

# PostgreSQL

Banco de dados para aplicações  
web modernas



© Casa do Código

Todos os direitos reservados e protegidos pela Lei nº 9.610, de 10/02/1998.

Nenhuma parte deste livro poderá ser reproduzida, nem transmitida, sem autorização prévia por escrito da editora, sejam quais forem os meios: fotográficos, eletrônicos, mecânicos, gravação ou quaisquer outros.

*Edição*

Vivian Matsui

*Revisão*

Antonio Pedro Loureiro

*Capa*

Design Alura

[2023]

Casa do Código

Rua Vergueiro, 3185 - 8º andar

04101-300 – Vila Mariana – São Paulo – SP – Brasil

[casadocodigo.com.br](http://casadocodigo.com.br)





Este livro possui a curadoria da Casa do Código e foi estruturado e criado com todo o carinho para que você possa aprender algo novo e acrescentar conhecimentos ao seu portfólio e à sua carreira.

A Casa do Código é a editora da Alura, escola online de tecnologia que nasceu da vontade de criar uma plataforma de ensino com o objetivo de incentivar a transformação pessoal e profissional através da tecnologia.

O ecossistema da Alura constrói uma verdadeira comunidade colaborativa de aprendizado em programação, negócios, design, marketing e muito mais, oferecendo inovação na evolução dos seus alunos e alunas através de uma verdadeira experiência de encantamento.

Venha conhecer os cursos da Alura e siga-nos em nossas redes sociais.

 [alura.com.br](http://alura.com.br)

 [@casadocodigo](https://www.instagram.com/casadocodigo)

 [@casadocodigo](https://twitter.com/casadocodigo)

# ISBN

Impresso e PDF: 978-85-5519-255-5

EPUB: 978-85-5519-256-2

MOBI: 978-85-5519-257-9

Caso você deseje submeter alguma errata ou sugestão, acesse  
<http://erratas.casadocodigo.com.br>.

# PREFÁCIO

## ESCREVENDO O LIVRO QUE EU GOSTARIA DE LER

Eu sempre consumi muitos livros de desenvolvimento de software brasileiros. Antes de conhecer a Casa do Código, eu tinha uma grande frustração com os livros dedicados ao desenvolvimento de software em português, e até mesmo com alguns internacionais.

Se você já leu algum livro da Casa do Código, ele é diferente desde a capa e todo seu conteúdo. Tem uma abordagem mais moderna e menos ortodoxa do que os outros livros possuem. Na minha opinião, livros da área de desenvolvimento de software deveriam ter essa pegada mais leve e gostosa de ler.

E o que me levou a escrever meu primeiro livro, lançado em 2015 pela Casa do Código, foi a vontade de criar um que eu gostaria de ler. Isso quer dizer, com um conteúdo prático, para que o(a) leitor(a) pudesse se desenvolver nível a nível sem se frustrar com o que estivesse começando a aprender — e o mais importante, na minha opinião: com cenários e problemas comuns do dia a dia do desenvolvedor.

Este livro é para quem está começando a se aventurar no maravilhoso mundo do desenvolvimento de software e quer começar a trabalhar com um banco de dados. Este livro é para quem já conhece SQL e quer se aperfeiçoar na utilização de um gerenciador de banco de dados. Este livro também é para quem

conhece o PostgreSQL e quer construir um projeto utilizando-o.

Do começo ao fim, vamos desenvolver um projeto que pode ser aplicado na prática. Em cada exemplo, busquei aplicar problemas comuns do dia a dia de uma pessoa desenvolvedora.

## CÓDIGO-FONTE

O código-fonte de todos os códigos gerados durante o nosso projeto neste livro estão disponíveis em meu repositório no GitHub. Lá você vai encontrá-los separados por capítulos.

[https://github.com/vinicioscdes/postgresql\\_codigos](https://github.com/vinicioscdes/postgresql_codigos)

## ENVIE SEU FEEDBACK

Feedback é muito importante para todos os profissionais. Após lançar meu primeiro livro, tive muitos feedbacks positivos e muitos que trouxeram oportunidades de melhoria que pude aplicar neste meu segundo livro.

Será um imenso prazer para mim saber o que você tem a dizer sobre este meu trabalho. Você pode enviar sua dúvida ou feedback para o e-mail a seguir:

viniciuscdes@gmail.com

Se preferir, pode acessar meu site pessoal também. Lá você encontrará todas as minhas redes sociais e contatos.

<https://www.vinicioscdes.com>

# AGRADECIMENTO

Quando lancei meu primeiro livro, uma das primeiras coisas que eu fiz foi ir até a faculdade na qual me formei para doá-lo à biblioteca da instituição através das mãos de uma professora, a qual também foi minha orientadora. Este ato singelo foi um pequeno gesto para demonstrar a minha gratidão por aqueles que se dedicam a compartilhar seu conhecimento todos os dias com centenas de pessoas durante todos os anos de sua vida: os **professores**.

Desde o primeiro dia que entrei na faculdade, sempre tive em minha mente que os melhores amigos que eu poderia fazer seriam os professores. Isso porque sabia que eles estavam dispostos a ensinar todos os dias e, de vez em quando, eu também conseguia compartilhar o que eu sabia e também ensiná-los. Durante a minha faculdade, sempre busquei essa troca de conhecimento que aquele ambiente nos proporciona.

Com os professores, desde pequeno, aprendi que compartilhar conhecimento nunca é demais. E cada vez que você compartilha algo, você aprende muito mais. Eu sempre fui inquieto, e me perguntei: como estou compartilhando o que aprendi durante todos esses anos, e como eu vou deixar para as outras pessoas esse conhecimento? Foi então que surgiu a grande vontade de escrever um livro.

Então, dedico este livro a todos os professores e professoras que eu tive durante todos esses anos de vida. Acredito que uma das grandes realizações de um(a) professor(a) é saber como estão os

alunos e alunas que passaram por suas turmas. Ser professor é algo, muitas vezes, estressante. É uma dedicação diária em tentar fazer a diferença em uma sala de aula.

Só gostaria de deixar registrado que vocês fizeram a diferença em minha vida. Sempre que encontro um(a) professor(a) antigo(a), tento passar essa mensagem. Creio que sirva de incentivo para que eles e elas continuem se dedicando e para mostrar que o trabalho que eles desenvolvem não é em vão.

Gostaria de agradecer aos meus primeiros professores: minha mãe, Juraci, meu pai, Nelson, e meus irmãos, Anderson, Judson e Nelson Jr. Além de serem professores das minhas primeiras palavras, são os de meu caráter.

Gostaria de agradecer à minha esposa, Thais, pelo incentivo em todos os meus projetos.

E gostaria de agradecer e também dedicar esse livro à pessoa mais importante na minha vida, a minha filha, Maia, que, desde o seu nascimento, me ensina muitas coisas e me faz buscar o meu melhor a cada dia. O papai te ama!

## SOBRE O AUTOR

Vinícius Carvalho teve seu primeiro contato com o computador em um curso de MS-DOS com Windows 95 e, desde então, apaixonou-se pela computação. Ao longo da adolescência, procurou aperfeiçoar-se e fazer alguns cursos até chegar a hora de escolher sua formação na faculdade. Essa parte foi fácil! Formou-se em Sistemas de Informações, pós graduou-se em Engenharia de Software e não parou de aprender coisas novas.

Entusiasta da busca pelo conhecimento, procura manter-se atualizado nas tendências de desenvolvimento de software e tecnologia e tem como meta aprender algo novo todos os dias. Tecnologista, é apaixonado por tecnologia e solução de problemas através do desenvolvimento de software e construção de produtos. Atua na área de tecnologia da informação desde 2010 e, nesse período, exerceu diferentes funções, como Desenvolvedor, Analista de Negócios, Gerente de Produto, Product Owner, Scrum Master e Gerente de Projetos.

Sempre em busca da melhoria contínua e de compartilhar conhecimento com a comunidade, buscando contribuir através de palestras e voluntariado em eventos de tecnologia. Tem o foco de seus estudos em métodos e metodologias ágeis e engenharia de dados. Lançou seu primeiro livro em 2015, *MySQL: Comece com o principal banco de dados open source do mercado*, que você pode encontrar em: <https://www.casadocodigo.com.br/products/livro-banco-mysql>. Sua página pessoal é <https://www.viniciuscdes.com>. Lá você pode conferir outras informações.

# Sumário

<b>1 Introdução</b>	<b>1</b>
1.1 Banco de dados	1
1.2 PostgreSQL	4
1.3 Instalando e configurando	10
1.4 Para pensar!	17
<b>2 Comece a desenvolver com o PostgreSQL</b>	<b>18</b>
2.1 PL/pgSQL	18
2.2 Data Types: do básico ao avançado	20
2.3 Para pensar!	29
<b>3 Nossa primeiro projeto</b>	<b>31</b>
3.1 Entendendo nossos dados	34
3.2 A estrutura das tabelas	35
3.3 Chaves primárias e chaves estrangeiras	37
3.4 Trabalhando com pgAdmin	39
3.5 Criando nossas tabelas	49
3.6 Constraints: integridade de seus dados	53
3.7 Criando sequências para as nossas tabelas	59

Sumário	Casa do Código
3.8 E os nossos registros? Já podemos inserir!	63
3.9 Consultando nossos registros	67
3.10 Para pensar!	74
<b>4 Functions — Agilizando o dia a dia</b>	<b>76</b>
4.1 Functions para poupar esforços	76
4.2 Utilizando a function	81
4.3 Functions sem return	84
4.4 Alterando functions	89
4.5 Excluindo functions	90
4.6 Vantagens da utilização das functions	90
4.7 Para pensar!	91
<b>5 Funções, operadores e operações</b>	<b>92</b>
5.1 Funções embutidas	92
5.2 Operadores lógicos	93
5.3 Operadores de comparação	97
5.4 Operadores e funções matemáticas	98
5.5 Funções de texto	103
5.6 Funções data/hora	107
5.7 Funções agregadoras	115
5.8 Consultas utilizando like	130
5.9 Para pensar!	136
<b>6 Banco de dados rápido nos gatilhos</b>	<b>137</b>
6.1 Triggers — Gatilhos para agilizar tarefas	137
6.2 Triggers insert, update e delete	138
6.3 Desabilitando, habilitando e deletando uma trigger	148

6.4 Para pensar!	151
<b>7 Turbinando as consultas com joins e views</b>	<b>153</b>
7.1 Subconsultas	153
7.2 Consultas entre duas ou mais tabelas através das joins	156
7.3 Views	163
7.4 Para pensar!	168
<b>8 Administração de banco de dados e outros tópicos</b>	<b>169</b>
8.1 Administrador(a) de banco de dados vs. desenvolvedor(a)	169
8.2 Comandos úteis	171
8.3 Backups	173
8.4 Índices e performance das consultas	182
8.5 Para pensar!	190
<b>9 Tipos de dados especiais</b>	<b>192</b>
9.1 Tipos de campos especiais	192
9.2 Campos array	193
9.3 Campos do JSON	198
9.4 Para pensar!	204
<b>10 Conclusão</b>	<b>206</b>
10.1 Para pensar e agradecer!	206

Versão: 27.7.7

## CAPÍTULO 1

# INTRODUÇÃO

*"Toda empresa precisa de gente que erra, que não tem medo de errar e que aprenda com o erro." — Bill Gates*

## 1.1 BANCO DE DADOS

Tecnologias de banco de dados dão suporte diário para operações e tomadas de decisões nos mais diversos níveis da empresa, da operação à gerência. Eles são vitais para as organizações modernas que querem se manter competitivas no mercado e no cenário atual de extrema concorrência.

