



BANCO DE DADOS

Osenias Oliveira

Que sou eu?

- Mais de 15 anos de experiência com Desenvolvimento de Software, Liderança de Projetos, Levantamento de Requisitos, Administração e Tuning de Banco de Dados.
- Analista de Sistemas Master na **FPF Tech** desde março de **2014**.
- Professor do Curso Técnico na Fucapi entre **2011** e **2016**.
- Certificação Scrum Master.
- Certificação Product Owner.
- Pós Graduado em Projeto e Administração de Banco de Dados pela **Uninorte**.
- Graduado em Desenvolvimento de Software pela **Uninorte**.

Ementa

- Porque usar PostgreSQL
- Quem usa PostgreSQL
- Conceitos básicos de modelagem relacional
- Configuração do ambiente
- Conceitos básicos
 - DDL – Data Definition Language
 - DCL – Data Control Language
 - DML – Data Manipulation Language
- SQL – Structure Query Language
- Um pouco do híbrido
- PL/PGSQL

Ementa

- Triggers
- Extensions
- Segmentação/Particionamento
- DBLink
- Dicas para criação de um banco de dados

Porque usar PostgreSQL

- Multiplataforma
- Open Source
- Alto desempenho
- Suporte a linguagens de programação
- Comunidade ativa
- Dados geométricos
- Suporte a JSON
- Híbrido
- Segmentação de forma transparente
- Suporte a replicação

Quem usa PostgreSQL

- Apple
- NASA
- Cisco
- Sony
- Fujitsu
- Skype
- Yahoo
- Caixa Econômica Federal
- FAB
- Metrô SP
- Governo do Ceará
- NTT (Telecom Japão)

Conceitos básicos de modelagem relacional

- No modelo entidade e relacionamento temos três principais conceitos aplicados:
 - Conjunto de entidades
 - Conjunto de relacionamentos
 - Atributos

Conceitos básicos de modelagem relacional

- **Entidades**
 - Compartilham as mesmas propriedades.
 - **Exemplo:**
 - Conjunto de pessoas, funcionários, empresas.

Conceitos básicos de modelagem relacional

- **Atributos**

- Cada entidade tem seus atributos.
- Atributos são propriedades particulares que descreve cada entidade.
- **Exemplo:**
 - A entidade funcionário pode ser descrita pelo nome, idade, salário e departamento.
- Uma entidade terá um valor para cada um de seus atributos.

Conceitos básicos de modelagem relacional

- Toda entidade possui uma super chave, quando convertemos o modelo conceitual, para o modelo lógico, uma entidade virá uma tabela e a **super chave** virá **chave primária**.
- Sabemos então que toda tabela deve possuir uma chave primária.
- É bem comum quando estudamos o modelo conceitual ouvirmos coisas como: **super chave** deve ser um atributo que identifica a entidade como única. Conceitualmente falando faz sentido, quando levamos isso para o mundo real é totalmente diferente.
- **Surrogate** é vida.

Conceitos básicos de modelagem relacional

- Criar chaves primárias com CPF por exemplo podem impactar em performance e mais armazenamento, uma vez que um CPF é composto sempre de um número no seguinte formato, **000.000.000-00**, enquanto um surrogate pode ser um número inteiro que vai ocupar menos em disco e vai ser mais rápido em consultas e relacionamentos (JOIN).
- Dentro do modelo conceitual existe o conceito de **chave alternativa** que é um atributo que tem potencial para ser uma **super chave**, mas não será, essa vai ser sempre a melhor opção, falando no modelo lógico esse atributo é definido com uma **UNIQUE**.

Conceitos básicos de modelagem relacional

- **Relacionamento**

- É uma associação entre duas ou mais entidades.

- **Exemplo:**

- Osenias **Trabalha_em** Projeto LevelUp

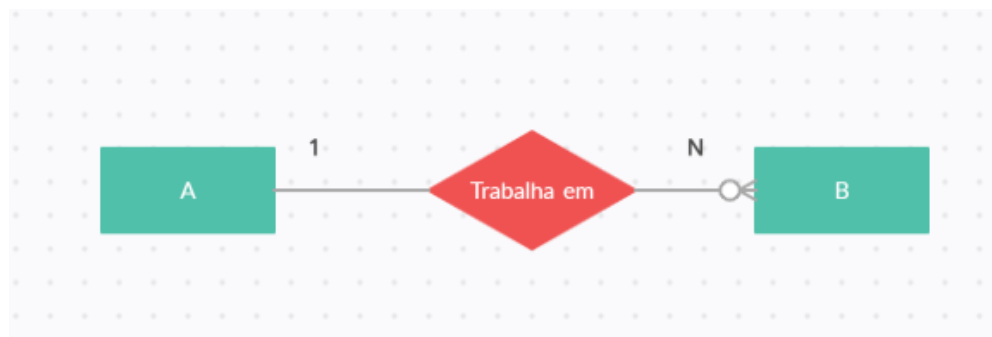


Conceitos básicos de modelagem relacional

- Quando falamos de relacionamento temos um conceito bem importante a cardinalidade.
- Cardinalidade é o grau em que uma entidade se relaciona com outra.
- Existem 3 tipos:
 - 1 para N
 - N para N
 - 1 para 1

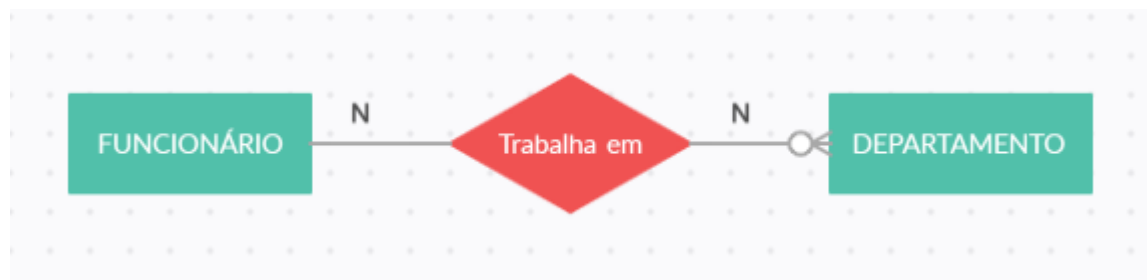
Conceitos básicos de modelagem relacional

- **1 para N**
- Em uma relação entre a entidade A e entidade B, onde 1 é do lado A e N é do lado B, a regra é que a **super chave** da entidade A será adicionada como **chave estrangeira** na entidade B quando a conversão de modelo conceitual para lógico for realizado.



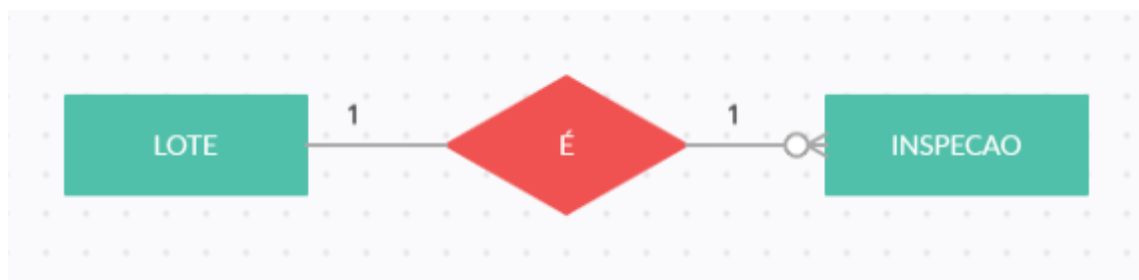
Conceitos básicos de modelagem relacional

- **N para N**
- Em uma relação entre a entidade Funcionário e entidade Departamento, a regra é que o relacionamento entre essas duas entidades quando convertido para o modelo lógico virá uma nova tabela, com um contexto geralmente de uma tabela associativa.



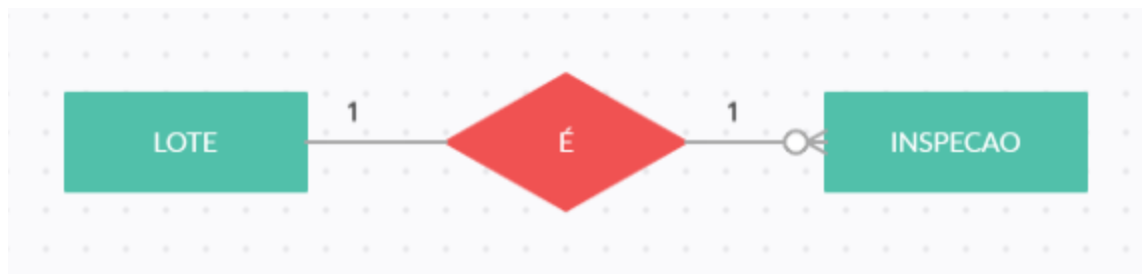
Conceitos básicos de modelagem relacional

- **1 para 1**
- Eu costumo dizer que a relação 1 para 1 nada mais é do que uma relação 1 para N com índice único, mas como assim?
- Vamos assumir que em um sistema de inspeção de lotes de produtos, um lote pode ser inspecionado uma única vez.
- Temos a seguinte modelagem.



Conceitos básicos de modelagem relacional

- Nesse cenário podemos escolher o lado da relação que será a chave estrangeira mas, geralmente a **super chave** do lote vai para inspeção como chave estrangeira, quando passado para o modelo lógico, e essa **chave estrangeira** fica como **UNIQUE**.



Configuração do ambiente

- Vamos utilizar a última versão 13 do PostgreSQL.
- Link para download
- <https://www.enterprisedb.com/downloads/postgres-postgresql-downloads>
- Você pode também utilizar um container **docker**.

Configuração do ambiente

- Como **client** para executarmos nossos comandos iremos usar o **PgAdmin** mas, você pode utilizar o **client** de sua escolha.
- O PgAdmin geralmente já vem com instalador do PostgreSQL mas, você também pode utilizar com o **docker** utilizando o compose abaixo.

Configuração do ambiente

- O compose.

```
version: '3.1'

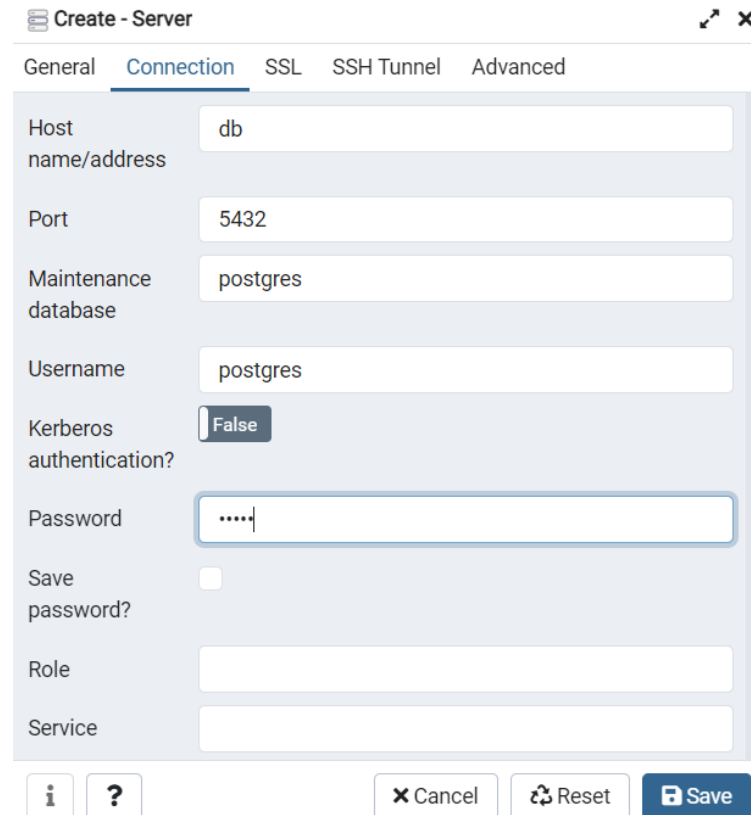
services:
  db:
    image: postgres
    ports:
      - 5434:5432
    restart: always
    environment:
      POSTGRES_PASSWORD: admin
    volumes:
      - dbdata:/var/lib/postgresql/data

  pg:
    image: dpage/pgadmin4
    restart: always
    ports:
      - 9090:80
    environment:
      PGADMIN_DEFAULT_EMAIL: admin@admin.com
      PGADMIN_DEFAULT_PASSWORD: 123456

volumes:
  dbdata:
```

Configuração do ambiente

- Configuração inicial do servidor.



