado o código do cliente por parâmetro, encontre o valor total das compras desse cliente. Como retorno, a função deve informar o nome do cliente concatenado com o valor da compra. Você deverá usar as tabelas cliente, pedido, item_pedido e produto.

### 1.6.7 Usando Comandos DML em Funções

As funções permite-nos usar comandos do tipo DML (Insert, Update e Delete) para manipulação de dados. A vantagem de usarmos comandos DML nas funções, é que podemos diminuir ainda mais o tráfego de dados pela rede, visto que você pode lêr dados de uma tabela e inseri-lo em outras sem a necessidade desses dados fazerem acesso ao meio de comunicação.

**Exemplo 1:** Implemente uma função que receba os valores por parâmetro e os insira na tabela de funcionários (seq_func). Como a chave primária da referida tabela é um ID, utilize a sequência criada na seção 1.3.2.

Repare no código a seguir o comando RETURNING ... INTO ... usado junto com o comando Insert. Ele possibilita que uma variável - no caso do exemplo, a variável resultado - receba o valor de um campo inserido. Isso possibilita saber se houve êxito ou não durante a operação. Execute o código abaixo:

*create or replace function f_InsereFuncionario (cpf varchar, nome varchar, endereco varchar, cidade varchar, salario numeric)*

```
returns Integer
AS
$$
Declare
    resultado integer;
Begin
     insert into seq_func (func_id, func_cpf, func_nome, func_ender, func_cidade, func_salario)
          values (nextval('sid_func'), cpf, nome, endereco, cidade, salario)
        RETURNING func_id INTO resultado;
     return resultado;
end;
$$
language plpgsql;
uso: select f_InsereFuncionario ('5221', 'Paulo Afonso', 'Rua das Acácias', 'Votuporanga', 9811)
```

### 1.6.8 Retornando Registros

É possível para as funções fazer retorno de registros de tabelas. O retorno pode ser de um único registro quanto de um conjunto. Assim, as funções trabalhariam como se fossem um comando *select* ou uma visão. O tipo de dado de retorno de **um registro** deve ser *record.* Para o retorno de vários de ser *setof record*.

1. Projete uma função que passado o código do cliente, retorne as informações nome, endereco, cidade, uf e cep em forma de registro. Implemente na função o controle, por meio de *Raise*, de cliente não encontrado:
   *create or replace function f_EncontraCliente (cod_clientepar cliente.codigo_cliente%type)* ***returns record***
   *as $$*
   *declare regcli RECORD;*
   *begin*
   *select nome_cliente, endereco, cidade, uf, cep into regcli*
   *from cliente where codigo_cliente = cod_clientepar;*
   *if not found then*
   *raise 'O cliente de código % não foi encontrado' , cod_clientepar using ERRCODE = 'ERR01';*
   *end if;*
   *return regcli;*
   *end;*
   *$$ language plpgsql;*
   *uso: select *from f_EncontraCliente(720) as (nome_cliente varchar, endereco varchar, cidade varchar, uf char(2), cep varchar);*

2. Desenvolva uma função para atualizar os valores dos produtos. A atualização será seletiva. Para produtos cuja unidade seja 'G', terão reajuste de 8%; para produtos cuja unidade seja 'M', terão reajuste negativo de 5%; as demais unidades não sofrerão reajuste. A função não terá parâmetro de entrada. O retorno dela serão os códigos do produtos, as descrições, unidade, os valores antigos e os valores novos.

```
create or replace function f_atualizaproduto () returns setof record
as $$
declare regprod record;
begin
for regprod in
select codigo_produto, descricao, unidade, val_unit "valor_antigo", val_unit "valor_novo"
from produto
loop
    if (regprod.unidade = 'G') then
        regprod.valor_novo = regprod.valor_novo*1.08;
        update produto
            set val_unit = regprod.valor_novo
            where codigo_produto = regprod.codigo_produto;
            return next regprod;
    elsif (regprod.unidade = 'M') then
        regprod.valor_novo = regprod.valor_novo*0.95;
        update produto
```

```
            set val_unit = regprod.valor_novo
            where codigo_produto = regprod.codigo_produto;
            return next regprod;
        end if;
    end loop;
    return;
    end;
    $$
    language plpgsql;
    uso: select * from f_atualizaproduto() as (codigo_produto numeric, descricao varchar, unidade
char(3), valor_antigo numeric, valor_novo numeric)
```

#### 1.6.8.1 Exercícios

1. Implemente uma função para calcular a média de uma aluno nas disciplinas cursadas. As tabelas usadas serão *Alunos, Disciplinas e Turmas_Matriculadas*. A função deverá seguir os seguintes requisitos:

   (a) O parâmetro de entrada deverá ser a matricula do aluno (campo *alunos.mat_alu*)

   (b) Os parâmetros de retorno e que deverão ser mostradas na tela são *alunos.nom_alu, disciplinas.nom_disc, turmas_matriculadas.ano, turmas_matriculadas.semestre, turmas_matriculadas.nota_1, turmas_matriculadas.n turmas_matriculadas.nota_3, turmas_matriculadas.nota_4, Media e turmas_matriculadas.cod_disc*;

(c) Entretanto, houve um erro na disciplina de código 500110 (ALGEBRA LINEAR I) e somente para essa disciplina, a média deve ser 0;

(d) Garanta que se uma matrícula de aluno não existir deverá ser gerado um erro com o comando *Raise*.

(e) Faça o teste com os alunos de matrícula número 915547 e 914830.

## 1.7 *TRIGGERS* (GATILHOS)

*Triggers* são rotinas disparadas de forma automática de acordo com determinados eventos. Assim, quando ocorre um evento que possui um *trigger* nele configurado, esse trigger é disparado - executado - de forma automática. Não é necessário fazer chamada direta do *trigger*.

A vantagem do uso do *trigger* é a automatização de certas operações quando determinadas situações ocorrem no banco de dados, como, por exemplo, um produto esgotando no estoque, um valor estimado ser alcançado no orçamento, etc. Uma aplicação poderia ser responsável por esse tipo de acompanhamento, mas teria que ser programada para executar de tempos em tempos.

É possível definir mais de um *trigger* para uma determinada condição. Neste caso, o Postgres utiliza a ordem alfabética dos nomes dos *triggers* para organizar uma fila de execução.

### 1.7.1 Eventos que disparam um Gatilho

Os eventos que podem disparar um *trigger* estão associados aos comandos do tipo DML: *Insert*, *Update* e *Delete*. O *trigger* é executado em momentos antes (*Before*) e depois (*After*) de um comando SQL. A lista de eventos são:

| Modo | Descrição |
|---|---|
| BEFORE INSERT | O trigger é disparado antes de uma ação de inserção |
| BEFORE UPDATE | O trigger é disparado antes de uma ação de alteração |
| BEFORE DELETE | O trigger é disparado antes de uma ação de exclusão |
| AFTER INSERT | O trigger é disparado depois de uma ação de inserção |
| AFTER UPDATE | O trigger é disparado depois de uma ação de alteração |
| AFTER DELETE | O trigger é disparado depois de uma ação de exclusão |

É possível ainda combinar alguns dos modos, desde que tenham a operação AFTER ou BEFORE em comum, assim mesclando duas ou mais opções. Veja o exemplo a seguir:

| Modo | Descrição |
|---|---|
| BEFORE INSERT OR UPDATE | O trigger é disparado antes de uma ação de inserção ou alteração. |

### 1.7.2 Acesso aos valores dos campos do Gatilho

A forma de acessar os valores dos campos que estão sendo processados é feita por meio dos identificadores:

- OLD: indica o valor corrente de uma coluna em operações que lidam com as instruções DELETE e UPDATE.
- NEW: refere-se ao novo valor da coluna nas operações INSERT e UPDATE.
- Veja no quadro abaixo, as variáveis e os comandos que podem ser usadas.

| **Identificador** | ***Insert*** | ***Update*** | ***Delete*** |
|---|---|---|---|
| *OLD* | Não | Sim | Sim |
| *NEW* | Sim | Sim | Não |

Abaixo, segue uma tabela comparativa dos identificadores [DevMedia].

| **DML** | ***OLD*** | ***NEW*** |
|---|---|---|
| *INSERT* | NULO | Valores Novos |
| *DELETE* | Valores antigos | NULO |
| *UPDATE* | Valores antigos | Valores Novos |

### 1.7.3 Criação de Gatilhos

A implementação de um gatilho é feita em uma função separada dele. Assim, para criarmos um trigger, primeiro deve ser criada uma função que retorna um tipo de dados *trigger* (***returns trigger***) e em seguida criarmos o gatilho propriamente dito.