Diariamente, geramos milhões de dados. A todo o tempo que estamos interagindo com um aplicativo no celular ou uma página na web, estamos gerando dados. Os dados se tornaram o bem mais valioso para uma empresa. Dizem que eles são o "novo petróleo". São a matéria-prima para a inovação. O entendimento dos dados de uma empresa é crucial para a formulação de consultas e perguntas para o negócio. Entretanto, poucas empresas estão tirando proveito das informações que elas possuem em seus bancos de dados e transformando isso em inteligência de negócio, devido a pouco conhecimento da gerência ou por não possuírem ferramentas necessárias.

Quando digo dados, estou querendo dizer todos os registros gravados em um banco de dados da empresa, seja esse banco conectado a um ERP, CRM, website, aplicativo de celular etc. A análise e uma boa administração desses dados são vitais para o negócio e tomadas de decisões dentro de uma organização. Volto a frisar a importância de ter uma boa ferramenta para administrar esse bem tão precioso da empresa. E será essa ferramenta para fazer a administração de seus dados que veremos neste livro: o PostgreSQL.

## Princípios de um SGBD relacional

Se você enviou um e-mail hoje, escreveu um post no Facebook ou no Twitter, ou enviou uma mensagem de celular, essas informações que você publicou ficaram lá armazenadas. E esse armazenamento é feito em um **banco de dados**.

Estamos conectados a diversos bancos de dados diariamente. Eles estão no computador, no celular, no *tablet*, no videogame e em até em alguns eletrodomésticos como algumas geladeiras modernas que salvam listas de compras.

Os bancos de dados gerenciam de forma automatizada os dados lá armazenados. Eles são conhecidos como Sistemas Gerenciadores de Banco de Dados Relacional (SGBDR), ou apenas Sistemas Gerenciadores de Banco de Dados (SGBD). O modelo de banco de dados relacional é o mais usado, principalmente por sua capacidade de manter a integridade dos dados quando existe alteração nas estruturas das tabelas. Isso porque seus mecanismos que interligam as tabelas relacionadas fazem com que seja muito seguro o trabalho com um SGBD relacional. Veremos esses

mecanismos no decorrer do livro.

O conceito básico de SGBD relacional é um conjunto de **tabelas** relacionadas, e estas são compostas por alguns elementos básicos: **colunas, linhas e campos**. Além desses elementos, o SGBD possui outros que também serão apresentados aqui. Cada um deles será demonstrado e analisado, não se preocupe em conhecê-los agora.

## **Importância do banco de dados no projeto de construção de software**

Os dados de uma empresa, se não forem o elemento mais precioso, estão entre eles. Uma informação armazenada incorretamente, ou de forma desordenada, pode custar todo o negócio. Sabendo disso, não tenha medo de desenhar esquemas, testar os esquemas das tabelas, trocar opiniões com outras pessoas desenvolvedoras na hora de modelar um banco de dados.

Realizar uma manutenção na estrutura de suas tabelas após o sistema em produção é um custo muito caro para o projeto. Além de ter um impacto na ocupação do tempo dos programadores, caso você esteja modelando o banco, custará o seu tempo de retrabalho, como também pode ter um impacto diretamente em seus usuários, podendo gerar muita reclamação ou o encerramento do seu projeto.

Sempre que tenho a oportunidade de falar sobre projetos de software, principalmente sobre a construção de banco de dados, deixo muito claro que essa etapa dirá muito sobre a qualidade do seu sistema no futuro. É claro, conforme o seu sistema vai crescendo, pode surgir a necessidade de fazer alterações em

algumas estruturas, mas se a modelagem for feita pensando em um cenário escalável, suas chances de sucesso vão aumentar consideravelmente.

## 1.2 POSTGRESQL

O PostgreSQL é um poderoso sistema gerenciador de banco de dados objeto-relacional de código aberto. Por muito tempo, foi descredibilizado no mundo dos bancos de dados, e o seu recente aumento de popularidade veio de usuários de outros bancos de dados em busca de um sistema com melhores garantias de confiabilidade, melhores recursos de consulta, mais operação previsível, ou simplesmente querendo algo mais fácil de aprender, entender e usar. Você encontrará no PostgreSQL todas essas coisas citadas e muito mais.

Com mais de 15 anos de desenvolvimento ativo e uma arquitetura que comprovadamente ganhou forte reputação de confiabilidade, integridade de dados e conformidade a padrões, o PostgreSQL tem como características:

- **É fácil de usar:** comandos SQL do PostgreSQL são consistentes entre si e, por padrão, as ferramentas de linha de comando aceitam os mesmos argumentos. Os tipos de dados não têm truncamento silencioso ou outro comportamento estranho. Surpresas são raras, e essa facilidade de utilização se generaliza para outros aspectos do sistema.
- **É seguro:** o PostgreSQL é totalmente transacional, incluindo mudanças estruturais destrutivas. Isso significa

que você pode tentar qualquer coisa com segurança dentro de uma transação, mesmo a exclusão de dados ou alterar estruturas de tabela, com a certeza de que, se você reverter a transação, cada mudança que você fez será revertida. Fácil backup e restauração tornam trivial clonar um banco de dados.

- **É poderoso:** o PostgreSQL suporta muitos tipos de dados sofisticados, incluindo JSON, XML, objetos geométricos, hierarquias, tags e matrizes. Novos tipos de dados e funções podem ser escritos em SQL, C, ou linguagens procedurais muito incorporadas, incluindo Python, Perl, TCL e outras. Extensões adicionam diversas capacidades rápida e facilmente, incluindo *full-text search*, acompanhamento de *slow query*, criptografia de senha e muito mais. Durante o livro, veremos exemplos acompanhados de uma explicação teórica para ficar fácil o entendimento.
- **É confiável:** o PostgreSQL é muito amigável tanto para o desenvolvimento de software quanto para a administração de banco de dados. Todas as conexões são processos simples e podem ser gerenciadas por utilitários do sistema operacional. Ele também fornece ao sistema operacional o que o banco e cada conexão estão fazendo. O layout de pasta padrão torna mais fácil de controlar onde os dados são armazenados para que você possa fazer o uso máximo do seu particionamento. Ele usa as facilidades de inicialização do sistema operacional em todas as plataformas.
- **É rápido:** o PostgreSQL faz uso estratégico de indexação e

consulta de otimização para trabalhar com o menor esforço possível. Ele tem um dos planejadores de consulta mais avançados de qualquer banco de dados relacional, e ainda expõe seu raciocínio interno através da funcionalidade `Explain`. Logo, você pode encontrar e corrigir problemas de desempenho se eles surgirem. PostgreSQL é referência com ótima performance em operações de leitura-escrita, conjuntos de dados massivos e consultas complexas.

## Onde, quando e como?

Como um banco de dados de nível corporativo, o PostgreSQL possui funcionalidades sofisticadas, como:

- O controle de concorrência multiversionado (MVCC, em inglês);
- Recuperação em um ponto no tempo (PITR, em inglês), *tablespaces*;
- Replicação assíncrona;
- Transações agrupadas (*savepoints*);
- Cópias de segurança quente (online/hot backup);
- Um sofisticado planejador de consultas (otimizador) e registrador de transações sequencial (WAL) para tolerância a falhas;
- Suporta conjuntos de caracteres internacionais;
- Codificação de caracteres *multibyte*, *Unicode* e sua ordenação por localização;
- Sensibilidade à caixa (maiúsculas e minúsculas) e formatação;
- É altamente escalável, tanto na quantidade enorme de dados que pode gerenciar quanto no número de usuários

concorrentes que pode acomodar. Existem sistemas ativos com o PostgreSQL em ambiente de produção que gerenciam mais de 4 TB de dados.

Resumindo, o que você precisar, não ultrapassando os limites apresentados na lista a seguir, o PostgreSQL poderá lhe oferecer com a excelência de um grande banco de dados *Open Source*.

Alguns limites do PostgreSQL estão incluídos na lista a seguir:

- **Tamanho máximo do banco de dados:** ilimitado;
- **Tamanho máximo de uma tabela:** 32 TB;
- **Tamanho máximo de uma linha:** 1.6 TB;
- **Tamanho máximo de um campo:** 1 GB;
- **Máximo de linhas por tabela:** ilimitado;
- **Máximo de colunas por tabela:** 250–1600, dependendo do tipo de coluna;
- **Máximo de índices por tabela:** ilimitado.

Com essas informações, respondendo às perguntas sobre onde, quando e como, podemos dizer que poderemos criar desde aplicativos até complexos sistemas ERP para gerenciar uma empresa (de pequeno até grande porte).

## Como usar o PostgreSQL?

Se você nunca usou um banco de dados relacional, respire fundo e você verá como as coisas são simples. PostgreSQL é realmente baseado em alguns conceitos bastante fáceis, aplicados com rigor.

Imagine uma tabela com alguns dados contida em uma

planilha do Excel. O PostgreSQL é como um sistema que gerenciará essas tabelas. Só que quando você tem uma planilha aberta, somente uma pessoa pode estar editando — diferentemente do SGBD, em que muitas pessoas podem estar mexendo.

Essas tabelas do banco de dados possuem uma estrutura rígida imposta pelo sistema de gestão de dados para que as informações contidas nelas não sejam corrompidas. Cada informação e cada estrutura inserida no banco de dados deve seguir uma série de especificações e padrões. Volto a frisar que veremos cada uma dessas especificações e padrões quando cada elemento for apresentado.

Você vai interagir com essas tabelas usando uma linguagem chamada *Structured Query Language* (SQL), que foi projetada para ser fácil de aprender e ler, sem sacrificar a potência. Se você já está usando um banco de dados relacional, começando com PostgreSQL, é fácil. Você só precisa instalá-lo. Mas não feche o livro e não desista do PostgreSQL, porque mais à frente veremos como fazer isso, deixar tudo pronto, aprender a criar usuários e bancos de dados e como se conectar.

A partir daí, é apenas uma questão de descobrir quais são as diferenças entre o seu banco de dados relacional antigo e o PostgreSQL e começar a fazer uso de novos e interessantes recursos que só ele tem. Se você já está usando um sistema não relacional, como um banco de dados NoSQL, seu caminho será semelhante, mas você também pode ter de aprender algo sobre como estruturar um banco de dados relacional.

Você vai descobrir que, com replicação, armazenando XML, JSON e usando o Ltree e extensões do PostgreSQL hstore, você

pode obter muitos benefícios de seu sistema NoSQL. Você verá que poderá utilizar em todos seus projetos, tanto online como offline, e testará sua imaginação muitas vezes para conseguir usar todas as suas funcionalidades.

## SQL no PostgreSQL muda alguma coisa?

SQL significa *Structured Query Language* e é a linguagem padrão utilizada pelos bancos de dados relacionais. Os principais motivos disso resultam de sua simplicidade e facilidade de uso. Mais uma vez, não entrarei no mérito histórico, mas algo relevante que você precisa conhecer são suas categorias de comandos.

Alguns autores divergem entre exatamente quais são. Ao pesquisar em um estudo diferente, você pode encontrar que alguns comandos citados por mim em uma categoria talvez estejam em outra. Eu separei três. Elas são:

- **DML — Linguagem de Manipulação de Dados:** esses comandos indicam uma ação para o SGBD executar. Usados para recuperar, inserir e modificar um registro no banco de dados. Seus comandos são: `INSERT` , `DELETE` , `UPDATE` , `SELECT` e `LOCK` .
- **DDL — Linguagem de Definição de Dados:** comandos DDL são responsáveis pela criação, alteração e exclusão dos objetos no banco de dados. São eles: `CREATE TABLE` , `CREATE INDEX` , `ALTER TABLE` , `DROP TABLE` , `DROP VIEW` e `DROP INDEX` .
- **DCL — Linguagem de Controle de Dados:** responsável pelo controle de acesso dos usuários, controlando as

sessões e transações do SGBD. Alguns de seus comandos são: COMMIT , ROLLBACK , GRANT e REVOKE .

Cada um dos comandos aqui citados será explicado ao longo do livro e aplicado em nosso projeto!

Pelos números apresentados, podemos dizer que as pessoas desenvolvedoras estão satisfeitas com o desempenho e uso do PostgreSQL. Eu, particularmente, mesmo sendo suspeito para falar, estou muito satisfeito com o que o PostgreSQL tem me retornado em projetos nos quais o estou utilizando.