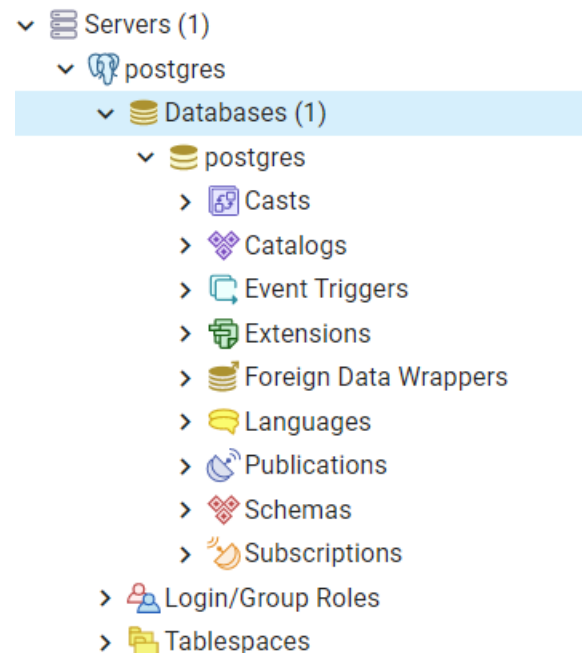
The screenshot shows a 'Create - Server' dialog box with the 'Connection' tab selected. The form contains the following fields and controls:

- Host name/address:** Text input field containing 'db'.
- Port:** Text input field containing '5432'.
- Maintenance database:** Text input field containing 'postgres'.
- Username:** Text input field containing 'postgres'.
- Kerberos authentication?:** A toggle switch currently set to 'False'.
- Password:** A text input field with masked characters '....'.
- Save password?:** An unchecked checkbox.
- Role:** An empty text input field.
- Service:** An empty text input field.

At the bottom of the dialog, there are three buttons: an information icon (i), a question mark icon (?), and a set of action buttons including 'Cancel', 'Reset', and 'Save'.

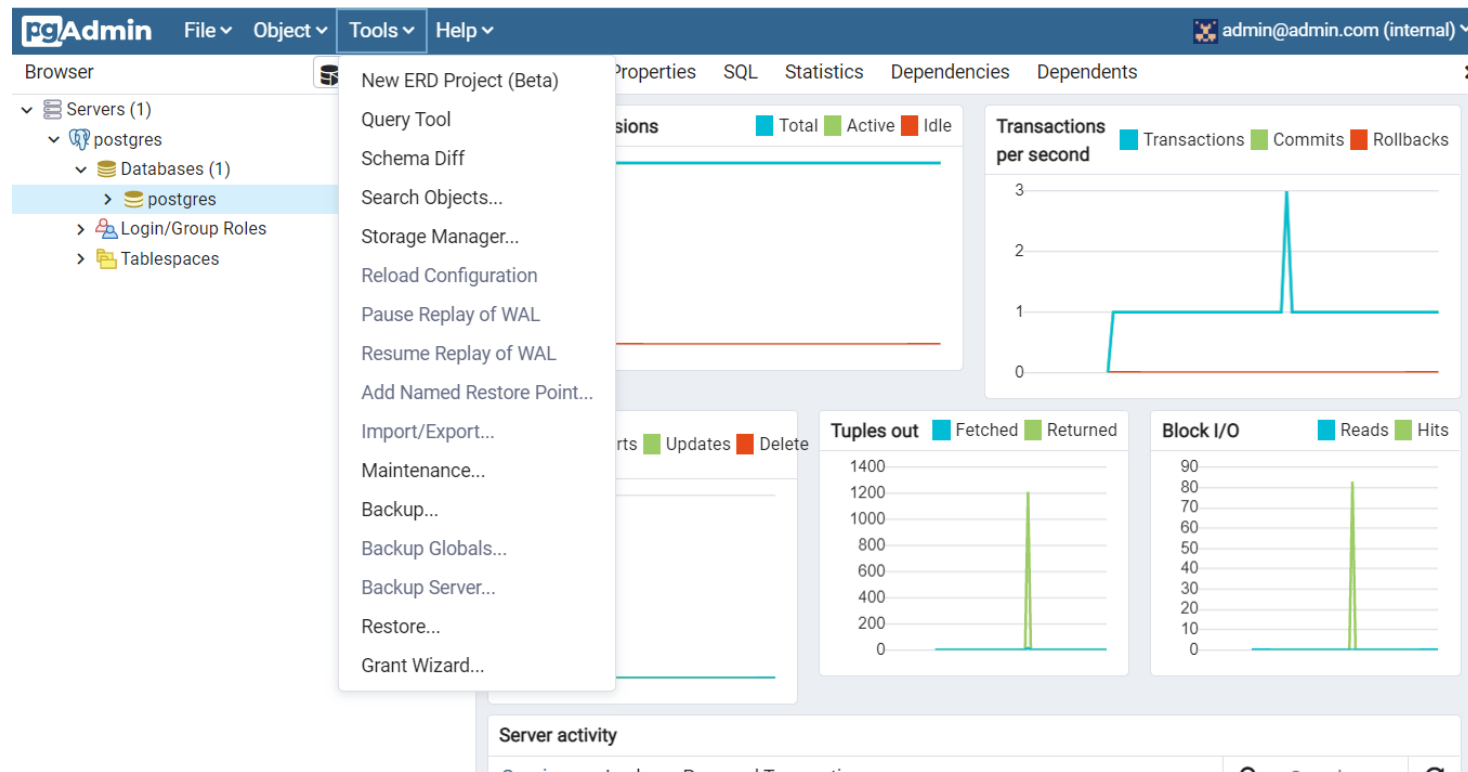
Configuração do ambiente

- Inicialmente existe apenas um banco de dados **postgres**, que é um banco de dados apenas para armazenamento de metadados do **postgres**. Não devemos criar nada ou manipular algo nesse banco de dados.



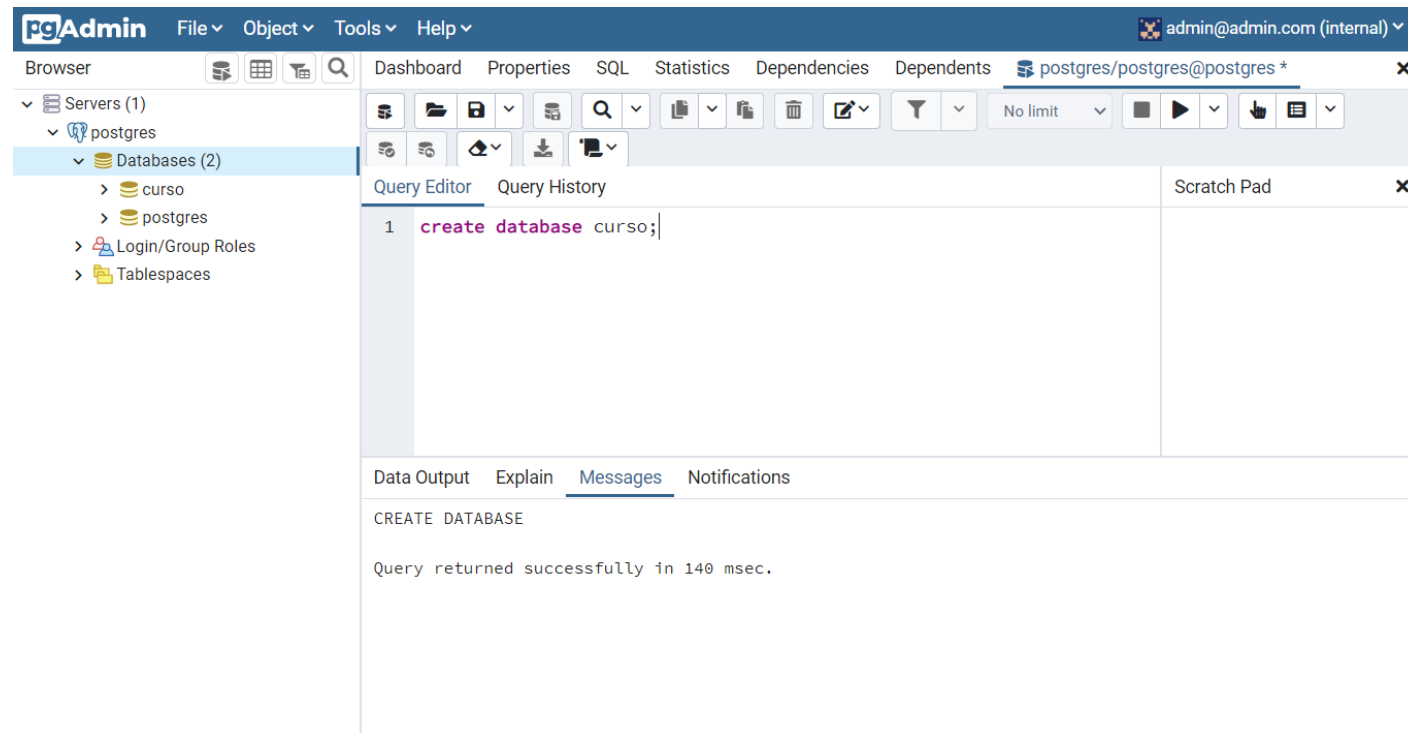
Configuração do ambiente

- Para criarmos nosso primeiro banco de dados vamos acessar a ferramenta para execução de comandos e fazê-lo. **Query Tool**.



Configuração do ambiente

- Basta executar o comando e pressionar **F5**, após isso basta fazer um **refresh** na árvore de banco de dados que seu novo banco de dados irá aparecer.



Configuração do ambiente

- Para fazermos nossas modelagens lógicas utilizaremos o ERD Concepts.
- Link para download.
- <https://www.erdconcepts.com/>

Conceitos básicos

- **DDL**
- Linguagem de definição de dados, estão associados a **CREATE, ALTER E DROP**, ou seja, está relacionado as estruturas criadas no seu banco de dados.

Conceitos básicos

- **CREATE**

```
create database curso;
```

```
create table cliente (  
    id integer not null,  
    nome varchar(100) not null  
);
```

```
create sequence incrementador start 1 increment 1;
```

```
create schema cadastros;
```

```
create user ozzy with encrypted password '123456';
```

Conceitos básicos

- **ALTER**

```
alter table cliente  
add column salario numeric(10, 2);
```

```
alter sequence incrementador restart 2;
```

```
alter user ozzy superuser;
```

```
alter table cliente  
add column data_aniversario date;
```

Conceitos básicos

- **DROP**

```
drop sequence incrementador;
```

```
drop user ozzy;
```

```
drop table cliente;
```

```
drop schema cadastros;
```

Conceitos básicos

- **PRIMARY KEY**
- Como já sabemos a chave primária é um conjunto de colunas que compõem a unicidade de um registro no banco de dados.
- Já sabemos também que usar CPF como PK, não.
- Em grande parte dos casos criaremos **surrogates**.
- Então vamos aprender como criar a estrutura no banco de dados.

Conceitos básicos

- **PRIMARY KEY**

```
create table cliente (  
    id integer not null primary key,  
    nome varchar(100) not null  
);  
  
create table funcionario (  
    id serial not null,  
    nome varchar(100) not null,  
    constraint pk_funcionario primary key (id)  
);
```

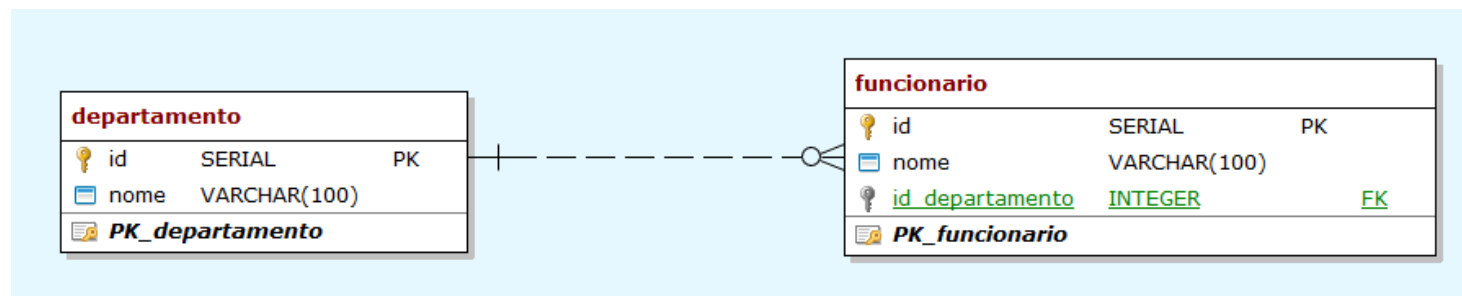
Conceitos básicos

- **PRIMARY KEY**
- Nossas chaves primárias também podem ser compostas de várias colunas, isso não quer dizer que nossa tabela tem mais de uma chave primária, isso nem é possível, mas sim uma chave primária composta.
- **Não recomendo!!!**

```
create table bairro (  
    nome varchar(100) not null,  
    zona varchar(20) not null,  
    constraint pk_bairro primary key (nome, zona)  
);
```


Conceitos básicos

- **FOREIGN KEY**
- Também chamada de chave estrangeira determina a relação entre duas tabelas.
- Um funcionário trabalha em um departamento e um departamento pode conter um ou mais funcionários.



