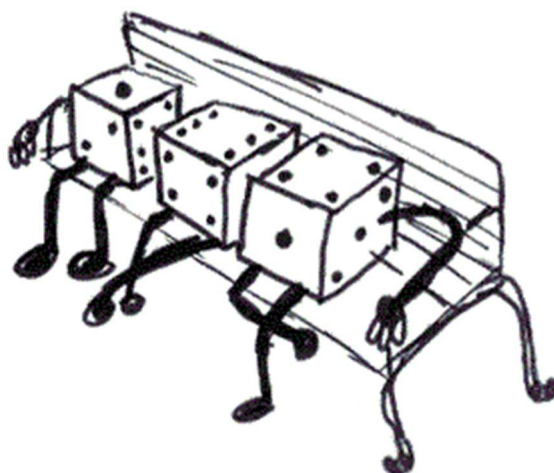


**UMC – Universidade de Mogi das Cruzes**

## *Banco de Dados*



## **Notas de Aula**

*Profa. MSc. Viviane Guimarães Ribeiro*

2023

## Sumário

1.	Banco de Dados.....	3
2.	Sistemas de Banco de Dados.....	4
2.1.	Problemas a serem resolvidos pelo SGBD.....	4
2.2.	Classificação de Banco de Dados .....	6
2.3.	Linguagem de Definição de Dados (DDL) .....	9
2.4.	Linguagem de Manipulação de Dados (DML) .....	9
2.5.	Linguagem de Controle de Dados (DCL).....	10
2.6.	Usuários do Banco de Dados.....	10
3.	O Projeto de Banco de Dados .....	11
3.1.	Modelo Conceitual .....	11
3.2.	Modelo Lógico .....	11
3.3.	Modelo Físico .....	12
4.	Modelo Entidade-Relacionamento .....	14
4.1.	Entidade (entidades-dados) .....	14
4.2.	Atributos.....	14
4.2.1.	Atributo Chave Primária.....	15
4.2.2.	Atributo Multivalorado .....	15
4.3.	Relacionamentos.....	16
4.3.1.	Relacionamentos 1 : 1 .....	16
4.3.2.	Relacionamentos 1 : N .....	17
4.3.3.	Relacionamentos N : N .....	18
4.3.4.	Auto-Relacionamento .....	18
4.3.5.	Relacionamento “n”-ário.....	19
4.3.6.	Agregação (conjuntos complexos) .....	19
4.3.7.	Mapeamento do MER para o Modelo Relacional (MR) .....	20
5.	Normalização.....	31
5.1.	Primeira Forma Normal – 1FN.....	31
5.2.	Segunda Forma Normal – 2FN .....	33
5.3.	Terceira Forma Normal – 3FN .....	34
5.4.	Forma Normal de Boyce-Codd – FNBC.....	36
5.5.	Quarta Forma Normal – 4FN .....	37
5.6.	Quinta Forma Normal – 5FN .....	38
6.	Conexão com o Banco de Dados .....	40
6.1.	Instalando o pgAdmin .....	40

6.2.	Acessando o pgAdmin .....	42
7.	Data Description Language – DDL .....	46
7.1.	Criação das Tabelas – CREATE TABLE .....	46
7.2.	Alteração das Tabelas – ALTER TABLE .....	47
7.3.	Remoção de Tabelas – DROP TABLE .....	49
8.	Data Manipulation Language - DML.....	51
8.1.	Inserção de Dados – Insert .....	51
8.2.	Exclusão de Dados – Delete .....	52
8.3.	Atualização de Dados – Update .....	53
8.4.	Consulta de Dados – Select .....	54
8.4.1.	Select com Junções .....	55
8.1.2.	Funções Agregadas.....	59
8.1.3.	Outras Funções e Cláusulas.....	60
9.	Sequence .....	62
10.	Insert com Query .....	64
11.	Índices .....	66
12.	Visões - View .....	68
13.	Transações.....	69
13.1.	Estados de uma transação .....	69
13.2.	Transações no PostgreSQL .....	70
14.	Stored Procedures em SQL (Functions).....	72
15.	Funções em Linguagens Procedurais- PL/PgSQL.....	75
16.	Trigger .....	77
Anexo A -	Tipos de Dados – PostgreSQL .....	82
	Tipos numéricos .....	82
	Tipos monetários.....	82
	Tipos para cadeia de caracteres.....	82
	Tipos para data e hora .....	82
	Tipo booleano .....	83

## 1. Banco de Dados

Um banco de dados pode ser definido como um conjunto de dados devidamente relacionados. Podemos compreender como dados os objetos conhecidos que podem ser armazenados e que possuem um significado implícito, porém o significado do termo banco de dados é mais restrito que simplesmente a definição dada anteriormente. Um banco de dados possui as seguintes propriedades:

- É uma coleção lógica coerente de dados com um significado inerente. Uma disposição desordenada dos dados não pode ser referenciada como banco de dados;
- Ele é projetado, construído e preenchido com valores de dados para um propósito específico;
- Um banco de dados possui um conjunto predefinido de usuários e de aplicações;
- Ele representa algum aspecto do mundo real, o qual é chamado de minimundo. Qualquer alteração efetuada no minimundo é automaticamente refletida no banco de dados.

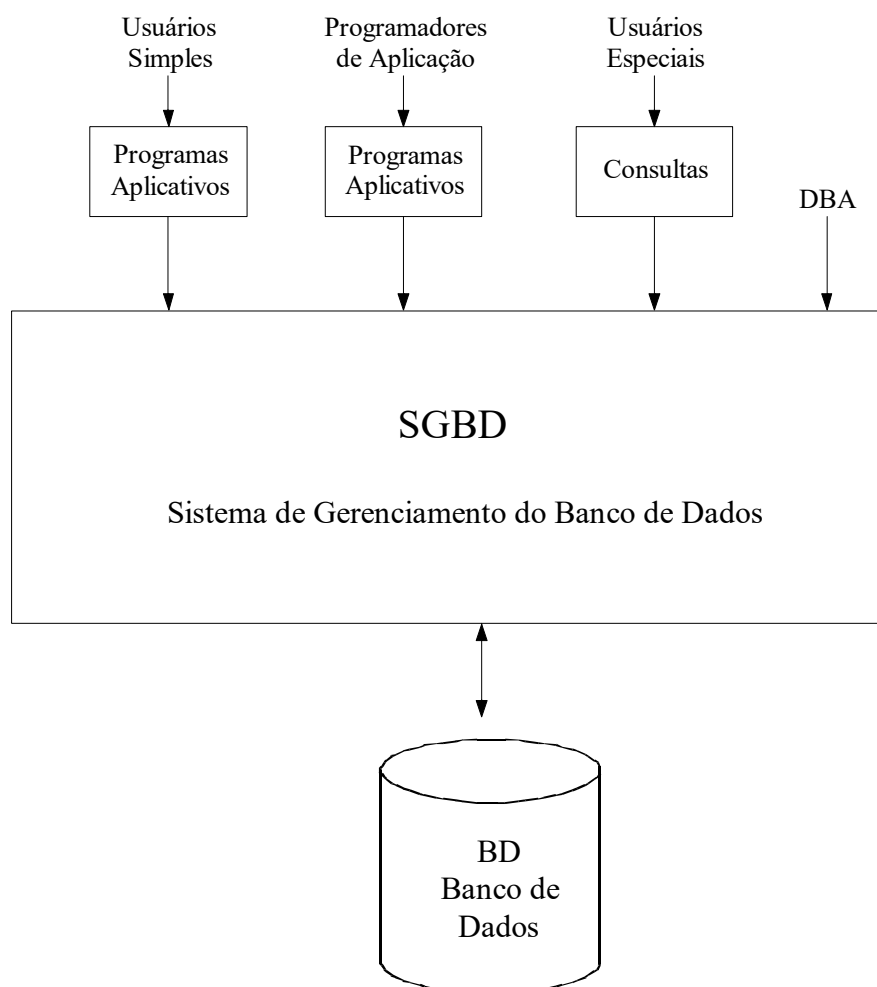
## 2. Sistemas de Banco de Dados

É um conjunto de elementos envolvidos no armazenamento e processamentos de grande volume de dados. Desde os dados propriamente ditos, até os usuários do software todos são componentes do sistema.

Um elemento fundamental de um sistema de banco de dados é o “Sistema Gerenciador de Banco de Dados (SGBD) => (Em inglês: Data Base Management System – DBMS). Podem variar entre fabricante, recursos e preço. EX: ORACLE, DBZ SQL SERVER.

O SGBD inclui ainda facilidades para:

- Manipulação de dados;
- Definição de estruturas de dados;
- Segurança contra falhas e acessos não autorizados;
- Controle de uso compartilhado dos dados por diversos usuários;
- Manter a integridade dos dados.



Não utilizar um sistema de gerenciamento de banco de dados significa ter um sistema de arquivos distintos, independentes, e uma série de programas específicos para cada tarefa desejada. Por exemplo, para cada processamento, cada consulta ou cada relatório pode ser necessária a criação de um novo programa.

### 2.1. Problemas a serem resolvidos pelo SGBD

Sistemas convencionais de processamento de arquivos são constituídos de:

- Arquivos (Ex: contas corrente, clientes);
- Programas de aplicação (Ex: Creditar/debitar uma conta; Acrescentar conta nova; Recuperar Saldo; Emitir Extratos etc.)

Ao longo do tempo novos programas precisarão ser adicionados (possivelmente por programadores diferentes, usando linguagens diferentes) e novos arquivos precisarão ser criados (possivelmente com formatos diferentes dos anteriores).

Os problemas dos sistemas de arquivos convencionais se originam de:

**a) Inconsistência e Redundância de Dados:**

Uma mesma informação poderá estar em mais de um arquivo.

A redundância leva a:

- Inconsistência dos dados (cópia de um mesmo dado com valores diferentes);
- Maior custo de armazenamento (Ex: endereço do cliente poderá estar em dois arquivos distintos: cliente e contas correntes).

O SGBD resolve esse problema, pois ele cria um modelo relacional onde o dado é um só.

**b) Dificuldade no acesso de dados:**

Acesso não previsto implica em:

- Desenvolvimento de um novo programa de aplicação;
- Ou trabalho manual.

Ex: Existe um arquivo com os nomes dos clientes. Listar os nomes dos clientes que vivem na área da cidade e cujo CEP é conhecido.

O SGBD possui uma ferramenta de CONSULTA própria que permite acesso aos dados. Além disso, o próprio usuário define o tipo de consulta que deseja, através de uma ferramenta específica.

**c) Isolamento dos dados:**

É difícil escrever novos programas quando os dados a serem manipulados estão espalhados, sem critérios adequados, em vários arquivos e podendo estar em formatos diferentes de um arquivo para outro.

**d) Múltiplos usuários:**

Múltiplos usuários atualizam dados simultaneamente.

Ex: saldo = 500; saque do cliente A = 50; saque do cliente B = 100;

Cliente A: lê saldo (saldo = 500);  
Cliente B: lê saldo (saldo = 500);  
Cliente A: calcula resultado (novo saldo = 450);  
Cliente B: calcula resultado (novo saldo = 400);  
Cliente A: armazena resultado (saldo = 450);  
Cliente B: calcula resultado (saldo = 400);  
Porém, o valor final do saldo é 400 ao invés de 350.

No SGBD o dado é único, e quando o primeiro usuário solicita o dado, ele trava o acesso ao dado de uma outra possível atualização (um de cada vez).

**e) Problemas de segurança:**

Controle de acesso aos dados. Nem todo usuário pode ser autorizado a acessar todos os dados.

**f) Integridade de dados:**

Restrições de integridade, ex: Saldo  $\geq 10,00$  (não pode assumir valor menor).  
Garantia da integridade fica difícil de ser mantida nos programas de aplicação porque ela precisaria ser implementada em todos os programas que atualizam saldo.

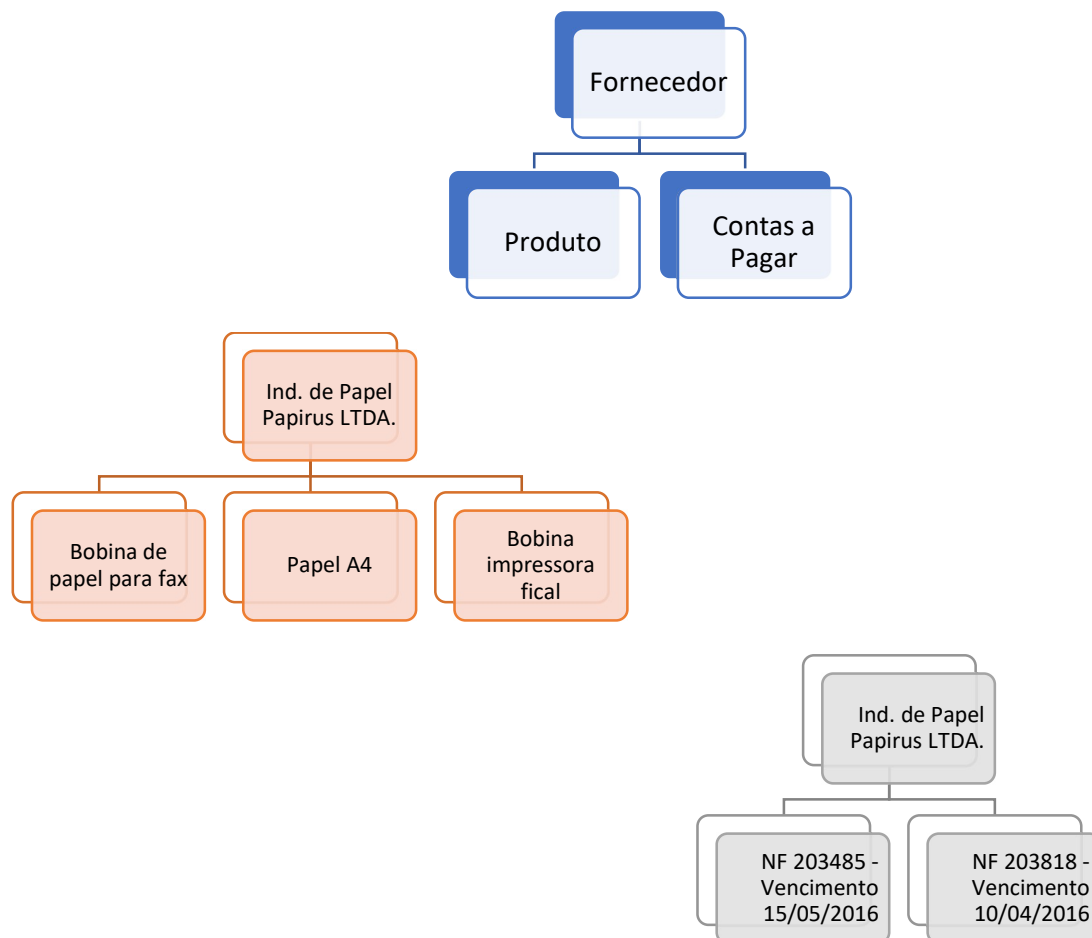
No SGBD as restrições são colocadas diretamente sobre os dados. Se por um acaso a restrição mudar, é só mudar em um único lugar e uma única vez.

## 2.2. Classificação de Banco de Dados

Os bancos de dados podem ser classificados por vários critérios, seguem os principais.

### Quanto ao Modelo de Dados

- Banco de Dados Hierárquico  
É considerado o primeiro tipo de banco de dados de que se têm notícias. Em sistemas de banco de dados hierárquicos encontramos dois conceitos fundamentais: registros e relacionamento pai-filho. O registro é uma coleção de valores que representam informações sobre uma dada entidade de um relacionamento. Os registros que antecedem outros na hierarquia têm a denominação PAI e os registros que o sucedem são chamados de FILHOS.



Podemos perceber que o esquema hierárquico é estruturado em árvore, em que o tipo de registro corresponde a um nó. Sendo assim, temos nós pais e nós filhos.

Neste tipo de banco de dados devemos nos referir a um relacionamento pai-fillho como um par ordenador, no qual temos o tipo de registro PAI e o tipo de registro FILHO, como nos exemplos (Fornecedor, Produto) e (Fornecedor, Contas a Pagar).

O primeiro sistema de banco de dados hierárquico que se tem conhecimento é o IMS da IBM, desenvolvido no fim da década de 1960.

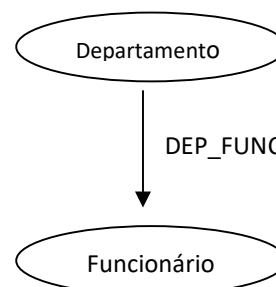
- Banco de Dados de Rede

Também conhecido como CODASYL ou sistemas DBTG. Estes sistemas são largamente utilizados em computadores de grande porte e à primeira vista se parecem com os sistemas hierárquicos, mas permitem que um mesmo registro participe de vários relacionamentos devido a eliminação da hierarquia. Outra característica que os diferencia do modelo hierárquico é a possibilidade de acesso direto a um determinado registro/nó da rede, enquanto no sistema hierárquico é necessário passar pela raiz obrigatoriamente.

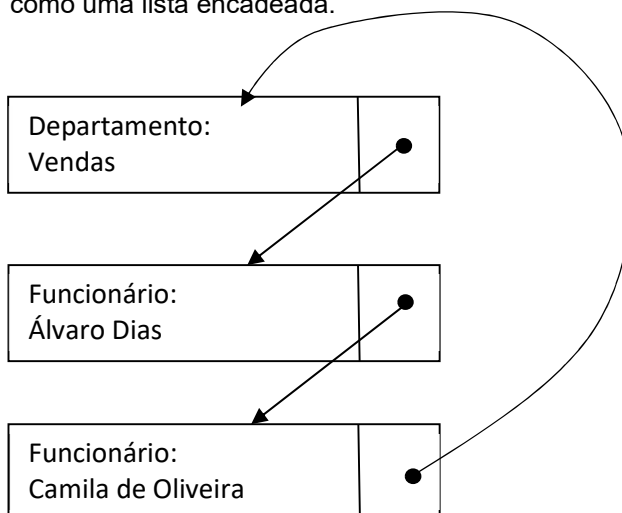
Os comandos de manipulação de registros devem ser incorporados a uma linguagem de programação hospedeira, sendo mais utilizada normalmente a COBOL, mas outras como Pascal e FORTRAN também podem ser empregadas. As duas estruturas fundamentais de um banco de dados de rede são os registros (records) e os conjuntos (sets). Os registros contêm dados relacionados e são agrupados em tipos de registros que armazenam os mesmos tipos de informações.

Os conjuntos são a forma de representação dos relacionamentos entre os diversos tipos de registros na forma 1:N.

Tipo de conjunto	DEP_FUNC
Tipo de registro proprietário	Departamento
Tipo de registro membro	Funcionário



Tecnicamente pode-se dizer que o registro proprietário possui um ponteiro que “aponta” para um registro membro. Esse registro, que é o primeiro do conjunto, “aponta”, para outro que também se relaciona com o mesmo registro proprietário, como uma lista encadeada.





- Banco de Dados Relacionais**  
 Um banco de dados relacional se caracteriza pelo fato de organizar os dados em tabelas (relações), formada por linhas e colunas. Estas tabelas relacionam as informações referentes a um mesmo assunto de modo organizado.  
 O Dr. Edgar F. Codd formulou os princípios básicos do sistema de banco de dados relacional em 1968, baseando-se na teoria de conjuntos e na álgebra relacional.  
 Um banco de dados relacional permite que tenhamos informações divididas entre várias tabelas. Porém, certas informações de uma tabela são obtidas a partir de outras. Para tanto, é necessário um campo comum entre elas.