Destarte, a sintaxe para criação de gatilhos é:

1. Criação da função que retorna o *trigger*:

   (a) *create or replace function nome_da_funcao (parâmteros)* ***returns triggers***
   *as $$*
   ...

***return null*** || ***new;***
*end;*
*$$ language plpgsql;*

2. Criação do *trigger*:

   (a) *create* ***trigger*** *nome_do_gatilho*
   *eventos_que_disparam_o_gatilho on tabela*
   *for tipo_execução*
   *execute procedure nome_da_funcao(parâmetros);*

Onde:

- *nome_do_gatilho*: é o nome que será atribuido ao *trigger*.
- *eventos_que_disparam_o_gatilho:* são os comandos DML que disparam o gatilho. São eles os comandos *insert, delete e update*.
- *tabela*: é a tabela do banco de dados a que o gatilho será configurado e disparado.
- *tipo_execução*: indica se o *trigger* deve ser executado uma vez por comando SQL ou deve ser executado para cada linha na tabela em questão.
  - *each statement*: dispara o gatilho uma única vez independente de quantas linhas forem alteradas pelo comando. Se nada for especificado, essa opção é utlizada.

- *each row*: dispara o gatilho para cada linha afetada pelos comandos DML.

- *Return Null || New*: dentro da função, você deverá indicar o retorno (*return*) sendo *Null* ou *New*. O primeiro, é usado quando o *trigger* é disparado depois (*after*) que o comando DML for executado. O *New* é utilizado quando o *trigger* é disparado antes (*before*) do comando DML ser executado e indica para o Postgres que quando terminar a execução da função, ele deve continuar a execução do comando DML.

#### 1.7.3.1 Exemplos de criação de triggers

**Exemplo 1:** Nesse exemplo, vamos permitir operações DML à tabela *conta* corrente no horário bancário. Das 10:00 às 15:00 horas.

```
create or replace function f_verifica_horario() returns trigger
as
$$
begin
  IF extract (hour from current_time) NOT BETWEEN 10 AND 15 THEN
        raise 'Operação não pode ser executada fora do horário bancário' using ERRCODE =
'EHO01';
  end if;
  return new;
end;
$$
```

*language plpgsql;*

*create trigger* ***trg_verifica_horario***
*before insert or update or delete*
*on conta for each row*
*execute procedure* ***f_verifica_horario();***

Execute o código: *insert into conta values (3, 'A-120', 600);*

Exemplo 2: Uma prática comum utilizada no processo de auditoria de sistemas é o registro das alterações realizadas nos salários dos funcionários. Dependendo do caso, é importante realizar o registro periódico de cada aumento ocorrido na remuneração de um empregado. Abaixo, segue o código de um *trigger* para registrar as alterações ocorridas na tabela de salário dos funcionários:

**1)** Inicialmente, cria-se as sequências *sid_func* para registro na tabela seq_funcionario e a sequencia sid_salreg para a tabela *seq_salario_registro*:

*create sequence sid_func;*
*create sequence sid_salreg;*

**2)** Criação da função do trigger que implementa a regra de negócio

*create or replace function* ***f_salario_registro() returns trigger***

*as $$*

```
begin
   insert into seq_salario_registro
        values (nextval('sid_salreg'), new.func_id, new.func_salario,current_date);
   return null;
end;
$$ language plpgsql;
```

**3)** Criação do trigger

*create trigger* ***tr_salario_registro***
*after insert or update*
*on seq_funcionario for each row*
*execute procedure* ***f_salario_registro()***

Para executar, insira ou altere um funcionário

```
insert into seq_funcionario values (2, '321', 'Pedro da Silva', 'Rua A', 'Votuporanga', 4000);
update seq_funcionario set func_salario = 6000;
```

### 1.7.4 Removendo um Gatilho

Para remover um trigger, usa-se o comando Drop Trigger da seguinte maneira:

```
Drop trigger [ IF EXISTS ] nome_trigger on tabela;
```

Para apagarmos o gatilho anterior, executemos o seguinte comando:

*Drop trigger if exists* ***tr_salario_registro*** *on seq_funcionario;*

### 1.7.5 Exercícios

1. Desenvolva um *trigger* que evite a venda de um produto cujo estoque seja menor que a quantidade vendida. Porém, caso haja estoque, deverá ser dado baixa no item no estoque. O *trigger* deverá ser criado sob a tabela *item_pedido*. Toda vez que um registro for inserido nela, antes da inserção (*before*), o *trigger* deverá verificar se existe estoque suficiente na tabela *produto*. Você deverá criar uma variável na função que receberá a quantidade atual em estoque (tabela produto). Em seguinda, deverá ser comparada a quantidade a ser vendida com a quantidade em estoque. Caso aquela seja menor ou igual a quantidade em estoque, será efetuada a baixa no estoque, caso contrário será gerado um erro com o comando *Raise* impossibilitando a operação.

2. (ENADE) Em um Banco de Dados PostgreSQL, Joana precisa criar uma trigger para inserir dados na tabela de auditoria chamada AGENTE_AUDIT todas as vezes que um registro da tabela AGENTE for efetivamente excluído. Para isso, considerando que a função "agente_removido()" já esteja implementada, Joana utilizará o comando:

   (a) CREATE TRIGGER audit_agente AFTER DELETE ON agente_audit FOR EACH STATEMENT EXECUTE PROCEDURE agente_removido();

(b) CREATE TRIGGER audit_agente AFTER EXCLUDE ON agente FOR EACH ROW EXECUTE PROCEDURE agente_removido();

(c) CREATE EVENT TRIGGER audit_agente AFTER DELETED ON agente FOR EACH ROW EXECUTE PROCEDURE agente_removido();

(d) CREATE TRIGGER audit_agente AFTER DELETE ON agente FOR EACH ROW EXECUTE PROCEDURE agente_removido();

(e) CREATE EVENT TRIGGER audit_agente AFTER DELETE ON agente_audit FOR EACH STATEMENT EXECUTE PROCEDURE agente_removido();

Gabarito: Questão 2 - Letra D

# Referências Bibliográficas

[Borello e Kneipp] Borello, F.; Kneipp, R. E. Álgebra Relacional. Sítio DevMedia. Acesso: http://www.devmedia.com.br/algebra-relacional/9229

[DevMedia] Criação de Triggers no Oracle. Sítio DevMedia. Acesso: http://www.devmedia.com.br/criacao-de-triggers-no-oracle/13039

[Guimarães] Guimarães, C. INF 325 - Álgebra Relacional x SQL: conexão prática. Acesso: http://www.ic.unicamp.br/~celio/inf325-2011/conexao-algebra-relacional-sql.html

[Passos] Passos, T. Exemplos práticos de conjuntos, utilizando União (UNION), Interseção (INTERSECT) e Diferença (EXCEPT) com o PostgreSQL. Acesso: http://blog.tiagopassos.com/2011/12/06/exemplos-praticos-de-conjuntos-utilizando-uniao-union-intersecao-intersect-e-diferenca-except-com-o-postgresql/

[Guedes] Guedes, G. B. Usando rules no PostgreSQL. Acesso: http://www.devmedia.com.br/usando-rules-no-postgresql/5940

# Capítulo 2

# Administrador de Banco de Dados (DBA)

## 2.1 INTRODUÇÃO

O trabalho do DBA contribui para a operação efetiva de todos os sistemas que rodam com o SGBD Postgres. O DBA dá suporte técnico para todos e espera-se que se torne fluente em todas as questões técnicas que aparecem no SGBD.

O DBA é responsável pelo seguinte:

- Dia-a-dia do banco de dados;
- Instalação e atualização do software;
- Otimização do desempenho;

- Estratégias de cópia e recuperação;
- Reunião com desenvolvedores.

## 2.2 USUÁRIO COM DIREITOS DE DBA NO POSTGRE

O Postgre possui um usuário que é automaticamente criado na instalação e possui direitos de administrador:

- Usuário *postgre:* É o proprietário do dicionário de dados do banco de dados.

## 2.3 FERRAMENTAS DE ADMINISTRAÇÃO (FA)

Gerenciar banco de dados (BD) não é tarefa fácil. Normalmente, o dia a dia de um BD ocorre sem grandes mudanças ou surpresas, porém, quando algo sai da rotina e o BD é afetado por uma queda ou grande lentidão, é necessário que o DBA utilize alguma ferramenta para poder fazer acesso ao SGBD e tentar garantir a normalidade do BD.