Conceitos básicos

- **FOREIGN KEY**

```
create table funcionario (  
    id serial not null,  
    nome varchar(100) not null,  
    id_departamento integer not null,  
    constraint pk_funcionario primary key (id),  
    constraint fk_funcionario_departamento foreign key (id_departamento) references departamento (id)  
);
```

ID	NOME
1	TI
2	FINANCEIRO

ID	NOME	ID_DEPARTAMENTO
1	OSENIAS	1
2	IZZIE	1
3	TATIANY	2
4	ELOAH	4

Conceitos básicos

- **FOREIGN KEY**

```
create table bairro (  
    nome varchar(100) not null,  
    zona varchar(20) not null,  
    constraint pk_bairro primary key (nome, zona)  
);
```

```
alter table funcionario  
add column nome_bairro varchar(100),  
add column zona_bairro varchar(20),  
add constraint fk_funcionario_bairro foreign key (nome_bairro, zona_bairro)  
references bairro (nome, zona);
```

Conceitos básicos






- **VALORES DEFAULT**

- Podemos definir valores default para as colunas, isso quer dizer que não informadas no momento da inserção dos dados elas serão preenchidas automaticamente.

```
create table produto (  
    id serial not null,  
    nome varchar(100) not null,  
    preco_venda numeric(10, 2) default 10.00,  
    ativo boolean default true,  
    constraint pk_produto primary key (id)  
);  
  
insert into produto (nome) values ('Cola-cola');  
insert into produto (nome) values ('Fanta laranja');  
insert into produto (nome) values ('Suco de uva');
```

Conceitos básicos

- **VALORES DEFAULT**
- Resultado

	Data Output	Explain	Messages	Notifications
	 id [PK] integer 	nome character varying (100) 	preco_venda numeric (10,2) 	ativo boolean 
1	1	Cola-cola	10.00	true
2	2	Fanta laranja	10.00	true
3	3	Suco de uva	10.00	true

Conceitos básicos

- Exercício
 - Criar um schema vendas onde nossa tabela deverá ser criada;
 - Criar uma tabela de fornecedor com **NOME, CNPJ**, não é para criar chave primárias nesse momento;
 - Adicionar uma coluna para identificar se o registro está ou não ativo, o valor default deve ser verdadeiro;
 - Adicionar uma coluna para data de cadastro, o valor default deve ser a data atual;
 - Criar um sequence para controlar o incremento do **ID** da tabela fornecedor;
 - Criar uma coluna **ID** e definir ela como chave primária da tabela de fornecedor e setar o valor default com o sequence.

Conceitos básicos

- **CHECK**

- Essa constraint serve para criar regras para as colunas no momento da manipulação dos dados.
- Por exemplo um funcionário que possui uma coluna para definir o sexo, sabemos que os valores possíveis são apenas **Masculino** e **Feminino**.
- Sabemos que o ideal é armazenarmos apenas **M** para masculino e **F** para feminino.
- Vamos então definir uma **CHECK CONSTRAINT** para determinar que apenas esses valores sejam possíveis na tabela de funcionário.

Conceitos básicos

- **CHECK**

```
create table funcionario (  
    id serial not null,  
    nome varchar(100) not null,  
    id_departamento integer not null,  
    constraint pk_funcionario primary key (id),  
    constraint fk_funcionario_departamento foreign key (id_departamento) references departamento (id)  
);  
  
alter table funcionario  
add column sexo varchar(1);  
  
alter table funcionario  
add constraint check_sexo_funcionario check ( sexo in ('M', 'F') );
```


Conceitos básicos

- **CHECK**

```
insert into departamento (nome) values ('TI');  
insert into departamento (nome) values ('Financeiro');  
  
insert into funcionario (nome, sexo, id_departamento) values ('Osenias', 'M', 1);  
insert into funcionario (nome, sexo, id_departamento) values ('Izzie', 'A', 1);
```

Data Output	Explain	<u>Messages</u>	Notifications
-------------	---------	-----------------	---------------

ERROR: new row for relation "funcionario" violates check constraint "check_sexo_funcionario"			
DETAIL: Failing row contains (2, Izzie, 1, A).			
SQL state: 23514			

Conceitos básicos

- **CHECK**
- Exercício
 - Criar uma coluna para o preço de custo do produto;
 - Criar uma regra para que o campo preço de custo seja maior que 0;
 - Criar uma regra para que o campo preço de venda seja maior que 0;
 - Criar uma regra para que o preço de venda seja maior que o preço de custo;

Conceitos básicos

- **Exercício de modelagem**
- Um pequeno país resolveu informatizar sua única delegacia de polícia para criar um banco de dados onde os criminosos deverão ser fichados, sendo que as suas vítimas também deverão ser cadastradas. No caso de criminosos que utilizem armas, estas deverão ser cadastradas e relacionadas ao crime cometido para possível utilização no julgamento do criminoso. O sistema, além de fornecer dados pessoais dos criminosos, das vítimas e das armas, também deve possibilitar saber:

Conceitos básicos

- **Exercício de modelagem**

- Quais crimes um determinado criminoso cometeu, lembrando que um crime pode ser cometido por mais de um criminoso;
- Quais crimes uma determinada vítima sofreu, lembrando que várias vítimas podem ter sofrido um mesmo crime;

Conceitos básicos

- **INDEX**
- Temos um índice específico para controle de unicidade UNIQUE INDEX.

```
CREATE UNIQUE INDEX ak_nome ON departamento (nome);
```

- Também temos índices relacionados a performance e devem ser usados de acordo com seus cenários.

Conceitos básicos

- **INDEX**
- Vamos fazer uma analogia do nosso banco de dados.
- Imagine que nosso banco de dados seja um livro com mil páginas, porém nosso livro não possui um sumário, a medida que precisemos consultar um determinado conteúdo, sentiremos dificuldade em encontrar tal conteúdo, pois, precisaríamos olhar página a página. Caso nosso livro tivesse um sumário seria bem mais fácil checar no sumário o conteúdo e ir direto na página desejada.

Conceitos básicos

- **INDEX**

- A ideia é que um índice vai cumprir é similar, inclusive para decidir em que momento criar um índice. Vamos supor que nosso livro tivesse apenas 5 páginas, não faria muito sentido criar um sumário, pois, seria bem rápido pesquisar qualquer conteúdo nesse livro, seria um esforço a mais para algo que relativamente seria simples e rápido resolver.
- Resumindo os índices vão nos ajudar a buscar de forma mais rápida um determinado conteúdo.

Conceitos básicos

- **INDEX**
- Grande parte dos cenários o tipo de índice **B-Tree** irá resolver, só usaremos um índice do tipo **HASH** quando o conjunto de campos determinados no índice é usado mais em comparações de igualdade.

```
create index idx_nome_funcionario on funcionario using btree (nome);
```

```
create index idv_sexo_funcionario on funcionario using hash (sexo);
```


Conceitos básicos

- **Quando devo criar índices?**
- 1) índices não devem ser criados no início do projeto de banco de dados;
- 2) precisa ter realmente uma grande massa de dados para que você decida criar índices;
- 3) avalie as suas consultas, verifique as condições usadas nas consultas (**where**) isso vai te dar uma boa métrica para tomar as decisões.

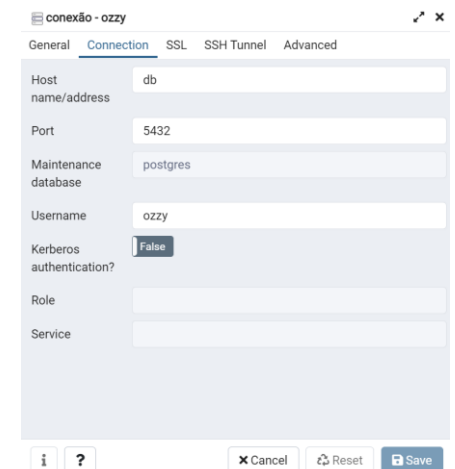
Conceitos básicos

- **Exercício**
- Na tabela de bairro criada anteriormente, onde nós definimos uma chave composta, faça as seguintes questões.
 - 1) apagar a chave primária composta;
 - 2) criar uma chave primária simples com um serial;
 - 3) criar um índice único composto por nome do bairro e nome da zona;
 - 4) criar um índice hash para o nome da zona;

Conceitos básicos

- **DCL**
- É a linguagem que define a parte de controle de estrutura e dados, geralmente relacionada aos comandos **GRANT E REVOKE**.
- Vamos criar um usuário sem acesso de super usuário e logo em seguida criar uma nova conexão para esse usuário.

```
create user ozzy with encrypted password '123';
```



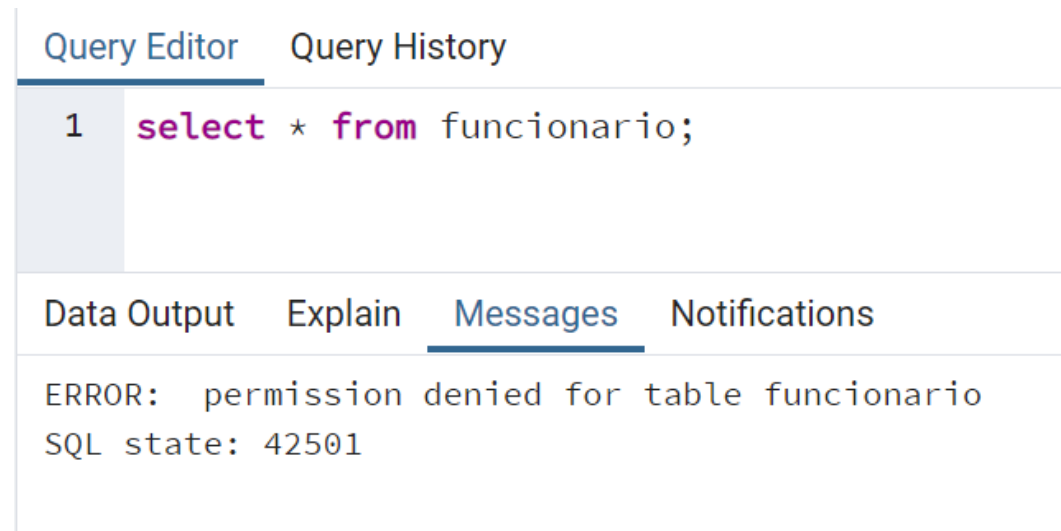
The screenshot shows a dialog box titled 'conexão - ozzy' with a close button (X) in the top right corner. Below the title bar are four tabs: 'General', 'Connection' (selected), 'SSL', 'SSH Tunnel', and 'Advanced'. The 'Connection' tab contains the following fields:

- Host name/address: db
- Port: 5432
- Maintenance database: postgres
- Username: ozzy
- Kerberos authentication?: False (checkbox)
- Role: (empty field)
- Service: (empty field)

At the bottom of the dialog are three buttons: 'Cancel', 'Reset', and 'Save'.

Conceitos básicos

- **DCL**
- Depois da conexão criada vamos tentar fazer uma consulta simples na tabela de funcionário.



The screenshot shows a database interface with two main sections. The top section, titled 'Query Editor' and 'Query History', contains a single line of SQL code: '1 select * from funcionario;'. The bottom section, titled 'Data Output', 'Explain', 'Messages', and 'Notifications', shows an error message: 'ERROR: permission denied for table funcionario' and 'SQL state: 42501'.

```
Query Editor  Query History
```

```
1 select * from funcionario;
```

```
Data Output  Explain  Messages  Notifications
```

```
ERROR: permission denied for table funcionario
SQL state: 42501
```

Conceitos básicos

- **DCL**
- Aqui começa a saga das permissões.
- O PostgreSQL possui um controle de permissões a nível de objetos, então você pode atribuir as permissões a cada objeto e indicando qual a operação desejada.

```
grant select on funcionario to ozzy;  
grant insert on funcionario to ozzy;  
grant update on funcionario to ozzy;  
grant delete on funcionario to ozzy;
```

Conceitos básicos

- **DCL**
- Outros tipos de permissão.

Privilege	Abbreviation	Applicable Object Types
SELECT	r ("read")	LARGE OBJECT, SEQUENCE, TABLE (and table-like objects), table column
INSERT	a ("append")	TABLE, table column
UPDATE	w ("write")	LARGE OBJECT, SEQUENCE, TABLE, table column
DELETE	d	TABLE
TRUNCATE	D	TABLE
REFERENCES	x	TABLE, table column
TRIGGER	t	TABLE
CREATE	C	DATABASE, SCHEMA, TABLESPACE
CONNECT	c	DATABASE
TEMPORARY	T	DATABASE
EXECUTE	X	FUNCTION, PROCEDURE
USAGE	U	DOMAIN, FOREIGN DATA WRAPPER, FOREIGN SERVER, LANGUAGE, SCHEMA, SEQUENCE, TYPE

Conceitos básicos

- **DCL**
- Outros objetos para atribuir permissão.

Object Type	All Privileges	Default PUBLIC Privileges	psql Command
DATABASE	CTc	Tc	\l
DOMAIN	U	U	\dD+
FUNCTION or PROCEDURE	X	X	\df+
FOREIGN DATA WRAPPER	U	none	\dew+
FOREIGN SERVER	U	none	\des+
LANGUAGE	U	U	\dL+
LARGE OBJECT	rw	none	
SCHEMA	UC	none	\dn+
SEQUENCE	rwU	none	\dp
TABLE (and table-like objects)	arwdDxt	none	\dp
Table column	arwx	none	\dp
TABLESPACE	C	none	\db+
TYPE	U	U	\dT+

Conceitos básicos

- **DCL**
- Olha que massa! Você consegue dar permissão de consulta em apenas uma coluna da tabela;

```
grant select (id) on produto to ozzy;
```


Conceitos básicos

- **DCL**
- Podemos também remover as permissões de um determinado usuário.

```
revoke select on produto from ozzy;
```

```
grant all on schema public to ozzy;
```

```
revoke all on schema public from ozzy;
```

Conceitos básicos

- **DCL**
- Podemos também criar grupos com suas permissões definidas e associar a uma determinado usuário;

```
create user izzie with encrypted password '123';
```

```
create role leitura;
```

```
alter group leitura add user ozzy;
```

```
alter group leitura add user izzie;
```

```
grant select on table produto to leitura;
```

- Criar as conexões com os usuários criados para testar o funcionamento das permissões.