Categoria de Produtos	
Código	Descrição
100	Eletrônicos
200	Brinquedos
300	Móveis

Fornecedores						
Código	Nome	Endereço	Bairro	Cidade	Estado	Telefone
1	ABC Móveis Domésticos	Rua Azul, 10	Centro	São Paulo	SP	12349876
2	Brinquedos Educar	Rua das Flores, 49	Jardim América	Osasco	SP	56782367

Produto				
Código	Nome	Estoque	CodCat	CodForn
234	Jogo de Dominó	15	200	2
567	Mesa vidro	2	300	1

- Banco de Dados Orientados a Objetos**  
 Surgiu em meados de 1980, em virtude da necessidade de armazenamento de dados que não era possível com os sistemas relacionais devido aos seus limites. Podemos citar como exemplo os sistemas de geoprocessamento (SIG) e CAD/CAM/CAE, que são baseados em tipos de dados complexo.  
 Basicamente, o modelo de dados orientado a objetos é caracterizado pela definição de banco de dados por meio de objetos, com suas propriedades e operações. Isso significa que um registro é mais parecido com uma classe definida na linguagem Java do que com uma tabela.

### Quanto ao Número de Usuários

- Monousuário**  
 Permite que apenas um usuário por vez acesse o banco de dados. São sistemas mais antigos e direcionados a um uso pessoal, como nos softwares dBASE III, dBASE IV, FoxBase e FoxPro, muito difundidos nas décadas de 1980 e 1990.
- Multiusuário**  
 Suporta o acesso de vários usuários ao mesmo tempo. Este suporte é possível graças ao controle de concorrência.

### Quanto à Localização

- **Centralizado**  
O sistema de gerenciamento e o banco de dados estão localizados em uma única máquina, denominada servidor de banco de dados. Mesmo sendo centralizado, ele pode ter suporte a acesso concorrente de vários usuários.
- **Distribuído**  
É caracterizado por ter o sistema gerenciador e o banco de dados armazenados em diferentes máquinas. O próprio SGBD pode ser distribuído em mais de um computador, todos interligados em rede.  
Uma tendência que vem crescendo nos últimos tempos é distribuir na arquitetura de SGBD vários bancos de dados de fornecedores diferentes, mas que são acessados por um único SGBD. São os chamados SGBDs Heterogêneos.

Independentemente de serem centralizados ou distribuídos, os SGBDs atualmente em uso trabalham dentro da arquitetura cliente/servidor.

## 2.3. Linguagem de Definição de Dados (DDL)

Através da DDL se expressa um conjunto de definições que especificam um esquema.

- a) O resultado da compilação/execução de instruções DDL afeta diretamente o dicionário de dados (tabelas no banco). O dicionário de dados é um arquivo especial que contém “dados” acerca dos dados “reais”, e é consultado pelo sistema para acesso ao BD.
- b) A estrutura de armazenamento e os Métodos de Acesso são também especificados por um conjunto de definições DDL. A compilação dessas definições resulta em um conjunto de instruções que especificam os detalhes de implementação dos esquemas.
- c) A DDL poderá ainda prover facilidades para descrever os domínios e as restrições de integridade.

Exemplo de DDL:

```
CREATE TABLE FUNCIONARIO (  
    NUMFUNC          SMALLINT NOTNULL,  
    NOME              VARCHAR (20),  
    SEXO              CHAR (1),  
    CPF               CHAR (11),  
    SALARIO           DECIMAL (8,2),  
    CODDEPTO          CHAR (3) NOT NULL  
);  
  
ALTER TABLE FUNCIONARIO ADD COLUMN DTNASC DATE;  
  
CREATE INDEX SALARIO_BUSCA ON FUNCIONARIOS (SALARIO DEC);
```

## 2.4. Linguagem de Manipulação de Dados (DML)

Por manipulação de dados entendemos:

- Recuperação de dados;
- Inserção de dados;
- Remoção de dados;
- Modificação de dados.

A DML é a linguagem que viabiliza o acesso ou a manipulação dos dados de forma compatível ao modelo de dados apropriado. São basicamente dois tipos:

- DML Procedural: exige que o usuário especifique quais dados são necessários, e como obtê-los;
- DML Não-Procedural: exige que o usuário especifique quais dados são necessários, sem especificar como obtê-los. Mais fácil de aprender a usar. Pode gerar códigos não muito eficientes.

Exemplos de DML em SQL

```
SELECT rua, cidade  
FROM cliente  
WHERE nome = 'Amanda';
```

## 2.5. Linguagem de Controle de Dados (DCL)

Apresenta uma série de comandos para controlar o acesso aos dados, usuários e grupos. Dentre alguns exemplos de comandos DCL destacam-se os comandos GRANT e REVOKE.

## 2.6. Usuários do Banco de Dados

**Programadores de Aplicação:** Utilizam comandos DML em programas escritos em linguagem do host (PASCAL, C, etc.). Esses programas são chamados de programas de aplicação. Os comandos DML são pré-compilados, isto é, convertidos em chamadas normais de rotinas na linguagem do host.

**Usuários Simples:** Usam programas de aplicação existentes (invocam programas em código objeto).

**Usuários Especiais (sofisticados):** Formulam requisições escrevendo suas consultas em Linguagem de Consulta (não escreve programas) - usam DML.

**Administrador do Banco de Dados (DBA):** Pessoa que possui o controle central dos dados e programas que acessam os dados.

Principais funções do DBA:

- Definição do esquema conceitual: criação do esquema original do Banco de Dados;
- Definição da estrutura de armazenamento e do método de acesso;
- Modificações do esquema e da organização física do Banco de Dados;
- Concessão de autorização para acesso aos dados: Define o nível de visão para os usuários;
- Especificação das restrições de integridade: Essas restrições são consultadas pelo gerenciador do Banco de Dados sempre que acontece uma atualização.

OBS: Uma pessoa, ou um grupo de pessoas, conhecedora de Sistemas de Banco de Dados e das aplicações deve ser competente, cuidadosa e responsável; de ações equilibradas e honestas.

### 3. O Projeto de Banco de Dados

O objetivo da modelagem de dados é transmitir e apresentar uma representação única, não redundante e resumida, dos dados de uma aplicação.

O projeto de um sistema de informações é atividade complexa que inclui planejamento, especificações e desenvolvimento de vários componentes.

O que se propõe é situar a sequência dessas atividades em uma ordem que possa resultar ganhos de produtividade e confiabilidade dos sistemas desenvolvidos, eliminando sensivelmente as falhas de sistemas após sua implantação.

Desafortunadamente as metodologias de projeto de banco de dados, para sistemas de aplicação, apesar de já serem muito populares entre a comunidade técnica, não são utilizadas corretamente.

Em várias organizações os profissionais utilizam-se de pequenas técnicas pessoais, ou ainda pior, de uma inexistência completa de metodologia para esses projetos e com distorções na utilização das técnicas, sendo esta uma das maiores causas de falhas nos sistemas de informação desenvolvidos, mesmo com a existência de modelos de dados.

A utilização da abordagem correta metodologias orientadas a modelagem de banco de dados envolve a estruturação nos três níveis de visão de dados existentes, ou seja, três etapas na execução de um projeto: conceitual, lógico e físico.

#### 3.1. Modelo Conceitual

Representa, descreve a realidade do ambiente do problema, constituindo-se em uma visão global dos principais dados e seus relacionamentos (estruturas de informação), completamente independente dos aspectos de sua implementação tecnológica.

O objetivo do modelo conceitual é descrever de forma simples e facilmente compreendida pelo usuário final as informações de um contexto de negócios, as quais devem ser armazenadas em um banco de dados.

Exemplo de um modelo conceitual textual:

##### 1) Cadastro de Clientes

Dados necessários: nome completo, tipo de pessoa (física ou jurídica), endereço, bairro, cidade, estado, telefone, e-mail, nome de contato.

##### 2) Pedido

Dados necessários: código do produto, quantidade, código do cliente, código do vendedor.

Registra QUE dados podem aparecer no banco, mas não registra COMO estes dados estão armazenados no SGBD.

#### 3.2. Modelo Lógico

Ele somente tem início após a criação do modelo conceitual, pois agora vamos considerar uma das abordagens possíveis da tecnologia de SGBD (relacional, hierárquico, rede ou orientado a objetos) para estruturação e estabelecimento da lógica dos relacionamentos existentes entre os dados definidos no modelo conceitual.

A elaboração direta de um modelo lógico de dados, independentemente de já sabermos a abordagem para banco de dados, para a qual estamos realizando um projeto, leva à vinculação tecnológica de nosso raciocínio, perturbando a interpretação pura e simples de um contexto. Sempre que analisamos um contexto sob a óptica tecnológica, temos a tendência de sermos técnicos demais, distorcendo a realidade, conduzindo às restrições da tecnologia empregada, o que sempre, e já estatisticamente

Compreende uma descrição das estruturas que serão armazenadas no banco e que resulta em uma representação gráfica dos dados de uma maneira lógica, inclusive nomeando os componentes e ações que exercem uns sobre os outros.

comprovado, leva a erros de interpretação da realidade, criando assim modelos de dados que não possuem aderência ao minimundo descrito.

Exemplo gráfico de um modelo lógico:



Exemplo textual de um modelo lógico:

```
TipoProduto (Codigo, Descricao)
Produto (Codigo, Descricao, Preco, CodTipoProd)
CodTipoProd referencia TipoProduto
```

A técnica de modelagem mais difundida é a abordagem entidade-relacionamento (ER). Nesta técnica, um modelo conceitual é usualmente representado através de um diagrama, chamado diagrama entidade-relacionamento (DER).

### 3.3. Modelo Físico

O modelo físico será construído a partir do modelo lógico e descreve as estruturas físicas de armazenamento de dados, tais como:

- Tipo e tamanho de campos;
- Índices;
- Domínio de preenchimento dos campos;
- Nomenclaturas;
- Exigência de conteúdos;
- Gatilhos; etc.

Esse modelo depende do SGBD que está sendo usado.

Esta é a etapa final do projeto de banco de dados, na qual será utilizada a linguagem de definição de dados do SGBD para a realização da sua montagem no dicionário de dados. Em ambiente de banco de dados relacional, denominamos script de criação do banco de dados o conjunto de comandos em SQL que será executado no SGBD para a criação do banco de dados correspondente.

Exemplo de um script:

```
CREATE TABLE TipoProduto (
    codigo      int,
    descricao   varchar(100),
    constraint pk_tipo primary key (codigo)
);

CREATE TABLE Produto (
    codigo      int,
    descricao   varchar(100),
    preco       numeric(10,2),
    codTipoProd int,
    constraint pk_prod primary key (codigo),
    constraint fk_prod_tipo foreign key (codTipoProd)
        reference TipoProduto (codigo)
);
```

Exemplo das tabelas em um banco de dados relacional:

**Tipo de produto**

Código	Descrição
1	Computador
2	Impressora

**Produto**

Código	Descrição	Preço	CódigoDoTipo
10	Desktop	1.200,00	1
20	Laptop	1.800,00	1
30	Impr. Jato Tinta	300,00	2
40	Impr. Laser	500,00	2

## Atividade Pontuada Teams – Lista 1

## 4. Modelo Entidade-Relacionamento

O Modelo Entidade-Relacionamento (MER), conhecido também como Diagrama de Entidades e Relacionamentos (DER), permite representar a estrutura do banco de dados de forma conceitual, ou seja, permite determinar quais dados armazenar e os relacionamentos entre esses dados. Do MER, pode-se derivar o modelo lógico relacional (dados vistos em tabelas), hierárquico (dados vistos em uma árvore genealógica), de rede (dados vistos em grafos) ou orientados a objetos (dados vistos como objetos).

Para isso, o MER utiliza-se de símbolos que permitem distinguir os elementos presentes em um banco de dados. Na literatura não existe um padrão universal para a criação do modelo. O que se vê é o mesmo diagrama sendo apresentado de formas diferentes por vários autores.

Originalmente, o MER foi concebido por Peter Chen, em 1976, e desde então vem sendo modificado e atualizado. Existem diversas propostas de extensão para o diagrama. Nesta disciplina adotaremos como padrão a notação de Chen (a simbologia mais empregada em banco de dados), mas apresentaremos algumas variações.

A elaboração do MER representa uma forma de validar a elaboração de um banco de dados antes de iniciar a criação de suas estruturas de armazenamento. Um projetista de banco de dados tem a oportunidade de debater com a equipe (ou até mesmo com os usuários) se o modelo está coerente e, principalmente, se irá atender às suas necessidades.

Ressalta-se que o MER não define o que será feito com os dados, isto é, ele define apenas os dados a serem armazenados e o relacionamento entre esses dados sem se preocupar com os procedimentos que serão aplicados aos dados.

### 4.1. Entidade (entidades-dados)

Para explicar o significado de entidades, vamos usar um exemplo comparativo. Considere um armário cheio de gavetas para guardar parafusos. Cada gaveta serve para guardar parafusos de certa medida e tamanho entre outras características. Uma entidade é como o armário usado para armazenar “coisas” de mesmas características. No MER, uma entidade se refere a uma representação usada para identificar “coisas” do mundo real que possuem as mesmas características. Quando escrevemos “coisas”, estamos considerando qualquer objeto do mundo real, seja ele concreto (produto, cliente, veículo, etc.) ou abstrato (viagem, empréstimo, pagamentos, etc.) que pode ser armazenado e manipulado em um banco de dados. (Observe a diferença para a Entidade Externa do DFD).



### 4.2. Atributos

Atributos das Entidades são os tipos de informações que o sistema precisa armazenar a respeito de um conjunto de entidades, ou seja, dados que o sistema precisa armazenar para que ele realize suas tarefas de processamento.

Por exemplo:

- a) Funcionário: id\_func, nome\_func, end\_func, CPF etc..
- b) Vendas: id\_vend, data, id\_func, valor, etc...
- c) Clientes: id\_cli, nom\_cli, tel\_cli, end\_cli, etc...

Serão representados no modelo da seguinte forma:



Os valores recebidos pelos atributos identificam cada uma das entidades do conjunto. Alguns autores denominam uma dada entidade determinada pelos valores dos atributos de instância de um conjunto de entidades.

Funcionário: João

Rua: XV, 200

4796-6895

226.652.988-45

#### 4.2.1. Atributo Chave Primária

É o atributo de um conjunto de entidades que recebe valores diferentes para entidades diferentes, ou seja, o seu valor identifica uma entidade de maneira única e vice-versa.

Na prática, pode-se escolher entre os atributos da entidade, um atributo que seja capaz de identificar unicamente um registro, por exemplo, na tabela funcionários pode-se utilizar o atributo CPF como chave primária. Também é possível criar um atributo que irá receber números inteiros sequenciais sob o controle do sistema.

Exemplo:

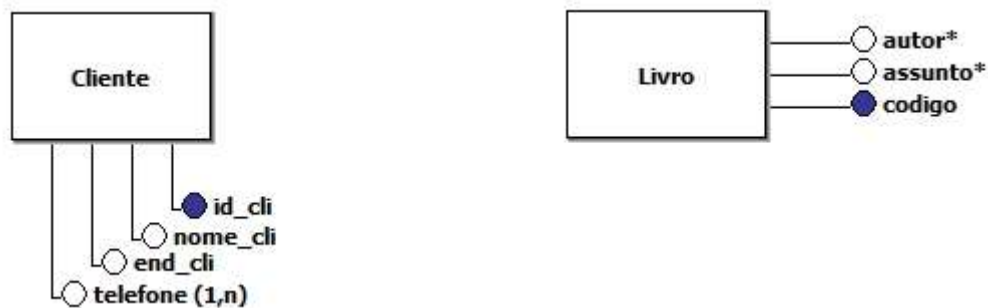


#### 4.2.2. Atributo Multivalorado

É um atributo que pode receber mais de um valor para uma mesma entidade (extensão do modelo de Chen). Poderá ser denotado por um \* ou (1,n).

Exemplo:





### 4.3. Relacionamentos

Relacionamento se refere a um fato que liga duas “coisas” no mundo real. Ou seja, um relacionamento pode interligar uma, duas ou mais entidades entre si. Vamos considerar o relacionamento entre as entidades Cliente e Pedido. A figura abaixo pode ser usada para representar o fato de que um Cliente faz Pedido. Na verdade, assim como o retângulo representa um conjunto de entidades, o losango representa um conjunto de relacionamentos, já que existem diversas ocorrências das mesmas entidades relacionadas.



No MER, o relacionamento é sempre designado por um verbo no interior do losango. No exemplo citado, o verbo **faz** descreve o relacionamento. A escolha do verbo deve ajudar a descrever a realidade do relacionamento. Como exemplo de outros relacionamentos entre entidades do mundo real, podemos citar:

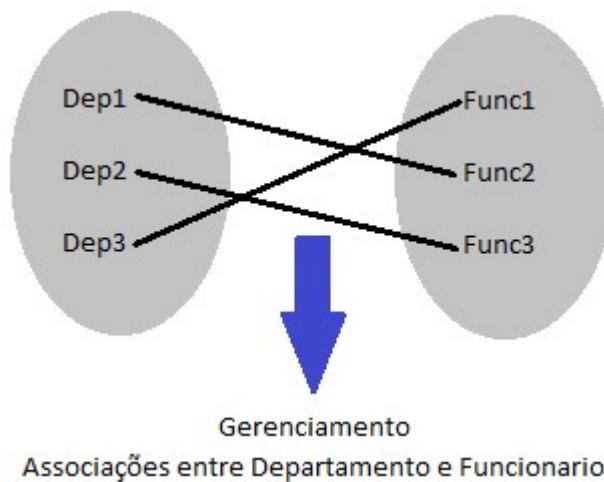
- Item **tem** Medicamento;
- Pedido **possui** Item;
- Estoque **contém** Medicamento;
- Pedido **gera** Movimento.

#### 4.3.1. Relacionamentos 1 : 1



O relacionamento é do tipo 1 : 1 se as regras da empresa forem :

- Um departamento é gerenciado por no máximo um funcionário.
- Um funcionário gerencia no máximo um departamento.



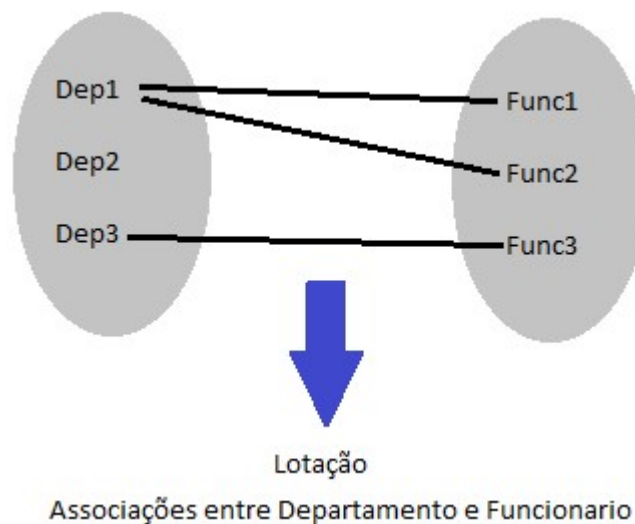
Existe um relacionamento entre departamento e funcionário que fará parte do modelo se o sistema precisar armazenar essa informação, ou seja, quais funcionários gerenciam determinados departamentos e/ou qual departamento é gerenciado por um dado funcionário.

#### 4.3.2. Relacionamentos 1 : N



Nesse caso subentende-se que:

- Um departamento lota N funcionários.
- Um funcionário é lotado em um departamento.



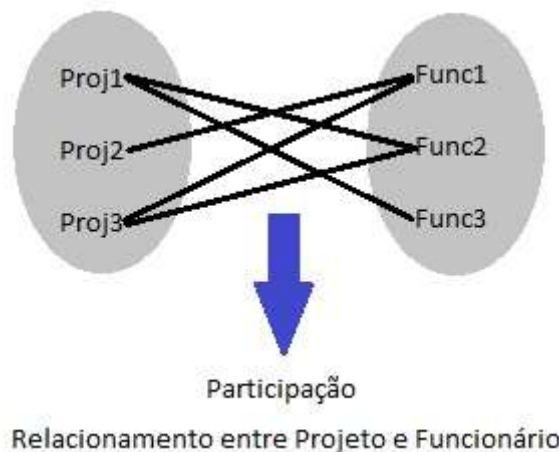
Existe relacionamento 1 para N entre departamento e funcionário, pois queremos saber quais funcionários trabalham em determinado departamento.