Tenho projetos em MySQL, Oracle e PostgreSQL. Cada um tem uma história e necessitava de uma estrutura. Os que estão usando PostgreSQL estão com um excelente desempenho, não tenho do que reclamar. São projetos de CRM de médio, grande porte, desenvolvidos para regras de negócios específicas.

## 1.3 INSTALANDO E CONFIGURANDO

Durante o livro, para desenvolvemos o nosso projeto, vamos utilizar a versão 14 do PostgreSQL. Descreverei como você poderá instalar nos três principais sistemas operacionais mais usados - Linux, Mac OS e Windows.

Para fazer o download das versões disponíveis, acesse o link: <https://www.postgresql.org/download>. Lá você poderá baixar a versão específica para o sistema operacional que desejar.

## Instalando no Mac OS

Para fazer a instalação, estou usando o Mac OS 12 Monterey. Para Mac temos algumas possibilidades para instalar o PostgreSQL. Você pode utilizar o instalador interativo criado pela empresa EDB, ou você pode utilizar o Homebrew (<https://brew.sh/>), que é o gerenciador de pacotes mais utilizado no Mac, ou ainda utilizar o aplicativo chamado **PostgreApp**, que você encontra na página de downloads, sendo uma maneira mais simples de usar o PostgreSQL no Mac. Ele roda como um serviço sem a necessidade de instalação. Você baixa e executa-o, muito simples. E será essa terceira opção que vamos utilizar.

Primeiro, vá até o site <https://postgresapp.com/> e faça o download. Quando ele estiver baixado, abra-o. Ao abrir, você verá que, na barra do topo, aparecerá uma imagem de um elefante que mostrará o serviço do PostgreSQL rodando. É possível iniciar ou parar o serviço através do ícone, como mostra a figura na sequência, ou abrir o app através da opção *Open Postgres*.

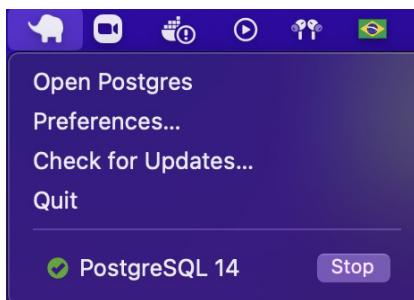


Figura 1.1: Serviço do PostgreSQL rodando.

Ao clicar em *Open Postgres*, vai abrir a janela mostrada na figura a seguir, onde é possível criar novos servidores do

PostgreSQL e iniciar ou parar um serviço. A figura exibe os servidores de banco de dados existentes. No caso abaixo, os servidores criados são: `postgres`, `template1` e `viniciuscarvalho`. Cada um deles é um servidor de banco de dados PostgreSQL.

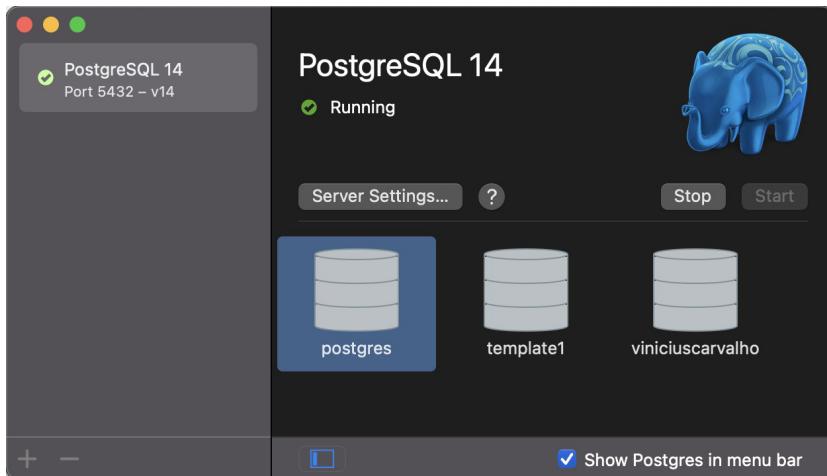


Figura 1.2: Tela de início do PostgreApp.

Com um duplo clique em `postgres`, abrirá um console que pode ser utilizado para escrever scripts e comandos de manipulação do PostgreSQL. Mas, durante o projeto, vamos utilizar uma ferramenta para trabalharmos com banco de dados que mostrarei no capítulo *Nosso primeiro projeto*: a ferramenta **pgAdmin**.

## Instalando no Linux

Na página de download, você encontrará versões disponíveis para diversas distribuições do Linux, como Red Hat, Debian, Ubuntu, Suse e para versões genéricas. Para realizar a instalação, vou utilizar o Ubuntu, versão 22.04.

No Ubuntu, temos as opções de baixar pacotes de instalação compilados, ou via comandos. Eu particularmente prefiro fazer a instalação via comandos, uma vez que é mais rápido e simples.

Primeiramente, atualizaremos os pacotes com:

```
$> sudo sh -c "echo 'deb http://apt.postgresql.org/pub/repos/apt / precise-pgdg main' > /etc/apt/sources.list.d/pgdg.list";  
$> wget --quiet -O - http://apt.postgresql.org/pub/repos/apt/ACCC  
4CF8.asc | sudo apt-key add -
```

Para manter sempre atualizados os pacotes de programas no Linux, utilizamos o comando:

```
$> sudo apt-get update  
$> sudo apt-get install postgresql-common
```

Após a atualização, podemos baixar a nova versão desejada com o seguinte comando:

```
$> sudo apt-get install postgresql-14;
```

Como queremos baixar uma versão específica, é preciso escrever como fizemos. Se tivéssemos escrito apenas:

```
$> sudo apt-get install postgresql;
```

Seria baixada a última versão liberada.

Pronto, já podemos utilizá-lo. Para acessá-lo, abra o terminal e digite o comando:

```
$> sudo -i -u postgres psql
```

O `postgres` é o nosso usuário e banco de dados criado por padrão do PostgreSQL. Ao logar, altere a senha do nosso usuário com o comando:

```
$> alter user postgres with password 'senha';
```

Agora saia do terminal usando `\q`, e acesse novamente, usando o comando:

```
$> psql -U postgres postgres -h localhost
```

Informe a senha, e pronto. Já podemos brincar com o nosso banco!

## Instalando no Windows

Depois de ter feito o download no site do PostgreSQL para o *Windows*, execute o arquivo. A instalação é bem intuitiva, siga clicando no botão *Next* até chegar à tela a seguir.

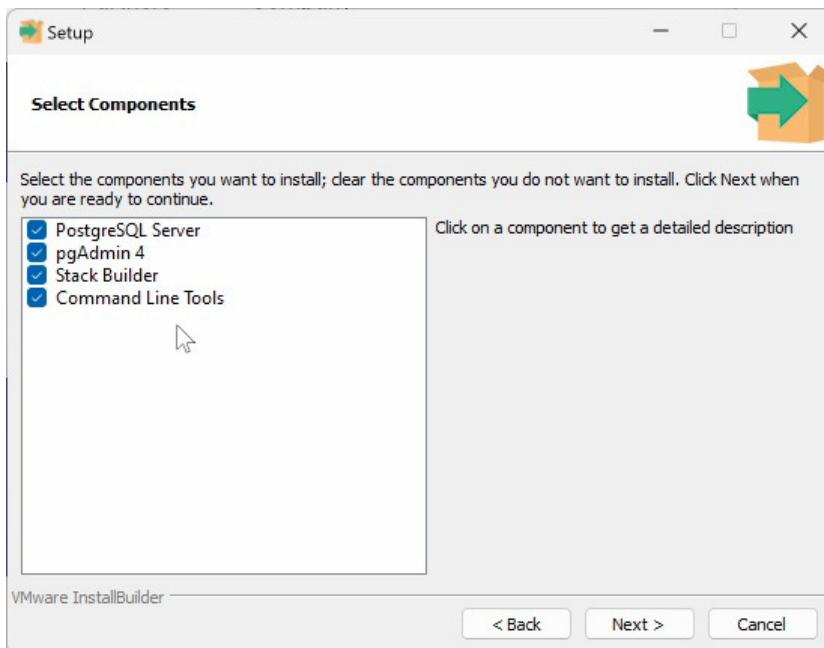


Figura 1.3: Instalação no Windows.

Nessa tela escolha o que você deseja instalar junto com o gerenciador de banco de dados. Por padrão, você pode deixar tudo marcado, pois em algum momento você poderá precisar de algum desses itens. O **PostgreSQL Server**, que é o nosso gerenciador de banco de dados; o **pgAdmin 4**, que será a nossa ferramenta de manipulação de banco; o **Stack Builder**, que é uma interface gráfica que facilita o download de módulos complementares; e as ferramentas de linhas de comando, que são necessárias quando não temos uma ferramenta com interface gráfica disponível.

Depois, digite uma senha de sua escolha para o usuário padrão `postgres` de seu banco de dados, e então clique em *Next*.

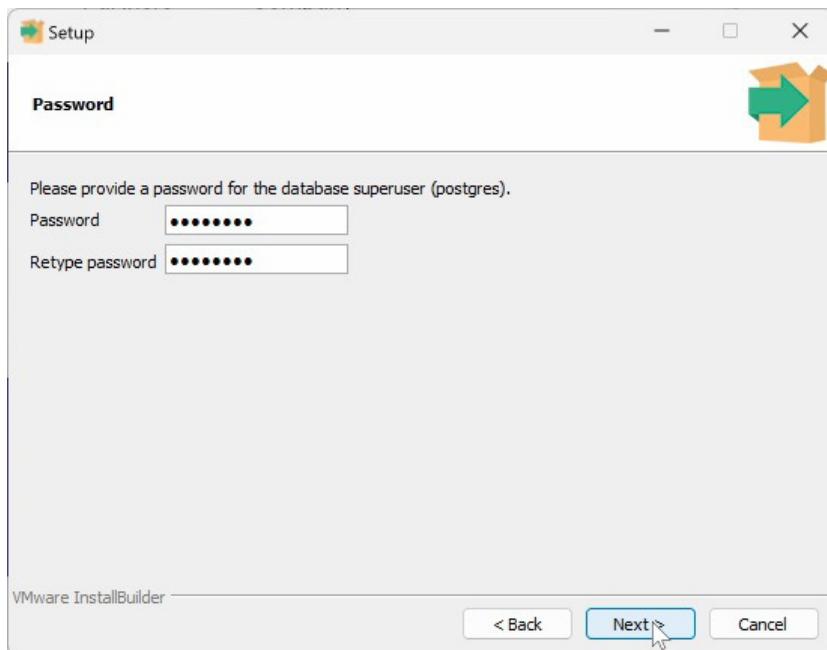


Figura 1.4: Instalação no Windows — Passo 5.

Na tela seguinte, terá um *input box* com a porta de acesso do gerenciador de banco de dados. Por padrão, o PostgreSQL utiliza a porta 5432. Se você não possuir muito conhecimento, aconselho deixar a padrão e clicar em *Next*.