Conceitos básicos

- **DCL**
- Tendo em vista que a consulta abaixo retorna as tabelas nos bancos da sua instância do PostgreSQL e que o exemplo em seguida faz concatenações de string, crie uma consulta para dar permissão de leitura para todas as tabelas para um determinado usuário.

```
select * from pg_tables;
```

schemaname name	tablename name	tableowner name	tablespace name	hasindexes boolean	hasrules boolean	hastriggers boolean	rowsecurity boolean
vendas	fornecedor	postgres	[null]	true	false	false	false
public	cliente	postgres	[null]	true	false	false	false

```
select concat('Brasil', ' ', 'Penta Campeão');
```

Conceitos básicos

- **DML**
- É a linguagem de manipulação dos dados refere-se aos comandos de **INSERT, UPDATE, DELETE** e controle de transação.
- Para realizarmos o INSERT é bem simples, basta você especificar o nome de colunas e valores que você deseja inserir.

```
insert into departamento (nome) values ('RH');  
insert into departamento (nome) values ('Teste');  
insert into departamento (nome) values ('Gestão');  
insert into departamento (nome) values ('Direção');
```

Conceitos básicos

- **DML**
- Para realizar um UPDATE simples você especifica a tabela e os campos que deseja alterar. É bem importante em um UPDATE determinar a clausula **WHERE**, caso contrário todos os registros serão afetados.

```
update departamento set nome = 'Direção' where id = 6;
```

```
update produto set nome = id::varchar || '-' || nome;
```

```
update departamento set nome = foo.id || ' - ' || foo.nome  
from (select * from departamento d) as foo  
where departamento.id = foo.id;
```

Conceitos básicos

- **DML**
- No delete basta você especificar a tabela e a clausula **WHERE**.
- Cuidado para não fazer besteira e executar um **DELETE** sem **WHERE**.

```
delete from departamento where id = 6;
```

```
delete from funcionario where id in (1, 2);
```




```
delete from departamento;
```

```
ERROR: update or delete on table "departamento" violates foreign key constraint "fk_funcionario_departamento" on table "funcionario"  
DETAIL: Key (id)=(2) is still referenced from table "funcionario".  
SQL state: 23503
```

Conceitos básicos

- **DML**
- Todo SGBD relacional possui um controle de transação, que garante que uma operação seja desfeita. Por padrão o PostgreSQL vem com uma configuração de **autocommit** = **true**, ou seja, as operações de DML que fazemos não poderão ser desfeitas, mas você pode de forma explícita controlar essas transações.

```
select * from departamento;
```

Data Output	Explain	Messages	Notific
			
id [PK] integer	nome character varying (104)		
1	1	1 - TI	
2	2	2 - Financeiro	
3	3	3 - RH	
4	4	4 - Teste	

Conceitos básicos

- **DML**
- Vamos fazer um DELETE utilizando o controle transacional.

```
begin;  
  delete from departamento where id = 4;
```

```
WARNING:  there is already a transaction in progress  
DELETE 1
```

```
Query returned successfully in 42 msec.
```


Conceitos básicos

- **DML**
- Vamos fazer novamente a consulta para ver como nossa tabela está, lembra que a transação ainda tá em progresso.

```
select * from departamento;
```



	Data Output	Explain	Messages	Notifica
	id [PK] integer		nome character varying (104)	
1		1	1 - TI	
2		2	2 - Financeiro	
3		3	3 - RH	

Conceitos básicos

- **DML**
- Agora você pode decidir se quer ou não confirmar a transação, para confirmar **COMMIT** para cancelar **ROLLBACK**.

```
rollback;
```

```
select * from departamento;
```

	Data Output	Explain	Messages	Notifica
	<div>id [PK] integer </div>		<div>nome character varying (104) </div>	
1	1	1	1 - TI	
2	2	2	2 - Financeiro	
3	3	3	3 - RH	
4	4	4	4 - Teste	

Conceitos básicos

- **Exercício**
- Criar uma coluna salário na tabela funcionário com valor default 0.00 obrigatória;
- Fazer um update para atualizar o salário para o id * 1000;
- Criar um bloco anônimo para gerar um loop de 1000 iterações e cadastrar vários funcionários.
 - O nome do funcionário deve ser no formato FUNCIONARIO_{CONTADOR};
 - O departamento deve ser sempre o de TI;
 - o sexo vai depender do valor do contador se for ímpar Masculino, senão feminino;
 - Salário deve ser gerado de forma randômica;
- Criar um usuário orelha que tem permissão de fazer **SELECT** na tabela de funcionário porém ele não pode em hipótese alguma visualizar o salário;

SQL

- Linguagem de consulta estruturada, é uma linguagem de pesquisa declarativa utilizada em bancos de dados relacionais.
- Estrutura para padrão para uma consulta SQL.

```
select colunas... from tabela  
where filtros...
```

```
select * from tabela
```

```
select coluna as apelido from tabela apelido_tabela
```

SQL

- Antes de começarmos vamos criar uma base de dados nova chamada **sale**.
- **Exercício**
 - Executar o script DDL para criar a estrutura da base de dados;
 - Criar um script para criar em todas as tabelas de vendas as seguintes colunas:
 - created_at TIMESTAMP valor padrão now();
 - modified_at TIMESTAMP valor padrão now();
 - active BOOLEAN valor padrão true;
 - Executar o script de DML para popular a base de dados;

SQL

- Para que possamos fazer os filtros em nossas consultas vamos ver alguns operadores.
- Operados matemáticos.

Operador	Descrição	Exemplo	Resultado
+	adição	2 + 3	5
-	subtração	2 - 3	-1
*	multiplicação	2 * 3	6
/	divisão (divisão inteira trunca o resultado)	4 / 2	2
%	módulo (resto)	5 % 4	1
^	exponenciação	2.0 ^ 3.0	8
/	raiz quadrada	/ 25.0	5
/	raiz cúbica	/ 27.0	3
!	fatorial	5 !	120
!!	fatorial (operador de prefixo)	!! 5	120
@	valor absoluto	@ -5.0	5
&	AND bit a bit	91 & 15	11
	OR bit a bit	32 3	35
#	XOR bit a bit	17 # 5	20
~	NOT bit a bit	~1	-2
<<	deslocamento à esquerda bit a bit	1 << 4	16
>>	deslocamento à direita bit a bit	8 >> 2	2

SQL

- Operadores lógicos

AND
OR
NOT

O SQL utiliza a lógica booleana de três valores, onde o valor nulo representa o "desconhecido". Devem ser observadas as seguintes tabelas verdade:

<i>a</i>	<i>b</i>	<i>a AND b</i>	<i>a OR b</i>
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	NULL	NULL	TRUE
FALSE	FALSE	FALSE	FALSE
FALSE	NULL	FALSE	NULL
NULL	NULL	NULL	NULL

<i>a</i>	NOT <i>a</i>
TRUE	FALSE
FALSE	TRUE
NULL	NULL

SQL

- Operadores de comparação

Operador	Descrição
<	menor
>	maior
<=	menor ou igual
>=	maior ou igual
=	igual
<> ou !=	diferente

SQL

- Outros operadores

Operador	Descrição
IN	Operador de igualdade utilizado para listas, ou seja, para saber se está dentro da lista
LIKE	Para busca de texto contendo. Aqui você utiliza % no início para indicar que é terminando com, % no fim para indicar que é iniciando com e % em ambas extremidades caso queira contendo em qualquer parte do texto.
ILIKE	Mesma função do like porém case insensitive.
BETWEEN	Para comparação de intervalos
IS	Utilizado para valores nulos ou booleanos.

SQL

- Utilizamos a clausula WHERE para realizar nossos filtros.
- Estrutura básica.

```
where campo = valor  
where campo = valor and 1 = 1
```

SQL

- **Exercício**
- Fazer uma consulta para retornar todos os funcionários que contenham silva no nome;
- Fazer um consulta para retornar todos os funcionários com o salário maior que R\$ 5.000;
- Fazer uma consulta para trazer todos os clientes que tem uma renda mensal inferior a R\$ 2.000,00;
- Fazer uma consulta para retornar todos os funcionários admitidos entre 2010 e 2021.

SQL

- **Exercício**

- Fazer uma consulta para retornar todos os funcionários casados ou solteiros;
- Fazer uma consulta para retornar todos os funcionários que ganham entre R\$ 1.000,00 e R\$ 5.000,00;
- Fazer uma consulta que retorne a diferença do preço de custo e preço de venda dos produtos;
- Fazer uma consulta para retornar todos os funcionários que não tenham salário entre R\$ 4.000,00 e R\$ 8.000,00;

SQL

- **Exercício**
- Fazer uma consulta para retornar todos os clientes que já tenham alguma venda, não pode usar JOIN;
- Fazer uma consulta para retornar todos os funcionários que já tenham alguma venda, não pode usar JOIN;
- Fazer uma consulta para retornar todas as vendas entre 2010 e 2021;

SQL

- Quando precisamos usar uma estrutura de decisão em nosso SQL?
- Temos como fazer utilizando a função CASE.

```
select
    case when condicao then 'Valor retornado'
         when outra_condicao then 'Valor retornado'
         else 'Valor default retornado'
    end
from tabela
```

SQL

- **Exercício**
- Fazer uma consulta para retornar o nome do funcionário e o sexo de forma descritiva;
- Fazer uma consulta que retorne o tipo de funcionário de acordo com a sua idade.
 - 18 - 25 - Jr.
 - 26 - 34 - Pl.
 - 35 ou mais - Sr.

SQL

- **Exercício**
- Fazer uma consulta para retornar um status para o funcionário de acordo com o tempo de casa.
 - Até 2 anos - Novato
 - Acima 2 anos e menor ou igual a 5 anos - Intermediário
 - Acima de 5 anos - Veterano

SQL

- E como eu faço para relacionar as minhas tabelas?
- Utilizamos os JOIN para fazer essa operação. Os JOIN utilizam a teoria de conjuntos para fazer a relação entre duas tabelas.
- Quando fazemos um JOIN estamos fazendo um produto cartesiano para realizar uma multiplicação onde o item do lado A deve existir no lado B.

SQL

- Nós temos 3 tipos de JOIN.
- INNER JOIN
 - Em um INNER JOIN os itens do lado A devem ser inclusivos no lado B.
 - Nós utilizamos a chave primária do lado A para relacionar com sua referência (chave estrangeira) para realizar a comparação.

```
select * from tabelaA a  
inner join tabelaB b on a.id = b.id_tabela_a;
```

```
select * from tabelaA a  
inner join tabelaB b on b.id_tabela_a = a.id;
```

SQL

- Nós temos 3 tipos de JOIN.
- LEFT JOIN
 - Em um LEFT JOIN nem todos os registro de A precisam ser inclusivos no lado B.
 - A relação que prevalece é a que está do lado esquerdo (ou primeira entidade) na comparação.

SQL

- Nesse exemplo todos os registros da tabelaA serão retornados mesmo que não existam do lado B.

```
select * from tabelaA a  
left join tabelaB b on b.id_tabela_a = a.id;
```

SQL

- Nós temos 3 tipos de JOIN.
- RIGHT JOIN
 - Em um RIGHT JOIN nem todos os registro de B precisam ser inclusivos no lado A.
 - A relação que prevalece é a que está do lado direito (ou segunda entidade) na comparação.