Em se tratando do Postgre, existe uma divisão no que refere-se às ferramentas de administração (FA):

- **Modo linha ou texto**: esse tipo de ferramenta é caracterizado pela ausência de interface gráfica. O *psql* é a FA do modo texto. A vantagem que se tem sobre este tipo de FA é sua presença certa em qualquer sistema operacional (SO) na qual o Postgre esteja instalado.

- **Modo gráfico**: uma ferramenta de administração deste tipo é o *PgAdmin*. Normalmente, também vem instalada na maioria dos SOs que o SGBD instalado.

Como já usamos o PgAdmin, vamos conhecer um pouco o psql.

### 2.3.1 Chamando o *psql*

Para chamarmos o *psql*, basta digitar no prompt do SO o comando:

```
psql -h localhost -U postgres
```

Após estar conectado ao *psql*, pode-se obter ajudar por meio dos comandos:

- \h :que mostra ajuda a respeito dos comando SQL.

  ```
  Exemplo: \h alter table;
  ```

- \? :que mostra ajuda a respeito dos comando do próprio *psql*. Conhecidos como comando do tipo "\".

#### 2.3.1.1 Realizando algumas tarefas no *psql* com os comando do tipo "\".

1. \l :trazer a lista dos *databases* existentes.

   ```
   \l
   ```

2. \c : conectar em um *database* específico.

```
\c aula
```

3. \i : importa um *script* no formato sql.

```
\i /home/aluno/adb/criabasedados.sql
```

4. \d : listar os objetos como tabelas, sequencias, etc do *database* em que está conectado.

```
\d
```

   (a) Variações:

```
\d tabela: descreve a estrutura de uma tabela.
\dt :lista somente os objetos do tipo tabela.
\dv : lista as visões do database.
\di : lista os índices do database.
\df : lista as funções desenvolvidas pelos programadores no database.
\sf nome_da_função: mostra a definição de uma função desenvolvida pelo programador.
\db nome_da_função: lista todas as tablespaces disponíveis.
```

Vimos alguns exemplos de como podemos utilizar o *psql* para realizar atividades do DBA dentro do SGBD Postgre. A seguir serão apresentadas algumas tarefas desempenhas por um DBA.

## 2.4 CONHECENDO A VERSÃO DO POSTGRE

Para sabendo a versão do SGBD que estamos trabalhando de dentro do *psql* digite:

*SELECT version();*

Na linha de comando do Sistema Operacional, digite:

*psql --version*

## 2.5 TEMPO EM QUE O SGBD ESTÁ NO AR

Para saber quanto tempo o SGBD Postgre está no ar, digite a consulta a seguir:

*SELECT date_trunc('second', current_timestamp - pg_postmaster_start_time()) as uptime;*

## 2.6 GERENCIAMENTO DE BANCO DE DADOS

Um banco de dados é um objeto do SGBD que armazena as tabelas, visões, gatilhos, sequences, etc. Na criação de um BD deve ser levado em conta a questão de armazenamento, desempenho, políticas de backup e restore, quantidade de acesso simultâneo, etc.

A seguir, serão apresentadas algumas operações que são feitas em cima de uma banco de dados:

### 2.6.1 Criando um Banco de Dados

Para se criar um novo banco de dados, utilizamos o comando

*CREATE DATABASE nome_do_banco_de_dados [parâmetros];*

em que os parâmetros podem ser:

- OWNER *usuário*: é possível informar qual usuário do servidor será responsável pelo banco de dados.
- ENCODING *valor*: esse argumento é responsável por indicar qual o conjunto de caracteres que o banco de dados irá usar. Para poder utilizar acentos da língua portuguesa, utilize o ENCODING *utf-8*.
- TABLESPACE *nome*: indica em qual *tablespace* o BD será armazenado.
- CONNECTION LIMIT *valor*: indica a quantidade máxima de usuários que poderão se conectar simultaneamente ao BD. O limite máximo só é observado para usuários que não sejam administradores, dessa forma, para o usuário *postgres*, não existe limite de conexão. O valor padrão é -1 indicando que não há limite de conexão simultânea.

#### 2.6.1.1 Exemplos

**Exemplo_1:** Criando o banco de dados *escola* com *encoding utf-8* :

*CREATE DATABASE escola encoding "utf-8";*
Lembrando que para o usuário *postgres* não existe limite de conexão.

**Exemplo_2:** Criando o banco de dados *sisvenda* com conexão simultânea de 1 no máximo usuário:

*CREATE DATABASE sisvenda connection limit 1;*
Lembrando que para o usuário *postgres* não existe limite de conexão.

### 2.6.2 Visualizando os Banco de Dados

Para visualizar os banco de dados no *psql*, digite:

\l

Já no pgAdmin, escreva o comando a seguir:

*select datname from pg_database;*

### 2.6.3 Conectando a um Banco de Dados

Para conectarmos a um banco de dados no psql, digite:

\c *nome_database*

Por exemplo, para no conectarmos ao database ABDA4, fazemos:

\c *ABDA4*

### 2.6.4 Alterando um Banco de Dados

Para alterar alguma propriedade do database, usamos o comando

*ALTER DATABASE nome_database OPÇÕES*

Por exemplo, para alteramos a quantidade de conexões simultâneas máxima, fazemos:

*ALTER DATABASE sisvenda CONNECTION LIMIT -1;*

Isso significa que esse *database* não terá limite de conexões simultâneas.

### 2.6.5 Determinando quantas Tabelas há no Banco de Dados

Para sabermos quantas tabelas existem no banco de dados, digitamos a consulta:

```
SELECT count(*)
FROM information_schema.tables
WHERE table_schema NOT IN ('information_schema', 'pg_catalog');
```

### 2.6.6 Determinando o Tamanho de um Banco de Dados

O tamanho de um *database* em que está conectado é obtido pela *query* a seguir:

*SELECT pg_database_size(current_database())*

Perceba que os valores estão em bytes. Para saber em megabytes acrescente a divisão a seguir:

```
SELECT (pg_database_size(current_database()))/1048576;
```

Entretanto, para saber o tamanho em megabytes de todos os *databases* do SGBD, faça:

```
SELECT (sum(pg_database_size(datname)))/1048576
from pg_database;
```

Já para conhecer o tamanho de uma determinada tabela dentro do banco de dados escreva:

```
select pg_relation_size('cliente');
```

Desta forma, para sabermos as tabelas que mais consomem espaço em um banco de dados, basta executar a consulta a seguir:

```
SELECT table_name,pg_relation_size(table_name) as size
FROM information_schema.tables
WHERE table_schema NOT IN ('information_schema', 'pg_catalog')
ORDER BY size DESC
LIMIT 10;
```

### 2.6.7 Excluindo um Banco de Dados

Para excluir um baco de dados, digite o comando:

*DROP DATABASE nome_banco_dados;*

Por exemplo, para excluir o database SIGAADM, faça:

*DROP DATABASE sigaadm;*

## 2.7 GERENCIAMENTO DE TABLESPACES

Os tablespaces (TS) são definições de locais para armazenamento lógico das informações do servidor. As TS permite que os banco de dados sejam criados em outros diretórios do servidor e não mais dentro do diretório padrão *../9.6/data* ou *../12.0/data*, etc.

Eles existem para que seja possível armazenar informações do servidor em locais distintos, o que pode ocorrer pelos mais diversos motivos: políticas de backup, organização, etc.

### 2.7.1 Criando Tablespaces

Para criar um tablespace utiliza o comando:

*CREATE TABLESPACE nome_do_tablespace LOCATION 'localização';*

Para utilizamos um tablespace, devemos indicar o diretório em que ele será criado (e os dados serão posteriormente armazenados). O usuário postgres deve ter proprietário desse diretório. Assim, para criarmos um TS dentro de um diretório chamado */opt/tmp*, devemos seguir os seguintes passos:

- Entre com o usuário root: *su <enter>*
- Crie o diretório: *mkdir /opt/tmp <enter>*
- Atribua o direito de propriedade ao usuário postgres: chown -R postgre /opt/tmp <enter>

Uma vez configurado o diretório em que será criado o TS, vamos criar um novo TS seguindo o código abaixo. Para isso pode ser utilizado tanto o psql quanto o PgAdmin:

*CREATE TABLESPACE ts_teste LOCATION '/opt/tmp';*

Agora, vamos criar um *database* dentro desse TS criado.

```
CREATE DATABASE "db_teste"
WITH OWNER "postgres"
ENCODING 'UTF8'
TABLESPACE = ts_teste;
```

Verifique que o *database db_teste* foi criado dentro do TS *ts_teste*.