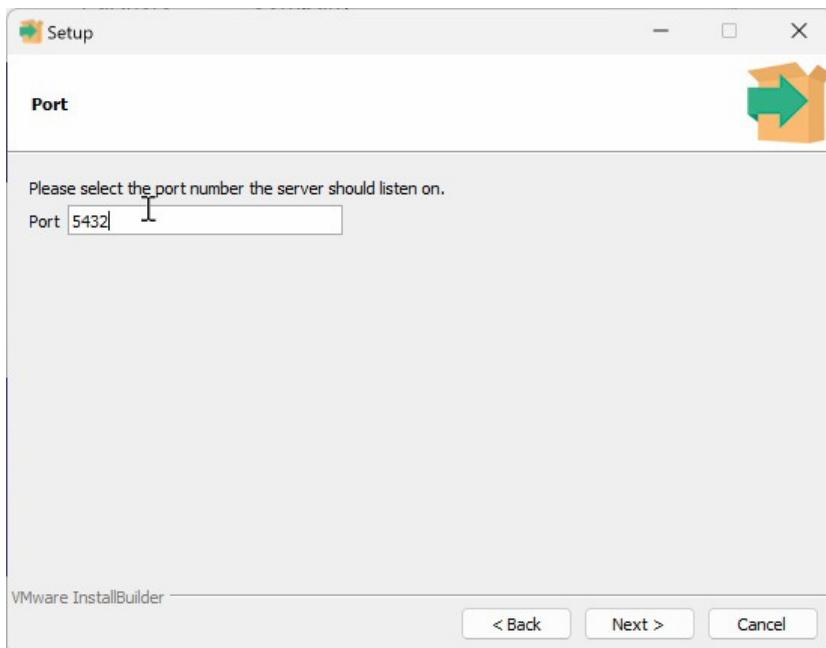


Figura 1.5: Instalação no Windows — Passo 6.

As duas últimas telas serão *Next* e depois *Finish* para você concluir a sua instalação e começar a utilizar o PostgreSQL.

Os comandos de criação de novos bancos e outras coisas serão os mesmos para todos os sistemas. Por isso, mostrarei mais à frente.

Durante o desenvolvimento do projeto deste livro, usarei um Mac como meu sistema operacional principal para

desenvolvimento. Se algum comando que eu fizer for diferente em outros sistemas, mostrarei ambos, não se preocupe. Você pode programar com o sistema operacional que mais lhe agradar. Além do mais, vamos utilizar o pgAdmin para trabalhar com o PostgreSQL, que é compatível com todos os sistemas operacionais.

## 1.4 PARA PENSAR!

Se você vai desenvolver um sistema que várias pessoas vão utilizar, você precisa instalar o PostgreSQL em um servidor. A maioria das pessoas desenvolvedoras prefere servidores com *Linux*, e realmente são melhores, mas nada impede que você tenha um servidor com *Windows*. Para o desenvolvimento, utilize o sistema operacional que você mais gostar e com o qual se sente bem. E, nesse caso, você tem essas três opções.

Seu projeto será *offline* ou *online*? Você conhece as ferramentas de desenvolvimento do sistema operacional que você usa? Se seu sistema rodar *offline*, você sabe como montar uma rede? Se ele for rodar na *web*, você conhece as plataformas de hospedagem?

Pense um pouco nas perguntas anteriores como preparação para iniciarmos o projeto que desenvolveremos durante o livro. Neste capítulo, conhecemos um pouco sobre o SQL e o PostgreSQL. No próximo, vamos conhecer um pouco mais sobre os padrões de dados do PostgreSQL e começar a esboçar o nosso projeto. Veremos mais detalhes na sequência.

## CAPÍTULO 2

# COMECE A DESENVOLVER COM O POSTGRESQL

*"As únicas grandes companhias que conseguirão ter êxito são aquelas que consideram os seus produtos obsoletos antes que os outros o façam." — Bill Gates*

## 2.1 PL/PGSQL

Se você trabalha ou já trabalhou com o Oracle, sabe o que é o PL/SQL. É a linguagem procedural do banco de dados Oracle que permite a inclusão de lógica no SGBD. Já o PL/pgSQL (*Procedural Language/PostgreSQL*) é a linguagem de programação procedural do PostgreSQL. Com ela, é possível inserir lógica em seu banco de dados.

Neste momento, você deve estar se perguntando por que precisaríamos de lógica ou de uma linguagem para trabalhar com banco de dados, já que os comandos SQL conseguem apenas manipular os dados em SGBD, e utilizamos alguns objetos quando precisamos criar processos automatizados ou que tenham a necessidade de serem aplicados rotineiramente. Mas eu explicarei e mostrarei esses processos nos capítulos *Functions — Agilizando o dia a dia, Funções, operadores e operações, Banco de dados rápido*

*nos gatilhos e Turbinando as consultas com joins e views.* Aproveite esses três primeiros capítulos para absorver bem a introdução de vários conceitos que são a base de seus estudos sobre banco de dados.

PL/pgSQL é uma linguagem procedural que você grava no sistema de banco de dados PostgreSQL. Os objetivos do PL/pgSQL foram criar uma linguagem procedural carregável que pode ser usada para criar funções e procedimentos de gatilhos, acrescentar estruturas de controle à linguagem SQL, poder realizar cálculos complexos e herdar todos os tipos, funções e operadores definidos pelo usuário. Ela pode ser definida para ser confiável para o servidor e é fácil de usar.

Funções criadas com PL/pgSQL podem ser usadas em qualquer lugar em que funções internas são utilizadas. Por exemplo, é possível criar funções de cálculo condicional complexas e depois usá-las em chamadas de *triggers*, como veremos no capítulo *Funções, operadores e operações*, ou em eventos agendados, como será mostrado no capítulo *Turbinando as consultas com joins e views*.

Nos próximos capítulos, entraremos em assuntos em que vamos usar PL/pgSQL. Conforme formos fazendo os exemplos, vou explicando a melhor maneira de utilizá-la em seus projetos e tudo vai ficar mais claro.

Neste capítulo, conheceremos algumas características dos tipos de dados que o PostgreSQL suporta e iniciaremos a descrição do nosso projeto que vamos desenvolver no decorrer deste livro.

## 2.2 DATA TYPES: DO BÁSICO AO AVANÇADO

Cada informação deve ser armazenada com o seu tipo correto. Isto é, um campo em que serão inseridos apenas números deverá ser do tipo *numérico*. É extremamente importante e crítico fazer a escolha do tipo de cada informação que você vai armazenar.

Quem está fazendo o projeto do banco de dados deve conhecer o tipo de informação e os tipos disponíveis no banco. Por exemplo, se quisermos armazenar um campo numérico do tipo real e criarmos para isso um campo numérico inteiro na tabela, seria um grande problema. Isso porque seria um erro tentar inserir um número decimal em um campo que apenas suporta números inteiros.

Outro problema que é muito comum entre desenvolvedores(a) é a utilização de campos do tipo `string`, que armazenam qualquer tipo de caractere, em campos que deveriam armazenar apenas números. Então, vamos conhecer os tipos de dados disponíveis no PostgreSQL.

### Campos do tipo `string`

Usados para armazenar campos alfanuméricicos. Entre os campos do tipo `string`, temos:

- `varying(n)` : variável do tipo `string`. Devemos informar o limite entre parênteses;
- `varchar(n)` : tipo de variável padrão para o tipo `string`. Devemos informar o limite entre parênteses;
- `character(n)` : tipo de variável `string`;
- `char(n)` : tipo de `string` que possui o tamanho fixo.

Entre parênteses, deve ser informado o número de caracteres;

- `text` : variável de tamanho ilimitado. Usamos campos `text` para armazenar informações no formato de texto, como uma descrição ou um campo de observação em formulários.

## Campos do tipo boolean

Variáveis do tipo `boolean` são utilizadas para testar se uma condição é verdadeira ou falsa. Quando iniciarmos os nossos exemplos, veremos como usá-las.

## Campos do tipo numérico

Utilizados para armazenar números. Não use campos do tipo `string` para armazenar números. Dê preferência para campos do tipo da informação que você vai armazenar. Com isso, evitará futuros transtornos e erros em seu software, principalmente erros de inconsistência de dados.

Entre os campos do tipo numérico, temos:

- `smallint` : capacidade de 2 *bytes* de armazenamento e pequena variação. Ele suporta números inteiros de -32768 até +32767;
- `integer` : capacidade de armazenamento de 4 *bytes*, principal escolha para utilizar em campos para armazenar números inteiros. Ele suporta números de -2147483648 até +2147483647;
- `bigint` : capacidade de armazenamento de 8 *bytes*. Possui uma grande capacidade de armazenamento. Pode

armazenar números inteiros de -9223372036854775808 até +9223372036854775807;

- `decimal` : tipo numérico usado especificamente quando for preciso armazenar números com precisão decimal. Pode armazenar exatos 131072 dígitos antes do ponto decimal e 16383 dígitos depois do ponto decimal;
- `numeric` : também utilizado para armazenar número com precisão decimal. Possui a capacidade de armazenar exatos 131072 dígitos antes da casa decimal e 16383 dígitos depois da casa decimal;
- `real` : com a capacidade de armazenamento de 4 *bytes*, esse tipo de campo armazena números reais e com até 6 dígitos decimais;
- `double precision` : com a capacidade de armazenamento de 8 *bytes*, esse campo pode armazenar até 15 dígitos nas casas decimais;
- `smallserial` : com a capacidade de armazenamento de 2 *bytes*, esse campo é um que se autoincrementa. Armazena números inteiros de 1 até 32767;
- `serial` : com a capacidade de armazenamento de 4 *bytes*, também é um campo de autoincremento. Armazena números inteiros de 1 até 2147483647;
- `bigserial` : com a capacidade de armazenamento de 8 *bytes*, é o campo autoincremental com a maior capacidade de armazenamento. Armazena números inteiros de 1 até 9223372036854775807.

## Campo autoincremental

São campos que possuem a capacidade de aumentar automaticamente. A cada novo registro em uma tabela, ele soma

+1 ao número anterior e o insere no campo. Veja um exemplo:

Tenho em uma tabela um campo com o nome de `controle` que é o tipo `serial`, e o campo `cidade` que é do tipo `varchar(10)`. Na tabela, temos:

controle	cidade
1	Americana
2	Brasília
3	Curitiba
4	Dracena
5	Eldorado

Eu não precisei inserir o registro no campo `controle`, pois ele é autoincremental. Conforme eu inseria os registros na coluna `cidade`, o campo ia se incrementando e inserindo os próximos registros.

## Campos do tipo data

Usados para armazenar datas. Outra gambiarra muito utilizada é o uso de campos `strings` para o armazenamento de datas. Fuja para bem longe desses "padrões" de desenvolvimento, pois estão errados.

Utilize o tipo correto para cada campo, pois, ao usar o tipo errado, isso pode lhe prejudicar em operações entre datas que você pode vir a utilizar posteriormente. Imagine que você crie um campo de data de aniversário do tipo `texto` e, por algum motivo, salva uma letra neste campo. Na sequência, você utiliza esse campo, que deveria receber somente datas, para fazer um cálculo

de diferença do número de dias entre dois valores. O seu sistema vai retornar um belo de um erro!

É por isso que o PostgreSQL tem alguns formatos para armazenar datas. São eles:

- `timestamp` : capacidade de armazenamento de 8 bytes.  
Armazena data e hora;
- `date` : capacidade de armazenamento de 4 bytes.  
Armazena apenas datas;
- `time` : capacidade de armazenamento de 8 bytes.  
Armazena apenas horas.

## Campo do tipo UUID

Em campos ID (identificadores únicos) das tabelas, tanto na literatura como na prática, é comum encontrarmos esse campo sendo autoincremento sequencial. A utilização de um campo ID como sequencial, principalmente em tabelas onde você expõe os dados através de API externa, pode trazer alguns riscos a sua aplicação. Vejamos o seguinte exemplo.

controle	cidade
1	Americana
2	Brasilia
3	Curitiba
4	Dracena
5	Eldorado

Uma das APIs que utilizamos na aplicação para listar os dados da tabela poderia ter a seguinte URL passando o ID como

parâmetro: <https://suaaplicao.com/clientes/1>

Agora perceba que é muito fácil e intuitivo. Para pesquisarmos outro registro de cliente, basta trocar o 1 por um número maior que ele e então a API vai retornar os dados do respectivo cliente. Isso exposto na internet pode ser um risco para a sua aplicação no quesito vazamento de dados, pois as pessoas podem, de uma forma simples, ter acesso a todos os dados de outros clientes. Agora, veja este outro exemplo:

<https://suaaplicao.com/clientes/328b0985-5241-43ba-9b16-e689639a4feb>

Com esse tipo de código fica muito mais difícil o vazamento de dados da sua aplicação. E esse código é o tipo de dado UUID. Ele é um identificador universal de 128 bits que é gerado por um algoritmo escolhido para que esse identificador não seja gerado em duplicidade por qualquer outra pessoa usando o mesmo sistema. Portanto, para sistemas distribuídos, esses identificadores fornecem uma garantia de exclusividade melhor do que os geradores de sequência, que são exclusivos apenas em um único banco de dados.