SQL

- Nesse exemplo todos os registros da tabelaB serão retornados mesmo que não existam do lado A.

```
select * from tabelaA a  
right join tabelaB b on b.id_tabela_a = a.id;
```

SQL

- Você pode no mesmo SQL ir encadeando novas relações.

```
select * from tabelaA a
inner join tabelaB b on a.id = b.id_a
inner join tabelaC c on a.id = c.id_a
```

- Em caso de chave composta basta você adicionar o operador AND na comparação.

```
select * from tabelaA a
inner join tabelaB b on a.id = b.id_a and a.id_2 = b.id_a2
```

SQL

- **Exercícios**

- Fazer uma consulta para retornar o nome do funcionário e o bairro onde ele mora;
- Fazer uma consulta para retornar o nome do cliente, cidade e zona que o mesmo mora;
- Fazer uma consulta para retornar os dados da filial, estado e cidade onde a mesma está localizada;
- Fazer uma consulta para retornar os dados do funcionário, departamento onde ele trabalha e qual seu estado civil atual;

SQL

- **Exercícios**

- Fazer uma consulta para retornar o nome do produto vendido, o preço unitário e o subtotal;
- Fazer uma consulta para retornar o nome do produto, subtotal e quanto deve ser pago de comissão por cada item;
- Fazer uma consulta para retornar o nome do produto, subtotal e quanto foi obtido de lucro por item;
- Criar um bloco anônimo que retorne o total de salário dos funcionários homens e total de salário das mulheres;

SQL

- Você pode limitar o resultado das suas consultas utilizando o **LIMIT**.

```
select * from tabela  
limit 10
```

- Você também consegue percorrer a consulta de forma posicional indicando a partir de onde quer começar utilizando o **OFFSET**.

```
select * from marital_status
```

	id [PK] integer	name character varying (64)	created_at timestamp without time zone	active boolean	modified_at timestamp without time zone
1	1	Solteiro	2021-07-21 19:22:06.018558	true	2021-07-21 19:22:06.018558
2	2	Casado	2021-07-21 19:22:06.018558	true	2021-07-21 19:22:06.018558
3	3	Divorciado	2021-07-21 19:22:06.018558	true	2021-07-21 19:22:06.018558
4	4	Viúvo	2021-07-21 19:22:06.018558	true	2021-07-21 19:22:06.018558

SQL

- Vamos usar para pegar os dois primeiros registros da consulta e depois os dois últimos.

```
select * from marital_status  
limit 2  
offset 0
```

	id [PK] integer	name character varying (64)	created_at timestamp without time zone	active boolean	modified_at timestamp without time zone
1	1	Solteiro	2021-07-21 19:22:06.018558	true	2021-07-21 19:22:06.018558
2	2	Casado	2021-07-21 19:22:06.018558	true	2021-07-21 19:22:06.018558

```
select * from marital_status  
limit 2  
offset 2
```

	id [PK] integer	name character varying (64)	created_at timestamp without time zone	active boolean	modified_at timestamp without time zone
1	3	Divorciado	2021-07-21 19:22:06.018558	true	2021-07-21 19:22:06.018558
2	4	Viúvo	2021-07-21 19:22:06.018558	true	2021-07-21 19:22:06.018558

SQL

- Ordenação de colunas é feita com o **ORDER BY** existem dois tipos de ordenação **ASC** do menor para o maior e **DESC** do maior para o menor, por padrão as ordenações são **ASC**.
- Com exceção de quando temos função de agrupamento em nosso **SELECT** o **ORDER BY** é sempre a última cláusula da consulta.

```
select * from tabela  
order by campo
```

```
select * from tabela  
order by campo desc
```

```
select * from tabela  
order by campo1 asc, campo2 desc
```

SQL

- **Exercício**
- Ranking dos 10 funcionários mais bem pagos;
- Ranking dos 20 clientes que tem a menor renda mensal;
- Trazer do décimo primeiro ao vigésimo funcionário mais bem pago;
- Ranking dos produtos mais caros vendidos no ano de 2021;

SQL

- **UNION**
- Tem o propósito de unir o resultado de duas ou mais consultas. Por padrão o **UNION** remove os registros duplicados, porém existe uma variação dele que nos permite trazer os registros duplicados **UNION ALL**.
- Vale ressaltar que a quantidade de colunas, ordem e tipos devem ser os mesmos nas consultas que englobam o **UNION**.

SQL

- **UNION**

```
select id, nome from tabela  
union  
select id, nome from tabela2
```

```
select id, nome from tabela  
union all  
select id, nome from tabela2
```

SQL

- **Exercício**
- Fazer uma consulta para retornar os 5 funcionários mais bem pagos juntamente com os 5 clientes com as melhores rendas mensais;
- Fazer uma consulta que retorne as 5 pessoas com piores rendas ou salários;
- Fazer uma consulta para retornar as 50 mulheres mais bem pagas entre funcionários e clientes;

SQL

- Vamos agora ver algumas funções de string nativas do PostgreSQL que pode nos ajudar na resolução de alguns problemas.

```
-- conversão
select id::text from employee;
select id::varchar from employee;

-- tamanho da string
select length(name) from employee;
select char_length(name) from employee;
select character_length(name) from employee;

-- transforma para minúsculo
select lower(name) from employee;

-- transforma para maiúsculo
select upper(name) from employee;

-- replace a partir de uma posição inicial e quantidade de caracteres
select overlay('Txxxxas' placing 'hom' from 2 for 4);
```

SQL

- Outras funções

```
-- posição da primeira ocorrência encontrada
```

```
select position('om' in 'Thomas');
```

```
-- obtém os caracteres de acordo com posição inicial e quantidade de caracteres ou expressão regular
```

```
select substring('Brasil', 1, 3);
```

```
select substr('Brasil', 1, 3);
```

```
select substring('Brasil' from 1 for 3);
```

```
select substring('8888-0000' from '^[0-9]{4}');
```

```
-- remove os espaços em branco ou outro caracter das extremidades, esquerda ou direita
```

```
select trim('  Brasil ');
```

```
select trim('xxBrasilxx', 'x');
```

```
select ltrim('  Brasil');
```

```
select ltrim('xxBrasil', 'x');
```

```
select rtrim('Brasil ');
```

```
select rtrim('Brasilxx', 'x');
```

SQL

- Outras funções

```
-- concatenação de valores
select concat('Brasil', 'penta', 'campeão');
select 'Brasil' || 'penta' || 'campeão';

-- concatenação de valores indicando o separador na primeira posição
select concat_ws(' ', 'Brasil', 'penta', 'campeão');

-- formatação de string
select format('Hello %s', 'Mundo');

-- primeira letra de cada palavra em maiúsculo
select initcap('brasil penta campeão');

-- retorna a quantidade de caracteres começando da esquerda
select left('brasil penta campeão', 2);

-- retorna a quantidade de caracteres começando da direita
select right('brasil penta campeão', 2);
```

SQL

- Outras funções

```
-- repete as ocorrências
select repeat('Pg', 4) ;

-- substitui as ocorrências
select replace('argentina penta campeão do mundo', 'argentina', 'brasil');

-- inverte a ocorrência
select reverse('lisarb');

-- completa a quantidade de vezes com o caracter especificado a esquerda ou direita
select lpad('5', 10, '0');
select rpad('5', 10, '0');

-- adicionar aspas simples em uma string
select quote_literal('Brasil');

-- iniciando com
select * from employee
where starts_with(name, 'A');

-- quebra em um array e retorna um posição específica
select split_part('brasil$penta$campeão', '$', 1);
```

SQL

- **Exercícios**
- Criar uma consulta para trazer o primeiro nome dos funcionários.
 - Sr. Sra. Dr. Dra. remover se tiver.
- Criar uma consulta para trazer o último nome dos clientes.
- Criar uma consulta que retorne os comandos para atualizar o nome de cada funcionário para o **NOME - SALÁRIO**.
- Criar uma consulta para rocar quem tenha silva no nome para **Oliveira**.

SQL

- **Funções de agrupamento**
- Nos ajudam a totalizar, gerar indicadores e relatórios.
- Temos algumas funções de agrupamento.
 - **AVG**
 - **MIN**
 - **MAX**
 - **COUNT**
 - **SUM**

SQL

- Podemos utilizar as funções totalizadoras sem para obter por exemplo o maior salário dos funcionários, o menor salário dos funcionários, a média dos salários dos funcionários, a quantidade de funcionários ou a soma do salários do funcionários.

```
select max(salary) from employee;  
select min(salary) from employee;  
select avg(salary) from employee;  
select count(*) from employee;  
select sum(salary) from employee;
```

SQL

- Nós podemos também utilizar as funções totalizadoras para realizar as devidas operação agrupando por determinado campo ou campos.
- Quando caímos nesse cenário, precisamos informar por quais campos desejamos fazer o **GROUP BY** o campo que for agrupado será o campo que obrigatoriamente aparecerá como rótulo, informando assim o agrupamento.

SQL

- Vamos supor que precisemos fazer uma consulta que retorne o total de funcionários por sexo.

```
select
    e.gender,
    count(*)
from employee e
group by e.gender;
```

	gender character (1)	count bigint
1	M	161
2	F	140

SQL

- A medida que você adicionando novos campos que deseja exibir os mesmos deverão entrar no **GROUP BY**.
- Você pode também utilizar mais de uma função de agrupamento, utilizando o mesmo campo para agrupamento.
- Vamos adicionar na consulta anterior a média de salários por sexo.

```
select
    e.gender,
    count(*),
    avg(e.salary)
from employee e
group by e.gender;
```

	gender character (1)	count bigint	avg numeric
1	M	161	10353.590062111801
2	F	140	10165.021428571429

SQL

- Nós podemos também fazer filtros no resultado já agrupado. Nesse caso precisamos utilizar o **HAVING** ao invés do **WHERE** e nesse caso o filtro dever ser feito após o **GROUP BY**.

```
select
    e.gender,
    count(*),
    avg(e.salary)
from employee e
group by e.gender
having count(*) > 150;
```

SQL

- **Exercícios**

- Criar uma consulta para trazer o total de funcionários por estado civil;
- Criar uma consulta para trazer o total vendido em valor R\$ por filial;
- Criar uma consulta para trazer o total vendido em valor R\$ por zona;
- Criar uma consulta para trazer o total vendido em valor R\$ por estado;
- Criar uma consulta para trazer o total vendido em quantidade por cidade, trazer apenas as cidades que tiveram vendas acima de quantidade 100;

SQL

- **Exercícios**
- Trazer a media de salários por sexo, o sexo deve está de forma descritiva;
- Fazer uma consulta que retorne os 5 grupos de produtos mais lucrativos em termos de valor, os grupos só entram na lista com lucros acima de R\$ 200,00;
- Uma consulta para trazer a média de salários por departamento;
- Uma consulta para trazer o total vendido em valor por ano;
- Uma consulta para trazer o total vendido em valor por idade de funcionário;

SQL

- **Subqueries**

- As subqueries podem nos ajudar a resolver problemas mais complexos.
- Quando precisamos fazer uma comparação no **where** por exemplo onde o resultado que deve ser comparado com a coluna vem do resultado de outra consulta, ou quando por exemplo precisamos saber se um cliente tem alguma venda, muitas vezes nesses cenários acabamos recorrendo para os **joins**, porém em grande maioria dos cenários uma subquery resolve melhor (mais rapidez) do que os próprios **joins**, lembre-se um **join** sempre faz um produto das relações.

SQL

- Para entendermos melhor vamos ver os operadores que geralmente utilizamos em **subqueries**.
- **EXISTS**
- **NOT EXISTS**
- **IN**
- **NOT IN**
- **ANY/SOME**
- **ALL**

SQL

- Estrutura utilizando **IN** ou **NOT IN**.
- Nesses cenários precisamos que a condição do WHERE seja comparado com os valores de uma lista, sendo que essa lista vem de uma outra consulta que pode ter seu próprios filtros, joins e tudo mais.

```
select * from tabelaA a
where a.id IN ( select b.id from subconsulta b )
```

```
select * from tabelaA a
where a.id NOT IN ( select b.id from subconsulta b )
```


SQL

- Estrutura utilizando **EXISTS** ou **NOT EXISTS**.
- Ao utilizar o **exists** ou **not exists** não utilizamos um campo no **where** da consulta principal para comparar e sim na subconsulta.
- Como não fazemos a comparação no **where** da consulta principal não precisamos especificar as colunas na subconsulta, podemos apenas utilizar o *, tendo em vista que o **exists** retorna um valor booleano.

```
select * from tabelaA  
where exists ( select * from subconsulta b )
```

```
select * from tabelaA  
where not exists ( select * from subconsulta b )
```

SQL

- Estrutura utilizando **ANY** ou **SOME**.
- Essa função basicamente funciona como um **IN** porém, você pode utilizar outros operadores além do igual.

```
select * from customer c
where c.id = any (select s.id_customer from sale s)
```

```
select * from customer c
where c.id <> any (select s.id_customer from sale s)
```

```
select * from customer c
where c.id <> some (select s.id_customer from sale s)
```

SQL

- Estrutura utilizando **ALL**.
- Condição deve ser satisfeita para todos os elementos do conjunto.
- Você pode também utilizar outros operadores fora o de igualdade.

```
select * from customer c
where c.id = all (select s.id_customer from sale s)
```

SQL

- Você pode também em um IN por exemplo utilizar mais de uma coluna na comparação utilizando o **ROW**.

```
select * from customer c
where row(c.id, c.active) in (select s.id_customer, s.active from sale s)
```

SQL

- Você pode também utilizar subconsultas como colunas do seu **select** porém, essas subconsultas devem estar limitadas a único de registro e única coluna.

```
select
    d.id,
    d.name,
    ( select array_agg(e.name) from employee e where e.id_department = d.id )
from department d
```

```
select
    (select b.coluna from subquery b limit 1)
from tabela a
```

SQL

- **Exercícios**

- Criar uma consulta pra trazer todas as zonas que possuem vendas no ano de 2012;
- Criar uma consulta para trazer todas as filiais que possuem uma venda no ano de 2013 para o grupo de produtos Alimentício;
- Criar uma consulta para trazer o total vendido por sexo nos anos de 2010, 2011, 2012, 2013, 2014, 2015, cada ano deve ser representado como uma coluna da sua consulta;
- Criar uma consulta para trazer todos os produtos que não foram vendidos no ano de 2020;

SQL

- **Exercícios**

- Criar uma consulta para trazer todos produtos que foram vendidos para clientes que não são casados no ano de 2015;
- Criar uma consulta para trazer o total de lucro por grupo de produto do ano de 2016 inteiro, cada mês deve ser considerado uma coluna em seu SQL;
- Criar uma consulta para trazer todos os fornecedores que não tiveram seus produtos vendidos no ano de 2021;
- Criar uma consulta com uma lista de produtos que foram vendidos no ano de 2019, deve ser exibido uma coluna que identifica o total vendido em quantidade por produto e estado civil, cada estado civil deve ser uma coluna diferente em sua consulta;

SQL

- **VIEW**

- Views nada mais são do que estruturas que armazenam um SQL, para que seja executada mais facilmente, a criação de views pode ser aplicadas principalmente para cenários onde tem muita repetição da mesma consulta, evitando assim que cada programados crie de uma forma diferente a sua consulta, e isso pode inclusive acelerar sua consulta, pois o SGBD vai sempre usar o mesmo plano de execução da query.
- Com a view criada você pode também aplicar filtros, joins e tudo mais.