### 2.7.2 Apagando Tablespaces

Para excluir um TS utilize a sintaxe a seguir:

*DROP TABLESPACE nome_do_tablespace;*

Não é possível excluir um tablespace que não esteja vazio. Assim, para apagarmos o TS *ts_teste* criado anteriormente, devemos fazer:

```
DROP DATABASE db_teste;
DROP TABLESPACE ts_teste;
```

### 2.7.3 Listando as Tablespaces

Para listar as tablespaces disponíveis, no psql digite:

*\db*

### 2.7.4 Criando o Banco de Dados *sigaadm* sobre a Tablespace *tstemp1*:

Para criar um Banco de Dados sobre a *tablespace tstemp1*, digite:

*CREATE DATABASE sigaadm tablespace tstemp1;*

## 2.8 EXERCÍCIOS

1. Determine qual a versão de seu SGBD?
2. Você é consultor de SGBD postgre e foi contratado para criar um novo *tablespace (TS)* dentro de um novo disco adquirido pela empresa contratante. O nome do novo TS será TSTEMP2 e, dntro dele, deverá ser criado um banco de dados de nome *BDTEMP2*.

3. Uma vez criado esse novo database, rode o script da disciplina dentro dele e verifique com que tamanho ele ficou.

4. Paulo Afonso, PA, gerente de TI, deconfia que algumas tabelas possuem tamanhos que justificam uma política de backup especial a elas. Destarte, determine quais são as cinco maiores tabelas da base de dados criada dentro do TS *TSTEMP2*.

5. Após reunião com a diretoria, ficou decido que não seria mais necessário o uso do *TS TSTEMP2*, pois outro TS foi criado para substitui-lo. Assim, Paulo Afonso, pediu a você removesse o referido TS.

6. Dentro das atividades para que você foi contratado, está a de que você deve criar um banco de dados de nome *BDTEMP3* que permita apenas 5 conexões simultâneas.

7. Uma vez que se esteja no terminal do *PostgreSQL* através do uso do comando psql, qual comando deve ser utilizado para exibir a ajuda geral dos comandos do psql?

   (a) \help (b) \? (c) \H (d) \q (e) \help *

8. Qual a ferramenta interativa de linha de comando padrão utilizado para acessar o SGBD PostgreSQL 9.1?

   (a) mysqld (b) sqlplus (c) pgcmd (d) psql (e) connect

9. (ENADE) Quanto ao PostgreSQL, identifique a alternativa que apresenta corretamente a instrução para renomear uma tabela:

   (a) ALTER TABLE nome_antigo RENAME TO nome_atuaI;

(b) RENAME TABLE nome_antigo TO nome_atuaI;

(c) ALTER TABLE nome_antigo TO nome_atuaI;

(d) RENAME TABLE nome_antigo ALTER TO nome_atuaI;

(e) UPDATE TABLE nome_antigo ALTER TO nome_atuaI;

Gabarito:

7 - B 8 - D 9 - A

# Capítulo 3

# Segurança do SGBD PostgreSQL: Usuários e Direitos de Acesso

## 3.1 INTRODUÇÃO

**Para** os exemplos desse capítulo, deve ser ter o script número 20171027 ou maior.

Uma das características que um SGBD deve possuir é o controle de acesso aos dados feito pelos usuário. O SGBD não pode deixar que usuários sem autorização acessem as informações de responsabilidade dele.

Para garantir esta restrição, o PostgreSQL só permite o acesso aos dados, os usuários previamente cadastrados. O PostgreSQL possui um usuário que é criado no momento da instalação:

- *postgres:* usuário com direito de administrador. É o que possui maior poder no SGBD.

Para saber quais usuários o Postgres possui digite:

*select usename,passwd from pg_shadow;*

Note que as senhas dos usuários aparecem criptografas pelo padrão MD5. Alem disso, é na tabela pg_shadow o SGBD Postgres armazena os usuários cadastrados nele.

Se precisar saber quais usuários tem direito de DBA, faça a consulta:

*select usename,usesuper from pg_shadow;*

**Dica:** para criptografar um campo senha do tipo varchar, use a função *md5()*:

- Inserindo um registro com senha criptografada
*insert into vendedor values (320, 'Amarildo', 6700, 'A', md5('123456'));*

- Selecionando o registro
*select nome_vendedor, senha*
*from vendedor*
*where senha = md5('123456');*

## 3.2 MANIPULAÇÃO DE USUÁRIOS

No *PostgreSQL*, o conceito de usuário é incorporado ao de *role*. *Role* (papel), grupos de usuários com determinadas permissões. Já os usuários são *papéis* com senha e que podem se conectar ao Postgre.

Quando se cria um usuário, esta se criando um *role*. Nós podemos criar um usuário, alterar suas propriedades e removê-lo.

### 3.2.1 Criação de Usuários

A sintaxe para se criar usuário é:

*CREATE USER usuário*
*WITH PASSWORD 'postdba';*

O CREATE USER é um aliás para CREATE ROLE.

Assim, para criarmos um usuário de nome *pauloafonso* procedemos:

*CREATE USER pauloafonso*
*WITH PASSWORD 'postdba';*

Outra maneira de vermos os usuários cadastrados é executando a consulta:

*select * from pg_user;*

### 3.2.2 Alteração De Usuário

Para alterar alguma propriedade do usuário, use o camando

*ALTER USER nome_usuário*
*opções*

#### 3.2.2.1 Exemplos

**Exemplo 1:** Para trocar a senha de um usuário

*ALTER USER usuario WITH PASSWORD '123456';*

**Exemplo 2:** Para de bloquear um usuário e assim ele não poder entrar no SGBD

*ALTER USER usuario*
*WITH NOLOGIN;*

**Exemplo 3:** Para de desbloquear um usuário e permitir seu login no SGBD

*ALTER USER usuario*
*WITH LOGIN;*

### 3.2.3 Remoção De Usuário

Para remover um usuário cadastrado, faz-se:

*DROP USER usuario;*

Deve-se observar que, quando um usuário possui objetos, deve-se eliminá-los ou atribuir sua propriedade (*owner*) a outro usuário e só depois remover o usuário desejado. Para alterar o proprietário de uma tabela faça:

*ALTER TABLE tabelausuario OWNER TO postgres;*

## 3.3 PRIVILÉGIOS DE USUÁRIO

Ao ser criado, um usuário tem apenas direito de atuar dentro de um banco de dados como usuário comum. Ele não possui direito de DBA e nem de acessar objetos, como tabelas, visões, etc, de outros usuários. Para que isso ocorra, ou seja, para que ele tenha o direito sobre objetos de outros usuários, é necessário atribuir concessões por meio do comando *grant* ao usuário recém criado.

O comando grant permite a concessão de direitos a objetos de usuários contro a objetos do próprio servidor. Com esse comando, é possível que um determinado usuário conceda privilégio de acesso aos seus objetos a outro determinado usuário.

### 3.3.1 Concedendo Privilégios

Para conceder privilégios aos usuários, usa-se o comando *grant* da seguinte forma:

*GRANT privilegio [, PRIVILEGIO,...]*
ON objeto
*TO usuario*

#### 3.3.1.1 Exemplos

Para observarmos algumas das propriedades funcionais do comando *grant*, vamos considerar os seguintes exemplos descritos a seguir. Porém, antes, vamos criar um usuário denominado de *usergrant* para realizações de nossos

testes:

*create user usergrant with password 'postdba';*

Feito isso, abra dois psql. Com um, conecte-se com o usuário *postgres* e com outro, conecte-se o usuário *usergrant*. Com o usuário recém criado não possui um banco de dados próprio, vamos conectar ambos os usuário ao ABDA4. Dessa forma, na linha de comando para chamar o psql, digite:

*psql -h localhost -U postgres -d ABDA4*
*psql -h localhost -U usergrant -d ABDA4*

**Exemplo 1:** tentar acessar os dados da **tabela de cliente**:

- Com o usúario ***usergrant***, digite o comando *select * from cliente;*
- Após o erro de falta de permissão, com o usuário ***postgres***, conceda a permissão da seguinte forma:
  - *grant **select on cliente** to usergrant;*
- Com o usuário ***usergrant***, digite, novamente, *select * from cliente;*

**Exemplo 2:** acessar e manipular **todos** os objetos de um banco de dados:

- Com o usúario ***usergrant***, digite o comando *select * from produto;*

- Após o erro de falta de permissão, com o usuário ***postgres*** **e dentro do *database* que você irá conceder todos os direitos** (neste exemplo, o comando a seguir deve ser executado dentro do *database* ABDA4) conceda a permissão da seguinte forma:
  - GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA public TO usergrant;
- Com o usuário ***usergrant***, digite, novamente, *select * from produto;*

### 3.3.2 Revogando privilégios

Uma concedido privilégios aos usuários, a maneira de tirá-los é através do comando *revoke*. Seu fomato é:

*REVOKE privilegio*
ON objeto
*FROM usuario;*

#### 3.3.2.1 Exemplo

Ainda considerando o mesmo ambiente criado na seção anterior com o usuário *usergrant*, vamos realizar alguns exemplos com o comando *revoke*.