Um UUID é escrito como uma sequência de dígitos hexadecimais minúsculos, em vários grupos separados por hifens, especificamente um grupo de 8 dígitos seguido por 3 grupos de 4 dígitos, seguidos por um grupo de 12 dígitos, para um total de 32 dígitos, representando 128 bits. Um exemplo de um UUID neste formulário padrão é: 328b0985-5241-43ba-9b16-e689639a4feb

Durante o livro, vamos nos aprofundar mais nesse tipo de dado, como e quando utilizá-lo.

## Campo do tipo ENUM

Os tipos de dados *ENUM* são tipos que compreendem um conjunto de valores permissíveis em uma coluna. Eles são equivalentes aos tipos de enumeração suportados em várias linguagens de programação. Um exemplo de um tipo de enumeração pode ser os dias da semana, assim teríamos em um campo possibilidades como: "Segunda-Feira", "Terça-Feira", "Quarta-Feira", "Quinta-Feira" e "Sexta-Feira".

## Campo do tipo Full-Text Search

PostgreSQL possui dois tipos de dados que são designados para dar suporte a *Full-Text Search*, que é a atividade de busca através de coleções na própria linguagem do SGBD, em registros locais, para localizar as semelhanças das consultas. Também é a técnica de indexação, pesquisa e relevância do PostgreSQL, que utiliza um conjunto de regras naturais para adicionar suporte a modo verbal (derivações de um verbo), através da utilização de dicionários e algoritmos específicos.

Podemos usar a busca completa, ou *text search*, para considerar nas pesquisas as derivações dos termos utilizados nas buscas, por exemplo, formas de conjugação de verbos, sinônimos e similaridade. O Full-Text Search pode ser usado em situações nas quais uma grande quantidade de texto precisa ser pesquisada e o resultado deve obedecer às regras linguísticas e ordenação por relevância.

Por exemplo, uma pesquisa por páginas dentro de um gerenciador de conteúdo web, ou por termos dentro da sinopse de um acervo de filmes. O Full-Text Search possui grandes vantagens

em relação a outras alternativas para pesquisas textuais, como o comando `LIKE`.

Nesse contexto, os tipos usados para *Full-Text Search* (ou FTS) são:

- `tsvector` : tipo de dados que representa um documento, como uma lista ordenada e com posições no texto;
- `tsquery` : tipo de dado para busca textual que suporta operadores booleanos.

Ainda falta aprendermos alguns conceitos para usar os tipos FTS . Mais à frente, no momento certo, vamos utilizar esse tipo de campo e você perceberá como ele pode lhe ajudar em tarefas complexas.

## Tipo XML

XML, ame ou odeie. Há uma grande quantidade de dados em formatos XML, e esse fato não está mudando rapidamente devido a um grande investimento em XML. Ocasionalmente, inserir dados XML em um banco de dados relacional pode render uma vitória quando a integração com fontes de dados externas fornece dados em XML.

PostgreSQL tem a capacidade de manipular dados XML com SQL, permitindo um caminho para integrar dados XML em consultas SQL. A declaração desse tipo de campo é como a dos outros tipos. Mais adiante, quando formos desenvolver exemplos desse tipo de campo, veremos na prática a sua utilização e como podemos nos beneficiar dele.

## Tipo JSON

PostgreSQL tem suporte a JSON já há algum tempo. Na versão 9.2, foi adicionado suporte nativo a esse tipo de dados, e os usuários desse poderoso gerenciador de banco de dados começaram a utilizar o PostgreSQL como um banco "NoSQL", que é um banco não relacional. Mas esse assunto fica de tarefa de casa para vocês.

Na versão 9.4, foi adicionada a funcionalidade para armazenar JSON como JSON binário (JSONB), que remove os espaços em branco insignificantes (não que seja um grande negócio), acrescenta um pouco de sobrecarga quando inserir dados, mas fornece um benefício enorme ao consultar.

Se você estava pensando em trocar o seu banco de dados relacional por um NoSQL, pode começar a rever seus conceitos. Isso porque, se você tem a possibilidade usar os benefícios de um banco NoSQL no PostgreSQL, então por que mudar?

Veremos na prática, em nosso projeto, como extrair do PostgreSQL os benefícios de um banco NoSQL utilizando campos do tipo JSON.

## Tipo array

PostgreSQL permite que colunas de uma tabela sejam definidas como matrizes multidimensionais de comprimento variável. Podem ser criadas matrizes de qualquer tipo de base definida pelo usuário, tipo de enumeração, ou tipo composto. Podemos então usar qualquer tipo de campo como um array.

Podemos definir um array unidimensional ou

multidimensional. Na próxima lista, seguem algumas maneiras de fazer a declaração desse tipo de campo. Veja que o `n` significa o seu campo, e cada `[]` significa as dimensões. Se não colocarmos um valor, o SGBD entenderá como valor indefinido.

- `integer[n]` : array do tipo inteiro com o tamanho igual `n` ;
- `varchar[n][n]` : array do tipo `varchar` bidimensional `n por n` ;
- `double array` : array do tipo `double` unidimensional de tamanho indefinido.

## Tipo composto

O tipo composto descreve a estrutura de uma linha ou registro. O PostgreSQL permite que os valores de tipo composto sejam utilizados de muitas maneiras idênticas às dos tipos simples. Por exemplo, uma coluna de uma tabela pode ser declarada como sendo de um tipo composto em outra tabela. Em outras palavras, posso ter um campo do tipo de uma outra tabela inteira.

## 2.3 PARA PENSAR!

Neste capítulo, conhecemos diversos tipos de campos, cada um para armazenar um tipo de informação. Citei várias vezes a importância da utilização do tipo de campo certo para cada informação que será armazenada. Se você estiver realizando algum projeto de banco de dados, tente observar se algum campo que você criou poderia ser alterado e melhorado; ou, com base no projeto proposto, pense em quais campos vamos criar para cada tabela e seus respectivos tipos. Tente também imaginar quais

seriam as implicações na escolha de um tipo errado para um campo.

Agora que já conhecemos os tipos de campos que podemos criar e os tipos de dados que podemos armazenar, podemos iniciar o desenvolvimento de nosso projeto. Vamos começar pelo projeto e sua descrição; depois, partiremos para a criação dos códigos.

## CAPÍTULO 3

# NOSSO PRIMEIRO PROJETO

*"Não se preocupe se não funcionar direito. Se tudo funcionasse, você estaria desempregado."* — Lei de Mosher da Engenharia de Software

Não me canso de falar como é produtivo e muito mais fácil aprender a programar quando temos de fazer um projeto real ou baseado na realidade de algum tipo de negócio. Durante o livro, não será diferente. Vamos criar um projeto que vai nos acompanhar durante toda a leitura e nos basear em sua regra de negócio para trabalhar com o PostgreSQL.

Vamos imaginar que fomos contratados para desenvolver um sistema para um restaurante. Então, precisamos pensar no esquema de tabelas para atender a essa demanda. Vamos criar algo que será simples, no entanto, conseguiremos testar todos os aspectos do banco de dados, inclusive todos os tipos de dados. Devemos imaginar tudo que será preciso para o funcionamento básico para vendas e pedidos de um restaurante, como: mesas, produtos, vendas, funcionários e as comissões dos funcionários.

Para visualizarmos melhor como ficará o *schema* do banco de

dados, vamos criar um D.E.R. (diagrama de entidade e relacionamento). Com isso, teremos uma forma visual para conhecer as tabelas e seus relacionamentos para criar os *scripts*.

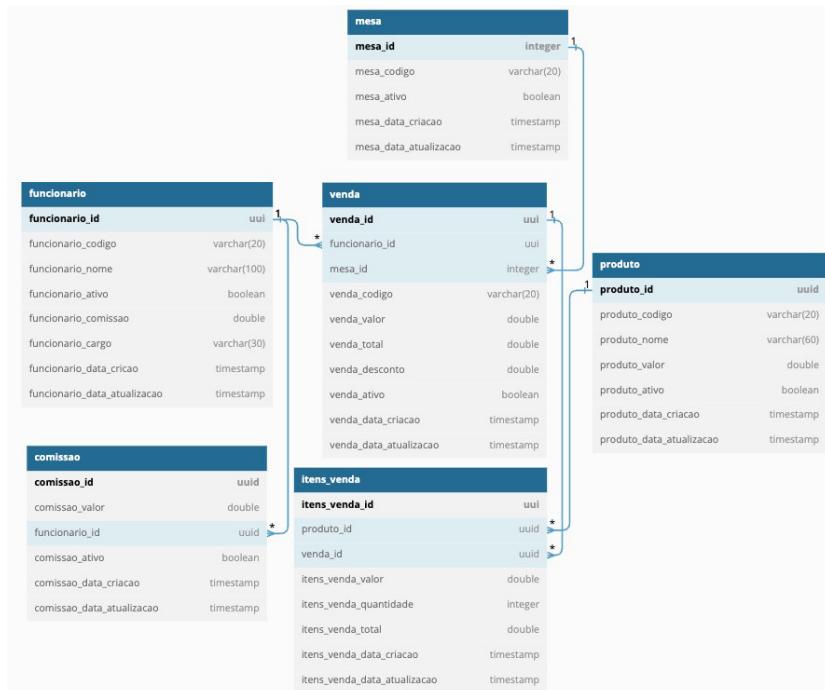


Figura 3.1: Diagrama de entidade e relacionamento.

Com o **DER** criado, podemos especificar o que cada tabela deverá armazenar.

- **MESA** : para cadastrarmos a quantidade de mesas que queremos que tenha no restaurante e seu respectivo código de identificação, pois assim podemos ampliar a quantidade de mesas em nosso restaurante apenas com um cadastro. No capítulo *Banco de dados rápido nos gatilhos*, vamos

criar um processo para gerar uma quantidade de mesas automaticamente, apenas passando por parâmetro a quantidade que quisermos.

- **VENDA** : cada pedido de uma mesa será relacionado a uma venda, que pode estar relacionada ou não a um funcionário ou a uma mesa, uma vez que podemos ter vendas que são realizadas diretamente no caixa. Incluiremos um funcionário na venda do tipo garçom para gerar sua comissão no final de cada dia.
- **ITENS\_VENDA** : cada item (produto) pedido por uma mesa será considerado um item de uma determinada venda, para que, ao concluir a venda, possamos somar todos os produtos e atualizar o total da venda. Cada item deverá ter um produto, sua quantidade e seu valor.
- **PRODUTO** : cada produto disponível para venda no restaurante deverá ser cadastrado na tabela de produtos. Vamos fazer um cadastro básico de produtos.
- **FUNCIONARIO** : essa tabela será para cadastrar os funcionários do restaurante. Nela poderemos cadastrar os garçons, gerentes, atendentes etc. Vamos distingui-los por um campo de tipo, assim poderemos incluí-los nas vendas que eles atendem e gerar a comissão. A comissão será de 10% no valor total da venda.

Com a evolução de todo projeto, o número de tabelas pode aumentar, pois é a tendência de todos os sistemas. Conforme surgir a necessidade de aumentarmos nosso sistema, criaremos novas tabelas e funcionalidades, principalmente porque temos de

criar muitos exemplos para testar as principais funcionalidades do banco de dados.