### 4.3.3. Relacionamentos N : N



Nesse caso subentende-se que:

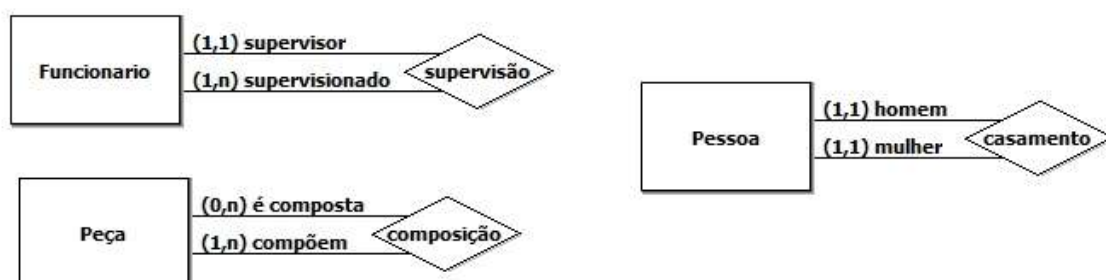
- Um projeto pode ter a participação N funcionários.
- Um funcionário pode participar de N projetos.



Existe relacionamento N para N para Projeto e Funcionario, pois queremos identificar quais funcionários participam de determinados projetos e quais projetos têm determinados funcionários, sendo que um funcionário pode participar de mais de um projeto, assim como o projeto pode ter mais de um funcionário.

### 4.3.4. Auto-Relacionamento

Representa uma associação entre ocorrências de uma mesma entidade e exige a identificação de papéis.

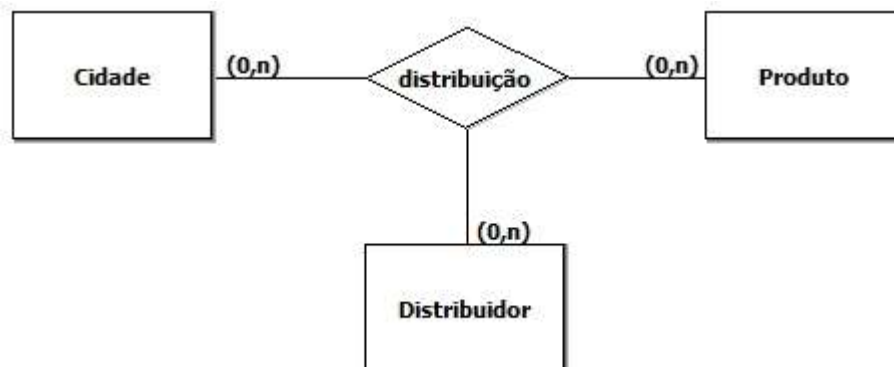


Neste caso subentendesse que:

- 1 funcionário supervisiona N funcionários.
- 1 funcionário é supervisionado por 1 funcionário.
- 1 peça poderá ser componente de várias outras peças.
- 1 peça poderá ter como componentes várias outras peças.
- 1 pessoa(homem) se casa com 1 pessoa(mulher).
- 1 pessoa(mulher) se casa com 1 pessoa(homem).

#### 4.3.5. Relacionamento “n”-ário

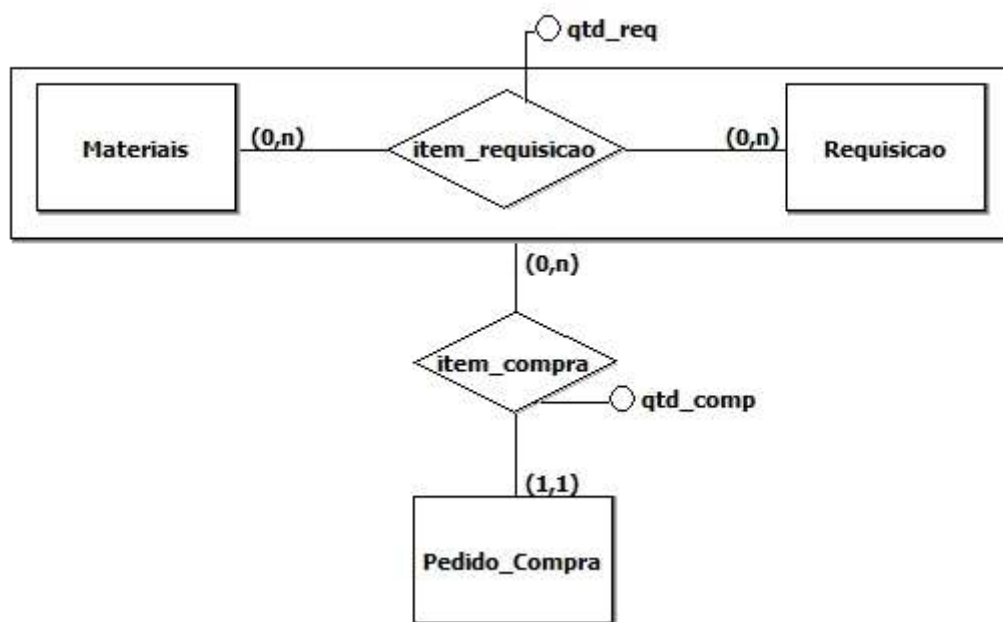
Representa uma associação entre “n” ocorrências de entidades.



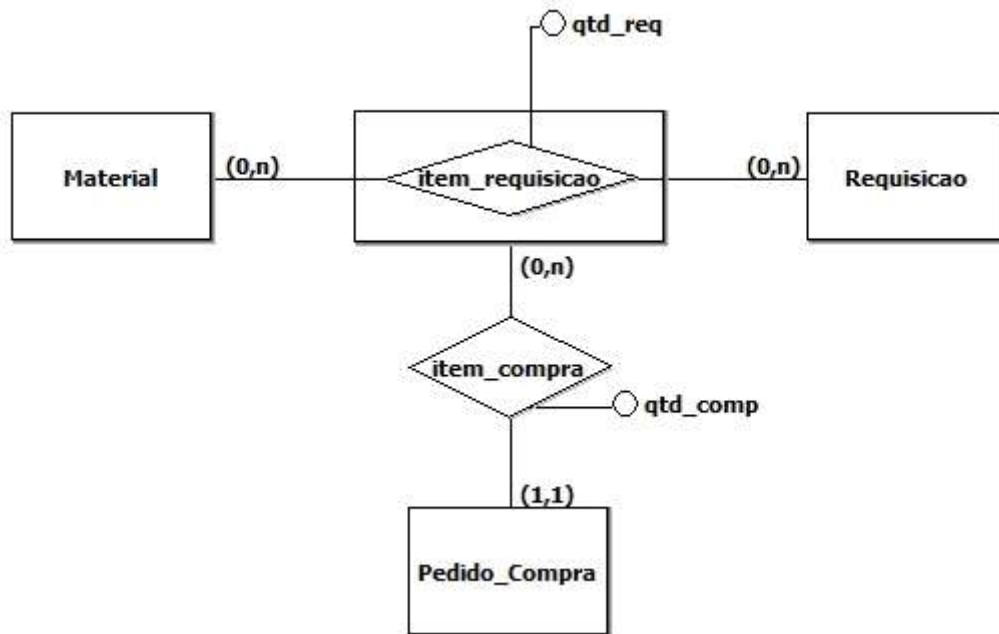
#### 4.3.6. Agregação (conjuntos complexos)

Dois, ou mais, conjuntos de entidades relacionadas podem ser agregadas e formar um conjunto complexo de entidades.

Formando um novo conjunto, eles podem se relacionar com outros conjuntos de entidades.



OU



Nesse exemplo Materiais e Requisições foram agrupados (ou seja, agregados) formando um conjunto complexo de entidades no qual cada entidade é composta por um par material e requisição que se relacionam de acordo com o item de requisição.

Esse par, representado pelo item de requisição, poderá formar um relacionamento com Pedido de compra através do relacionamento itens de compra.

Uma requisição pode ser uma relação de vários materiais (N : N).

As requisições de materiais geram pedidos de compra. Mas nem sempre isso acontece, pois uma parte do material requisitado poderá estar no almoxarifado. Portanto, os materiais relacionam-se com as requisições independentes dos pedidos de compra.

Um pedido de compra pode constar de vários materiais e se for necessário, para cada um deles, que se conheça em qual requisição ela constava, o pedido de compra deve então relacionar-se com ambos: material e requisição. (itens de requisição).

A agregação é um recurso de modelagem útil para alguns sistemas onde os relacionamentos dos tipos 1:1, 1:N e N:N não sejam suficientes para representar adequadamente as informações que o sistema precisa manter.

#### 4.3.7. Mapeamento do MER para o Modelo Relacional (MR)

O Modelo Relacional (MR) é um modelo de dados do nível lógico onde conjuntos de dados são representados por “tabelas” bidimensionais denominadas de “relações”. As “tabelas” no MR são organizadas em linhas e colunas. Cada “linha” representa um elemento do conjunto de dados. Cada “item de dados” de um elemento é representado pelo valor que se encontra no cruzamento da linha correspondente ao elemento com a coluna correspondente ao item.

O esquema de uma tabela é constituído pelo: Nome da Tabela + Nomes de suas colunas.

Exemplo:

Funcionario( CPF, Nome, End, Telef, Salario )

**Funcionario**

<u>CPF</u>	Nome	End	Telef	Salario
12345678910	Maria da Silva	Rua dos Sonhos, 12	47474747	12000,00
98765432109	José Maria	Av. Brasil, 100	47984798	10000,00
76540987123	Vera Lúcia	Rua dos Colibris, 54	47901234	10000,00

O projeto do BD tem por objetivo organizar os dados, na forma conveniente para serem implementados utilizando-se um gerenciador de BD. Se os dados forem modelados sob a forma de Entidades e Relacionamento e depois mapeados criteriosamente para tabelas (MR), as tabelas resultantes organizam os dados de forma conveniente para quase todos os tipos de sistema. Dessa maneira, cada tabela estará em um nível de qualidade elevada, sem informações repetidas, com facilidade para alterar, inserir e retirar dados, ou seja, no mínimo na 3ª Forma Normal.

Quando se usa um gerenciador de BD Relacional para implementar as relações, o MER deve ser mapeado para o Modelo Relacional (MR), ou seja, para tabelas.

Quando não se utiliza um BD Relacional, por exemplo, um sistema de arquivos, deve-se acrescentar o projeto de arquivos para então realizar a implementação.

### Chave Primária de uma Tabela (ou simplesmente Chave)

Uma coluna, ou uma concatenação de colunas, é denominada chave primária, se para uma linha qualquer da tabela não existir outra linha com o mesmo valor nessa coluna (ou na concatenação de colunas).

O valor da chave primária identifica uma linha da tabela, ou seja, identifica um elemento de forma única.

Exemplo:

Funcionário(CPF, Nome, End, Telef, Salario)

Chave primária

**Funcionario**

CPF	Nome	End	Telef	Salario
12345678910	Maria da Silva	Rua dos Sonhos, 12	47474747	12000,00
98765432109	José Maria	Av. Brasil, 100	47984798	10000,00
76540987123	Vera Lúcia	Rua dos Colibris, 54	47901234	10000,00

Cada CPF identifica unicamente um funcionário. Portanto, não é permitida a repetição deste valor dentro da tabela

### Chave estrangeira

A coluna de uma tabela que contém valores de chave primária de outra tabela é denominada "chave estrangeira". Como o valor da chave representa uma entidade, na coluna chave estrangeira ele indica que existe o relacionamento dessa entidade com uma outra entidade cujos dados estão na mesma linha.

Exemplo:

Funcionário( CPF, Nome, End, Telef, Salario)

Dependente( Id\_dep, Nome\_dep, ..... , Id\_Func)

Chave estrangeira

### Funcionario

CPF	Nome	End	Telef	Salario
12345678910	Maria da Silva	Rua dos Sonhos, 12	47474747	12000,00
98765432109	José Maria	Av. Brasil, 100	47984798	10000,00
76540987123	Vera Lúcia	Rua dos Colibris, 54	47901234	10000,00

### Dependente

Id_dep	Nome_dep	.....	Id_Func
1	Carlos Eduardo Lopes		76540987123
2	Helena da Silva		76540987123
3	Renato Cavalcante		12345678910
4	Carla Rodrigues		98765432109

### Modelo ER (Entidade Relacionamento) X Modelo Relacional (Tabelas)

Modelo ER		Modelo Relacional
Conjunto de entidades	→	Relação (ou tabela)
Entidade	→	Linha
Atributo	→	Coluna
Atributo chave	→	Chave
Atributo multivalorado	→	Relação auxiliar
Relacionamento	→	Coluna de chave estrangeira ou tabela com colunas de chaves estrangeiras

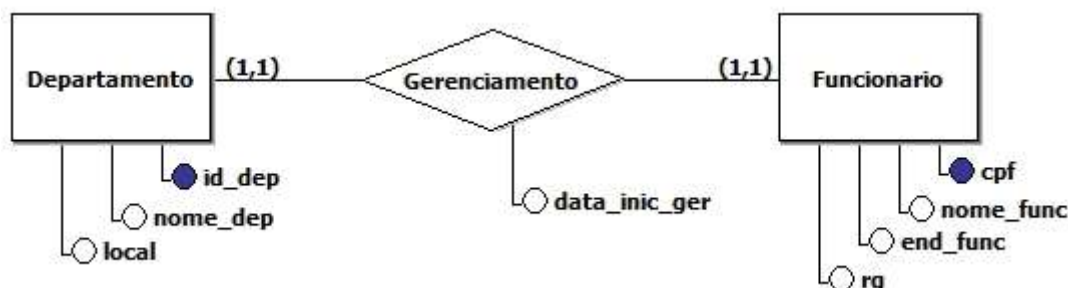
#### 4.3.7.1. Mapeamento do Relacionamento 1:1

Se os conjuntos de entidades puderem ser unidos, eles serão representados por uma única relação. Senão, cria-se uma tabela para cada conjunto de entidades (cada conjunto de entidades é mapeado para uma tabela).

Cria-se uma coluna de chave estrangeira em uma das tabelas (esta coluna conterá valores de chave da outra tabela).

Em qual tabela deve-se criar a chave estrangeira dependerá das formas de acesso para ganhar eficiência no processamento, ou da forma como o relacionamento acontece e assim economizar espaço da memória. Na prática essa otimização só faz sentido para grandes conjuntos de entidades e onde ter uma eficiência de processamento seja importante para a aplicação.

Se o relacionamento tiver atributos, para cada um deles será criado uma coluna adicional na tabela onde a chave estrangeira foi criada.



Opções de mapeamento:

Opção 1: Coluna de chave estrangeira na tabela Departamento

Funcionario (**cpf**, nome\_func, end\_func, rg)

Departamento (id\_dep, nome\_dep, local, data\_inic\_ger, **cpf\_func**)

Nesse caso, a coluna de chave estrangeira **id\_func** representa o relacionamento Gerenciamento e acrescentou-se também uma coluna correspondente ao atributo do relacionamento **data\_inic\_ger**.

Opção 2: Coluna de chave estrangeira na tabela Funcionario

Departamento (id\_dep, nome\_dep, local)

Funcionario (**cpf**, nome, end, rg, data\_inic\_ger, **id\_dep**)

Nesse caso, a coluna de chave estrangeira **id\_dep** representa o relacionamento Gerenciamento.

OBS:

a) Como o número de funcionários é muito maior do que o número de departamentos, a tabela Funcionario da opção 2 terá muitos espaços vazios nas colunas **id\_dep** e **data\_inic\_ger**.

b) Por outro lado, se algum dia um funcionário puder ser gerente de mais de um departamento (ou seja, o relacionamento se torna 1 : N) a solução 2 terá problemas, mas a solução 1 não.

#### 4.3.7.2. Mapeamento do Relacionamento 1 : N

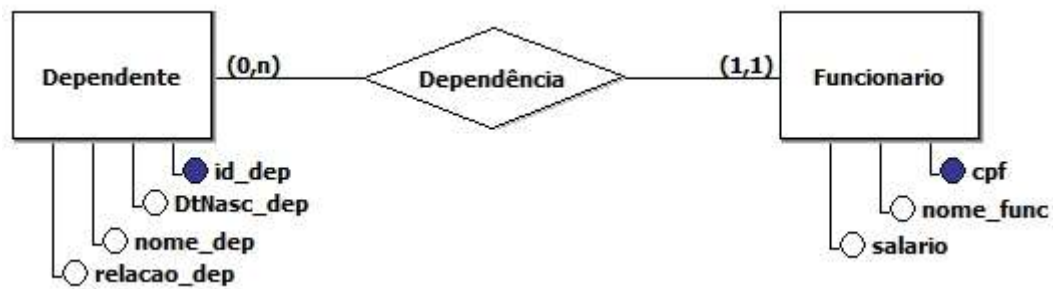
Opção 1: Mapeamento 1 : N para duas tabelas

Cada um dos conjuntos de entidades deverá ser mapeado para uma tabela. No exemplo abaixo para as tabelas: Funcionario e Dependente.

O relacionamento será mapeado para uma coluna de chaves estrangeiras na tabela correspondente ao lado N. No exemplo abaixo, o relacionamento Dependência foi mapeado para a coluna de chave estrangeira **num\_func** na tabela Dependente (para conter valores da chave primária de funcionário).

Se o relacionamento tiver atributos estes também serão mapeados para colunas da tabela do lado N.





### Funcionario

<u>CPF</u>	Nome_Func	Salario
340784879	Maria	4000
124510212	Antonio	3000
144875400	Pedro	3400
487878789	João	2500
157879877	José	4200

### Dependente

<u>Id_Dep</u>	Nome_Dep	Dt_Nasc_Dep	Relacao_dep	<u>Id_Func</u>
1	Joana	23/03/88	Filha	340784879
2	Rita	02/09/90	Filha	144875400
3	Antonia	31/07/78	Esposa	157879877

Funcionario (cpf, nome\_func, salario)

Dependente (id\_dep, nome\_dep, dt\_nasc\_dep, relacao\_dep, id\_func)

Opção 2: Mapeamento 1 : N para três tabelas

O relacionamento é mapeado para uma terceira tabela que conterá duas colunas de chaves estrangeiras. Uma chave estrangeira correspondente a cada conjunto de entidades. No exemplo abaixo, a tabela Dependência com as colunas de chaves estrangeiras id\_depend e id\_func.

Se o relacionamento tiver atributos, eles farão parte dessa terceira tabela.

Funcionario (cpf, nome\_func, salario)

Dependente (id\_dep, nome\_dep, data\_nasc\_dep, relacao\_dep)

Dependencia ( id\_depend, id\_func, id\_dep )

### Funcionario

<u>CPF</u>	Nome_Func	Salario
340784879	Maria	4000
124510212	Antonio	3000
144875400	Pedro	3400
487878789	João	2500
157879877	José	4200

### Dependencia

<u>Id_Depnd</u>	CPF	Id_Dep
100	340784879	1
101	144875400	2
102	157879877	3

### Dependente

<u>Id_Dep</u>	Nome_Dep	Dt_Nasc_Dep	Relacao_dep
1	Joana	23/03/88	Filha
2	Rita	02/09/90	Filha
3	Antonia	31/07/78	Esposa

OBS: Se um dia o relacionamento 1 : N for alterado para N : N, o segundo caso permite que isso seja feito alterando a rotina que implementa a restrição de integridade.

### 4.3.7.3. Mapeamento do Relacionamento N : N

Cria-se uma tabela correspondente ao relacionamento e com duas colunas de chaves estrangeiras, uma para cada conjunto de entidades que se relacionam. No exemplo abaixo, a tabela Participacao com suas colunas id\_proj e cpf\_func.

Se houver atributos no relacionamento eles também serão mapeados para colunas da tabela criada. No exemplo, a coluna periodo\_part.