SQL

- Estrutura de uma view.

```
create or replace view vw_exemplo as  
select * from tabela a
```

SQL

- **Exercício**
- Criar uma view para todas as consultas criadas nas questões de subqueries.

Um pouco do híbrido

- O PostgreSQL além de todos os conceitos de um banco de dados puramente relacional, nos fornece alguns conceitos de banco de dados No SQL.
- Um dos primeiros recursos são os **arrays**, isso mesmo com o PostgreSQL podemos ter campos do tipo **array**.
- Podemos por exemplo usar os campos **array** para substituir atributos multivalorados, que aprendemos no modelo conceitual.

Um pouco do híbrido

- Declaração de uma coluna como **array**.

```
create table cliente (  
    id serial not null primary key,  
    nome varchar(100) not null,  
    telefones varchar[]  
);
```

Um pouco do híbrido

- Inserir dados em uma coluna **array**.

```
insert into cliente (nome, telefones) values ('Osenias', array['9999-9999', '0000-0000']);  
insert into cliente (nome, telefones) values ('Izzie', '{9999-9999,0000-0000}');
```

Um pouco do híbrido

- Acessando valores de um **array**.





```
select nome, telefones[1], telefones[2] from cliente;
```

	nome character varying (100) 🔒	telefones character varying 🔒	telefones character varying 🔒
1	Osenias	9999-9999	0000-0000
2	Izzie	9999-9999	0000-0000

Um pouco do híbrido

- Atualizando um **array**.





```
update cliente set telefones = array_append(telefones, '8888-8888');
```

	 id [PK] integer 	nome character varying (100) 	telefones character varying[] 
1	1	Osenias	{9999-9999,0000-0000,8888-8888}
2	2	Izzie	{9999-9999,0000-0000,8888-8888}

Um pouco do híbrido

- Atualizando um **array**.

```
update cliente set telefones[1] = '2222-2222';
```

	 id [PK] integer 	nome character varying (100) 	telefones character varying[] 
1	1	Osenias	{2222-2222,0000-0000,8888-8888}
2	2	Izzie	{2222-2222,0000-0000,8888-8888}

Um pouco do híbrido

- Consultando em um **array**.
- Basicamente conseguimos utilizar todas as funções de subqueries para os campos do tipo **array**.

```
select * from cliente  
where '2222-2222' = any (telefones);
```

Um pouco do híbrido

- Algumas funções para se utilizar em **arrays**.

Função	Descrição
array_append	Concatena elementos ao array
array_cat	Concatena dois arrays
array_length	Tamanho do array
array_position	Retorna a posição da primeira ocorrência
array_positions	Retorna um array com todas as posições encontradas
array_prepend	Adiciona um elemento no início do array
array_remove	Remove todos os elementos encontrados
array_replace	Replace de elementos dentro do array

Um pouco do híbrido

- Algumas funções para se utilizar em **arrays**.

Função	Descrição
array_to_string	Transforma um array em string + separador
string_to_array	Transforma uma string em array a partir de um delimitador
unnest	Transforma um array em tuplas

Um pouco do híbrido

- Algumas funções para se utilizar em **arrays**.

```
select array_append(array[1,2], 3);
```

```
select array_cat(array[1,2,3], array[4,5]);
```

```
select array_length(array[1,2,3], 1);
```

```
select array_position(array['sun', 'mon', 'tue', 'wed', 'thu', 'fri', 'sat'], 'mon');
```

```
select array_positions(array['A','A','B','A'], 'A');
```

```
select array_prepend(1, array[2,3]);
```

Um pouco do híbrido

- Algumas funções para se utilizar em **arrays**.

```
select array_remove(array[1,2,3,2], 2);
```

```
select array_replace(array[1,2,5,4], 5, 3);
```

```
select array_to_string(array[1, 2, 3, NULL, 5], ',', '*');
```

```
select string_to_array('xx~~yy~~zz', '~~', 'yy');
```

```
select unnest(array[1,2]);
```

Um pouco do híbrido

- Outro tipo de dados bem usado em um banco de dados No SQL é o json, com o PostgreSQL podemos também criar e manipular campos desse tipo.

```
create table cliente (  
    id serial not null primary key,  
    nome varchar(100) not null,  
    outras_informacoes jsonb  
);
```

Um pouco do híbrido

- Inserindo dados com um campo **json**.




```
insert into cliente (nome, outras_informacoes) values ('Osenias', '{"idade": 33, "email": "oseniasjunior@gmail.com"}');
```

	<div>id</div> <div>[PK] integer</div>	<div>nome</div> <div>character varying (100)</div>	<div>outras_informacoes</div> <div>jsonb</div>
1	1	Osenias	{"email": "oseniasjunior@gmail.com", "idade": 33}

Um pouco do híbrido

- Acessando valores em um **json**.

```
select
    nome,
    outras_informacoes->'email',
    outras_informacoes->'idade'
from cliente;
```

	nome character varying (100) 	?column? jsonb 	?column? jsonb 
1	Osenias	"oseniasjunior@gmail.com"	33

Um pouco do híbrido

- Acessando valores em um **json**.

```
select jsonb_extract_path(outras_informacoes, 'idade') from cliente;  
select jsonb_extract_path_text(outras_informacoes, 'email') from cliente;
```

	jsonb_extract_path_text	🔒
	text	
1	oseniasjunior@gmail.com	

Um pouco do híbrido

- Atualizando dados em um campo **json**.

```
update cliente set outras_informacoes = outras_informacoes || '{"cpf": "000.000.000-00"}';
```

	<div>id</div> <div>[PK] integer</div>	<div>nome</div> <div>character varying (100)</div>	<div>outras_informacoes</div> <div>jsonb</div>
1	1	Osenias	{"cpf": "000.000.000-00", "email": "oseniasjunior@gmail.com", "idade": 33}

Um pouco do híbrido

- Filtro em um **json**.

```
select * from cliente where outras_informacoes @@ '$.cpf == "000.000.000-00";
```

```
select * from cliente  
where jsonb_extract_path_text(campos_personalizados, 'cpf') ilike '%000%';
```

Um pouco do híbrido

- Criando um **json** com um array dentro.

```
create table cliente (  
    id serial not null primary key,  
    nome varchar(100) not null,  
    campos_personalizados jsonb  
);
```

```
insert into cliente (nome, campos_personalizados)  
values ('Osenias', '{"email": "oseniasjunior@gmail.com", "telefones": ["9200000-0000", "9511111-1111"]}');
```

Um pouco do híbrido

- No caso de um array que fica localizado dentro de um campo do tipo json a forma de acesso muda para um campo que é nativamente um **array field**, além disso o índice desse começa em 0.

```
select
    campos_personalizados->'telefones'->>0,
    campos_personalizados->'telefones'->>1
from cliente;
```

```
select
    jsonb_extract_path(campos_personalizados, 'telefones')->>0,
    jsonb_extract_path(campos_personalizados, 'telefones')->>1
from cliente;
```

Um pouco do híbrido

- Alguns operadores que podem ser usados no **json**.

Operator	Description
->	Get JSON array element (indexed from zero, negative integers count from the end)
->	Get JSON object field by key
->>	Get JSON array element as text
->>	Get JSON object field as text
#>	Get JSON object at the specified path
#>>	Get JSON object at the specified path as text
@>	Does the left JSON value contain the right JSON path/value entries at the top level?
<@	Are the left JSON path/value entries contained at the top level within the right JSON value?
?	Does the <i>string</i> exist as a top-level key within the JSON value?
?	Do any of these array <i>strings</i> exist as top-level keys?
?&	Do all of these array <i>strings</i> exist as top-level keys?
	Concatenate two jsonb values into a new jsonb value
-	Delete key/value pair or <i>string</i> element from left operand. Key/value pairs are matched based on their key value.
-	Delete multiple key/value pairs or <i>string</i> elements from left operand. Key/value pairs are matched based on their key value.
-	Delete the array element with specified index (Negative integers count from the end). Throws an error if top level container is not an array.
#-	Delete the field or element with specified path (for JSON arrays, negative integers count from the end)
@?	Does JSON path return any item for the specified JSON value?
@@	Returns the result of JSON path predicate check for the specified JSON value. Only the first item of the result is taken into account. If the result is not Boolean, then null is returned.

Um pouco do híbrido

- Algumas funções **json**.

Função	Descrição
json_build_object	Criar um objeto json
jsonb_object	Criar um objeto json
jsonb_each_text	Converte o json em tuplas com chave e valor
json_object_keys	Retorna apenas as Keys de um json

Um pouco do híbrido

- Algumas funções **json**.

```
select jsonb_build_object('foo', 1, 2, row(3, 'bar'));
```

```
select json_object('{a, 1, b, "def", c, 3.5}');
```

```
select * from jsonb_each_text('{"a":"foo", "b":"bar"}')
```

```
select * from json_object_keys('{"f1":"abc", "f2":{"f3":"a", "f4":"b"}}');
```


Um pouco do híbrido

- Nós podemos também criar tipos compostos e declarar nossas colunas com esses tipos.

```
create type informacoes as (  
    email varchar(100),  
    cpf varchar(11)  
);
```

```
create table cliente (  
    id serial not null,  
    nome varchar(100) not null,  
    dados informacoes  
);
```

Um pouco do híbrido

- Inserindo dados tabelas com colunas de tipos compostos.

```
insert into cliente (nome, dados) values ('Osenias', row('oseniasjunior@gmail.com', '000000000000'));
```

Um pouco do híbrido

- Atualizando dados tabelas com colunas de tipos compostos.

```
update cliente set dados.email = 'osenias.oliveira@fpf.br';
```

```
update cliente set dados.cpf '11111111111';
```

```
update cliente set dados = row('osenias.oliveira@fpf.br', '222222222');
```

Um pouco do híbrido

- Acessando os dados de uma coluna baseada em um tipo composto.

```
select
    (dados).email,
    (dados).cpf
from cliente;
```

	email character varying (100) 🔒	cpf character varying (11) 🔒
1	osenias.oliveira@fpf.br	000000000000

Um pouco do híbrido

- Você pode também criar tipos como ENUMS.
- O ENUM aqui poderia substituir uma **check constraint** para validar se os valores aceito para coluna sexo.

```
create type sexos as enum ('M', 'F');
```

```
create table cliente (  
    id serial not null primary key,  
    nome varchar(100) not null,  
    sexo sexos not null  
);
```

PL/PGSQL

- O PL/PGSQL é a linguagem do PostgreSQL para desenvolvimento de algoritmos dentro do banco de dados, é claro que cada caso é um caso, a ideia de você utilizar algoritmos dentro do banco de dados está diretamente relacionado a muitas queries executadas para resolver um determinado problema, geralmente recorremos para esse recurso quando realmente precisamos resolver um problema mais complexo e que precisa ser performático ou quando precisamos encapsular um cálculo um pouco mais complexo que é realizado dentro de uma query.
- **Não é o ideal utilizar PL/PGSQL para tudo.**

PL/PGSQL

- Nós podemos utilizar o **PL/PGSQL** dentro de uma função ou dentro de um bloco anônimo por exemplo.
- O PostgreSQL nas versões mais atuais absorveu o conceito também de procedures, uma procedure e uma função são parecidas vamos dizer assim. A ideia dos dois é a execução de algum procedimento, porém a procedure por natureza tem um papel mais de executor sem se preocupar muito com retornos. Uma procedure no PostgreSQL não utiliza a linguagem **PL/PGSQL** e sim **SQL** puro.
- Na prática uma função pode resolver o que uma procedure resolve, então faça suas análises para saber se faz sentido ou não usar as procedures, as vezes com um bloco anônimo você pode resolver o problema.