### 3.1 ENTENDENDO NOSSOS DADOS

Algo que aprendemos conforme vamos ganhando experiência em desenvolvimento de software é que, antes de começarmos a codificar, precisamos conhecer bem o negócio e o público para o qual vamos desenvolver a aplicação. Isso porque, quando estamos atrás do nosso computador, é comum não conseguirmos imaginar como os usuários vão utilizar o que estamos desenvolvendo, e a aplicação pode ficar a desejar. Por isso, devemos fazer uma especificação dos requisitos a serem desenvolvidos. É o princípio da engenharia de software.

A modelagem de dados sempre deve estar contida em sua especificação de requisitos, e sempre dê uma atenção especial para ela. Além de a experiência do usuário entender os dados do seu sistema, é muito importante entender como os dados se relacionam, o que será armazenado em cada tabela e em cada campo, entender como armazenar cada informação e como elas serão apresentadas para o usuário. Tudo isso é vital para o seu projeto.

Esses cuidados estão diretamente ligados ao desempenho e à longevidade de sua aplicação. Isso porque, se o banco estiver mal projetado, você não conseguirá escalar sua aplicação, e isso terá um custo alto no futuro.

Costumo desenhar e criar rascunhos das tabelas do projeto antes de iniciar o desenvolvimento. Além de escrever, na mão

mesmo, crio testes de mesas para entender o fluxo dos dados e ver se o resultado é o esperado. E quando crio as tabelas no banco, faço uma inserção de dados manualmente para ver se realmente o meu projeto está consistente.

Se estiver com dúvida, mostre para outros(as) desenvolvedores(as). Não tenha medo de mostrar seu código. Se tiver algo errado, será bom, pois você vai aprender e corrigirá seu erro. Sempre que tiver a oportunidade, compartilhe conhecimento e mostre seus códigos para alguém. Você só terá a ganhar.

## 3.2 A ESTRUTURA DAS TABELAS

Após a concepção, modelagem e revisão do seu projeto de banco de dados, chegou a hora de escrever os códigos para criar os objetos no seu SGBD. Volto a frisar o quanto importante é essa etapa no processo de desenvolvimento de um aplicativo. Mas o que são tabelas? E como são suas estruturas?

Segundo o escritor e especialista em banco de dados Bob Bryla, uma tabela é uma unidade de armazenamento em um banco de dados. Sem tabelas, um banco de dados não tem valor para uma empresa. Independentemente do tipo de tabela, os seus dados são armazenados em linhas e colunas, similar ao modo como os dados são armazenados em uma planilha. Mas as semelhanças terminam aí. A robustez de uma tabela de banco de dados torna uma planilha uma segunda opção ineficiente ao decidir sobre um local para armazenar informações importantes.

Imagine um banco de dados como um grande armário com várias gavetas, e cada gaveta como uma tabela. Cada gaveta

armazenará um tipo de objeto, e no caso do banco, cada tabela armazenará um tipo de informação. Se tem uma gaveta apenas para as camisetas, terá uma tabela para armazenar as informações sobre as camisetas no estoque de uma empresa. E diferentemente de uma gaveta, uma tabela vai conter  $n$  colunas, que conterão diversas informações sobre essas camisetas.

Para identificarmos e conseguirmos buscar registro de uma tabela no SGBD, temos de determinar uma identificação única para cada registro, assim como identificar o relacionamento entre as tabelas. Esses dois elementos são o que chamamos de chave primária e chave estrangeira, pois, em vez de duplicarmos uma informação em uma tabela, nós criamos tabelas com valores que não vão mudar e fazemos a referência em outras tabelas.

Vamos imaginar a nossa tabela para guardar as informações sobre as camisetas. Imagine quais informações podemos armazenar. Eu escolhi colocar em nossa tabela uma coluna que será a identificação única de nossa tabela, uma coluna para descrever a cor de nossa camiseta, uma coluna para informarmos o tamanho e uma para informar o tipo do tecido.

Em vez de repetirmos o tipo do tecido várias vezes, o melhor é criarmos uma tabela de tecidos, a qual terá todos os tipos e, em nossa tabela de camisetas, apenas fazer uma referência a esta. Veja a figura a seguir.

<b>Tabela: Gaveta</b>					
ID	COR	TAMANHO	TECIDO	ID	Descrição Tecido
1	PRETO	M	2	1	ALGODÃO
2	AZUL	P	1	2	JEANS
3	BRANCO	G	2		
4	AMARELO	GG	1		
5	VERDE	XG	1		

Figura 3.2: Identificação única da tabela e referência a outra tabela.

Com isso, já começamos a ter uma base sobre o que é chave primária, ou *primary key* (PK), e chave estrangeira, ou *foreign key* (FK). Detalharemos cada uma delas a seguir, com novos exemplos.

### 3.3 CHAVES PRIMÁRIAS E CHAVES ESTRANGEIRAS

#### Chave primária

Como já citado, o PostgreSQL é um banco de dados relacional. O princípio dos bancos de dados relacionais é o relacionamento entre uma ou mais tabelas, que é feito por meio de uma chave única, chamada de *primary key* (PK, ou *chave primária*).

Vamos exemplificar esse cenário. Vamos imaginar que uma tabela é a representação da rua na qual você mora, e sua casa e as demais são os registros da tabela. Geralmente, temos apenas uma sequência numérica, única, de casas em uma rua — ao menos, deveríamos ter. Imagine o número da casa como sendo uma "chave" única, algo que vai identificá-la das demais.

Nas tabelas do nosso banco, é exatamente isso que acontece. Temos de ter uma chave única para identificar os registros que se encontram em uma determinada tabela. Dentro desse cenário,

conseguiremos manter a integridade dos dados.

Você deve ter uma chave primária, pois será um registro que não sofrerá alteração nem se repetirá. Sempre será único e imutável. Só assim você terá a consistência de seus dados. Para alterar um registro na tabela, você deve buscá-lo por sua PK, assim não ocorrerá de alterar um registro incorreto.

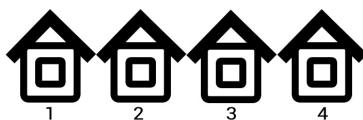


Figura 3.3: A identificação única das casas em uma rua.

Em uma tabela de banco de dados, teríamos:

Tabela: Gaveta				
ID	COR	TAMANHO	TECIDO	
1	PRETO	M	2	
2	AZUL	P	1	
3	BRANCO	G	2	
4	AMARELO	GG	1	
5	VERDE	XG	1	

pk

Figura 3.4: A identificação única dos registros em uma tabela.

## Chave estrangeira

As chaves estrangeiras (*foreign keys*, ou FK) são identificadores únicos que fazem referência à chave primária de outra tabela. Se tivermos uma FK em nossa tabela, não conseguimos inserir um registro que não esteja contido na tabela referenciada.

O exemplo a seguir mostra duas tabelas: uma de funcionários e uma de cargos. O `ID` da tabela `CARGOS`, que é uma chave primária, passa a ser uma chave estrangeira na tabela `FUNCIONARIOS`, chamada `CARGO_ID`. Ela, por sua vez, não vai permitir a inserção de nenhum cargo que não esteja cadastrado na tabela `CARGOS`, dando consistência para a sua tabela e evitando erros.

Tabela: Funcionários		
ID	NOME_FUNCIONARIO	CARGO
1	VINICIUS	1
2	DANIEL	1
3	BRUNA	2
4	THOMAS	1
5	HECTOR	1
6	VICTORIA	2

Tabela: Cargos	
ID	DESCRIÇÃO
1	DESENVOLVEDOR
2	DESIGNER

Figura 3.5: PK sendo referenciada como uma FK.

Sabendo a importância e o objetivo da utilização de PKs e FKs nas tabelas, já podemos finalmente criar nossos códigos. Então, agora, mãos no teclado e vamos começar.

### 3.4 TRABALHANDO COM PGADMIN

Podemos trabalhar com nosso banco de dados tanto utilizando o terminal de comandos quanto utilizando uma ferramenta visual como o pgAdmin. Há quem prefere o terminal de comandos e quem prefere utilizar outras ferramentas com interface. Durante o nosso projeto, vamos utilizar o pgAdmin como ferramenta padrão.

O pgAdmin é uma ferramenta open source que tem suporte para Windows, Linux e Mac OS. Você pode baixá-la em <https://www.pgadmin.org/>; lá, você pode escolher a versão para o seu sistema operacional.

Após baixar e instalar, ao abrir pela primeira vez, do lado esquerdo o pgAdmin vai exibir os servidores do PostgreSQL que estão registrados em sua máquina, como mostra a figura a seguir. No meu caso, tenho o servidor **PostgreSQL14**.

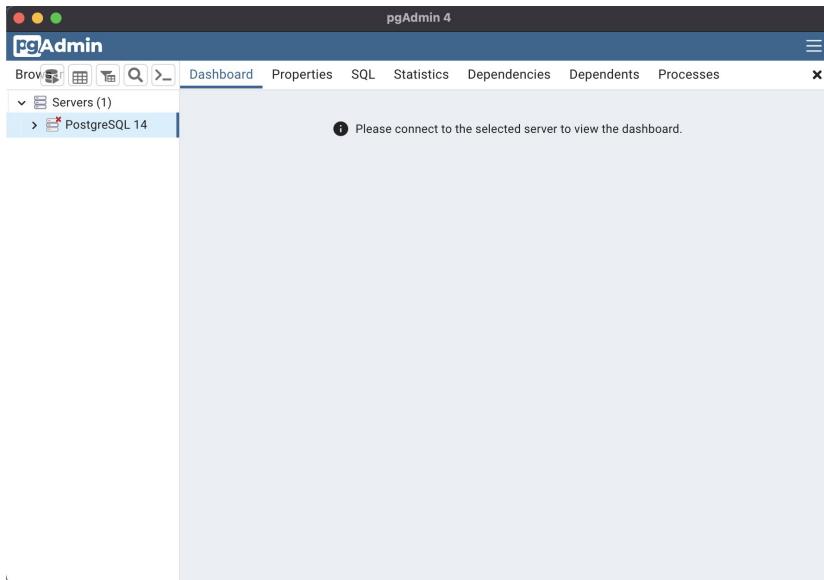


Figura 3.6: Tela inicial do pgAdmin.

Caso o pgAdmin não tenha identificado automaticamente um servidor, nós precisamos registrar um novo. Para isso, vá em `Object > Register > Server`, ou através do clique com o botão direito em `Servers > Register > Server`, e então ele abrirá a janela a seguir.

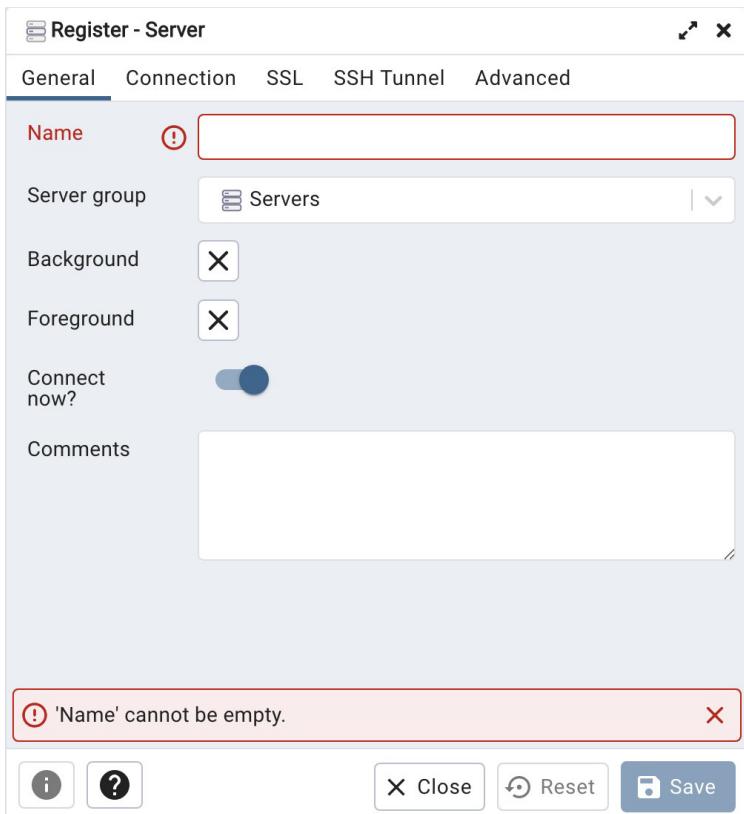


Figura 3.7: Registrando um novo servidor - General.