**Exemplo 1:** Para tirar o direito do usuário *usergrant* de acessar os dados da tabela *cliente*.

*revoke select on cliente from usergrant;*

**Exemplo 2:** Para tirar o direito do *usergrant* de acessar todos os objetos do banco de dados ABDA4. Você deve executar de dentro do database ABDA4:

- *revoke ALL PRIVILEGES ON ALL TABLES IN SCHEMA public from usergrant* ;

## 3.4 Exercícios


Trabalhando com usuário e privilégios

1. Você ficou ficou encarregado de criar o ambiente para a instalação de um software de controle de processos empresariais denominado SIGASYS. Assim, crie um banco de dados denominado de SIGASYS e execute nele o *script* da base de dados da disciplina.

2. Uma vez criado o ambiente no exercício anterior, você irá executar as seguintes tarefas:

   (a) Criar um usuário denominado *"secretaria"* que possui somente o direito de visualizar os dados das tabelas de CLIENTE e VENDEDORES. Ela não deverá poder criar, atualizar ou deletar nenhum objeto do banco.

   (b) Criar um usuário denominado *vendedor* que possui os direitos de atualizar, criar e visualizar os dados na tabela de CLIENTE; criar, atualizar, visualizar e apagar dados da tabela de PEDIDO, atualizar, criar e visualizar os dados na tabela de ITEM_PEDIDO e atualizar e visualizar dados da tabela PRODUTO.

(c) Criar um usuário *gerente* podendo visualizar, inserir, atualizar e deletar dados das tabelas do banco de dados SIGASYS.

# Capítulo 4

# Cópias de Segurança (Backup)

## 4.1 Introdução

A realização de cópias de segurança, denominadas de *backup*, é uma tarefa essêncial no dia-a-dia do DBA. Uma vez que o DBA é o responsável pelo Banco de Dados, qualquer problema que venha ocorrer neste, é de responsbilidade do DBA. Ele é o responsável por deixar o banco sempre em produção e se ocorrer algum problema de perda de informação, ele deve providenciar e recuperar os dados o mais rápido possível.

Existem diversos fatores que podem causar perdas de dados, alguns deles são:

- Comandos mal utilizados como drop, delete, update, etc....
- Usuários mal-intencionados.

- Roubos de dados
- Virus
- Falha no hardware, etc

## 4.2 TIPOS DE BACKUPS

Existem várias formas de realizar backup. A mais simples é dar *shutdown* no banco e copiar todos os arquivos necessários com o banco *Offline*. Isso seria um *COLD backup*. Embora seja uma das formas mais fáceis para posterior *restore/recover*, não é a mais recomendada, pois existem bases trabalham 24/7 ou seja, parar um banco de dados é difícil[**?** ]. A outra forma de fazer backup é o *HOT backup*, realizado com o banco de dados em operação.

Basicamente, existem três tipos de backups que podem ser feitos em um SGBD, são eles[**?** ]:

- ***Backups* Completos**: este tipo **consiste na cópia de todos os arquivos** para a mídia de *backup*. Se os dados sendo copiados nunca mudam, cada backup completo será igual aos outros, ou seja, todos os arquivos serão copiados novamente. Isso ocorre devido o fato que um backup completo não verifica se o arquivo foi alterado desde o último backup; copia tudo indiscriminadamente para a mídia de backup, tendo modificações ou não. Esta é a razão pela qual os backups completos não são feitos o tempo todo. Por este motivo os backups incrementais foram criados.

- **Backups Incrementais**: ao contrário dos backups completos, os backups incrementais primeiro verificam se o horário de alteração de um arquivo é mais recente que o horário de seu último backup. Se não for, o

arquivo não foi modificado desde o último backup e pode ser ignorado desta vez. Por outro lado, se a data de modificação é mais recente que a data do último backup, o arquivo foi modificado e deve ter seu backup feito. Os backups incrementais são usados em conjunto com um backup completo frequente (ex.: um backup completo semanal, com incrementais diários).

  - A vantagem principal em usar backups incrementais é que rodam mais rápido que os backups completos. A principal desvantagem dos backups incrementais é que para restaurar um determinado arquivo, pode ser necessário procurar em um ou mais backups incrementais até encontrar o arquivo. Para restaurar um sistema de arquivo completo, é necessário restaurar o último backup completo e todos os backups incrementais subsequentes. Numa tentativa de diminuir a necessidade de procurar em todos os backups incrementais, foi implementada uma tática ligeiramente diferente. Esta é conhecida como backup diferencial.

- **Backups Diferenciais**: são similares aos backups incrementais, pois ambos podem fazer backup somente de arquivos modificados. No entanto, os backups diferenciais são acumulativos, em outras palavras, no caso de um backup diferencial, uma vez que um arquivo foi modificado, este continua a ser incluso em todos os backups diferenciais (obviamente, até o próximo backup completo). Isto significa que cada backup diferencial contém todos os arquivos modificados desde o último backup completo, possibilitando executar uma restauração completa somente com o último backup completo e o último backup diferencial. Assim como a estratégia utilizada nos backups incrementais, os backups diferenciais normalmente seguem a mesma tática: um único backup completo periódico seguido de backups diferenciais mais frequentes. O efeito de usar backups diferenciais desta maneira é que estes tendem a crescer um pouco ao longo do tempo (assumindo

que arquivos diferentes foram modificados entre os backups completos). Isto posiciona os backups diferenciais em algum ponto entre os backups incrementais e os completos em termos de velocidade e utilização da mídia de backup, enquanto geralmente oferecem restaurações completas e de arquivos mais rápidas (devido o menor número de backups onde procurar e restaurar).

## 4.3 FORMAS DE BACKUPS NO POSTGRE

O Postgre fornece duas formas de *backup* por padrão. Essas formas possibilitam o DBA a manter as cópias de seguranças atualizadas e prontas para serem usadas caso haja necessidade. As maneiras de *backups* possíveis no Postgre são:

- *pg_dump*: programa por linha de comando do S.O. que gera *backups* para posterior recuperação com o comando *pg_restore*. Essa opção é denominda de *backup* lógico.
- cópia dos arquivos de sistema: em que o DBA faz uma cópia dos arquivos direta do S.O.. O DBA utiliza comandos do próprio sistema operacional. No caso do Linux, é o comando *cp* e do Windows, o comando *copy*. Essa segunda opção é denominda de *backup* físico.

Nesse curso, iremos trabalhar **somente com *backups* lógicos** com o comando *pg_dump*.

## 4.4 REALIZANDO BACKUP COM O PROGRAMA PG_DUMP

Nessa seção, veremos como realizar diversos tipos de backup por meio do programa *pg_dump*.

### 4.4.1 Backup Completo do Servidor

Para exportar todos os banco de dados existentes em seu servidor Postgres é necessário utlizar o *pg_dumpall*, que realiza a mesma função do programa *pg_dump*, porém para todos os databases do seu servidor.

*pg_dumpall > copia_seguranca.bkp*
onde:
- copia_seguranca.bkp é o nome do arquivo de *backup*.

### 4.4.2 Backup de um Banco de Dados Específico

Para exportar um banco de dados específico no Postgres é necessário utilizar o *pg_dump* da seguinte maneira:

*pg_dump -h localhost -U postgres -v -Fc -f ../../../../tmp/base.backup database_copiado*
onde:
-h: é o host em que se encontra o servidor Postgres.
-U: o usuário que fará o backup.
-v: mostra o que está sendo copiado.
-Fc: cópia no format .backup / -Fp: cópia no formato SQL
-f: destino do arquivo com a cópia de segurança
*database_copiado*: é o banco de dados em que será realizado *backup*.

#### 4.4.2.1 Exemplos

Para realização dos exemplos de backups, crie um diretório chamado *backup_aula* para colocar as cópias de segurança dentro dele.