Opção 1: cria-se uma chave primária auto-incremento

Projeto (id\_proj, nome\_proj, orcam\_proj)

Funcionario (cpf, nome\_func, salario)

Participacao (id\_participacao, periodo\_par, id\_proj, cpf\_func)

### Projeto

<u>id_proj</u>	nome_proj	orcam_proj
1	Projeto A	4000
2	Projeto B	3000
3	Projeto C	4200

### Participacao

<u>id_participacao</u>	periodo_par	id_proj	cpf_func
100	1 ano	1	340784879
200	6 meses	1	144875400
300	1 ano e 8 meses	3	340784879
400	3 anos	2	157879877

### Funcionario

<u>CPF</u>	Nome_Func	Salario
340784879	Maria	4000
124510212	Antonio	3000
144875400	Pedro	3400
487878789	João	2500
157879877	José	4200

Opção 2: cria-se uma chave primária composta por dois campos

Projeto (id\_proj, nome\_proj, orcam\_proj)


Funcionario (cpf, nome\_func, salario)

Participacao (id\_proj, cpf\_func, periodo\_par)

**Projeto**


<u>id_proj</u>	nome_proj	orcam_proj
1	Projeto A	4000
2	Projeto B	3000
3	Projeto C	4200

**Participacao**



<u>id_proj</u>	<u>cpf_func</u>	periodo_par
1	340784879	1 ano
1	144875400	6 meses
3	340784879	1 ano e 8 meses
2	157879877	3 anos

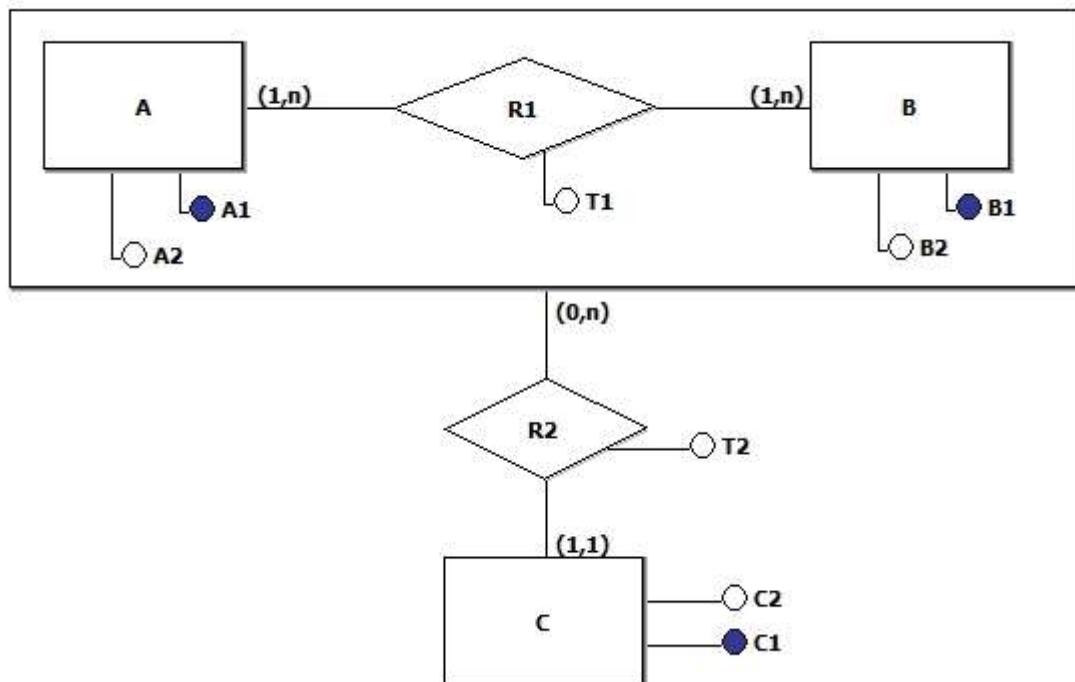
**Funcionario**



<u>CPF</u>	Nome_Func	Salario
340784879	Maria	4000
124510212	Antonio	3000
144875400	Pedro	3400
487878789	João	2500
157879877	José	4200

OBS: Na solução apresentada na opção 1, deverá ser criada uma restrição de unicidade para garantir que um mesmo relacionamento não seja inserido mais de uma vez. Já na solução apresentada na opção 2, este cuidado não é necessário.

#### 4.3.7.4. Mapeamento de Agregações



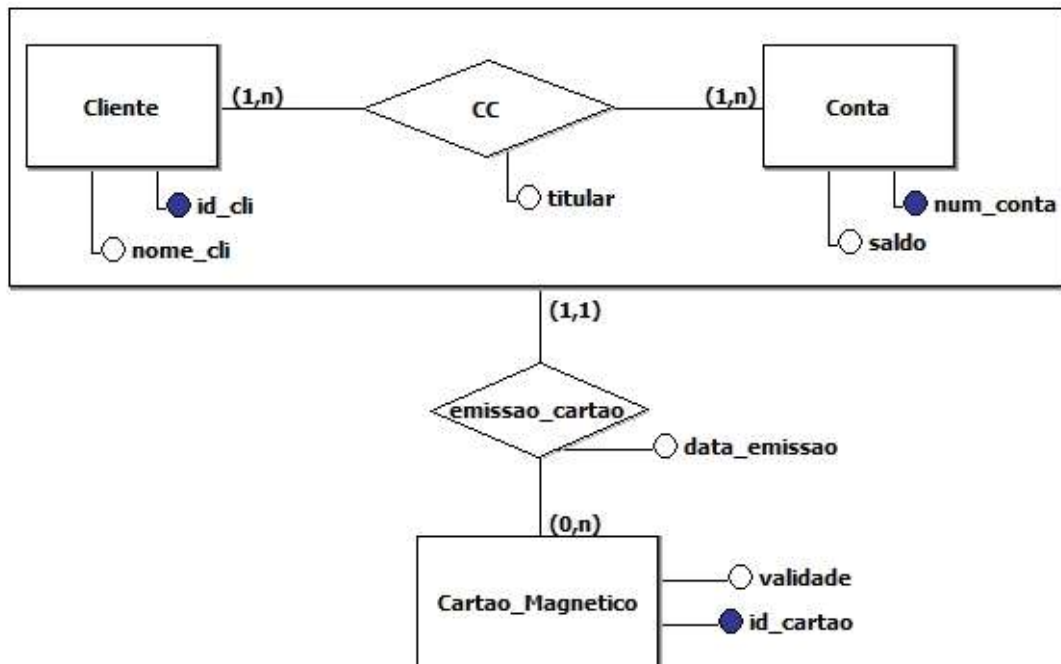
##### Opção 1: Mapeamento da agregação para cinco tabelas

Além das 3 tabelas correspondentes aos três conjunto de entidades A, B e C, cria-se uma relação correspondente ao relacionamento R1 com duas colunas de chaves estrangeiras correspondentes aos conjuntos de entidades “A” e “B” e uma coluna para cada atributo de R1.

Cria-se outra tabela correspondente ao relacionamento R2 com 3 colunas de chaves estrangeiras correspondentes aos conjuntos de entidades A, B e C e uma coluna para cada atributo de R2.

##### Opção 2: Mapeamento da agregação para quatro tabelas

Além das 3 tabelas correspondentes aos três conjuntos de entidades A, B e C, criar-se somente uma tabela para representar os relacionamentos R1 e R2. Essa tabela deverá conter 3 colunas de chaves estrangeiras correspondentes aos conjuntos de entidades A, B e C e uma coluna para cada atributo dos relacionamentos R1 e R2.



Opção 1 : (cinco tabelas)

Cliente (id\_cli,...)

Conta (num\_conta, ...)

CC( id\_cli, num\_conta, titular)

Cartao\_Magnetico (id\_cartao,...)

emissao\_cartao (id\_emissao, id\_cli, num\_conta, id\_cartao, data\_emissao)

Opção 2 : (quatro tabelas)

Cliente (id\_cli,...)

Conta (num\_conta, ...)

Cartao\_Magnetico (id\_cartao,...)

CCC (id\_ccc, id\_cli, num\_conta, id\_cartao, titular, data\_emissao)

OBS: Para optar por umas das possíveis soluções deve-se considerar:

- Tipos (ou formas) de processamento
- Frequência do processamento
- Quantidade de dados (tamanho das tabelas)

#### 4.3.7.5. Mapeamento de Atributos Multivalorados

Cria-se uma tabela correspondente a cada atributo multivalorado acrescentando uma coluna de chave estrangeira correspondente à entidade.

Exemplo:



Livro (id\_livro, titulo)

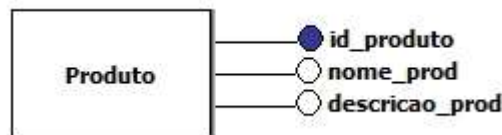
Autor (id\_autor, nome\_autor, id\_livro)

Assunto (id\_assunto, nome\_assunto, id\_livro)

#### 4.3.7.5. Mapeamento de Atributos Extensos

Um atributo extenso de um conjunto de entidades, por exemplo, um texto, poderá tornar lenta a manipulação constante dos dados da tabela que contém esse atributo em uma coluna e/ou ocupar maiores espaços de memória se ele tiver que ser armazenado diversas vezes. Para melhorar essa situação cria-se uma tabela adicional para o atributo extenso incluindo nela uma coluna chave correspondente ao atributo. No exemplo, a tabela Descrição\_Produto. Adiciona-se uma coluna de chave estrangeira na tabela correspondente à entidade para conter valores de chave correspondente ao atributo extenso. No exemplo, a coluna Id\_Descr\_Prod.

Exemplo 1:



Considerar que o conjunto de entidades Produto possui um atributo extenso descrição\_prod. Ao invés de ser mapeado para uma tabela como:

Produto (id\_prod, nome\_prod, descricao\_prod)

Cria-se uma tabela Descricao\_Produto e adiciona-se uma coluna de chave estrangeira na tabela Produto:

Produto (id\_prod, nome\_prod, descricao\_prod)  
Descricao\_produto (id\_prod, texto\_descricao)

#### 4.3.7.6. Mapeamento de Auto-Relacionamento

Além da tabela correspondente ao conjunto de entidades, cria-se uma tabela correspondente ao relacionamento entre entidades desse conjunto. Esta tabela terá duas colunas para conter os valores de chaves das entidades que se relacionam.

Se o relacionamento tiver atributos, eles darão origem às colunas da segunda tabela criada.



Produto (cod\_produto, descricao)  
Componente (produto, matéria, quantidade)

## Atividade Pontuada Teams – Lista 2

## 5. Normalização

Antes de entender o que é a Normalização de um banco de dados, é necessário entender o que é a chamada **Dependência Funcional**.

A dependência funcional (DF) é um conceito muito importante em banco de dados relacionais e que consiste em uma restrição entre dois conjuntos de atributos de uma mesma entidade/relação.

Uma dependência funcional é representada pela expressão  $X \rightarrow Y$ , em que X e Y são subconjuntos de atributos de uma relação qualquer. Isso impõe uma restrição na qual um componente Y de uma tupla é dependente de um valor do componente X (ou é determinado por ele). Do mesmo modo, os valores do componente X determinam de forma unívoca os valores do componente Y. Resumindo, Y é dependente funcionalmente de X.

Ex:

MatriculaAluno	CodigoCurso	CodigoDisciplina	NomeAluno	DataMatricula	NomeCurso	NomeDisciplina	NotaProva
----------------	-------------	------------------	-----------	---------------	-----------	----------------	-----------

É possível estabelecer 4 DF:

- $\text{MatriculaAluno} \rightarrow \{\text{NomeAluno}, \text{DataMatricula}\}$
- $\text{CodigoCurso} \rightarrow \text{NomeCurso}$
- $\text{CodigoDisciplina} \rightarrow \text{NomeDisciplina}$
- $\{\text{MatriculaAluno}, \text{CodigoCurso}, \text{CodigoDisciplina}\} \rightarrow \text{NotaProva}$

A **normalização** é um processo de refinamento do esquema de banco de dados, procurando eliminar possíveis redundâncias (dados repetidos em entidades), sanar problemas de dependências parciais entre atributos e reduzir ao mínimo as anomalias de inclusão, alteração e exclusão.

Tabela Aluno		
Nome	Atividade	Taxa
José	Musculação	30,00
Pedro	Judô	35,00
Manoel	Judo	35,00

Anomalia de exclusão: ao excluir a tupla do aluno José, perdemos, além do nome do aluno, as informações referentes à atividade Musculação.

Anomalia de inclusão: ao implantar uma nova atividade, não podemos inseri-lo até que um aluno tenha a disposição de fazê-lo.

Anomalia de alteração: como Judô foi grafado de diferentes formas, teremos dificuldade na atualização do valor a ser cobrado como taxa.

### 5.1. Primeira Forma Normal – 1FN

Uma tabela está na primeira forma normal se e somente se todas as colunas possuem um único valor, e não existem grupos repetitivos (colunas) em uma linha ou atributos compostos.

Aplicar a 1FN consiste em retirar da estrutura os elementos repetitivos, ou seja, aqueles dados que podem compor uma estrutura do tipo vetor.



Tabela-Pedido												
NumPedido	CodCliente	Cliente	Endereco	Cidade	UF	CodProduto	NomeProduto	Qtd	ValorU	ValorT	CodVendedor	Vendedor
347	100	Francisco· Moura	Av. Nove-de- Julho, 19 - Jd. Paulista	São- Paulo	SP	1234	Caderno-200-f	2	8,00	16,00	23	Alvaro
347	100	Francisco· Moura	Av. Nove-de- Julho, 19 - Jd. Paulista	São- Paulo	SP	5678	Caneta-Azul	1	1,50	1,50	23	Alvaro
347	100	Francisco· Moura	Av. Nove-de- Julho, 19 - Jd. Paulista	São- Paulo	SP	9876	Caneta- Vermelha	1	1,50	1,50	23	Alvaro
347	100	Francisco· Moura	Av. Nove-de- Julho, 19 - Jd. Paulista	São- Paulo	SP	5432	Papel-Sulfite- A4	1	12,00	12,00	23	Alvaro
347	100	Francisco· Moura	Av. Nove-de- Julho, 19 - Jd. Paulista	São- Paulo	SP	1357	Envelope-A4	10	0,50	5,00	23	Alvaro

A tabela Pedido não está na 1FN por possuir valores repetidos em várias tuplas.

Colocando na 1FN

Tabela Pedido							
NumPedido	CodCliente	Cliente	Endereco	Cidade	UF	CodVendedor	Vendedor
347	100	Francisco Moura	Av. Nove de Julho, 19 – Jd. Paulista	São Paulo	SP	23	Álvaro

Tabela ItemPedido						
NumPedido	CodProduto	NomeProduto	Qtd	ValorU	ValorT	
347	1234	Caderno 200 fl	2	8,00	16,00	
347	5678	Caneta Azul	1	1,50	1,50	
347	9876	Caneta Vermelha	1	1,50	1,50	
347	5432	Papel Sulfite A4	1	12,00	12,00	
347	1357	Envelope A4	10	0,50	5,00	



## 5.2. Segunda Forma Normal – 2FN

Consiste em retirar das estruturas de dados que possuem chaves compostas todos os campos que são funcionalmente dependentes de somente alguma parte dessa chave.

Podemos afirmar que uma estrutura está na 2FN, se ela estiver na 1FN e não possuir campos que sejam funcionalmente dependentes de parte da chave.

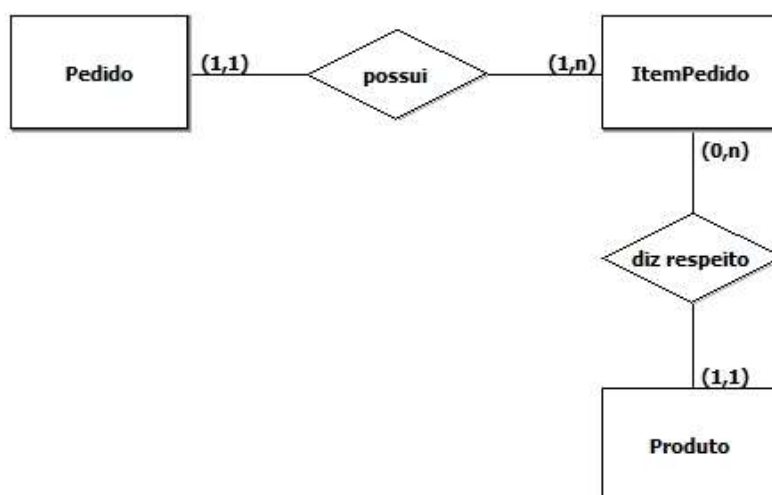
Tabela ItemPedido						
NumPedido	CodProduto	NomeProduto	Qtd	ValorU	ValorT	
347	1234	Caderno 200 fl	2	8,00	16,00	
347	5678	Caneta Azul	1	1,50	1,50	
347	9876	Caneta Vermelha	1	1,50	1,50	
347	5432	Papel Sulfite A4	1	12,00	12,00	
347	1357	Envelope A4	10	0,50	5,00	

A tabela ItemPedido, apesar de estar na 1FN, não está na 2FN, pois as colunas NomeProduto, e ValorU dependem funcionalmente de parte da chave primária (CodProduto)

Tabela Pedido							
NumPedido	CodCliente	Cliente	Endereco	Cidade	UF	CodVendedor	Vendedor
347	100	Francisco Moura	Av. Nove de Julho, 19 – Jd. Paulista	São Paulo	SP	23	Álvaro

Tabela ItemPedido			
NumPedido	CodProduto	Qty	ValorT
347	1234	2	16,00
347	5678	1	1,50
347	9876	1	1,50
347	5432	1	12,00
347	1357	10	5,00

Tabela Produto		
CodProduto	NomeProduto	ValorU
1234	Caderno 200 fl	8,00
5678	Caneta Azul	1,50
9876	Caneta Vermelha	1,50
5432	Papel Sulfite A4	12,00
1357	Envelope A4	0,50



### 5.3. Terceira Forma Normal – 3FN

A terceira forma normal determina que não devem existir atributos com dependência funcional transitiva em uma tabela, pois podem provocar anomalias de inclusão, alteração e deleção.

A aplicação da 3FN consiste em retirar das estruturas os campos que são funcionalmente dependentes de outros campos que não são chaves.

Podemos afirmar que uma estrutura está na 3FN se ela estiver na 2FN e não possuir campos dependentes de outros campos não chaves.

Tabela Pedido							
NumPedido	CodCliente	Cliente	Endereco	Cidade	UF	CodVendedor	Vendedor
347	100	Francisco Moura	Av. Nove de Julho, 19 – Jd. Paulista	São Paulo	SP	23	Álvaro

Apesar de estar na 2FN, a tabela Pedido não está na 3FN, pois os atributos Cliente, Endereco, Cidade e UF são dependentes funcionalmente de CodCliente (dependência transitiva) e o campo Vendedor é dependente funcionalmente de CodVendedor (dependência transitiva).

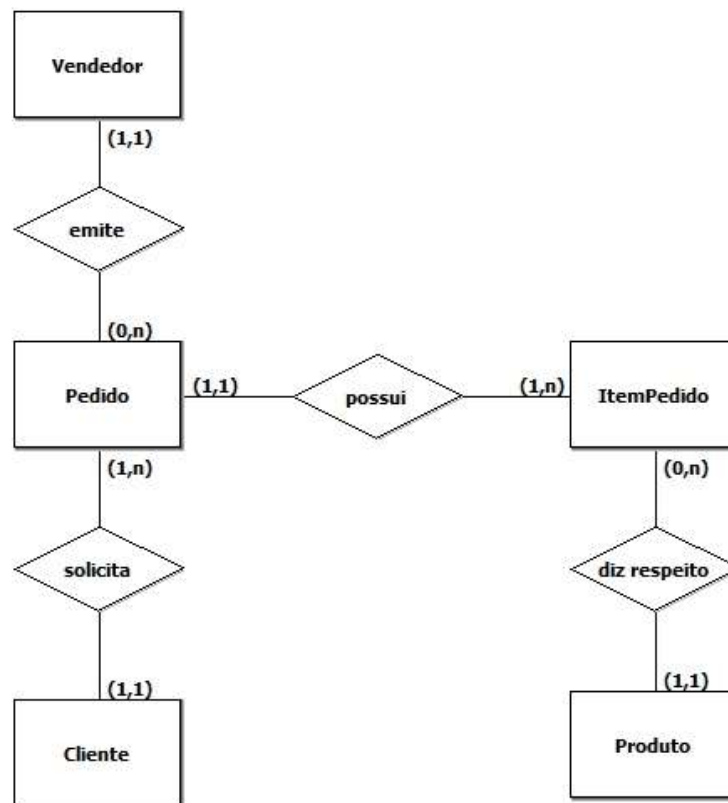
Tabela Pedido		
NumPedido	CodCliente	CodVendedor
347	100	23

Tabela Cliente				
CodCliente	Cliente	Endereco	Cidade	UF
100	Francisco Moura	Av. Nove de Julho, 19 – Jd. Paulista	São Paulo	SP

Tabela Vendedor	
CodVendedor	Vendedor
23	Álvaro

Tabela ItemPedido			
NumPedido	CodProduto	Qty	ValorT
347	1234	2	16,00
347	5678	1	1,50
347	9876	1	1,50
347	5432	1	12,00
347	1357	10	5,00

Tabela Produto		
CodProduto	NomeProduto	ValorU
1234	Caderno 200 fl	8,00
5678	Caneta Azul	1,50
9876	Caneta Vermelha	1,50
5432	Papel Sulfite A4	12,00
1357	Envelope A4	0,50



## 5.4. Forma Normal de Boyce-Codd – FNBC

Na verdade a FNBC é uma extensão da 3FN, que não resolvia certas anomalias presentes na informação contida em uma entidade. O problema foi observado porque a 2FN e a 3FN só tratavam dos casos de dependência parcial e transitiva de atributos fora de qualquer chave, porém quando o atributo observado estiver contido em uma chave (primária ou candidata), ele não é captado pela verificação da 2FN e 3FN.