Nessa tela e nessa primeira aba, **General**, informe o nome do servidor em *Name*. Eu vou adicionar o nome **ServerPostgre**. Após essa etapa, clique na aba **Connection** e adicione o *Host name/address* de seu servidor. Por padrão, podemos colocar *localhost*. Precisamos apenas desses dois parâmetros para começar a utilizar o PostgreSQL.

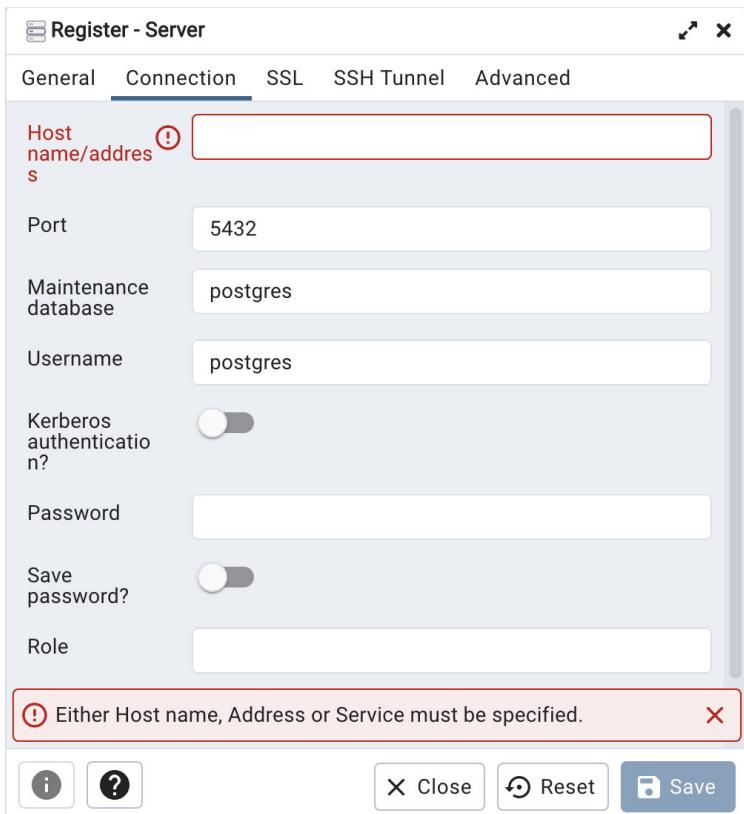


Figura 3.8: Configurando um novo servidor - Connection.

Agora você pode clicar em **Save** para salvar essas informações e voltar para a tela anterior com o seu novo servidor registrado e rodando.

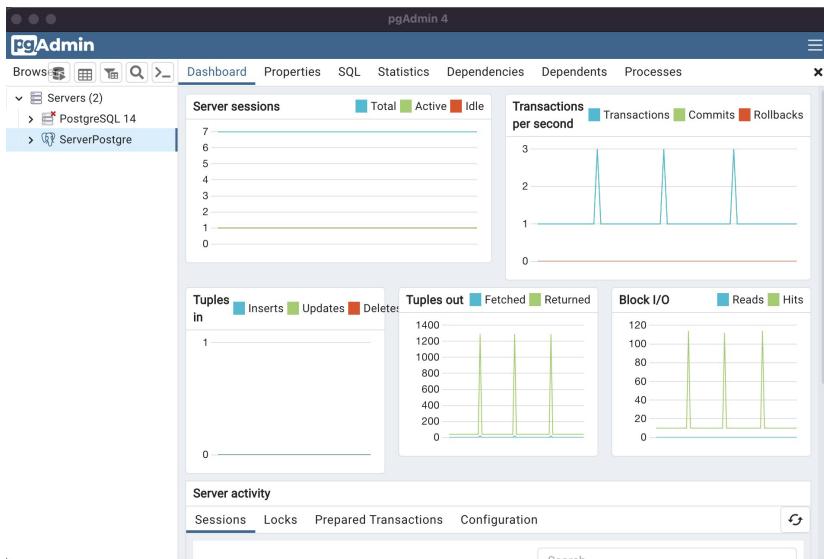


Figura 3.9: Servidor registrado.

Quando finalizamos o registro do servidor, ele se conecta automaticamente ao PostgreSQL; caso ele não se conecte automaticamente, precisamos clicar no nome do servidor (no nosso caso, `ServerPostgre`) e a ferramenta solicitará a senha que você criou para o seu banco de dados durante a instalação do PostgreSQL. Desde o começo do nosso projeto, a minha senha é `senha`. A figura a seguir mostra onde você vai inserir a sua senha. Caso você não tenha criado uma senha, basta clicar em `OK`.



Figura 3.10: Informando a senha para se conectar ao servidor.

Após estar conectado, a página inicial do pgAdmin apresenta algumas informações estatísticas do banco de dados (as quais você pode conhecer e saber mais no site da ferramenta) e, do lado esquerdo, você encontrará a estrutura dos bancos que seu servidor possui. Conforme vamos expandindo os itens, conseguimos visualizar os objetos do nosso banco de dados.

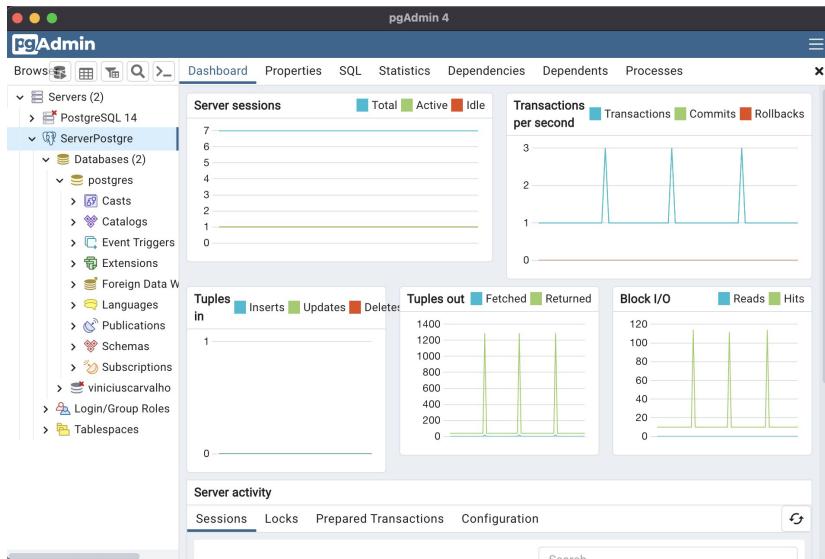


Figura 3.11: Tela inicial do pgAdmin.

Para criar um banco de dados, conforme mostra a imagem adiante, temos as seguintes opções:

- Do lado esquerdo, clicar com o botão direito em Databases > Create > Database... ; ou
- Ir até Object > Create > Database....

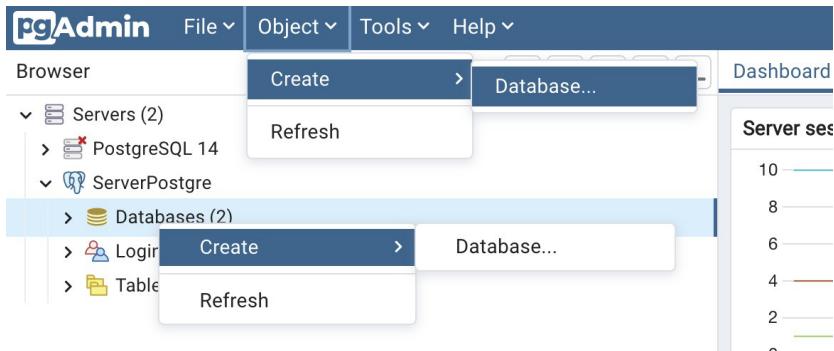


Figura 3.12: Criando um novo banco de dados pela ferramenta.

Isso abrirá uma tela de configurações do banco de dados. Vamos inserir o nome do novo banco de dados db\_restaurant no campo Database e clicar no botão Save .

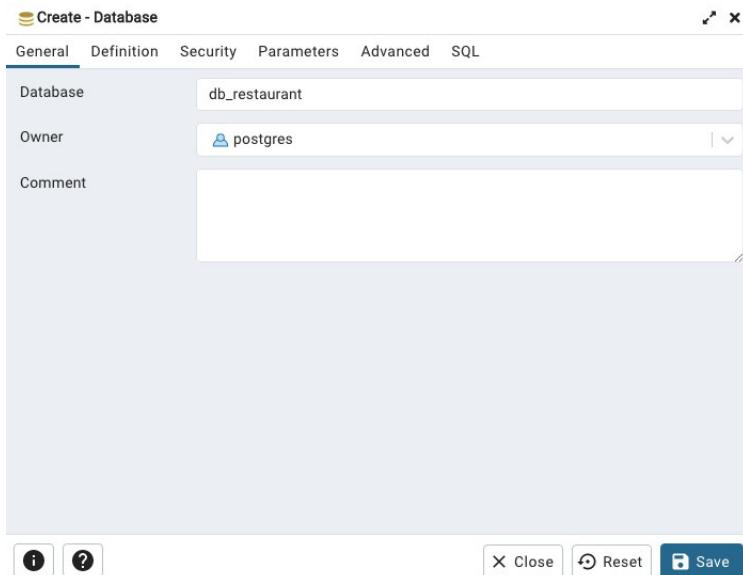


Figura 3.13: Inserindo o nome do novo banco.

Após salvar a criação do novo banco, podemos visualizá-lo na lista dos objetos do servidor.

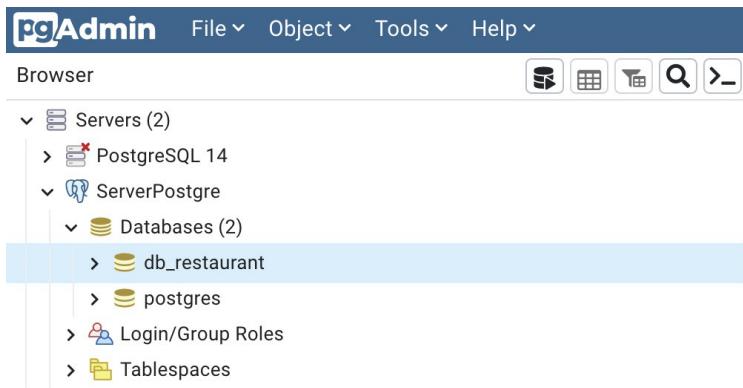


Figura 3.14: Criando um novo banco de dados pela ferramenta.

## Onde executamos os códigos?

No pgAdmin, os comandos são executados na ferramenta através da Query Tool. Para acessá-la, é preciso deixar selecionado o banco de dados que criamos e ir à opção do menu Tools > Query Tool , ou clicar com o botão direito em cima do banco de dados e ir à opção Query Tool .