**Exemplo 01:** Realizando o *backup* do *database* ABDA4:

*pg_dump -h localhost -U postgres -v -Fc -f data_do_dia_ABDA4.bkp ABDA4*

**Exemplo_02:** Realizando o *backup* do *database* postgres:

*pg_dump -h localhost -U postgres -v -Fc -f data_do_dia_postgres.bkp postgres*

### 4.4.3 Backup de umaTabela Específica

Para realizar o backup de uma tabela específica, execute o *pg_dump* como o exemplo a seguir:

**Exemplo 01:** Realizando o *backup* da tabela *cliente* do *database* ABDA4:

*pg_dump -h localhost -U postgres -v -Fc -f data_do_dia_tab_cliente.bkp ABDA4 -t cliente*

**Exemplo 02:** Realizando o *backup* da tabela *seq_salario_registro* do *database* ABDA4 no format SQL:

*pg_dump -h localhost -U postgres -v -**Fp** -f data_do_dia_tab_seq_salario_registro.sql ABDA4 -t seq_salario_registro*

## 4.5 RESTAURANDO BACKUPS COM O PG_RESTORE

Para realizar a restaruração dos backups feitos com o pg_dump, utilizamos o programa pg_restore.

O pg_restore é o responsável por importar as informações de um arquivo criado pelo pg_dump para dentro do SGBD.

## 4.6 REALIZANDO RESTORES COM O PROGRAMA PG_RESTORE

Nessa seção, veremos como realizar diversos tipos de restaurações por meio do programa *pg_restore*.

### 4.6.1 Restauração Completa do Servidor

Para importar todo o conteúdo de um arquivo de backup utilizando o *pg_restore*, faça:

*pg_restore [opções] arquivo_de_backup*

### 4.6.2 Restaurando um Banco de Dados Específico

Para restaurar o backup um banco de dados específico, utilize a o comando pg_restore da seguinte maneira:

*pg_restore -h localhost -U postgres -v -d Database_Destino -Fc data_do_dia_database.bkp*

onde:

-d: é o nome do *database* em que a cópia de segurança será restaurada.
-Fc: cópia no format .backup

#### 4.6.2.1 Exemplos

Para realização dos exemplos de restauração das cópias de segurança, utilize os arquivos criados no diretório chamado *backup_aula.*

**Exemplo 01:** Realizando a restauração do *database* ABDA4 para o database ABDA4_Restaurado:

*pg_restore -h localhost -U -v postgres -d* ABDA4_Restaurado *-Fc data_do_dia_ABDA4.bkp*

**Exemplo_02:** Realizando a restauração do banco de dados postgres para o postgres_restaurado:

*pg_restore -h localhost -U -v postgres -d* postgres_restaurado *-Fc data_do_dia_postgres.bkp*

### 4.6.3 Restaurando umaTabela Específica

Para realizar a restauração de uma tabela específica, execute o *pg_restore* como os exemplos a seguir:

**Exemplo 01:** Realizando a restauração da tabela *cliente* do *database* ABDA4_Restaurado:

*pg_restore -h localhost -U postgres -v -d ABDA4_Restaurado -Fc 20171104_ABDA4.bkp -t cliente*

Repare que as chaves estrageiras que faziam referência à tabela cliente não foram restauradas.

**Exemplo 02:** Realizando o *backup* da tabela *seq_salario_registro* do *database* ABDA4_Restaurado no format SQL. Note que nesse caso, a restauração deve ser feita com o programa *psql* como mostrado a seguir:

***psql** -h localhost -U postgres -v -d ABDA4_Restaurado -f 20171104_tab_seq_salario_registro.sql*

## 4.7 MIGRANDO BANCO DE DADOS ENTRE VERSÕES DIFERENTES DO POSTGRES

Fonte: http://pgdocptbr.sourceforge.net/pg80/migration.html

Como regra geral, o formato interno de armazenamento dos dados está sujeito a alterações entre versões principais do PostgreSQL (onde muda o número após o primeiro ponto). Isto não se aplica às versões secundárias sob a mesma versão principal (onde muda o número após o segundo ponto); estas versões sempre possuem formatos de armazenamento compatíveis. Por exemplo, as versões 7.0.1, 7.1.2 e 7.2 não são compatíveis, enquanto as versões 7.1.1 e 7.1.2 são compatíveis. Quando é feita a atualização para uma versão compatível, pode-se simplesmente substituir os executáveis e reutilizar o diretório de dados no disco. Caso contrário, é necessário fazer a cópia de segurança dos dados e "restaurá-la" no novo servidor. A cópia de segurança tem de ser feita utilizando o ***pg_dump***.

O menor tempo de parada pode ser obtido instalando o novo servidor em um diretório diferente, e executando tanto o servidor novo quanto o antigo em paralelo, em portas diferentes. Depois pode ser executado algo como:

*pg_dumpall -p **5432** > copia_seguranca.bkp*
*pg_restore -p **5433** arquivo_de_backup*

## 4.8 EXERCÍCIOS

Para os exercícios de backups, o aluno deve criar dois banco de dados: bdbackup1 e bdbackup2. Em seguida, salve o *script* de criação da base de dados da disciplina em */home/aluno/aulaABD*. Quem usar o Windows, crie o diretório *c:\aluno\aulaABD*. Realize a tarefa de criação do *database* e as seguintes por meio do ***psql***.

1. Você acabou de instalar um novo SGBD Postgre e irá, agora, rodar o *script* da disciplina para criar as tabelas iniciais no *database bdbackup1*. Faça isso com a opção **\i** do psql. O **\i** permite que você indique o caminho que contém o arquivo SQL para ser executado. Considerando que o *script* foi salvo no local indicado, use o \i assim:
   \i */home/aluno/aulaABD/criabaseABD_Postgres.sql*

2. O passo a seguir é garantir que, se o servidor de problema, você tenha uma cópia de segurança. Para isso, utilize o comando *pg_dump* diretamente da linha de comando do S.O. para realizar a cópia de segurança do *database bdbackup1*. Faça cópia no formato *backup*.

3. Simule uma ação do estagiário que, corajosamente, apagou a tabela de *vendedor*. A seguir, recupere somente essa tabela do *backup*.

# Capítulo 5

# Índices e Otimização de Banco de Dados

## 5.1 INTRODUÇÃO

Índices são estrutura de dados inseridas no banco de dados com o objetivo de melhorar o desempenho de acesso às tabelas. Para realizamos os exemplos desta apostila, precisaremos executar o *script* ***sp_insere_cliPostgresOTIMIZACAO.sql*** disponível no Github.

Sua função é reduzir o I/O em disco utilizando uma estrutura de árvore *B (B-Tree)* para localizar rapidamente os dados.

Índices são utilizados durante comandos que possuam as cláusulas **where, order by** e **group by**.

Considere

Imagine uma tabela criada da seguinte forma:

*create table teste(*

*c1 char(30),*

*c2 char(30),*

*c3 char(30),*

*...*

*c26 char(30));*

A tabela teste possui 26 campo de 30 bytes cada, sendo assim, o SGBD precisa de 780 bytes (26x30).
A cada 10 registros, a tabela ocuparia 7.8 Kbytes.

Usando **Índice**

Agora imagine uma estrutura auxiliar que contivesse 2 campos: C1 (Campo indexado) e o endereço físico do bloco no disco que armazena o registro.

Cada registro ocupando 40 bytes (c1 + endereço do bloco de tamanho de 10 bytes). Assim os 7.8 bytes poderiam armazenar 195 registros (7800/40).

## 5.2 PÁGINAS DO DISCO

A primeira definição que veremos está ligada diretamente com o **Sistema Operacional**. Para o SGBD manipular informações, ele deve acessar tanto a memória (RAM) quanto o disco (HD).

Todos os acessos feitos a estes dois recursos sempre serão feitos através do Sistema Operacional, pois é ele que faz este meio de campo entre as aplicações e os recursos.

O sistema operacional precisa trabalhar com blocos de informações para serem lidos ou gravados, de modo a otimizar o acesso à memória e ao disco. Estes blocos de informações são chamados de página.

## 5.3 ÁRVORE B+

A estrutura de árvore B+ é a mais utilizada dentre as estruturas de índices, pois mantém sua eficiência mesmo sob pesada carga de inserções e remoções de dados. O nome árvore vem da semelhança da estrutura do índice com árvore de cabeça para baixo, onde a raiz está localizada no topo da estrutura e é usada como ponto inicial nos processos de buscas.

Essa estrutura é composta por páginas, ou nós, divididas em dois grupos: páginas internas e folhas. As páginas internas são usadas para guiar o processo de busca até as páginas folhas, e estas últimas contêm o valor da chave de busca e a localização da página de dados no disco referente ao registro procurado.

É importante frisar que na avaliação de custo de uma consulta que use uma árvore B+, considera-se o custo de recuperar uma página do disco e trazê-la para a memória RAM.