A definição da FNBC é a seguinte: uma entidade está na FNBC se e somente se todos os determinantes forem chaves candidatas<sup>1</sup>. Note que esta definição é em termos de chaves candidatas e não sobre as chaves primárias.

Emprestimo			
NomeAgencia	NomeCliente	NumEmp	Valor
Rio Branco	Luis Sampaio	902230	1500,00
Rio Branco	Carlos Pereira	902230	1500,00
Rio Branco	Luis Paulo Souza	902240	1200,00
Ipanema	José Alves	902289	3000,00
Ipanema	Luis Paulo Souza	902255	850,00
Acaraí	Calos Pereira	902290	700,00
Acaraí	José Alves	202212	400,00

<sup>1</sup> Uma chave candidata representa um conjunto mínimo de atributos que podem identificar uma tupla dentro de uma relação, da mesma forma que faria uma chave primária. Elas são chamadas de candidata porque poderiam perfeitamente ser chaves primárias da relação, mas não foram escolhidas para tanto.

Esta estrutura de dados, entretanto, não satisfaz a FNBC, pois se considerarmos somente NumEmp como sua chave candidata, pode haver um par de linha representando o mesmo empréstimo, desde que fossem duas as pessoas participantes dele, como na tabela apresentada:

NomeAgencia	NomeCliente	NumEmp	Valor
Rio Branco	Luis Sampaio	902230	1500,00
Rio Branco	Carlos Pereira	902230	1500,00

Como existe dependência multivalorada neste caso, temos que aplicar a FNBC.

Decompomos a entidade que não está na FNBC em duas entidades sem perder a capacidade de junção delas.

Emprestimo		
NomeAgencia	NumEmp	Valor
Rio Branco	902230	1500,00
Rio Branco	902240	1200,00
Ipanema	902289	3000,00
Ipanema	902255	850,00
Acaraí	902290	700,00
Acaraí	202212	400,00

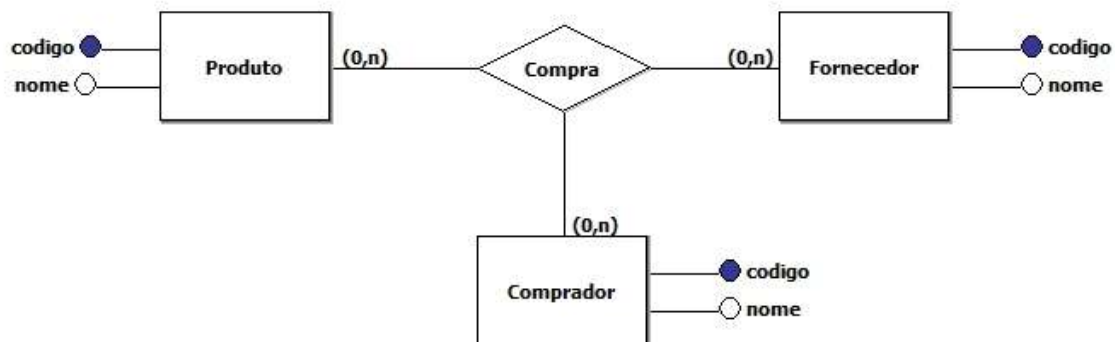
Devedor	
NomeCliente	NumEmp
Luis Sampaio	902230
Carlos Pereira	902230
Luis Paulo Souza	902240
José Alves	902289
Luis Paulo Souza	902255
Calos Pereira	902290
José Alves	202212

## 5.5. Quarta Forma Normal – 4FN

Uma entidade que esteja na 3FN também está na 4FN, se ela não contiver mais do que um fato multivalorado (não estamos falando de atributo multivalorado) a respeito da entidade descrita.

Vamos imaginar o conteúdo da tabela Compra.

CodFornecedor	CodProduto	CodComprador
101	BA3	01
102	CJ10	05
110	88A	25
530	BA3	01
101	BA3	25



Embora esteja na 3FN, pois não existem dependências transitivas na tabela, ao conter mais de um fato multivalorado, sua atualização torna-se muito difícil.

Para passarmos a entidade anterior para a 4FN, é necessário a realização de uma divisão da entidade original (Compra), em duas outras, ambas herdando a chave CodProduto e concatenada, em cada nova entidade, com os atributos CodFornecedor e CodComprador.

CodFornecedor	CodProduto
101	BA3
102	CJ10
110	88A
530	BA3

CodProduto	CodComprador
BA3	01
CJ10	05
88A	25
BA3	25

A aplicação da 4FN foi realizada neste caso, para corrigir um erro de modelagem de dados, pois no primeiro modelo utilizamos um relacionamento ternário de forma desnecessária, associando o comprador somente ao processo de compra, e não aos produtos que ele compra.

## 5.6. Quinta Forma Normal – 5FN

A aplicação da 5FN consiste em encontrar a Dependência de Junção que permite decompor uma relação sem perdas.

Advém das dependências multivaloradas que ocorrem entre os atributos de uma relação. A verificação da 5FN somente precisa ser empreendida em relações que tenham 3 ou mais atributos como parte da chave.

A 5FN trata da situação em que a informação permite ser reconstruída a partir de componentes menores que possam ser mantidos com uma redundância menor. Ela generaliza os casos não cobertos pela segunda, terceira e quarta formas normais.

Abaixo segue o exemplo da relação Fornecimento, que não pode ser decomposta em 2 relações, pois a junção entre elas geraria tuplas espúrias (tuplas não existentes na relação original). Assim, usamos a 5FN para decompor a relação em três (R1, R2, e R3), de modo que uma junção feita sobre essas relações mantenha a propriedade de junção sem perdas.

Produto	PedidoCompra	Fornecedor
Padrão B2	7801	341
Padrão B2	7801	108
Poste duplo T 9M	7801	108
Padrão B2	7802	108

Se desmembrarmos esta relação, chegamos às seguintes entidades:

R1	
Produto	PedidoCompra
Padrão B2	7801
Poste duplo T 9M	7801
Padrão B2	7802

R2	
PedidoCompra	Fornecedor
7801	341
7801	108
7802	108

R3	
Produto	Fornecedor
Padrão B2	341
Padrão B2	108
Poste duplo T 9M	108

```
select r1.Produto, r2.PedidoCompra, r3.Fornecedor
from r1,r2,r3
where r1.PedidoCompra = r2.PedidoCompra and
r2.Fornecedor = r3.Fornecedor and
r3.Produto = r1.Produto
```

## Atividade Pontuada Teams – Lista 3



## 6. Conexão com o Banco de Dados

PostgreSQL é um sistema gerenciador de banco de dados objeto relacional (SGBDOR), desenvolvido como projeto de código aberto. O PostgreSQL pode ser adquirido gratuitamente através do endereço <https://www.postgresql.org/download/>

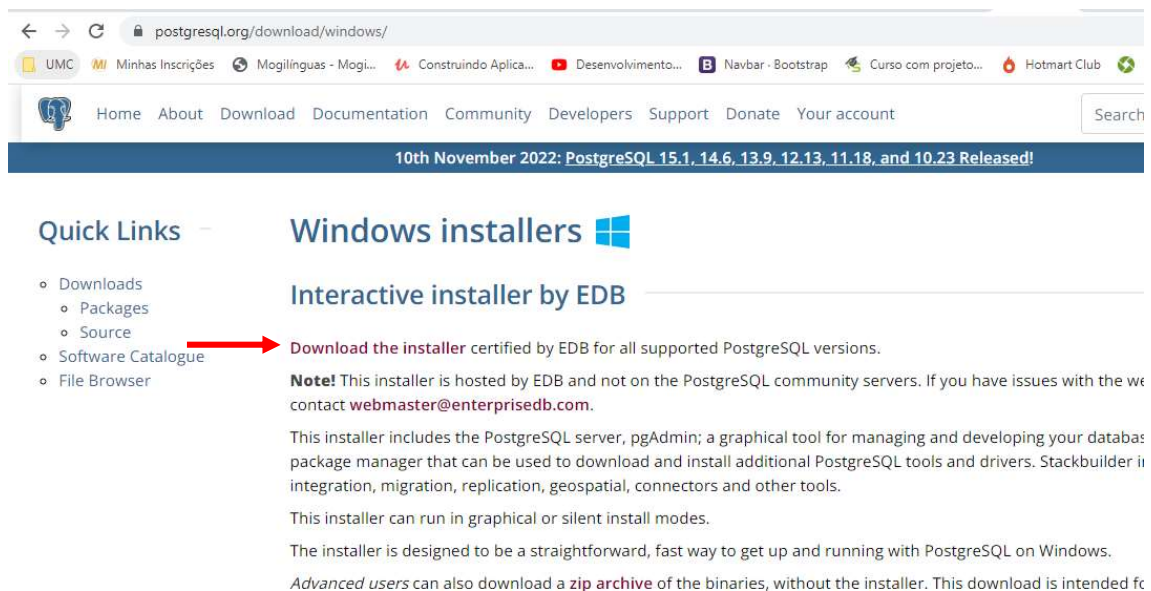
Hoje, o PostgreSQL é um dos SGBDs (Sistema Gerenciador de Bancos de Dados) de código aberto mais avançados, contando com recursos como:

- Consultas complexas;
- Chaves estrangeiras;
- Integridade transacional;
- Controle de concorrência multi-versão;
- Suporte ao modelo híbrido objeto-relacional;
- Gatilhos;
- Visões;
- Linguagem Procedural em várias linguagens (PL/pgSQL, PL/Python, PL/Java, PL/Perl) para Procedimentos armazenados;
- Indexação por texto;
- Estrutura para guardar dados Georeferenciados PostGIS;

O PostgreSQL possui uma ferramenta para administração do banco, chamada pgAdmin. Esta ferramenta permite entre outras funções, o carregamento de arquivos de linguagem SQL.

### 6.1. Instalando o pgAdmin

Fazer download do PostgreSQL acessando a URL <https://www.postgresql.org/download/>  
Clicar em “Download the installer”.



10th November 2022: PostgreSQL 15.1, 14.6, 13.9, 12.13, 11.18, and 10.23 Released!

**Quick Links**

- Downloads
- Packages
- Source
- Software Catalogue
- File Browser

**Windows installers**

**Interactive installer by EDB**

**Download the installer** certified by EDB for all supported PostgreSQL versions.

**Note!** This installer is hosted by EDB and not on the PostgreSQL community servers. If you have issues with the webmaster@enterprisedb.com.

This installer includes the PostgreSQL server, pgAdmin; a graphical tool for managing and developing your database package manager that can be used to download and install additional PostgreSQL tools and drivers. Stackbuilder integration, migration, replication, geospatial, connectors and other tools.

This installer can run in graphical or silent install modes.

The installer is designed to be a straightforward, fast way to get up and running with PostgreSQL on Windows.

Advanced users can also download a zip archive of the binaries, without the installer. This download is intended for

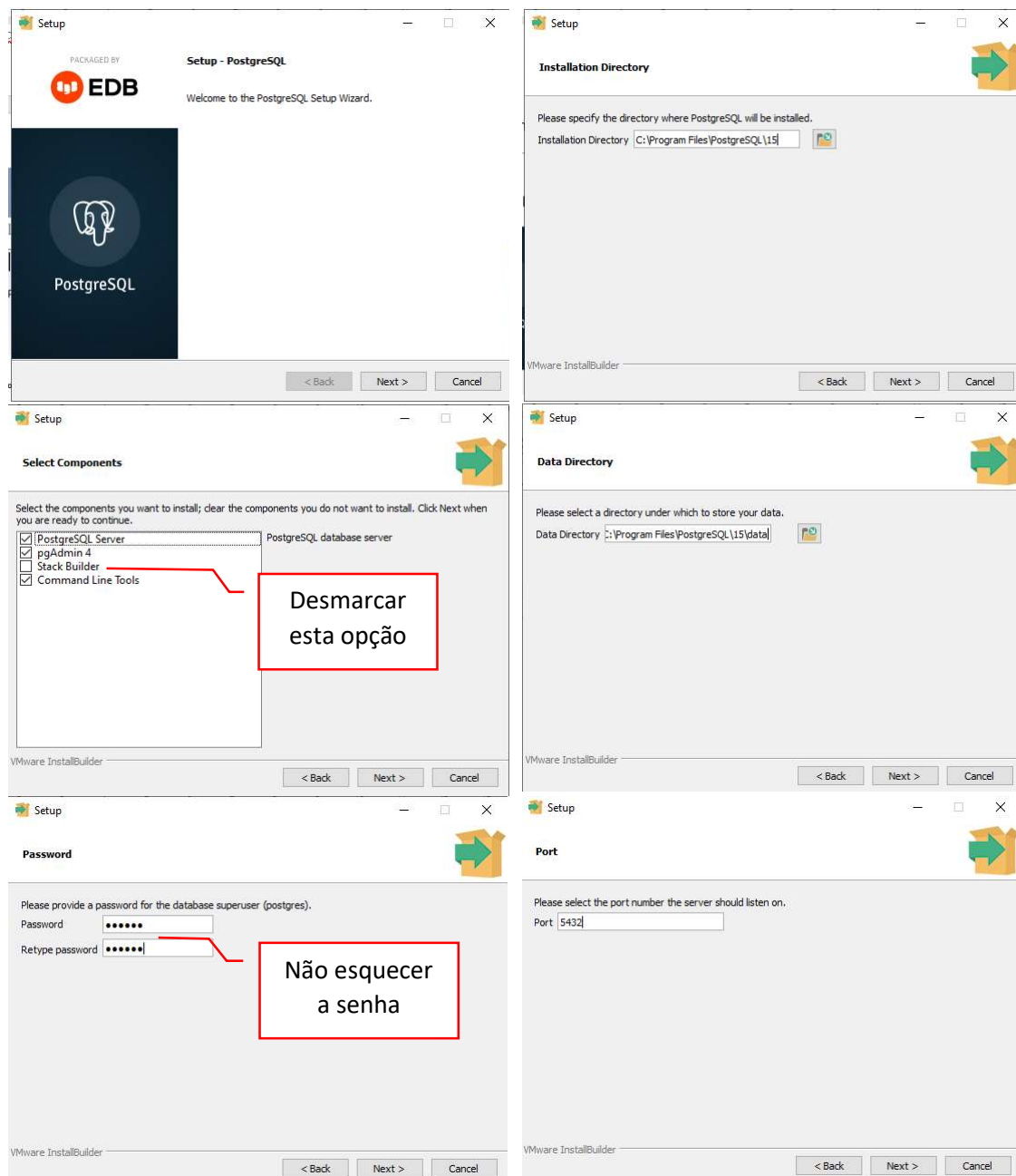
Escolher a versão desejada.

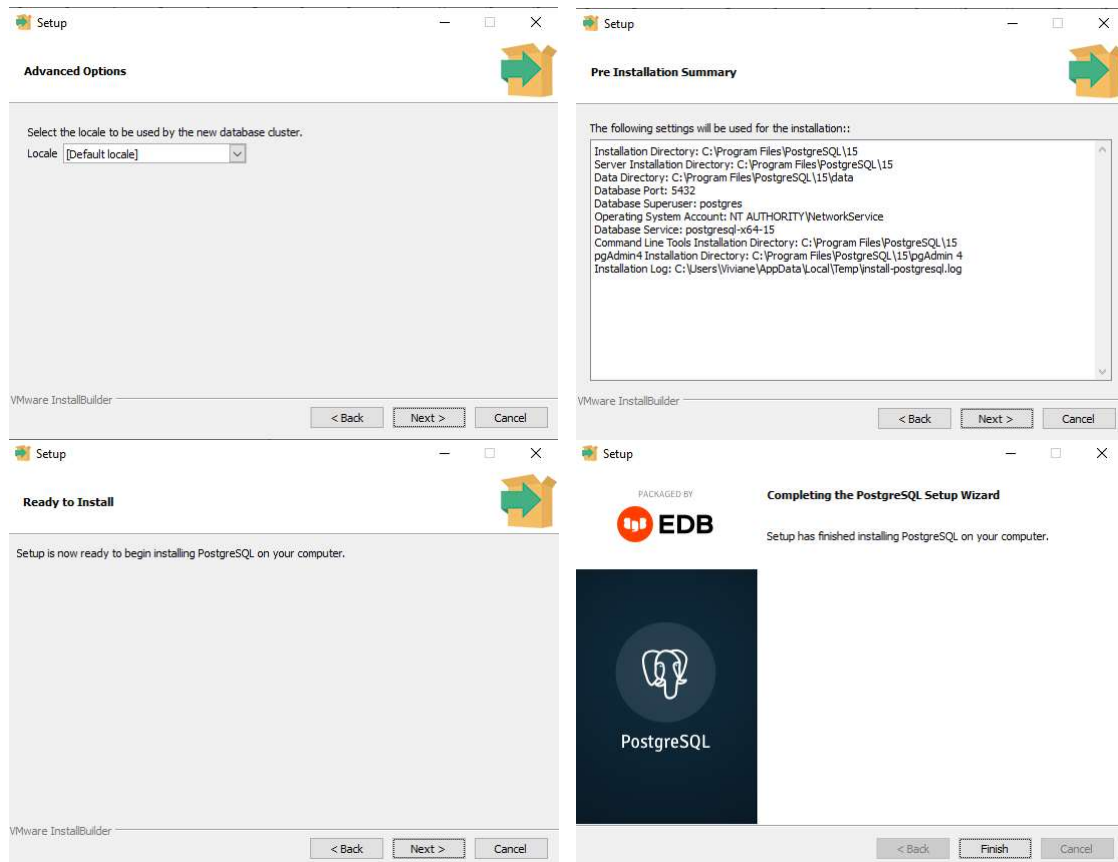
### Download PostgreSQL

Open source PostgreSQL packages and installers from EDB

PostgreSQL Version	Linux x86-64	Linux x86-32	Mac OS X	Windows x86-64	Windows x86-32
15.1	<a href="https://www.postgresql.org/ftp/nightly/v15.1/linux-x86_64/postgresql-15.1-linux-x86_64.tar.gz">postgresql.org</a>	<a href="https://www.postgresql.org/ftp/nightly/v15.1/linux-x86_32/postgresql-15.1-linux-x86_32.tar.gz">postgresql.org</a>			Not supported
14.6	<a href="https://www.postgresql.org/ftp/nightly/v14.6/linux-x86_64/postgresql-14.6-linux-x86_64.tar.gz">postgresql.org</a>	<a href="https://www.postgresql.org/ftp/nightly/v14.6/linux-x86_32/postgresql-14.6-linux-x86_32.tar.gz">postgresql.org</a>			Not supported

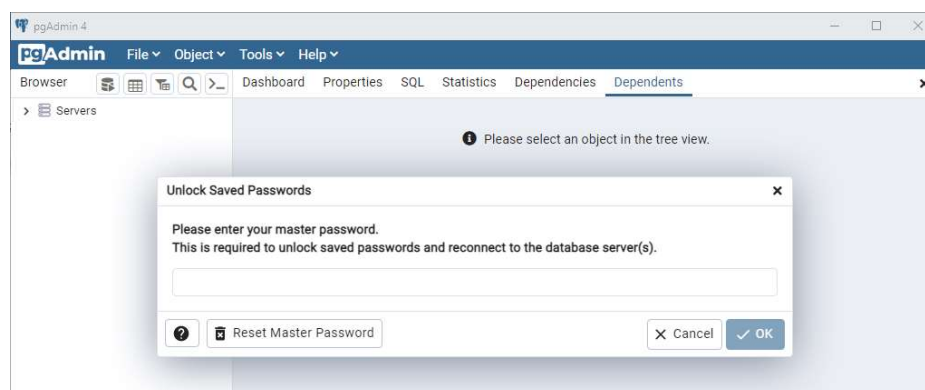
Após download, seguir com a instalação.