PL/PGSQL

- Exemplo de uma procedure e sua execução.

```
create procedure inserir_dados(nome varchar[])  
language sql  
as $$  
    insert into cliente (nome) values (nome[1]);  
    insert into cliente (nome) values (nome[2]);  
$$;
```

```
call inserir_dados(array['osenias', 'izzie']);
```


PL/PGSQL

- Agora vamos para as funções.

```
create or replace function nome_funcao(parametro_a integer) returns varchar as -- definição do retorno
$$
declare
    -- aqui você declara suas variáveis
begin
    -- aqui você criar o corpo da função
end
$$
-- definição da linguagem
language plpgsql;
```

```
select nome_funcao(10);
```

PL/PGSQL

- **Exercícios**
- Criar uma função que receba o nome e retorne Olá {nome}.
- Criar uma função para somar dois números.

PL/PGSQL

- Estruturas de decisão
- Dentro da sua função você também pode utilizar estruturas de decisão.

```
create or replace function fn_decisao(valor integer) returns boolean as
$$
declare
begin
    if valor > 10 then
        return true;
    else
        return false;
    end if;
end
$$
language plpgsql;
```

PL/PGSQL

- Estruturas de decisão

```
create or replace function fn_decisao_2(valor integer) returns varchar as
$$
declare
begin
    raise notice 'valor: %s', valor;
    if valor between 1 and 10 then
        return 'criança';
    elsif valor between 11 and 17 then
        return 'adolescente';
    else
        return 'adulto';
    end if;
end
$$
language plpgsql;
```

PL/PGSQL

- Estruturas de decisão

```
create or replace function fn_decisao_3(valor integer) returns varchar as
$$
declare
    msg varchar default '';
begin
    case
        when valor between 1 and 10 then
            msg := 'criança';
        when valor between 11 and 17 then
            msg := 'adolescente';
        else
            msg := 'adulto';
        end case;
    return msg;
end
$$
language plpgsql;
```

PL/PGSQL

- **Exercício**
- Criar uma função para receber um valor inteiro e retornar se é um número par ou ímpar;
- Criar uma função para receber um salário e partir de um range determinado definir seu perfil;
 - 1 - 2000 - Jr;
 - 2001 - 5000 - Pl;
 - > 5000 - Sr;
 - Usar essa função em um **SELECT** de funcionário que traga seu nome, salário e perfil;

PL/PGSQL

- Estruturas de repetição

```
create or replace function fn_repeticao() returns void as
$$
declare
    counter integer default 1;
begin
    loop
        if counter > 100 then
            exit;
        end if;
        counter := counter + 1;
        raise notice 'counter: %', counter;
    end loop;
end
$$
language plpgsql;
```

PL/PGSQL

- Estruturas de repetição

```
create or replace function fn_repeticao_2() returns void as
$$
declare
    counter integer default 1;
begin
    loop
        exit when counter > 100;
        raise notice 'counter: %', counter;
        counter := counter + 1;
    end loop;
end
$$
language plpgsql;
```


PL/PGSQL

- Estruturas de repetição

```
create or replace function fn_repeticao_3() returns void as
$$
declare
    counter integer default 1;
begin
    while true loop
        exit when counter > 100;
        raise notice 'counter: %', counter;
        counter := counter + 1;
    end loop;
end
$$
language plpgsql;
```

PL/PGSQL

- Estruturas de repetição

```
create or replace function fn_repeticao_4() returns void as
$$
declare
    i integer;
begin
    for i in 1..10 loop
        raise notice 'i: %', i;
    end loop;
end
$$
language plpgsql;
```

PL/PGSQL

- Estruturas de repetição

```
create or replace function fn_repeticao_5() returns void as
$$
declare
    i integer;
begin
    for i in 1..10 by 2 loop
        raise notice 'i: %', i;
    end loop;
end
$$
language plpgsql;
```

PL/PGSQL

- Estruturas de repetição

```
create or replace function fn_repeticao_6() returns void as
$$
declare
    i integer;
begin
    for i in reverse 10..1 by 2 loop
        raise notice 'i: %', i;
    end loop;
end
$$
language plpgsql;
```

PL/PGSQL

- Estruturas de repetição

```
create or replace function fn_repeticao_7(numeros int[]) returns void as
$$
declare
    indice integer;
begin
    foreach indice in array numeros loop
        raise notice 'indice: %', indice;
    end loop;
end
$$
language plpgsql;
```

PL/PGSQL

- Estruturas de repetição
- Nós podemos também fazer com que nossa função retorne uma coleção (tuplas), especificando o **SETOF** no retorno e dentro da estrutura de repetição o **RETURN NEXT**.

```
create or replace function gerar_serie(inicial integer, final integer) returns setof integer as
$$
declare
    numero integer;
begin
    for numero in inicial..final
        loop
            return next numero;
        end loop;
    return;
end;
$$
language plpgsql;
```

PL/PGSQL

- **Exercício**
- Criar uma função para receber uma palavra e deve retornar a quantidade de vogais e consoantes;
- Criar uma função para receber uma palavra e inverter a mesma;
- Criar uma função para receber uma lista de inteiros e some o valor das posições pares e posições ímpares;
- Criar uma função para gerar uma série de valores inteiros. Deve receber o valor inicial, o valor final e retornar as tuplas com os números gerados. Em caso de funções que retornam mais de uma linha você deve usar o **SETOF** para especificar que é uma lista, deve ter um **RETURN NEXT** dentro da estrutura de repetição e um **RETURN;** no final da função;

PL/PGSQL

- **Exercício**
- Criar uma função que receba um **SQL** como **VARCHAR** por exemplo **SELECT * FROM funcionario**, o outro parâmetro deve ser um **JSON** com os filtros, essa função deve retornar o SQL que será executado;

PL/PGSQL

- **Estruturas de repetição**
- O loop a seguir será baseado em uma consulta para isso vamos necessitar de um tipo especial.
- O PostgreSQL possui um tipo chamado **RECORD** que é específico para armazenar resultados de consultas.

PL/PGSQL

- Estruturas de repetição

```
create or replace function usando_record() returns void as
$$
declare
    consulta record;
begin
    for consulta in select name from employee loop
        raise notice 'nome: %', consulta.nome;
    end loop;
end;
$$
language plpgsql;
```

PL/PGSQL

- Nós também podemos fazer com que nossa função retorne um tipo customizados, criados por nós dentro do banco de dados.
- Para isso precisamos, além do tipo criado no banco de dados, utilizar uma função específica para indicar que cada linha de nosso retorno terá essa tipagem.

```
create type exemplo as (  
    data date,  
    numero integer  
);
```

PL/PGSQL

```
create or replace function usando_tipos() returns setof exemplo as
$$
declare
    linha exemplo%rowtype;
    contador integer;
begin
    for contador in 1..100 loop
        linha.data := current_date + interval '1 day' * contador;
        linha.numero := contador;
        return next linha;
    end loop;
    return;
end;
$$

language plpgsql;

select * from usando_tipos();
```

PL/PGSQL

- Você pode também fazer com que sua função retorne um **TABLE**.

```
create or replace function retornando_table() returns table (id_employee integer, name_employee varchar, salary_employee numeric, salary_10 numeric) as
$$
declare
    consulta record;
begin
    for consulta in select id, name, salary from employee loop

        id_employee := consulta.id;
        name_employee := consulta.name;
        salary_employee := consulta.salary;
        salary_10 := round(consulta.salary + consulta.salary / 0.10, 2);

        return next;
    end loop;
    return;
end;
$$
language plpgsql;
```

PL/PGSQL

- Chamando a função

```
select * from retornando_table();
```

Data Output		Explain	Messages	Notifications				
	id_employee integer		name_employee character varying		salary_employee numeric		salary_10 numeric	
1		1	Ana Luiza Cunha		2097.00		23067.00	
2		2	Vicente Dias		1078.00		11858.00	
3		3	Sr. Henrique Martins		17585.00		193435.00	

PL/PGSQL

- Outro conceito bem importante é o tratamento de exceções.
- O PostgreSQL também proporciona mecanismos para que você possa fazer os devidos tratamentos.
- Existem outros tipos exceções, mas na dúvida pode utilizar o **others**.

```
create or replace function divisao(a numeric, b numeric) returns numeric as
$$
declare
begin
    begin
        return a / b;
    exception
        when division_by_zero then
            raise exception 'Não pode dividir por zero';
    end;
end;
$$
language plpgsql;
```

Data Output Explain Messages Notifications

ERROR: Não pode dividir por zero
CONTEXT: PL/pgSQL function divisao(numeric,numeric) line 9 at RAISE
SQL state: P0001

PL/PGSQL

- **Blocos anônimos**
- O PostgreSQL também permite que você crie seus algoritmos usando o PL/PGSQL sem precisar criar uma função, usando apenas um bloco anônimo.
- Quando usamos um bloco anônimo não podemos especificar parâmetros ou retornos, devemos apenas fazer uma determinada operação.

PL/PGSQL

- Blocos anônimos

```
do
$$
declare
    consulta record;
begin
    for consulta in select name from department loop
        raise notice 'nome: %', consulta.name;
    end loop;
end
$$
```

PL/PGSQL

- **Exercício**
- Crie uma função para retornar o total vendido por produto e estado civil, cada estado civil deve vir como uma coluna.
- Crie uma função que receba como parâmetro um determinado ano, por exemplo 2020, a função deve avaliar as vendas mês a mês de cada produto. A função deve retornar uma projeção de vendas mês a mês para o ano atual, considerando que o valor da venda será o analisado + 10% por ano corrido, caso o valor de um determinado mês seja 0, o valor da venda daquele mês será automaticamente de o preço de venda do produto * 1000.

TRIGGERS

- Trigger são ações que são executadas de forma espontânea de acordo com um determinado evento.
- Quando falamos de eventos estamos nos referindo a operações de **INSERT, UPDATE, DELETE ou TRUNCATE**, bem como, em que momento as ações devem ser executadas **BEFORE ou AFTER**.
- Uma informação importante as ações das triggers são executadas a partir de funções que retornam um tipo específico para **TRIGGERS**.

TRIGGERS

- O cenário.

```
create table cliente (  
    id serial not null primary key,  
    nome varchar(100) not null  
);  
  
create table historico (  
    notas text not null,  
    data date not null default current_date  
);
```

TRIGGERS

- A função que será executada.
- Veja que utilizamos o **new** para indicar que estamos pegando o novo valor cadastrado, no caso da tabela cliente, podemos também utilizar o **old** quando tivermos o cenário de atualização e precisarmos pegar o valor antigo.

```
create or replace function fn_salvar_historico() returns trigger as
$$
begin
    insert into historico (notas) values (format('Cadastrando o cliente %s', new.nome));
return new;
end;
$$
language plpgsql;
```

TRIGGERS

- A criação da **trigger** baseado na tabela de cliente.

```
create trigger tg_fn_salvar_historico  
  before insert on cliente  
  for each row  
  execute function fn_salvar_historico();
```

TRIGGERS

- Inserindo os dados na tabela cliente e consultando a tabela histórico para checarmos se o evento funcionou.

```
insert into cliente (nome) values ('Osenias');  
  
select * from historico;
```

	notas text	data date
1	Cadastrando o clinente Osenias	2021-07-30

TRIGGERS

- Você pode também especificar a alteração para um campo específico.

```
create trigger check_update  
  before update of balance on accounts  
  for each row  
  execute function check_account_update();
```


TRIGGERS

- Você pode também especificar uma condição.

```
create trigger check_update
before update on accounts
for each row
when (old.balance is distinct from new.balance)
execute function check_account_update();
```

EXTENSIONS


- Extensões no PostgreSQL funcionam como plugins, que podem ser adicionados no seu banco de dados quando necessário.
- Temos algumas funções bem conhecidas e usadas no dia a dia.

```
create extension if not exists unaccent;
```

EXTENSIONS

- Uso da extensão **UNACCENT**.

```
select unaccent('éáéá');
```

	unaccent text 
1	eaea

EXTENSIONS

- Extensão usada para gerar **UUID**.

```
create extension if not exists "uuid-ossf";
```

```
select uuid_generate_v4();
```

uuid_generate_v4	
uuid	
1	93b8b2c1-f702-4bb8-85e4-2f0c103200d3

EXTENSIONS

- Extensão usada para **PIVOT TABLE**.

```
create extension if not exists "tablefunc";
```

```
select * from crosstab(  
$$  
    select  
        d.name as departamento,  
        ms.name as estado_civil,  
        count(*)  
    from employee e  
    inner join department d on e.id_department = d.id  
    inner join marital_status ms on ms.id = e.id_marital_status  
    group by 1, 2  
$$,  
$$  
    select ms.name from marital_status ms order by ms.id  
$$  
) as (departamento varchar, solteiro integer, casado integer, divorciado integer, viuvo integer);
```

SEGMENTAÇÃO/PARTICIONAMENTO

- Uma prática muito comum quando temos tabelas com grandes números de tuplas é o uso de segmentação.
- Por exemplo, imagine uma tabela de vendas com vendas desde o 2000, temos 21 anos de vendas, isso pode impactar em performance até mesmo em consultas em dados mais recentes.
- Nesse cenário criamos podemos por exemplo criar várias tabelas de vendas para cada ano **vendas_2000, vendas_2001** e por ai vai. Caso precisemos de vendas de mais de um ano nada impede de utilizarmos por exemplo um **UNION**.