A Figura 5.1 mostra uma árvore B+ com 3 níveis e de ordem N=4. Todas as páginas folhas aparecem no nível 3. Em cada página, as chaves de busca ocorrem em ordem crescente, permitindo utilizar o algoritmo de busca binária (ver nota 1) na procura da chave dos registros lidos para a memória.

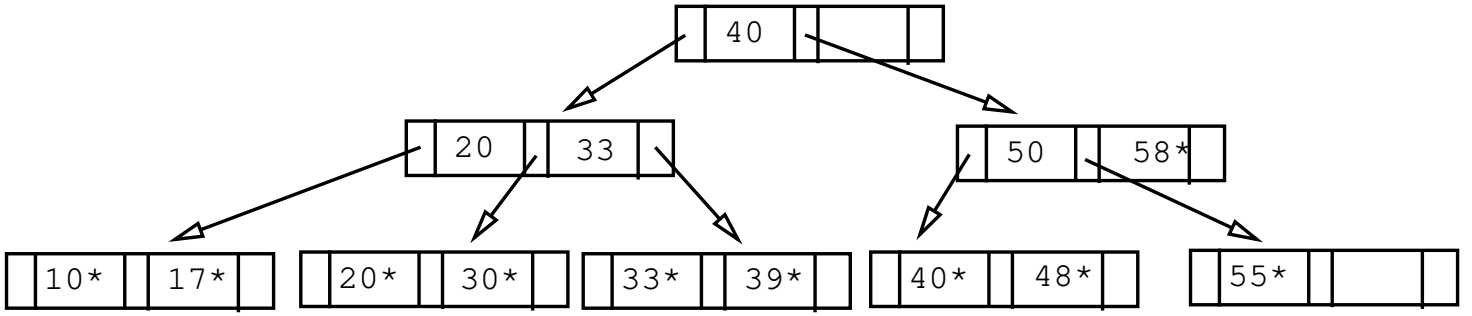


Figura 5.1: Exemplo de Árvore B+

## 5.4 DESVANTAGEM DO USO DE ÍNDICES

Um índice é uma estrutura criada a parte no banco de dados. Para cada índice criado, uma estrutura é criada, ocupando espaço e consumindo recursos do SGBD.

O recurso consumido está no que se **refere a atualização** dos mesmos, pois para cada atualização em uma tabela, todos os índices ligado a ela, devem ser atualizados. Se uma tabela tiver muitos índices. maior será o tempo gastos com as atualizações.

## 5.5 CRIANDO ÍNDICES

O comando para criar índices é o *create index*. Seu uso é da seguinte forma:

*create index idx_nome_indice on tabela(campo1);*

Além disso, se um índice contiver valores que não podem ser repetidos, devemos colocar a claúsula *unique*. Fazemos isto da seguinte forma:

*create **unique** index idx_nome_indice on tabela (campo1);*

**Exemplos:**

*create index idx_cidade on cliente (cidade);*
Cria um índice em sobre o campo cidade da tabela cliente.

*create unique idx_cnpj on cliente (cnpj);*
Cria um índice único em sobre o campo obs1 da tabela cliente.

### 5.5.1 Criando índices compostos

**Para** criar índices compostos utiliza-se o comando *create index* da seguinte forma:

*create index idx_nome_indice on tabela(campo1, campo2,...,campoN);*

**Exemplo:**

*create index idx_nome_endereco on cliente (nome_cli, rua_cli);*
Cria um índice composto sobre os campos *nome_cli* + *rua_cli*.

## 5.6 REMOVENDO ÍNDICES

**Para** remover um índice, basta utilizar o comando *drop index* da seguinte forma:

*drop index idx_nome_indice;*

**Exemplo**

*drop index idx_cidade;*

## 5.7 SELETIVIDADE DE ATRIBUTOS

Um conceito importante que afeta a decisão sobre usar ou não um índice é a seletividade do atributo. A seletividade estima, em média, a porcentagem dos registros que deverão aparecer na resposta dado um determinado atributo. Ela pode ser calculada pela função:

*Seletividade (atributo)= 1 - (registros-na-resposta-filtrada-pelo-atributo / total-de-registros)*
Registro retornados = 1 - Seletividade

Quanto mais próximo de 1, melhor é a seletividade de um atributo. Vejamos um exemplo: suponha uma tabela de clientes com 500 clientes. Vamos realizar uma consulta pelo CPF. Sabemos que o CPF não se repete e que existe um índice sobre este campo. Então teremos:

*Seletividade(CPF) = 1 - (1/500) = 0,998*
*Registro retornados = 1 - 0,998 = 0,012*

Isto indica que CPF é um excelente atributo para realização de consultas através do uso do índice, pois cada valor de CPF descarta quase todos os registros da tabela. Agora vamos realizar uma consulta pelo atributo cidade, sabendo que existem clientes de 110 cidades diferentes e que também existe um índice sobre esse campo.

*Seletividade(Cidade) = 1 - (110/500) = 0,78*
*Registro retornados = 1 - 0,78 = 0,22*

Esta pesquisa irá retornar em média 22% dos registros. Isto pode indicar para o otimizador não utilizar o índice sobre o campo cidade e sim fazer uma varredura total sobre a tabela.

Considere a tabela cliente composta pelos campos:

*Cliente = {nome_cliente, rua_cli, cidade, obs1, obs2, obs3}*

A tabela possui um total de 100.000 registros armazenados. Foi criado um índice de árvore B+ sobre o campo cidade, onde existem 95.000 registros com valores iguais a "Sao Paulo" e 5000 registros com valores iguais a "Fernandopolis". Além disto, também existe um índice B+ sobre o campo chave *nome_cliente*. Agora considere a seguinte consulta:

*select nome_cliente from cliente where cidade = "Sao Paulo";*

Uma vez que 95% dos registros possuem o campo cidade igual a "Sao Paulo", o uso do índice torna-se dispendioso, indicando para o otimizador que é mais econômico fazer uma varredura em toda a tabela cliente do que utilizar o índice sobre o campo indexado.

Uma vez que somente 5% dos registros possuem o valor "Fernandopolis" para o atributo cidade, é interessante para o otimizador fazer o uso do índice para resolver esta consulta.

## 5.8 ANALISANDO O USO DE ÍNDICES EM CONSULTAS

O Postgre disponibiliza o comando *EXPLAIN* que possibilita analisar previamente consultas SQL para saber como será o comportamento das mesmas no momento de sua execução e assim determinar qual a melhor maneira de realizá-las.

Para utiliza-lo basta acrescentá-lo antes da consulta da seguinte forma:

*explain select * from tabela;*

Os exemplos a seguir mostram algumas situações de quando o SGBD faz ou não uso dos índices. É importante ter em mente, que o *EXPLAIN* faz uso de estatítiscas das tabelas e se estas estatísticas estiverem desatualizadas, o SGBD pode executar uma busca pela maneira ineficiente.

### 5.8.1 Exemplos

Os exemplos a seguir mostram como usar o *explain* para analisarmos uma consulta:

**Exemplo 1:** selecionando os dados dos clientes de 'Fernandopolis' que possui alta seletividade.

*explain*
*select * from cliente_ind*

*where cidade_cli = 'Fernandopolis';*

–**Houve o uso do índice** idx_cidade, pois o valor 'Fernandopolis' tem alta seletividade.

**Exemplo 2:** selecionando os dados dos clientes de 'Sao Paulo' que possui baixa seletividade.

*explain*
*select * from cliente_ind*
*where cidade_cli = 'Sao Paulo';*

–**Não houve o uso** do índice idx_cidade pois existem muitas tuplas com valor igual a 'Sao Paulo' indicando que esse valor tem **baixa seletividade**.

**Exemplo 3:** selecionando os dados dos clientes de 'Fernandopolis' **e** o cliente de nome = 'nome100'. Como o valor 'nome100' tem maior seletividade, o SGBD deixa de usar o índice *idx_cidade* e só usa o *pk_cliente*.

*explain*
*select **
*from cliente_ind*
*where cidade_cli = 'Fernandopolis'*
*and nom]e_cli = 'nome100';*

–Note que **não houve** o uso do **indice idx_cidade,** mas **sim** do índice **pk_cliente** (chave primária) que mais é restritivo para localizar os dados desejados.

**Exemplo 4:** selecionando os dados dos clientes de 'Fernandopolis' **e** o cliente de nome *like* 'nome100%'. Como o operador like força uma *range* (faixa), o SGBD usa o índice *idx_cidade* e descarta o *pk_cliente*.