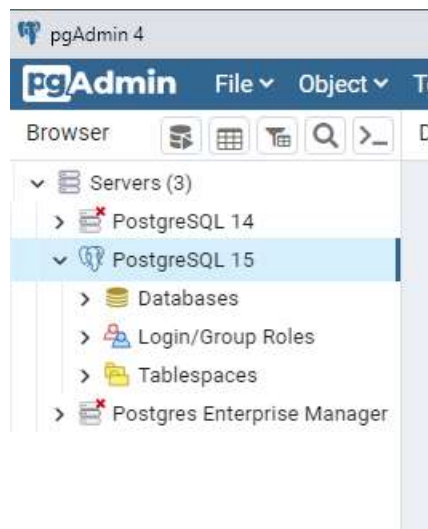
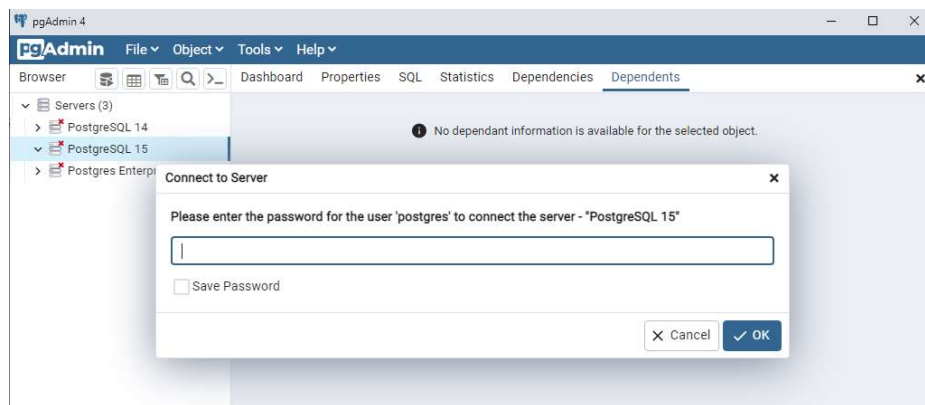


## 6.2. Acessando o pgAdmin

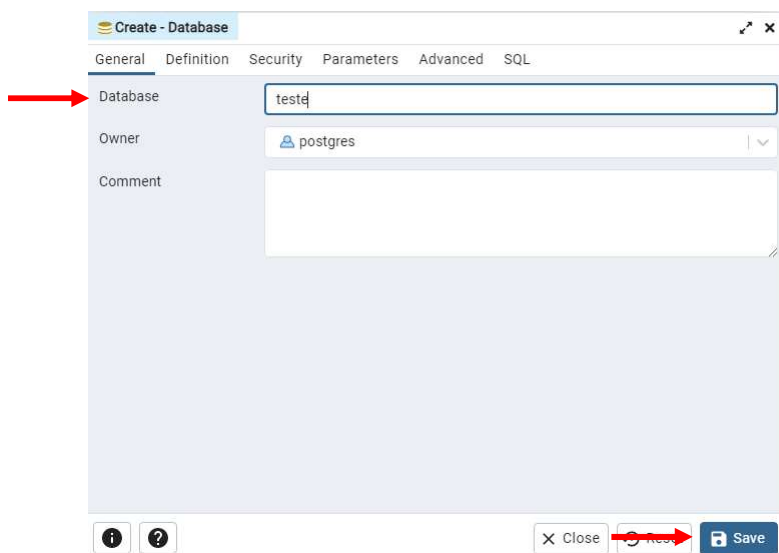
Para acessar, abra o aplicativo pgAdmin4. Será solicitada a senha usada na instalação.



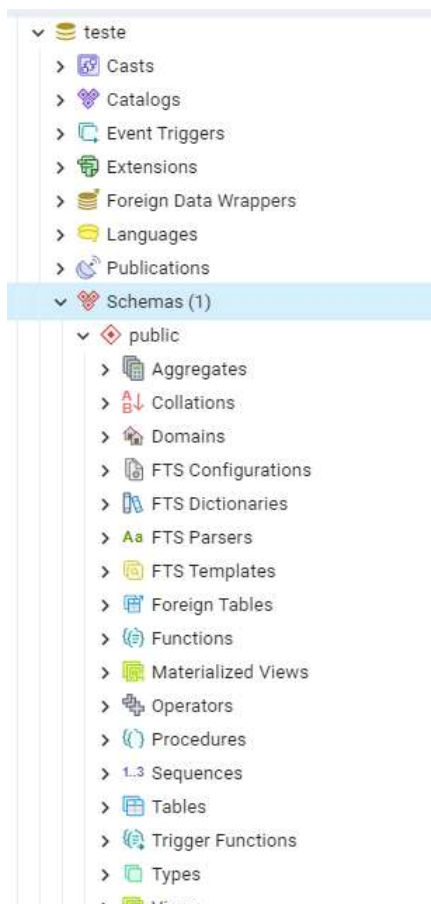
Clique em “Servers/PostgreSQL 15”, em algumas máquinas será solicitada novamente a digitação da senha.



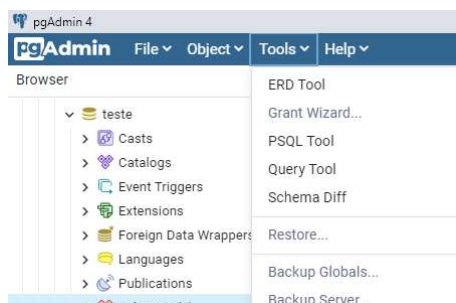
Com o botão direito, clique em Database/Create/Database...



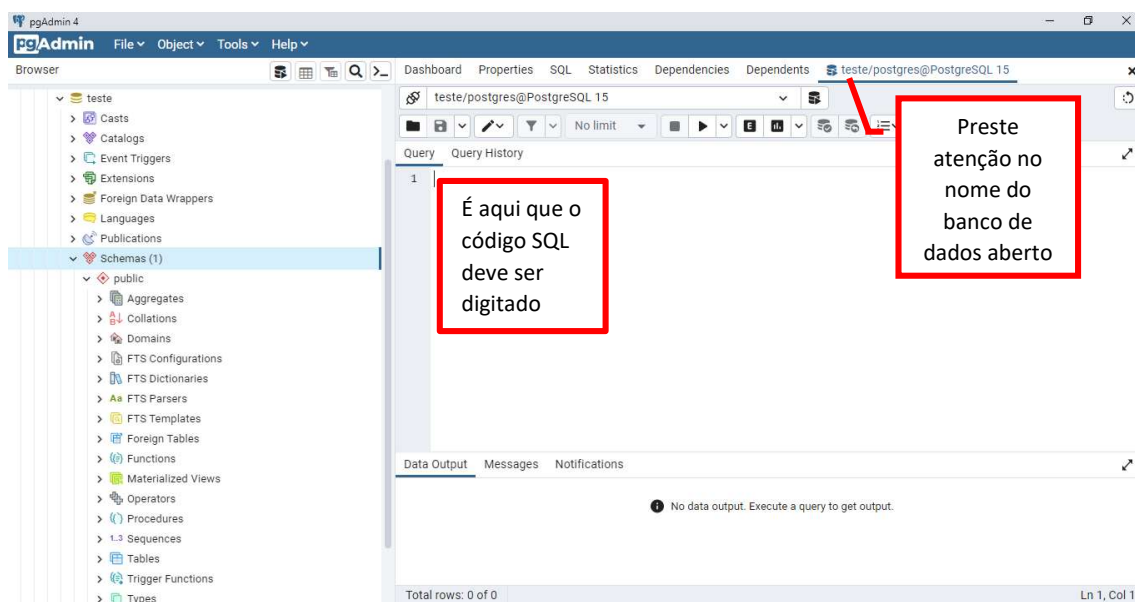
Digite o nome do banco de dados. Este nome deve ser único no servidor e não conter caracteres especiais. Clique em "Save" e aguarde a criação da estrutura do banco.




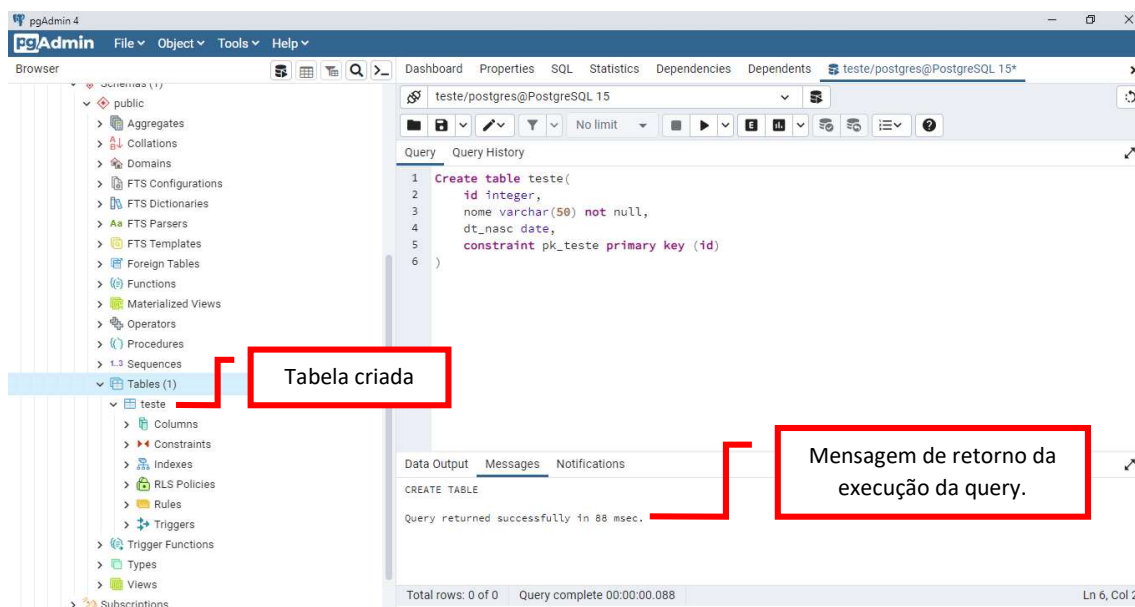
Para cada banco existente no servidor será criada uma estrutura como a da figura ao lado. É no item “Tables” que irão aparecer as tabelas criadas.




Para abrir a área de digitação de código, clique em “Tools/Query Tool”.



Para testar o código digitado clique em  ou pressione F5. O resultado da execução da query será exibido na aba “Data Output” ou “Messages”.



Para salvar o código digitado clique em , escolha o local/pasta onde o arquivo será salvo e digite o nome do arquivo, a extensão será .sql. Um arquivo .sql pode ser aberto em qualquer editor, incluindo o bloco de notas.

## 7. Data Description Language – DDL

Linguagem de Definição de Dados responsável pela criação e manutenção da estrutura do banco de dados, ou seja, das tabelas e campos que formam o banco de dados.

### 7.1. Criação das Tabelas – CREATE TABLE

O comando CREATE TABLE cria uma tabela nova, inicialmente vazia, no banco de dados atual.

Sintaxe:

```
Create Table <nome_da_tabela>
(
    <nome_do_campo><tipo_e_tamanho_de_Dados><aceitação>,
    <nome_do_campo><tipo_e_tamanho_de_Dados><aceitação>,
    constraint <nome_da_constraint> Primary Key(<nome_do_campo>),
    constraint <nome_da_constraint> Foreign Key(<nome_do_campo>)
        references <nome_da_tabela>(<nome_do_campo>),
    constraint <nome_da_constraint> unique (<nome_do_campo>)
);
```

Parâmetros:

**<nome\_da\_tabela>**

O nome da tabela a ser criada.

**<nome\_do\_campo>**

O nome da coluna/campo a ser criada na nova tabela.

**<Tipo e tamanho\_do\_dado>**

O tipo de dado da coluna e o tamanho máximo que pode assumir.

**<Aceitação>**

Tipo de aceitação do dado, ou seja, se ele pode ou não receber valores nulos ou não.

NOT NULL - Valores nulos não são permitidos na coluna.

NULL - Valores nulos são permitidos na coluna. Este é o padrão.

CHECK – Para validar o valor inserido no campo de acordo com alguma condição.

DEFAULT – Valor a ser inserido de forma automática, quando não for passado de forma explícita no comando da inserção.

**Constraint <nome\_da\_constraint>**

Um nome atribuído para a restrição da coluna ou da tabela.

**Primary Key(<nome\_do\_campo>)**

A restrição de chave primária especifica que a coluna, ou colunas, da tabela pode conter apenas valores únicos (não duplicados) e não nulos.

Tecnicamente a chave primária (PRIMARY KEY) é simplesmente uma combinação de unicidade (UNIQUE) com não nulo (NOT NULL), mas identificar um conjunto de colunas como chave primária também fornece metadados sobre o projeto do esquema, porque chaves primárias indicam que outras tabelas podem depender deste conjunto de colunas como um identificador único para linhas.

Somente uma chave primária pode ser especificada para uma tabela, seja como uma restrição de coluna ou como uma restrição de tabela.

**FOREIGN KEY (<nome\_do\_campo>)**

**References <nome\_da\_tabela>(<nome\_do\_campo>)**



A restrição REFERENCES especifica que um grupo de uma ou mais colunas da nova tabela deve conter somente valores correspondentes aos valores das colunas referenciadas da tabela referenciada.

- **<nome\_da\_tabela>** nome da tabela referenciada
- **<nome\_do\_campo>** nome do campo referenciado nesta tabela

#### **Unique (<nome\_do campo>, <nome\_do campo2>..., <nome\_do campoN>)**

A restrição UNIQUE especifica a regra onde um grupo de uma ou mais colunas distintas de uma tabela podem conter apenas valores únicos, ou seja, o valor não pode ser duplicado na tabela.

Cada restrição de unicidade da tabela deve abranger um conjunto de colunas diferentes do conjunto de colunas abrangido por qualquer outra restrição de unicidade e da chave primária definida para a tabela (Senão, seria apenas a mesma restrição declarada duas vezes).

Exemplo:

```
Create Table EXEMPLO
(
    codigo integer not null,
    nome varchar(60) not null,
    valor numeric(5,2) not null check (valor > 0),
    constraint pk_exemplo_matricula Primary Key(codigo),
    constraint uk_exemplo_varios unique (valor, nome)
);

Create Table EXEMPLO2
(
    codigo integer,
    nome varchar(60) not null,
    data_compra timestamp DEFAULT CURRENT_TIMESTAMP,
    codigo_exemplo integer,
    constraint pk_exemplo2 Primary Key(codigo),
    constraint fk_exemplo2 Foreign Key(codigo_exemplo)
        references Exemplo (codigo)
);
```

## **7.2. Alteração das Tabelas – ALTER TABLE**

O comando ALTER TABLE altera a definição de uma tabela existente. É necessário ser o dono da tabela para executar o comando ALTER TABLE, exceto para o super usuário. Existem várias formas alternativas:

**ADD COLUMN**→Esta forma adiciona uma nova coluna à tabela usando a mesma sintaxe do comando CREATE TABLE.

**DROP COLUMN**→Esta forma exclui uma coluna da tabela. Note que os índices e as restrições da tabela que referenciam a coluna também serão automaticamente excluídos. Será necessário especificar CASCADE se algum objeto fora da tabela depender da coluna --- por exemplo, chaves estrangeiras, visões, etc.

**SET/DROP NOT NULL**→Estas formas mudam a definição da coluna para permitir valores nulos ou para rejeitar valores nulos. A forma SET NOT NULL somente pode ser usada quando não existem valores nulos na coluna.

**RENAME**→A forma RENAME muda o nome de uma tabela (um índice, uma sequência ou uma visão) ou o nome de uma coluna da tabela. Não produz efeito sobre os dados armazenados.



**ADD definição\_de\_restrição\_de\_tabela**→Esta forma adiciona uma nova restrição para a tabela usando a mesma sintaxe do comando CREATE TABLE.

**DROP CONSTRAINT**→Esta forma exclui a restrição da tabela. Atualmente as restrições de tabela não necessitam ter nomes únicos, portanto pode haver mais de uma restrição correspondendo ao nome especificado. Todas estas restrições serão excluídas.

### Sinopse

```
ALTER TABLE tabela
    ADD [ COLUMN ] column tipo

ALTER TABLE tabela
    DROP [ COLUMN ] coluna [ RESTRICT | CASCADE ]

ALTER TABLE tabela
ALTER [ COLUMN ] coluna { SET | DROP } NOT NULL

ALTER TABLE tabela
RENAME [ COLUMN ] coluna TO novo_nome_da_coluna

ALTER TABLE tabela
    RENAME TO novo_nome_da_tabela

ALTER TABLE tabela
ADD CONSTRAINT nome_da_restrição

ALTER TABLE tabela
DROP CONSTRAINT nome_da_restrição [ RESTRICT | CASCADE ]
```

### Entradas

tabela →O nome (opcionalmente qualificado pelo esquema) da tabela existente a ser alterada.

coluna →O nome de uma coluna nova ou existente.

tipo →O tipo de dado da nova coluna.

novo\_nome\_da\_coluna →O novo nome para a coluna existente.

novo\_nome\_da\_tabela →O novo nome para a tabela.

definição\_de\_restrição\_de\_tabela →A nova restrição de tabela (table constraint) para a tabela.

nome\_da\_restrição →O nome da restrição existente a ser excluída.

CASCADE→Exclui, automaticamente, os objetos dependentes da coluna ou restrição excluída (por exemplo, visões fazendo referência à coluna).

RESTRICT→Recusa excluir a coluna ou a restrição se existirem objetos dependentes. Este é o comportamento padrão.

### Saídas

ALTER TABLE→Mensagem retornada se o nome da coluna ou da tabela for alterado com sucesso.

ERROR→Mensagem retornada se a tabela ou a coluna não existir.

### Exemplos

1. Para adicionar uma coluna do tipo varchar em uma tabela:  

```
ALTER TABLE distribuidores  
ADD COLUMN endereco VARCHAR(30);
```
2. Para excluir uma coluna de uma tabela:  

```
ALTER TABLE distribuidores  
DROP COLUMN endereco RESTRICT;
```
3. Para mudar o nome de uma coluna existente:  

```
ALTER TABLE distribuidores  
RENAME COLUMN endereco TO cidade;
```
4. Para mudar o nome de uma tabela existente:  

```
ALTER TABLE distribuidores  
RENAME TO fornecedores;
```
5. Para adicionar uma restrição NOT NULL a uma coluna:  

```
ALTER TABLE distribuidores  
ALTER COLUMN logradouro SET NOT NULL;
```
6. Para remover a restrição NOT NULL da coluna:  

```
ALTER TABLE distribuidores  
ALTER COLUMN logradouro DROP NOT NULL;
```
7. Para remover uma restrição de verificação de uma tabela e de todas as suas descendentes:  

```
ALTER TABLE distribuidores  
DROP CONSTRAINT chk_cep;
```
8. Para adicionar uma restrição de chave estrangeira a uma tabela:  

```
ALTER TABLE distribuidores  
ADD CONSTRAINT fk_dist FOREIGN KEY (endereco)  
REFERENCES enderecos(endereco);
```
9. Para adicionar uma restrição de unicidade à tabela:  

```
ALTER TABLE distribuidores  
ADD CONSTRAINT unq_dist_id_cod_cep UNIQUE (cod_cep);
```
10. Para adicionar uma restrição de chave primária a uma tabela com o nome gerado automaticamente, levando em conta que a tabela somente pode possuir uma única chave primária:  

```
ALTER TABLE distribuidores  
ADD PRIMARY KEY (dist_id);
```

### 7.3. Remoção de Tabelas – DROP TABLE

O comando DROP TABLE remove tabelas do banco de dados. Somente o criador pode remover a tabela. A tabela poderá ficar sem linhas, mas não será removida, usando o comando DELETE.