SEGMENTAÇÃO/PARTICIONAMENTO

- Conforme falado essa é uma prática bem adotada para resolver problemas de performance, porém, todo o controle das tabelas é feito de forma manual.
- No PostgreSQL existe um mecanismo de particionamento que depois de criada a estrutura adequada e configurada, fica transparente para o usuário a divisão das tabelas ou responsabilidade de fazer os **UNIONS**.
- Quando fizermos uma consulta numa determinada tabela e ela for particionada o PostgreSQL se encarregará de fazer todo o trabalho.

SEGMENTAÇÃO/PARTICIONAMENTO

- Vamos criar uma tabela de vendas apenas com um campo observações e outro referente a data da venda.

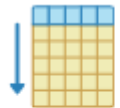
```
create table venda (  
    id serial not null,  
    observacao varchar(256) not null,  
    data timestamp default now()  
);
```

```
do  
$$  
declare  
    contador integer;  
    ano integer;  
begin  
    for contador in 1..10000 loop  
        for ano in 1..10 loop  
  
            insert into venda (observacao, data) values (  
                format('venda %s no ano %s', contador, ano),  
                now() + interval '1 year' * ano  
            );  
        end loop;  
    end loop;  
end  
$$
```


SEGMENTAÇÃO/PARTICIONAMENTO

- Vamos ver o plano de execução dessa consulta.

```
explain(format json)
select * from venda v
where v.data between '2021-01-01' and '2022-12-31';
```



venda

SEGMENTAÇÃO/PARTICIONAMENTO

- Agora vamos criar uma tabela particionada para verificar o plano de execução da mesma consulta quando a tabela é particionada.

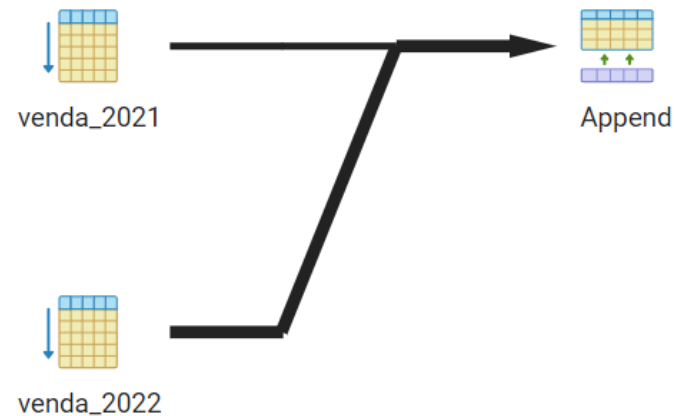
```
create table venda (  
    id serial not null,  
    observacao varchar(256) not null,  
    data timestamp default now()  
) partition by range (data);
```

```
create table venda_2021 partition of venda for values from ('2021-01-01') to ('2021-12-31');  
create table venda_2022 partition of venda for values from ('2022-01-01') to ('2022-12-31');  
create table venda_2023 partition of venda for values from ('2023-01-01') to ('2023-12-31');  
create table venda_2024 partition of venda for values from ('2024-01-01') to ('2024-12-31');  
create table venda_2025 partition of venda for values from ('2025-01-01') to ('2025-12-31');  
create table venda_2026 partition of venda for values from ('2026-01-01') to ('2026-12-31');  
create table venda_2027 partition of venda for values from ('2027-01-01') to ('2027-12-31');  
create table venda_2028 partition of venda for values from ('2028-01-01') to ('2028-12-31');  
create table venda_2029 partition of venda for values from ('2029-01-01') to ('2029-12-31');  
create table venda_2030 partition of venda for values from ('2030-01-01') to ('2030-12-31');  
create table venda_2031 partition of venda for values from ('2031-01-01') to ('2031-12-31');
```

SEGMENTAÇÃO/PARTICIONAMENTO

- Agora vamos ver como fica a consulta de forma particionada.

```
explain(format json)
select * from venda v
where v.data between '2021-01-01' and '2022-12-31';
```



DBLINK

- O DBLink é um recurso que nos permite comunicar com outra base de dados, seja no mesmo servidor ou em outro servidor.
- Para que possamos usar precisamos criar a extensão.

```
create extension dblink;
```

DBLINK

- A consulta.

```
select * from dblink(  
    'dbname=delegacia port=5432 host=127.0.0.1 user=postgres password=123456',  
    'select id, numero_serie, descricao from arma'  
) as (id integer, numero_serie varchar, descricao varchar);
```

EXERCÍCIO

- Transformar a tabela de vendas particionada por ano. Lembre-se de verificar todos os anos possíveis para criar as partições de forma correta;
- Crie um **PIVOT TABLE** para saber o total vendido por grupo de produto por mês referente a um determinado ano;
- Crie um **PIVOT TABLE** para saber o total de clientes por bairro e zona;
- Crie uma coluna para saber o preço unitário do item de venda, crie um script para atualizar os dados já existentes e logo em seguida uma trigger para preencher o campo;

EXERCÍCIO

- Crie um campo para saber o total da venda, crie um script para atualizar os dados já existentes, em seguida uma trigger para preencher o campo de forma automática;
- Baseado no banco de dados de crime vamos fazer algumas questões.
 - 1 - Criar o banco de dados;
 - 2 - Criar o DDL para estrutura das tabelas;
 - 3 - Criar um script para criar armas de forma automática, seguindo os seguintes critérios: O número de série da arma deve ser gerado por o UUID, os tipos de armas são, 0 - Arma de fogo, 1 - Arma branca, 2 - Outros.

EXERCÍCIO

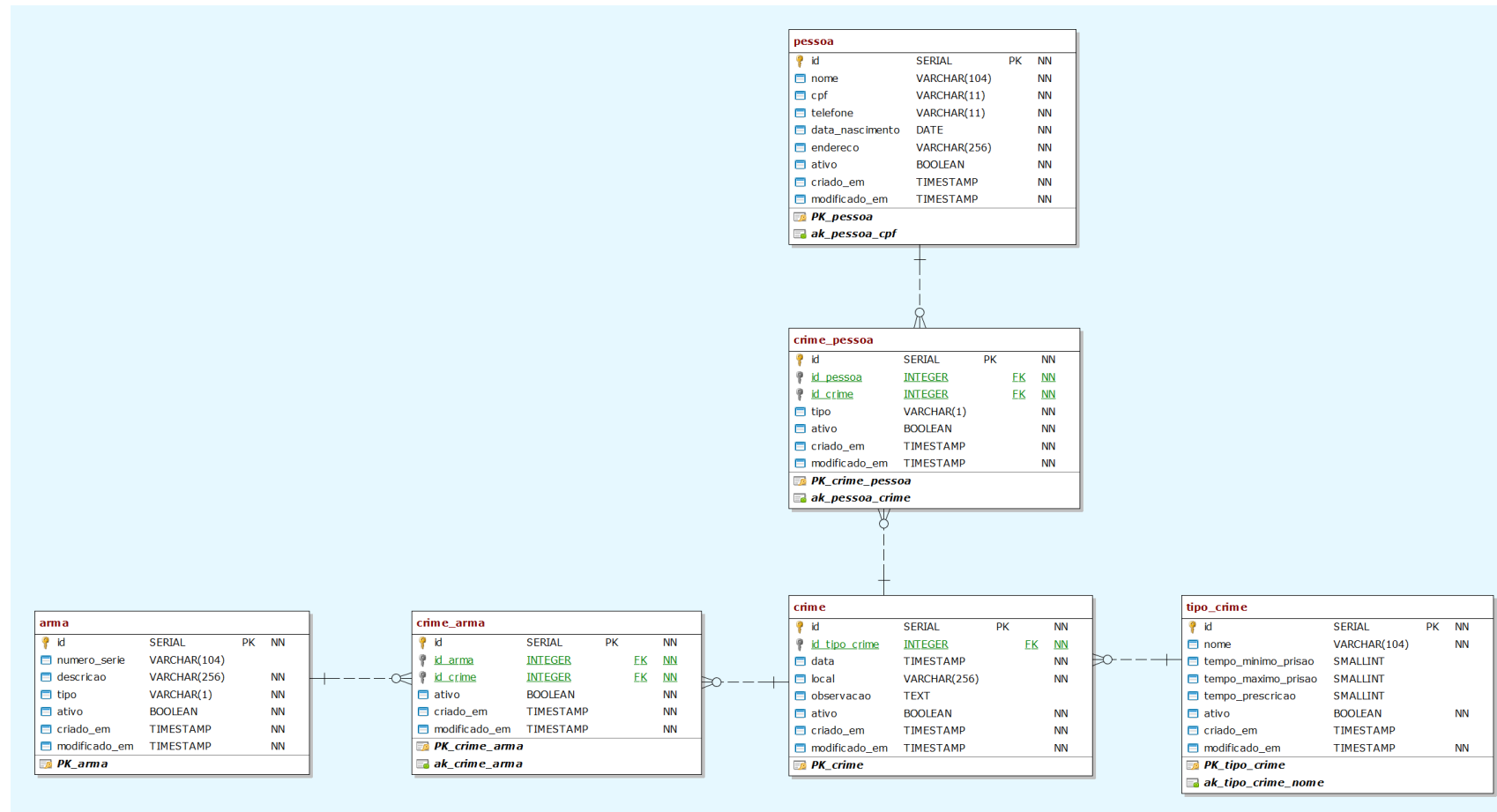
- **Algumas armas de fogo.**
 - Pistola, Metralhadora, Escopeta.
- **Algumas armas brancas.**
 - Faca, Facão, estilete.
- **Outros**
 - Corda, Garrafa.
- A ideia é que você use essas tipos de armas para de forma aleatória gere as armas os **INSERTS**.

EXERCÍCIO

- Faça um script para migrar todos os clientes e funcionários da base de vendas como pessoas na base de dados de crimes. Os campos que por ventura não existirem, coloque-os como nulo ou gere de forma aleatória.

EXERCÍCIO

- Base de dados de crime



DICAS PARA CRIAÇÃO DE UM BANCO DE DADOS

- Faça a modelagem do banco de dados com a equipe, modelagem do banco de dados é de extrema importância no histórico do ciclo de vida do Software e facilita manutenção.
- Modele o banco de dados sem pensar em futuros problemas relacionados a Framework de persistência.
- Criar padrões para nomenclaturas de campos pode ser uma boa.
- Esqueça aquele ditado de que caso a entidade possua algo (atributo) que a defina como única, esse atributo deve ser chave primária, Surrogate é legal :).

DICAS PARA CRIAÇÃO DE UM BANCO DE DADOS

- Utilize índices UNIQUE para definir unicidade de uma coluna que determina a identidade de uma entidade.
- Evite chave composta.
- Defina bem o tamanho das colunas do tipo VARCHAR, em um grande volume de dados isso pode impactar na performance do banco de dados.
- Procure definir o tamanho das colunas VARCHAR com valores múltiplos de 4, o algoritmo de busca do SGBD é baseado em múltiplos de 4.

DICAS PARA CRIAÇÃO DE UM BANCO DE DADOS

- Uma base de dados é algo evolutivo. Pode ser que lá na frente você precise criar índices para melhorar performance.
- Índices BTREE não devem ser criados logo de cara, o volume de dados que vai determinar a criação dos índices BTREE.
- Índices BTREE devem ser criados para colunas que são muito utilizadas em WHERE.
- Índices BTREE devem ser criados para colunas que são muito utilizadas com LIKE ou ILIKE.
- Prefira o EXISTS ou NOT EXISTS ao IN e NOT IN, dependendo do SQL ou volume de dados isso impacta em 75% da performance.

DICAS PARA CRIAÇÃO DE UM BANCO DE DADOS

- Ao realizar um SQL que possua um JOIN e tenha filtro, verifique se o filtro pode ser acrescentado diretamente na comparação do JOIN, isso pode impactar em 75% da performance do SQL.
- Nomear as constraints e índices pode ser uma boa, isso facilita a encontrar exceções.
- Crie um script para realizar VACCUM E REINDEX semanalmente ou mensalmente, isso também impacta na criação de novos comando de DDL e QUERY.

DICAS PARA CRIAÇÃO DE UM BANCO DE DADOS

- O PostgreSQL assim como qualquer SGBD relacional possui funções de agrupamentos, evite realizar query simples e realizar os cálculos na aplicação, faça Query complexas que retorne calculado, apenas para ser exibido na aplicação.
- Avalie suas regras de negócios, que acessam muitas tabelas com muitos dados, algumas funções que tem essa natureza serão com certeza mais rápidas e eficazes, no banco de dados, PROCEDURES, TRIGGERS, VIEW, FUNCTIONS utilizem-nas.

DICAS PARA CRIAÇÃO DE UM BANCO DE DADOS

- Avalie suas queries, você sempre vai encontrar uma maneira melhor de fazer.
- A criação de **CHECK CONSTRAINT** pode ser uma boa, principalmente quando sua base é acessada por muitos usuários.
- Utilize os conceitos de objeto relacional que o PostgreSQL pode te oferecer.
- Em caso de uma base dados com volume de dados muito grande aplique alguns conceitos de DATAWAREHOUSE, como por exemplo quebre uma data em dia, mês e ano e modifique suas queries.

DICAS PARA CRIAÇÃO DE UM BANCO DE DADOS

- Não tenha medo de "desnormalizar" as tabelas, na maioria das vezes a performance é o fator crítico.