*explain*
*select **
*from cliente_ind*
*where cidade_cli = 'Fernandopolis'*
*and nome_cli like 'nome100%';*
–Note que **não houve** o uso do indice pk_cliente, pois o SGBD prefere localizar comparações de igualdade do que procurar uma *range* indicada pelo operador *like*.

**Exemplo 5:** selecionando os dados dos clientes de 'Fernandopolis' **ou** o cliente de nome *like* 'nome100%'. Como o operador ***OR*** foi empregado na consulta, ele descarta o uso de índices.

*explain*
*select **
*from cliente_ind*
*where cidade_cli = 'Fernandopolis'*
***or** nome_cli like 'nome100%';*
–Note que **não houve** o uso dos dois indices, pois o **operador OR** inabilita o uso dde índices

**Exemplo 6:** selecionando os dados dos clientes diferentes de 'Fernandopolis' **ou** o cliente de nome *like* 'nome100%'. Como o operador ***OR*** foi empregado na consulta, ele descarta o uso de índices. O operador **<>** também inabilita o uso de índices.

*explain*
*select **

*from cliente_ind*
*where cidade_cli <> 'Fernandopolis'*
***or** nome_cli like 'nome100%';*
–Note que **não houve** o uso dos dois indices, pois o **operador OR** inabilita o uso de índices. O operador **<>** também inabilita o uso de índices.

**Exemplo 7:** selecionando os dados dos clientes de 'Fernandopolis' **e** a rua igual a 'rua a'. O índice *idx_cidade* foi usado pelo SGBD e posteriormente foi feito um filtro pela 'rua a'.

*explain*
*select **
*from cliente_ind*
*where cidade_cli = 'Fernandopolis'*
***and** rua_cli = 'rua a';*
–Note **que houve** o uso do índice *idx_cidade*, pois, depois de localizar os registros com 'Fernandopolis', ele filtrou quem reside na 'rua a'.

**Exemplo 8:** selecionando os dados dos clientes de 'Fernandopolis' **ou** a rua igual a 'rua a'. Como o operador ***OR*** foi empregado na consulta, ele descarta o uso de índices.

*explain*
*select **
*from cliente_ind*

*where cidade_cli = 'Fernandopolis'*
***or*** *rua_cli = 'rua a';*
–Note que **não houve** o uso do indice *idx_cidade,* pois o **operador *OR*** inabilita o uso dde índices

**Exemplo 9:** selecionando os dados dos clientes cujos nomes são filtrados pelo operado ***IN***. Houve o uso de índice cujo custo foi de **0.42 a 24.10**.

*explain*
*select **
*from cliente_ind*
*where nome_cli in ('nome100*', 'nome 200', 'nome300');
–Note **que houve o uso** do indice pk_cliente. Assim, o operador **IN** aplicado sobre um campo com índice faz uso desse.

**Exemplo 10:** selecionando os dados dos clientes cujos nomes são filtrados pelo operado ***or*** sobre o mesmo campo. Na prática, o resultado da busca é o mesmo que no exemplo com o operador **IN**, porém o custo é maior. Houve o uso de índice cujo custo foi de **13.28 a 25.21**.

*explain*
*select **
*from cliente_ind*
*nome_cli = 'nome100' or nome_cli = 'nome 200' or nome_cli = 'nome300';*
–Note **que houve o uso** do indice pk_cliente. Apesar de usar o operador OR, o SGBD optou pela utilização dos índices.

## 5.9 COMANDO VACUUM PARA DESFRAGMENTAÇÃO DOS DADOS

Uma característica gerada pelos tipos de dados *varchar* e *char* que sofrem constantes alterações, ora aumentando o tamanho do valor armazenado, ora diminuindo o tamanho, é a **fragmentação** do campos com esses tipos de dados.

A fragmentação é uma espécie de 'buracos' no disco rígido que impacta no desempenho das buscas do SGBD. Um banco de dados muito fragmentado pode ter seu tempo de busca prejudicado e assim causar mal estar com os usuários.

Para contornar esse problema, o Postgre dispõe do comando *vacuum* cuja sintaxe é a seguinte:

VACUUM [VERBOSE] [ANALYSE] [TABELA]
onde:
[VERBOSE]: ao habilitar essa opção, informações detalhadas serão exibidas.
[ANALYSE]: além de desfragmentar, atualiza as estatística. Assim, o comando *explain* pode dar resultados diferentes dos mostrados antes da atualização das estatísticas.
[TABELA]: indica a tabela que se deve fazer a desfragmentação. Caso não seja informada a tabela, todo o banco de dados será desfragmentado.

### 5.9.1 Exemplos

Segue exemplos dos do uso do comando *vacuum*.

**Exemplo 1:** desfragmentando a tabela de *cliente*_ind e atualizando as estatísticas.

*vacuum verbose analyse cliente_ind;*

**Exemplo_2:** desfragmentando a tabela de *historicos_escolares*_ind sem atualizar as estatísticas.

*vacuum verbose historicos_escolares;*

## 5.10 RESUMO E DICAS PARA CRIAÇÃO DE ÍNDICES

Agora que aprendemos como verificar se o SGBD fará ou não o uso de índices, vamos a algumas dicas de escrita do SQL que fazem ou não o uso dos índices.

Considere o comando select abaixo, nele temos a cláusula where que possue diversas comparações:

```
1    select campo1
2    from tabela
3    where campo1 = 3
4    and campo2 > 4
5    and campo3 <> 7
6    and campo4 between 10 and 20
7    and campo5 + 10 = 15
8    and campo8 like 'Re%'
9    and campo9 like '%ana'
10    and campo10 is null
```

- Linha 3: Faz o uso de índice.
- Linha 4: Faz o uso de índice.
- Linha 5: Não faz uso de índice.
- Linha 6: Faz o uso de índice.
- Linha 7: Não faz o uso de índice.
- Linha 8: Faz o uso de índice.
- Linha 9: Não faz o uso de índice.
- Linha 10: Não faz uso de índice.

### 5.10.1 Quando criar índices

Siga estas dicas para decidir se há ou não a necessidade da criação de índices:

**Tabelas pequenas**: Nao há necessidade de criamos índices para tabelas pequenas. É mais economico para o SGBD fazer um *full table scan*, do que usar o um índice para tabelas pequenas.

**Chave estrageira**: É importante utilizar índices em chaves estrangeiras, já que estes são muito utilizados em joins. O índice será útil quando a tabela que possui uma chave estrangeira tentar acessar dados na tabela que dá

suporte à esta chave. Também será útil no caso em que for excluir determinada linha de uma tabela que possui uma chave estrangeira, terá que ser feita uma leitura na tabela de onde vem os dados para a chave estrangeira para excluir os dados nesta segunda tabela no caso em que foi definido ON DELETE CASCADE.

**Garantia de Integridade de chave Candidata**: A integridade de chaves candidatas é garantida através de índices. Isto pode ser feito de duas maneiras:

1. **Caso:**
   *create table teste(*
   *campo1 integer not null,*
   *campo2 varchar2(40),*
   *campo3 varchar2(50),*
   *constraint pk_teste* ***primary key****(campo1),*
   *constraint idx_campo2* ***unique*** *(campo2)*
   *constraint fk_teste_mama* ***foreign key*** *(campo3) references mama);*
   Nesta tabela temos os seguintes índices:
   **- Primary Key**: Toda *primary key* é um índice.
   **- Unique**: As restrições de unicidade são feitas através de índices
   **- Foreign Key**: Nem sempre a chave estrageira é um índice.

2. **Caso**:
   *create unique index idx_campo2 on teste (campo2);*

Cria um índice sobre o *campo2* da tabela teste.

3. **Campos de busca com alta seletividade**: Antes de criar um índice sobre um determinado campo, verifique sua seletividade. Campos com baixa seletividade não são candidatos a terem índices.

## 5.11 Exercícios

1. Crie índices para as chaves estrangeiras das tabelas *pedido* e *item_pedido*.
2. O índice, em um banco de dados, encapsula tarefas repetitivas, aceita parâmetros de entrada e retorna um valor de status, para indicar aceitação ou falha na execução.
3. Em sistema gerenciador de banco de dados, os índices são estruturas que permitem agilizar a busca dos registros no disco.
4. (ENADE) Os bancos de dados em geral são armazenados fisicamente como arquivos de registros em discos magnéticos. Para acessá-los, existem diferentes técnicas que eficientemente usam vários algoritmos. Sobre conhecimento técnico em administração de dados, algumas técnicas de acesso requerem estruturas de dados auxiliares que são chamadas: **A)** matrizes. **B)** esquemas. **C)** tabelas. **D)** índices. **E)** hashing.

### 5.11.1 Gabarito

2-E; 3-C; 4-D