O comando DROP TABLE sempre remove todos os índices, regras, gatilhos e restrições existentes na tabela. Entretanto, para remover uma tabela referenciada por uma restrição de chave estrangeira de outra tabela deve ser especificado o CASCADE (O CASCADE remove a restrição de chave estrangeira, e não a tabela).

#### Sinopse

```
DROP TABLE nome [, ...] [ CASCADE | RESTRICT ]
```

**Entradas**

nome→O nome (opcionalmente qualificado pelo esquema) de uma tabela existente a ser removida.

CASCADE→Remove automaticamente os objetos que dependem da tabela (como visões).

RESTRICT→Recusa remover a tabela se existirem objetos dependentes. Este é o padrão.

**Saídas**

DROP TABLE →Mensagem retornada se o comando for executado com sucesso.

ERROR: table "nome" does not exist →Se a tabela especificada não existe no banco de dados.

**Exemplo**

1. Destruir as tabelas filmes e distribuidores:  
`DROP TABLE filmes, distribuidores;`

<b>Atividade Pontuada Teams – Lista 4</b>
---

## 8. Data Manipulation Language - DML

Linguagem de Manipulação de Dados responsável pela inclusão, alteração, exclusão e consulta dos dados armazenados no banco.

### 8.1. Inserção de Dados – Insert

O comando INSERT insere uma ou mais tuplas em tabelas do banco de dados.

Sintaxe:

```
insert into <nome_da_tabela> [<coluna(s)>] values <valores>
```

Parâmetros:

**<nome\_da\_tabela>**

O nome da tabela onde os dados serão inseridos.

**[<coluna(s)>]**

Lista com o nome dos campos (colunas) onde os valores serão inseridos. Este parâmetro pode ser omitido.

**<valores>**

Lista de dados a serem inseridos na tabela.

Exemplos:

```
insert into cidades values (1, 'Casca', 'RS');
```

```
insert into cidades (cod_cid, nome_cid) values (1, 'Casca');
```

```
insert into cidades (nome_cid, cod_cid) values ('Casca', 1);
```

Execute o script abaixo:

```
CREATE TABLE cliente (  
    cod_cli      numeric(2),  
    nome         varchar(50),  
    cpf          numeric(11),  
    rua          varchar(50),  
    num          varchar(5),  
    bairro       varchar(50),  
    cidade       varchar(50),  
    uf           varchar(2),  
    cep          varchar(9),  
    CONSTRAINT pk_cliente PRIMARY KEY (cod_cli)  
);
```

Execute os comandos abaixo anotando os resultados. Em caso de erro, explique o motivo e faça a correção necessária:

```
INSERT INTO cliente  
(cod_cli, nome, cpf, rua, num, bairro, cidade, uf, cep) VALUES  
(1, 'GILBERTO', '12345678901', 'AV. Astronautas', '1333',  
 'Cidade Jardim', 'S. J. Dos Campos', 'SP', '12227-220');
```

```
INSERT INTO cliente  
(cod_cli, nome, cpf, rua, num, bairro, cidade, uf, cep) VALUES  
(1, 'EDUARDO', '12345678902', 'AV. X', '1456', 'Vila', 'Rio de Janeiro',  
 'RJ', '35000000');
```

Omitindo a lista de campos:

### Exemplo1:

```
INSERT INTO cliente VALUES
(2, 'BRUNO', '12345678902', 'Rua Albino Sartori', '95', 'V. S. J.', 'Ouro
Preto', 'MG', '35400-000');
```

### Exemplo2:

```
INSERT INTO cliente VALUES(3, 'RICARDO', '12345678903', 'Rua Ceci');
```

### Exemplo3:

```
INSERT INTO cliente VALUES('ANTONIO', '12345678904');
```

## 8.2. Exclusão de Dados – Delete

O comando DELETE exclui as linhas que satisfazem a cláusula WHERE na tabela especificada.

Se a *condição* (cláusula WHERE) estiver ausente, o efeito é a exclusão de todas as linhas da tabela. O resultado será uma tabela válida, porém vazia.

### Sintaxe:

```
DELETE FROM <nome_da_tabela> [ WHERE condição ]
```

### Parâmetros:

#### <nome\_da\_tabela>

O nome da tabela onde os dados serão excluídos.

#### [WHERE condição]

Uma condição de consulta do SQL que retorna as linhas a serem excluídas.

### Saídas:

#### DELETE contador

Mensagem retornada se as linhas foram excluídas com sucesso. O *contador* representa o número de linhas excluídas. Se *contador* for 0, então nenhuma linha foi excluída.

### Exemplos:

#### 1. Remover todos os filmes, exceto os musicais:

```
DELETE FROM filmes WHERE tipo <> 'Musical';
SELECT * FROM filmes;
```

cod	titulo	did	data_prod	tipo
UA501	West Side Story	105	1961-01-03	Musical
TC901	The King and I	109	1956-08-11	Musical
WD101	Bed Knobs and Broomsticks	111		Musical

(3 rows)

## 2. Esvaziar a tabela *filmes*:

```
DELETE FROM filmes;
SELECT * FROM filmes;
```

```
cod | titulo | did | data_prod | tipo | duracao
-----+-----+-----+-----+-----+-----
(0 rows)
```

## 8.3. Atualização de Dados – Update

O comando **UPDATE** muda os valores das colunas especificadas em todas as linhas que satisfazem a condição. Somente as colunas a serem modificadas devem aparecer na lista de colunas da declaração.

Sintaxe:

```
UPDATE <nome_da_tabela> SET coluna = expressão [, ...]
[ WHERE condição ]
```

Parâmetros:

**<nome\_da\_tabela>**

O nome da tabela onde os dados serão atualizados.

**coluna**

O nome de uma coluna da tabela.

**expressão**

Uma expressão válida ou um valor a ser atribuído à coluna.

**[WHERE condição]**

Condição para a modificação.

Saídas:

**UPDATE contador**

Mensagem retornada se o comando for executado com sucesso. O *contador* representa o número de linhas atualizadas. Se o *contador* for 0 então nenhuma linha foi atualizada.

Exemplos:

### 1. Mudar a palavra *Drama* por *Suspense* na coluna *tipo*:

```
UPDATE filmes
SET tipo = 'Suspense'
WHERE tipo = 'Drama';
```

```
SELECT *
FROM filmes
WHERE tipo = 'Drama' OR tipo = 'Suspense';
```

```
cod | titulo | did | data_prod | tipo | tempo
-----+-----+-----+-----+-----+-----
BL101 | The Third Man | 101 | 1949-12-23 | Suspense | 01:44
P_302 | Becket | 103 | 1964-02-03 | Suspense | 02:28
M_401 | War and Peace | 104 | 1967-02-12 | Suspense | 05:57
T_601 | Yojimbo | 106 | 1961-06-16 | Suspense | 01:50
DA101 | Das Boot | 110 | 1981-11-11 | Suspense | 02:29
```

## 8.4. Consulta de Dados – Select

Seleciona os registros de uma tabela.

Sintaxe:

```
SELECT <retorno>  
FROM <tabela>  
[WHERE <condicao>]  
[ORDER BY <campo> [ASC | DESC]];
```

Parâmetros:

### **<retorno>**

Campo da tabela que será retornado para visualização.

Se eu colocar o \* significa que retornará todos os campos da tabela.

### **<tabela>**

O nome da tabela onde a consulta será feita.

### **[WHERE condição]**

Condição que fará um filtro nos dados, ou seja, serão exibidos apenas os registros que atenderem a condição imposta.

A cláusula WHERE cria uma condição para que a operação seja efetuada caso o registro atenda a condição imposta.

Ou seja, ela pode ser utilizada para Filtrar os dados de uma consulta (SELECT), para impor uma restrição em uma exclusão (DELETE) e para impor restrições nos registros atualizados (UPDATE).

As condições criadas podem ser combinadas com alguns comandos ou com expressões simples:

### **LIKE**

Comparação se existe o valor no campo testado.

Ex: `SELECT * FROM Aluno WHERE nome like 'Ana%';`

No exemplo utilizamos o % para comparar os registros que comecem com Ana. Podemos utilizar o % como um “coringa” onde não sabemos (ou não importa) o início ou o fim da palavra a ser localizada. É importante ressaltar que o comando like é case sensitive, ou seja, faz diferenciação entre letras maiúsculas e minúsculas.

### **ILIKE**

Comparação se existe o valor no campo testado.

Ex: `SELECT * FROM Aluno WHERE nome ilike 'Ana%';`

Funciona com o comando LIKE, porém não é case sensitive.

### **AND**

Lógica booleana.

A verificação das condições só será verdadeira quando as duas condições forem satisfeitas.

Ex: `SELECT * Aluno WHERE idade=23 AND sexo='masculino';`

Só exibirá os registros de alunos homens que possuem idade igual a 23.

### **OR**

Lógica booleana.

A verificação das condições só será verdadeira quando qualquer uma das condições for satisfeita.

Ex: `SELECT * Aluno WHERE idade=23 OR sexo='masculino';`

Só exibirá os registros de todos os alunos homens e todos os alunos que possuem idade igual a 23. Só não serão exibidos os registros de mulheres que não tem 23 anos.

### **IN (valores)**

Verifica se o conteúdo está dentro de uma lista de valores passados.

Ex: `SELECT * Aluno WHERE nome IN ('Ana Lucia', 'Ana Claudia', 'Ana Maria');`

Exibirá apenas os registros dos alunos que possuíram nome Ana Lucia, Ana Claudia ou Ana Maria.

### **BETWEEN**

Verifica se o conteúdo está entre as duas condições impostas.

Ex: `SELECT * Aluno WHERE media BETWEEN 3.0 AND 5.0;`

Só exibirá os registros de alunos que possuem média maior ou igual a 3 e menor ou igual a 5.

### **Operadores**

=	igualdade
<>	desigualdade
<	menor que
>	maior que
>=	maior ou igual
<=	menor ou igual

### **[ORDER BY <campo> [ASC | DESC]]**

Define a ordem em que os registros serão exibidos. O padrão é ASC (ascendente/ordem crescente). Para exibir em ordem decrescente (descendente) a cláusula DESC é obrigatória.

Ex: `SELECT * Aluno WHERE media BETWEEN 3.0 AND 5.0 ORDER BY nome DESC;`

Exibe os registros ordenados pelo campo nome de forma decrescente.

## **8.4.1. Select com Junções**

O modelo relacional permite o agrupamento de dados de tabelas separadas em novos e não-previstos relacionamentos.

Os relacionamentos tornam-se explícitos quando os dados são manipulados: ao consultar o banco de dados e não durante sua criação.

Você pode unir dados a partir de quaisquer colunas nas tabelas, desde que os tipos dos dados sejam iguais e a operação faça sentido.

Esses dados não precisam ser chaves, apesar de normalmente o serem.

### **Boas junções:**

A coluna de junção normalmente é a coluna de chave: primária ou estrangeira

As colunas de junção devem ter tipos de dados compatíveis

Supondo que desejamos mostrar em nossa consulta o nome e o rgm do aluno e o nome da turma que ele está alocado.



### Turma

Codigo	turma	Descrição
01	SisInfo	Sistema de Informação
02	TDS	Tecnólogo Processamento de Dados
03	CIECO	Ciência da computação

### Alunos

RGM	Nome	Telefone	cod_turma
27898	Antonio Jose	80900909	01
899090	Luiza Rossi	09-08989	02
6767879	Renato Faria	767567	03

```
Select rgm, nome, turma
from ALUNOS, TURMA
where cod_turma = codigo;
```

Agora vamos fazer uma junção entre 3 tabelas. Desejamos mostrar o nome e salário do funcionário, a cidade onde ele mora e o departamento que trabalha.

### Cidade

Codigo	Nome	Estado
01	Mogi das Cruzes	SP
06	Taubaté	SP
09	São José dos Campos	SP

### Departamento

Codigo	Nome
04	Compras
08	Informática
98	Vendas

### Funcionarios

Codigo	Nome	telefone	Salario	Cod_cidade	Cod_depto
01909	Carlos Allan	8788099	10.000,00	01	04
9909	Manuel	67989090	800,00	09	08
5787	Renato	6789899	4000,00	06	98

```
Select f.nome, f.salario, c.nome, d.nome
from Funcionário as f, Cidade as c, Departamento as d
where f.cod_cidade = c.codigo AND
      f.cod_depto = d.codigo;
```

### Como funciona?

O comando de junções entre tabelas faz um produto cartesiano entre as tabelas consultadas e retorna apenas aquelas que satisfaçam a condição de junção (igualdade entre campos das tabelas)

**Alunos**

RGM	Nome	Telefone	Cod_turma
27898	Antonio Jose	80900909	01
899090	Luiza Rossi	09-08989	02
6767879	Renato Faria	767567	03

**Turma**

Codigo	Turma	Descrição
01	SisInfo	Sistema de Informação
02	TDS	Tecnólogo Processamento de Dados
03	CIECO	Ciência da computação

Rgm	Nome	Telefone	Cod_turma	Código	Turma	Descricao
27898	Antonio Jose	80900909	01	01	SisInfo	Sistema de Informação
27898	Antonio Jose	80900909	01	02	TDS	Tecnólogo Processamento de Dados
27898	Antonio Jose	80900909	01	03	CIECO	Ciência da computação
899090	Luiza Rossi	09-08989	02	01	01	SisInfo
899090	Luiza Rossi	09-08989	02	02	TDS	Tecnólogo Processamento de Dados
899090	Luiza Rossi	09-08989	02	03	CIECO	Ciência da computação
6767879	Renato Faria	767567	03	01	01	SisInfo
6767879	Renato Faria	767567	03	02	TDS	Tecnólogo Processamento de Dados
6767879	Renato Faria	767567	03	03	CIECO	Ciência da computação

Mais exemplos:

**Pais**

cod	nome
10	Brasil
20	Italia

**Cidade**

cod	nome	cod_pais
1	Mogi das Cruzes	10
2	SãoPaulo	10
3	Suzano	
4	Roma	20

**Aluno**

rgm	nome	cod_cid
100	Maria	3
200	José	1
300	Ricardo	

Este comando exibe o produto cartesiano entre as tabelas aluno e cidade.

```
select aluno.nome, cidade.nome
from aluno, cidade;
```

1 - Este comando elimina as tuplas espúrias (registros falsos) criadas pelo produto cartesiano.

```
select aluno.nome, cidade.nome
from aluno, cidade
where aluno.cod_cid = cidade.cod;
```

2- Equivale ao select 1, porém utiliza apelidos (alias) para as tabelas aluno e cidade este apelido são definidos na cláusula FROM.

```
select a.nome, c.nome
from aluno a, cidade c
where a.cod_cid = c.cod;
```

3- Equivale ao select 1, porém utiliza a cláusula INNER JOIN para ligar as tabelas eliminando as tuplas espúrias.

```
select a.nome, c.nome
from aluno a inner join cidade c on a.cod_cid = c.cod;
```

4- Equivale ao select 3, escrever INNER JOIN ou apenas JOIN é a mesma coisa.

```
select a.nome, c.nome
from aluno a join cidade c on a.cod_cid = c.cod;
```

5- Mostra todos os alunos, mesmo que eles não tenham a cidade cadastrada, portanto, dá prioridade aos registros da tabela que aparece a ESQUERDA da cláusula JOIN.

```
select a.nome, c.nome
from aluno a left join cidade c on a.cod_cid = c.cod;
```

6- Mostra todas as cidades, mesmo que elas não tenham alunos ligados a ela, portanto, dá prioridade aos registros da tabela que aparece a DIREITA da cláusula JOIN.

```
select a.nome, c.nome
from aluno a right join cidade c on a.cod_cid = c.cod;
```

7- Mostra todos os registros das duas tabelas.

```
select a.nome, c.nome
from aluno a full join cidade c on a.cod_cid = c.cod;
```

8- Mostra que é possível incluir na cláusula WHERE outras condições além da necessárias para eliminação de tuplas espúrias.

```
select a.nome, c.nome
from aluno a, cidade c
where a.cod_cid = c.cod and
      c.nome <> 'Suzano';
```

9- Unindo várias tabelas.

```
select a.nome, c.nome, p.nome
from aluno a, cidade c, pais p
where a.cod_cid = c.cod and
      c.cod_pais = p.cod;
```

10- Usando apelidos para as TABELAS e para as COLUNAS.

```
select a.nome "Nome Aluno", c.nome "Nome Cidade", p.nome "Nome País"
from aluno a, cidade c, pais p
where a.cod_cid = c.cod and
      c.cod_pais = p.cod;
```

### 8.1.2. Funções Agregadas

**COUNT( )** – Retorna o número de linhas.

Ex: Select COUNT(codigo) from cliente;

**SUM( )** – Devolve a soma.

Ex: Select SUM(salario) from cliente;

**MAX( )** – Retorna o maior valor.

Ex: Select MAX(salario) from cliente;

**MIN( )** – Retorna o menor valor.

Ex: Select MIN(salario) from cliente;

**AVG( )** – Retorna o valor médio.

Ex: Select AVG(salario) from cliente;

**GROUP BY** – O uso da cláusula GROUP BY faz com que os registros sejam retornados na ordem crescente do(s) campo(s) usado(s) para agrupar o resultado da função agregada. Mais um detalhe, todos os campos na cláusula SELECT devem aparecer após a cláusula GROUP BY separados por vírgula, ou você terá um erro de sintaxe na consulta.

Ex: Select SUM(salario) from clientes group by salario;

**HAVING** – A cláusula HAVING vem a complementar a cláusula GROUP BY. Quando usamos GROUP BY, os registros retornados serão todos os que satisfizerem ao critério informado após a palavra WHERE. Porém, podemos querer fazer uma segunda filtragem após termos os resultados dos cálculos das funções agregadas. Neste caso, usamos a cláusula HAVING para fazer esta filtragem posterior.

Ex.: SELECT SUM(salario) from clientes GROUP BY salário HAVING SUM(salario) < 2000;

#### Exemplos:

Create table vendas(

id	numeric(10),
vendedor	varchar(50),
data	date,
valor	numeric(10,2),
constraint pk_vendas primary key (id)	

);

```
insert into vendas values(1,'Fulano','2011-11-14',600);
insert into vendas values(2,'Fulano','2011-11-16',1000);
insert into vendas values(3,'Fulano','2011-11-18',800);
insert into vendas values(4,'Beltrano','2011-11-15',1200);
insert into vendas values(5,'Beltrano','2011-11-17',1600);
insert into vendas values(6,'Cicrano','2011-11-14',1000);
insert into vendas values(7,'Cicrano','2011-11-15',1200);
insert into vendas values(8,'Cicrano','2011-11-16',1400);
insert into vendas values(9,'Cicrano','2011-11-17',1600);
insert into vendas values(10,'Cicrano','2011-11-18',1800);
```

1)Qual o total das vendas?

```
SELECT SUM(valor)
FROM vendas;
```

2)Qual o total das vendas de cada vendedor?

```
SELECT vendedor, SUM(valor)
FROM vendas
GROUP BY vendedor;
```

3)Quantas vendas foram feitas?

```
SELECT COUNT(*)
FROM vendas;
```

4)Quantas vendas cada vendedor fez?

```
SELECT vendedor, COUNT(*)
FROM vendas
GROUP BY vendedor;
```

5)Qual a menor e a maior venda?

```
SELECT MIN(valor), MAX(valor)
FROM vendas;
```

6)Qual a menor e a maior venda de cada vendedor?

```
SELECT vendedor, MIN(valor), MAX(valor)
FROM vendas
GROUP BY vendedor;
```

7)Qual a média das vendas dos dias 15 e 17?

```
SELECT data, AVG(valor)
FROM vendas
WHERE data IN ('2011-11-15', '2011-11-17')
GROUP BY data;
```

8)Em quais dias as vendas superaram 3.000?

```
SELECT data, SUM(valor)
FROM vendas
GROUP BY data
HAVING SUM(valor) >3000;
```

9)Em quais dias, no período de 14 a 16/11/2011, a média das vendas foi menor que 2000?

```
SELECT data, AVG(valor)
FROM vendas
WHERE data BETWEEN '2011-11-14' AND '2011-11-16'
GROUP BY data
HAVING AVG(valor) <2000;
```

10)Que vendedores fecharam mais de 2 vendas nos dias 14, 16 e 18 de Novembro de 2011?

```
SELECT vendedor, COUNT(*)
FROM vendas
WHERE data IN ('2011-11-14', '2011-11-16', '2011-11-18')
GROUP BY vendedor
HAVING COUNT(*) >2;
```

### 8.1.3. Outras Funções e Cláusulas

**DISTINCT** - Em uma tabela, algumas das colunas podem conter valores duplicados, para remover os registros duplicados e exibir apenas uma ocorrência de cada valor utiliza-se a cláusula DISTINCT.

Ex: Selecione todos os clientes que fizeram compras no ano de 2013.

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

**NOW( )** – Retorna a data e hora do sistema

Ex: Select NOW( )

**IS NULL** – Retorna os campos nulos

Ex: Select \* from clientes where nome IS NULL;

Select \* from clientes where nome IS NOT NULL;

**LOWER( )** – Esta função converte o valor de um campo com letras minúsculas.

Ex: Select LOWER(nome) from clientes;

**UPPER( )** – Esta função converte o valor de um campo com letras maiúsculas.

Ex: Select UPPER(nome) from clientes;

**ROUND( )** – Esta função arredonda um campo numérico para um número de casas decimais especificado.

Ex: Select ROUND(salário,2) from clientes;

**SQRT( )** – Esta função extrair a raiz quadrada do valor especificado.

EX: SELECT sqrt(salario), numFROM cliente;

**TRUNC( )** – Esta função retorna a parte inteira de um valor especificado.

EX: SELECT trunc(salario), num FROM clientes;

**date\_part( )** – Esta função retorna a parte inteira de um valor especificado.

EX:SELECT \* FROM passagem WHERE date\_part('month',data\_compra)=4;

## Atividade Pontuada Teams – Lista 5

## 9. Sequence

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

Uma Sequence possui três funções de acesso:

- Nextval('nome\_sequence');
- Currval('nome\_sequence');
- Setval('nome\_sequence', novo\_valor);

Exemplo 1: Criação de sequences:

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

Exemplo 2: Utilizando uma sequence como padrão em um campo:

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

Exemplo 3: Utilizando o comando insert:

```
CREATE TABLE produtos(  
    codigo      numeric(4) not null,  
    nome        varchar(100) not null  
);  
  
CREATE SEQUENCE minha_sequence;  
  
INSERT INTO produtos (codigo, nome)  
VALUES (nextval('minha_sequence'), 'caneta');
```

Partes de uma Sequence:

INCREMENT BY → Podemos definir o incremento da nossa sequence.

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

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

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

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

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

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

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

Exemplo 4: Apagando uma sequence:

```
DROP SEQUENCE contador_cli;
```

Exemplo 5: Alterando uma sequence:

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

Exemplo 6: Utilizando um campo serial:

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

Deste modo, o PostgreSQL criará uma sequence que terá como nome nomeTabela\_nomeCampo\_seq, iniciando de 1 e com incremento de 1.



## 10. Insert com Query

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

### Exemplo 1: Cópia de dados:

```
CREATE TABLE cliente(  
  cpf          numeric(11,0),  
  nome         character varying(100) NOT NULL,  
  rua          character varying(100),  
  numero       character varying(10),  
  cidade       character varying(50),  
  tel          character varying(15),  
  email        character varying(20),  
  CONSTRAINT pk_cliente PRIMARY KEY (cpf),  
  CONSTRAINT uk_email UNIQUE (email)  
);  
  
INSERT INTO cliente VALUES  
  (12345678910,'Maria','Rua A','10','Mogi','4747-8765','maria@mail');  
INSERT INTO cliente VALUES  
  (98745678910,'José','Rua B','11','Suzano','8765-8765','jose@mail');  
INSERT INTO cliente VALUES  
  (12312378910,'Clara','Rua C','12','Guararema','3102-8765','clara@mail');  
INSERT INTO cliente VALUES  
  (83645678910,'Ricardo','Rua D','13','Mogi','4747-6298','ricardo@mail');  
INSERT INTO cliente VALUES  
  (12345639710,'Lucia','Rua E','14','Mogi','4747-1739','lucia@mail');
```

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

### Exemplo 2: Valores de chave estrangeira:

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

```
CREATE TABLE func(  
    codigo        integer,  
    nome          varchar(100) NOT NULL,  
    depto         integer,  
    CONSTRAINT pk_func PRIMARY KEY (codigo),  
    CONSTRAINT fk_func_depto FOREIGN KEY (depto) REFERENCES depto(codigo)  
);  
  
INSERT INTO depto VALUES (nextval('seq_depto'),'Compras');  
INSERT INTO depto VALUES (nextval('seq_depto'),'RH');  
INSERT INTO depto VALUES (nextval('seq_depto'),'CPD');  
INSERT INTO depto VALUES (nextval('seq_depto'),'Jurídico');  
  
SELECT * FROM depto;  
  
INSERT INTO func VALUES  
    (nextval('seq_func'),'Maria',  
     (SELECT codigo FROM depto WHERE nome = 'Compras'));  
INSERT INTO func VALUES  
    (nextval('seq_func'),'João',  
     (SELECT codigo FROM depto WHERE nome = 'Jurídico'));  
INSERT INTO func VALUES  
    (nextval('seq_func'),'José',  
     (SELECT codigo FROM depto WHERE nome = 'RH'));  
INSERT INTO func VALUES  
    (nextval('seq_func'),'Claudia',  
     (SELECT codigo FROM depto WHERE nome = 'CPD'));  
INSERT INTO func VALUES  
    (nextval('seq_func'),'Luana',  
     (SELECT codigo FROM depto WHERE nome = 'CPD'));  
INSERT INTO func VALUES  
    (nextval('seq_func'),'Valéria',  
     (SELECT codigo FROM depto WHERE nome = 'Compras'));  
  
SELECT * FROM func;
```

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

## 11. Índices

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

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

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

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

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

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

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

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

Sintaxe:

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

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

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

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

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

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

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

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

Exemplo 4: Destruindo um índice:

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

## Atividade Pontuada Teams – Lista 6

## 12. Visões - View

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

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

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

Vantagens de uso:

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

### Exemplo1: View sobre a junção cliente e passagem

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

SELECT * FROM cliente_passagem;
```

### Exemplo2: Destruindo uma view

```
DROP VIEW cliente_passagem;
```

## 13. Transações

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

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

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

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

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

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

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

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

### 13.1. Estados de uma transação

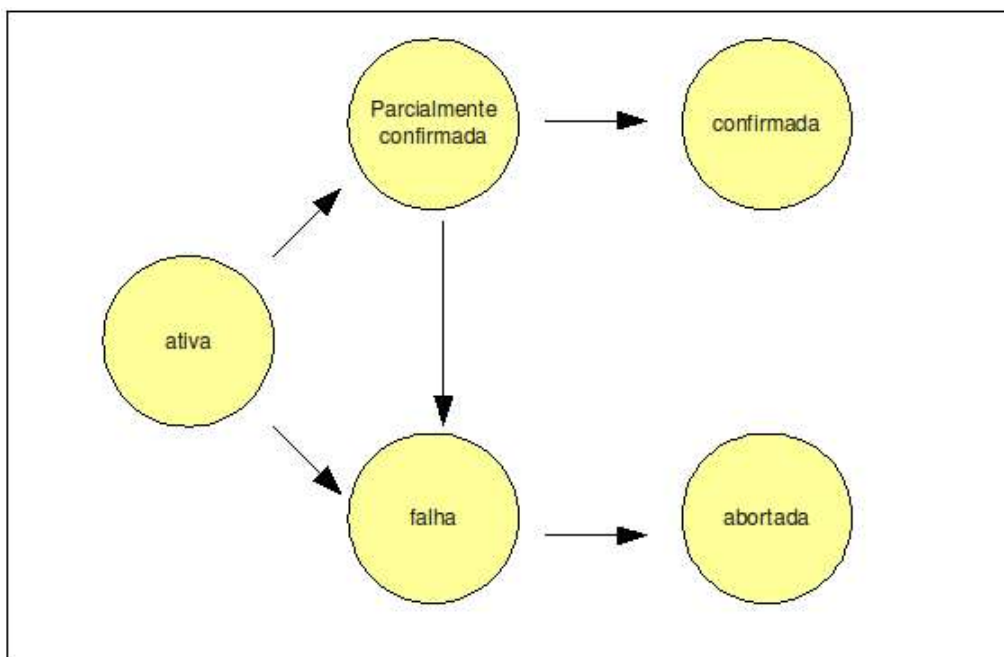
Na ausência de falhas, todas as transações são completadas com sucesso. Porém, uma transação nem sempre pode completar sua execução com sucesso. Caso isso ocorra, essa transação é considerada abortada.

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

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

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

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



## 13.2. Transações no PostgreSQL

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

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

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

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

Exemplo:

```

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

```

CREATE TABLE filme(
    id          numeric(5),
    nome        varchar(100),
  
```

```

        duracao          varchar(20),
        sinopse          varchar(200),
        data             date,
        genero           numeric(5),
        constraint pk_filme primary key (id),
        constraint fk_filme_genero foreign key (genero) references genero(id)
    );

CREATE SEQUENCE id_genero;
CREATE SEQUENCE id_filme;

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

SELECT * FROM filme;

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

SELECT * FROM filme;

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

SELECT * FROM filme;

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

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

SELECT * FROM genero;
SELECT * FROM filme;

```

## Atividade Pontuada Teams – Lista 7



## 14. Stored Procedures em SQL (Functions)

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

O PostgreSQL conta com três formas diferentes de criar funções:

- **Funções em Linguagem SQL:** são funções que utilizam a sintaxe SQL e se caracterizam por não possuírem estruturas de condição (if, else, case), estruturas de repetição (while, do while, for), não permitem a criação de variáveis e utilizam sempre algum dos seguintes comandos SQL: SELECT, INSERT, DELETE ou UPDATE.

- **Funções de Linguagens Procedurais:** ao contrário das funções SQL, aqui é permitido o uso de estruturas de condição e repetição e o uso de variáveis. As funções em linguagens procedurais caracterizam-se também por não possuírem apenas uma possibilidade de linguagem, mas várias. Normalmente a mais utilizada é conhecida como PL/PgSQL, linguagem fortemente semelhante ao conhecido PL/SQL utilizado no Oracle.

- **Funções em Linguagens Externas ou de Rotinas Complexas:** são funções normalmente escritas em C++ que trazem consigo a vantagem de utilizarem uma linguagem com diversos recursos, na qual pode-se implementar algoritmos com grande complexidade. Tais funções são empacotadas e registradas no SGBD para seu uso futuro.

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

```
CREATE [ OR REPLACE ] FUNCTION nome ( [ tipo_do_parametro1 [, ...] ] )  
RETURNS tipo_retornado AS
```

```
,  
    Implementação_da_função;  
,
```

```
LANGUAGE 'SQL';
```

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

Na qual CREATE FUNCTION é o comando que define a criação de uma função, [OR REPLACE] informa que se acaso existir uma função com este nome, a atual função deverá sobrescrever a antiga. RETURNS tipo\_retornado informa o tipo de dado que será retornado ao término da função. Tais tipos de retornos são os convencionais como o INTEGER, FLOAT, VARCHAR, etc. As funções em SQL também permitem o retorno de múltiplos valores e para isso informa-se como retorno SETOF. Implementação\_da\_função, como o nome mesmo diz, traz as linhas de programação para a implementação da stored procedure. LANGUAGE está avisando em que linguagem está sendo implementada a função.

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

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

```
CREATE SEQUENCE idpessoas;
```

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

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

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

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

Para visualizar o resultado dessa função podemos usar:

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

Quando passamos parâmetros à função, não utilizamos nome nas variáveis que estão dentro dos parênteses da assinatura da função. Utilizamos apenas, separados por vírgulas, o tipo da variável de parâmetro. Para acessarmos o valor de tais parâmetros, devemos utilizar o '\$' mais o número da posição que ocupa nos parâmetros, seguindo a ordem da esquerda para a direita:

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

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

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

Quando desejar excluir uma função do sistema utilize o comando:

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

Para excluir uma função é necessário passar toda a sua assinatura (nome + parâmetros):

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

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

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

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

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

```
SELECT nome_da_funcao();
```

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

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

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

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

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

## 15. Funções em Linguagens Procedurais- PL/PgSQL

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

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

### **Alguns exemplos**

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

**select primeira\_funcao();**

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

**select \* from soma(5,1);**

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

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

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

```
select * from subtracao2 (5,1);
```

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

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

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

```
select mostra_valor();
```

## 16. Trigger

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

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

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

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

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

Argumentos:

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

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

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

Argumentos:

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

Para excluir uma trigger basta executar o comando abaixo:

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

Argumentos:

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

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

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

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

- ✓ **NEW** → Tipo de dado RECORD. Variável armazenando a nova linha do banco de dados para as operações de INSERT/UPDATE nos gatilhos no nível de linha. Esta variável é nula nos gatilhos no nível de declaração.
- ✓ **OLD** → Tipo de dado RECORD. Variável armazenando a linha antiga do banco de dados para as operações de UPDATE/DELETE nos gatilhos no nível de linha. Esta variável é nula nos gatilhos no nível de declaração.
- ✓ **TG\_NAME** → Tipo de dado name. Variável contendo o nome do gatilho realmente disparado.
- ✓ **TG\_WHEN** → Tipo de dado text. Uma cadeia de caracteres contendo BEFORE ou AFTER dependendo da definição do gatilho.
- ✓ **TG\_LEVEL** → Tipo de dado text. Uma cadeia de caracteres contendo ROW ou STATEMENT dependendo da definição do gatilho.
- ✓ **TG\_OP** → Tipo de dado text. Uma cadeia de caracteres contendo INSERT, UPDATE ou DELETE informando para qual operação o gatilho foi disparado.
- ✓ **TG\_RELID** → Tipo de dado oid. O ID de objeto da tabela que causou o disparo do gatilho.
- ✓ **TG\_RELNAME** → Tipo de dado name. O nome da tabela que causou o disparo do gatilho.
- ✓ **TG\_NARGS** → Tipo de dado integer. O número de argumentos fornecidos ao procedimento de gatilho pela declaração CREATE TRIGGER.
- ✓ **TG\_ARGV[]** → Tipo de dado matriz de text. Os argumentos da declaração CREATE TRIGGER. O contador do índice começa com 0 (zero). Índices inválidos (menor que 0 ou maior ou igual a TG\_NARGS) resultam em um valor nulo.

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

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

Se for retornado um valor não nulo, então a operação prossegue com este valor de linha. Retornar um valor de linha diferente do valor original de NEW altera a linha que será inserida ou atualizada (mas não tem efeito direto no caso do DELETE). Para alterar a linha a ser armazenada, é possível substituir valores únicos diretamente no NEW e retornar o NEW modificado, ou construir uma linha (registro) nova completa para retornar.

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

Considere a tabela pessoa para os exemplos 1 e 2:

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

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

```
CREATE SEQUENCE seq_log1;

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

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

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

END;
,
Language 'plpgsql';
```

Em seguida deve-se criar a trigger para a função acima. Antes de deletar ou atualizar uma linha na tabela pessoa execute a função func\_log().

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

SELECT * FROM PESSOA;
INSERT INTO PESSOA VALUES (1,'ZE',1123,95762);
UPDATE PESSOA SET NOME = 'JOSÉ' WHERE ID = 1;
DELETE FROM PESSOA WHERE ID = 1;
```



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

```
CREATE SEQUENCE seq_log2;

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

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

END;
,
Language 'plpgsql';

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

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

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

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

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

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

    --Verifica se o salário é negativo
    IF NEW.salario < 0 THEN
```

```
        RAISE EXCEPTION "% não pode ter um salário negativo", NEW.nome_emp;  
    END IF;  
  
    --Registrar quem alterou o pagamento e quando  
    NEW.ultima_data := "now";  
    NEW.ultimo_usuario := current_user;  
    RETURN NEW;  
END;  
,  
Language 'plpgsql';  
  
CREATE TRIGGER emp_carimbo BEFORE insert or update ON emp  
    FOR EACH ROW EXECUTE PROCEDURE emp_carimbo();  
  
INSERT INTO EMP (id,nome_emp,salario) VALUES (1,'BRUNO',-10);  
INSERT INTO EMP (id,nome_emp,salario) VALUES (1,'BRUNO',10000)
```

## Atividade Pontuada Teams – Lista 8

## Anexo A - Tipos de Dados – PostgreSQL

As informações a seguir foram extraídas da documentação oficial do PostgreSQL, que pode ser obtida através do seguinte endereço web:

<http://pgdocptbr.sourceforge.net/pg80/index.html>

### Tipos numéricos

Nome	Tamanho de armazenamento	Descrição	Faixa de valores
<i>smallint</i>	2 bytes	inteiro com faixa pequena	-32768 a +32767
<i>integer</i>	4 bytes	escolha usual para inteiro	-2147483648 a +2147483647
<i>bigint</i>	8 bytes	inteiro com faixa larga	-9223372036854775808 a 9223372036854775807
<i>decimal</i>	variável	precisão especificada pelo usuário, exato	sem limite
<i>numeric</i>	variável	precisão especificada pelo usuário, exato	sem limite
<i>real</i>	4 bytes	precisão variável, inexacto	precisão de 6 dígitos decimais
<i>double precision</i>	8 bytes	precisão variável, inexacto	precisão de 15 dígitos decimais
<i>serial</i>	4 bytes	inteiro com auto-incremento	1 a 2147483647
<i>bigserial</i>	8 bytes	inteiro grande com auto-incremento	1 a 9223372036854775807

### Tipos monetários

Nome	Tamanho de Armazenamento	Descrição	Faixa
<i>money</i>	4 bytes	quantia monetária	-21474836.48 a +21474836.47

### Tipos para cadeia de caracteres

Nome	Descrição
<i>character varying(n), varchar(n)</i>	comprimento variável com limite
<i>character(n), char(n)</i>	comprimento fixo, completado com brancos
<i>text</i>	comprimento variável não limitado

### Tipos para data e hora

Nome	Tamanho de Armazenamento	Descrição	Menor valor	Maior valor	Resolução
<i>timestamp [ (p) ] [ without time zone ]</i>	8 bytes	tanto data quanto hora	4713 AC	5874897 DC	1 microssegundo / 14 dígitos
<i>timestamp [ (p) ] with time zone</i>	8 bytes	tanto data quanto hora, com zona horária	4713 AC	5874897 DC	1 microssegundo / 14 dígitos
<i>interval [ (p) ]</i>	12 bytes	intervalo de tempo	-178000000 anos	178000000 anos	1 microssegundo / 14 dígitos
<i>date</i>	4 bytes	somente data	4713 AC	32767 DC	1 dia
<i>time [ (p) ] [ without time zone ]</i>	8 bytes	somente a hora do dia	00:00:00.00	23:59:59.99	1 microssegundo / 14 dígitos
<i>time [ (p) ] with time zone</i>	12 bytes	somente a hora do dia, com zona horária	00:00:00.00+12	23:59:59.99-12	1 microssegundo / 14 dígitos

## ***Tipo booleano***

Os valores literais válidos para o estado "verdade" são:

TRUE  
't'  
'true'  
'y'  
'yes'  
'1'

Para o estado "falso" podem ser utilizados os seguintes valores:

FALSE  
'f'  
'false'  
'n'  
'no'  
'0'

A utilização das palavras chave TRUE e FALSE é preferida (e em conformidade com o padrão SQL).