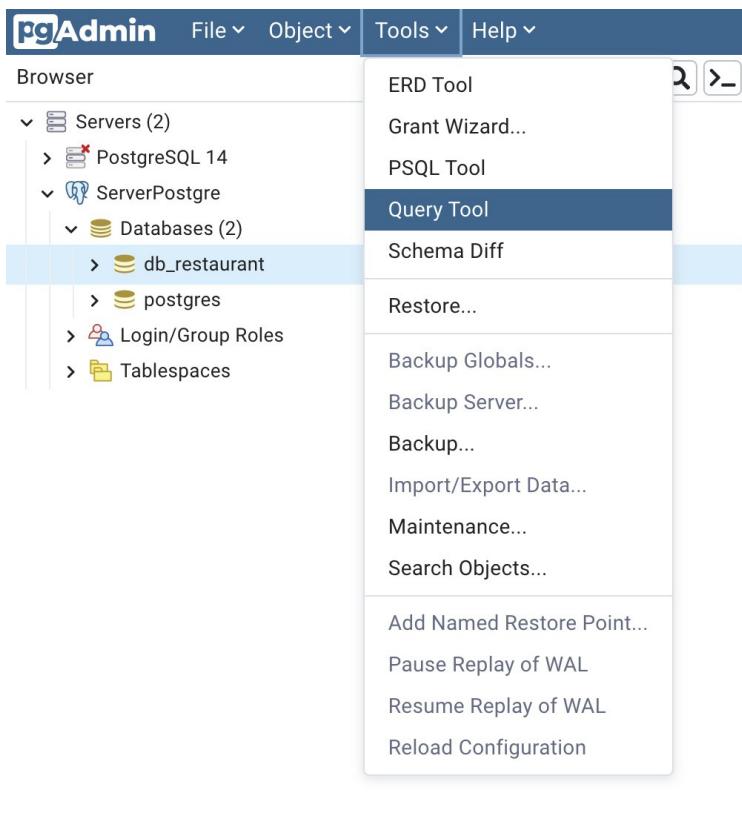


Figura 3.15: Executando o Query Tool.

Ao executar a Query Tool, abrirá um espaço no qual podemos

escrever os comandos e mandar executar, conforme mostra a figura seguinte. É nesse espaço em branco que você vai escrever o código e, para executá-lo, basta clicar no botão Execute/Refresh ou usar a tecla F5 .

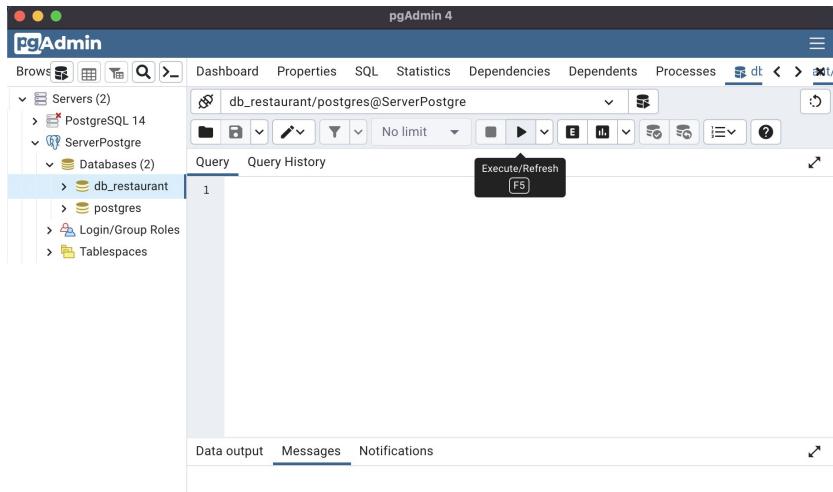


Figura 3.16: Área para executar os comandos.

Para executar o terminal de comandos pelo pgAdmin, os passos são semelhantes. Selecione o banco de dados e, depois, Tools > PSQL Tool , ou dê um clique direito no banco de dados e depois em PSQL Tool .

Muito simples, não é mesmo? Essa ferramenta facilita bastante a nossa vida. Depois que você já conhece os comandos, não precisa ficar escrevendo-os na mão. Dependendo do cenário em que você estiver trabalhando, talvez até necessite utilizar o console do SGBD, mas, em outras ocasiões, poderá usar uma ferramenta visual como o pgAdmin. Há quem prefira trabalhar apenas pelo console. Fica a seu critério.

Agora vamos começar a criação dos códigos do nosso projeto.

## 3.5 CRIANDO NOSSAS TABELAS

Abra o terminal de comandos (instalado durante a instalação do PostgreSQL) para criarmos os objetos em nosso banco.

O `create table` é o comando usado para criar tabelas. Sabendo disso, vamos em frente.

```
-- DROP TABLE IF EXISTS mesa;

CREATE TABLE IF NOT EXISTS mesa(
    mesa_id          INTEGER NOT NULL,
    mesa_codigo      VARCHAR(20),
    mesa_ativo       BOOLEAN DEFAULT true,
    mesa_data_criacao  TIMESTAMP WITHOUT TIME ZONE,
    mesa_data_atualizacao  TIMESTAMP WITHOUT TIME ZONE,
    CONSTRAINT mesa_pkey PRIMARY KEY (mesa_id));

-- DROP TABLE IF EXISTS funcionario;

CREATE TABLE IF NOT EXISTS funcionario(
    funcionario_id        UUID NOT NULL DEFAULT gen_random_
_uuid(),
    funcionario_codigo    VARCHAR(20),
    funcionario_nome      VARCHAR(100),
    funcionario_ativo     BOOLEAN DEFAULT true,
    funcionario_comissao  DOUBLE PRECISION,
    funcionario_cargo      VARCHAR(30),
    funcionario_data_criacao  TIMESTAMP WITHOUT TIME ZONE,
    funcionario_data_atualizacao  TIMESTAMP WITHOUT TIME ZONE,
    CONSTRAINT funcionario_pkey PRIMARY KEY (funcionario_id));

-- DROP TABLE IF EXISTS venda;

CREATE TABLE IF NOT EXISTS venda(
    venda_id          UUID NOT NULL,
    venda_codigo      VARCHAR(20),
    venda_valor       DOUBLE PRECISION,
    venda_total        DOUBLE PRECISION,
```

```

venda_desconto      DOUBLE PRECISION,
venda_ativo         BOOLEAN DEFAULT true,
venda_data_criacao TIMESTAMP WITHOUT TIME ZONE,
venda_data_atualizacao TIMESTAMP WITHOUT TIME ZONE,
funcionario_id      UUID,
mesa_id             INTEGER,
CONSTRAINT venda_pk PRIMARY KEY (venda_id),
CONSTRAINT venda_funcionario_fk FOREIGN KEY (funcionario_id)
    REFERENCES funcionario (funcionario_id) MATCH SIMPLE
    ON UPDATE NO ACTION
    ON DELETE NO ACTION,
CONSTRAINT venda_mesa_fk FOREIGN KEY (mesa_id)
    REFERENCES mesa (mesa_id) MATCH SIMPLE
    ON UPDATE NO ACTION
    ON DELETE NO ACTION
    NOT VALID);

-- DROP TABLE IF EXISTS produto;

CREATE TABLE IF NOT EXISTS produto(
    produto_id          UUID NOT NULL,
    produto_codigo       VARCHAR(20),
    produto_nome         VARCHAR(60),
    produto_valor        DOUBLE PRECISION,
    produto_ativo        BOOLEAN NOT NULL DEFAULT true,
    produto_data_criacao TIMESTAMP WITHOUT TIME ZONE,
    produto_data_atualizacao TIMESTAMP WITHOUT TIME ZONE,
    CONSTRAINT produto_pkey PRIMARY KEY (produto_id));

-- DROP TABLE IF EXISTS itens_venda;

CREATE TABLE IF NOT EXISTS itens_venda(
    itens_venda_id        UUID NOT NULL,
    produto_id            UUID,
    venda_id              UUID,
    itens_venda_valor     DOUBLE PRECISION,
    itens_venda_quantidade INTEGER,
    itens_venda_total      DOUBLE PRECISION,
    itens_data_criacao    TIMESTAMP WITHOUT TIME ZONE,
    itens_data_atualizacao TIMESTAMP WITHOUT TIME ZONE,
    itens_venda_comissionado BOOLEAN DEFAULT FALSE,
    CONSTRAINT itens_venda_pk PRIMARY KEY (itens_venda_id),
    CONSTRAINT itens_venda_produto_fk FOREIGN KEY (produto_id)
        REFERENCES produto (produto_id) MATCH SIMPLE
        ON UPDATE NO ACTION

```

```

        ON DELETE NO ACTION,
CONSTRAINT itens_venda_venda_fk FOREIGN KEY (venda_id)
    REFERENCES venda (venda_id) MATCH SIMPLE
ON UPDATE NO ACTION
ON DELETE NO ACTION);

-- DROP TABLE IF EXISTS comissao;

CREATE TABLE IF NOT EXISTS comissao(
    comissao_id              UUID NOT NULL,
    funcionario_id            UUID,
    comissao_valor            DOUBLE PRECISION,
    comissao_ativo            BOOLEAN DEFAULT true,
    comissao_data_criacao     TIMESTAMP WITHOUT TIME ZONE,
    comissao_data_atualizacao TIMESTAMP WITHOUT TIME ZONE,
    CONSTRAINT comissao_pk    PRIMARY KEY (comissao_id),
    CONSTRAINT comissao_funcionario_fk FOREIGN KEY (funcionario_i
d)
    REFERENCES funcionario (funcionario_id) MATCH SIMPLE
    ON UPDATE NO ACTION
    ON DELETE NO ACTION);

```

### DICA!

Em tabelas onde não terá uma exposição através de URLs, você pode optar por utilizar mecanismos de autoincremento nas chaves primárias. Assim, não precisará se preocupar em inserir e/ou criar uma forma de incrementar e não repetir a sequência. Ainda neste capítulo, vou demonstrar como criar uma *SEQUENCE* que lhe auxiliará nesse quesito. E, também, caso você utilize o tipo UUID em sua chave primária, veremos como gerará automaticamente.

Depois de criar todas as nossas tabelas, para você ver se realmente estão criadas no banco de dados, utilize o menu de

navegação da lateral esquerda do pgAdmin com seu banco de dados selecionado, db\_restaurant > Schemas > public > Tables . Isso vai abrir a lista de tabelas criadas, como mostra a figura a seguir.



Figura 3.17: Lista das tabelas criadas no banco na barra lateral esquerda.

Você também pode utilizar o comando \dt no terminal de comando, como a figura a seguir. Uma lista com as tabelas será mostrada para você:

```
psql (14.4, server 14.5)
Type "help" for help.
```

```
db_restaurant=# \dt
              List of relations
 Schema |      Name      |   Type   |  Owner
-----+-----+-----+-----+
 public | comissao    | table   | postgres
 public | funcionario | table   | postgres
 public | itens_venda | table   | postgres
 public | mesa        | table   | postgres
 public | produto     | table   | postgres
 public | venda       | table   | postgres
(6 rows)
```

Figura 3.18: \dt para listar as tabelas criadas no banco.

## 3.6 CONSTRAINTS: INTEGRIDADE DE SEUS DADOS

Os tipos de dados são uma forma para limitar quais tipos podem ser armazenados em uma tabela. Para muitas aplicações, contudo, a restrição que eles fornecem é demasiadamente grosseira. Por exemplo, uma coluna contendo preços de produtos provavelmente só pode aceitar valores positivos. Mas não há nenhum tipo de dados padrão que aceite apenas números positivos. Outra questão é que você pode querer restringir os dados de uma coluna com relação a outras colunas ou linhas, como: em uma tabela contendo informações sobre o produto deve haver apenas uma linha para cada número de produto.

Assim, o SQL permite definir restrições em colunas e tabelas. Restrições darão tanto controle sobre os campos como em suas tabelas, como você desejar. Se um usuário tentar armazenar dados em uma coluna que possa violar uma restrição, será gerado um erro. Na sequência, demonstrarei os tipos de `CONSTRAINTS` e os tipos de erros que podemos enfrentar.

As *constraints* podem ser para a validação de valores como de chave primária e estrangeira. Observe que, na criação de nossas tabelas, nós especificamos quais os campos que seriam as PKs e as FKs de cada uma.

Para visualizarmos as *constraints* de uma tabela, na barra de navegação lateral no pgAdmin, clique na seta da tabela `venda` para expandir as informações e, depois, clique na seta do item `CONSTRAINTS` para expandi-lo também. Assim, serão exibidas as *constraints* da tabela, conforme a figura a seguir.

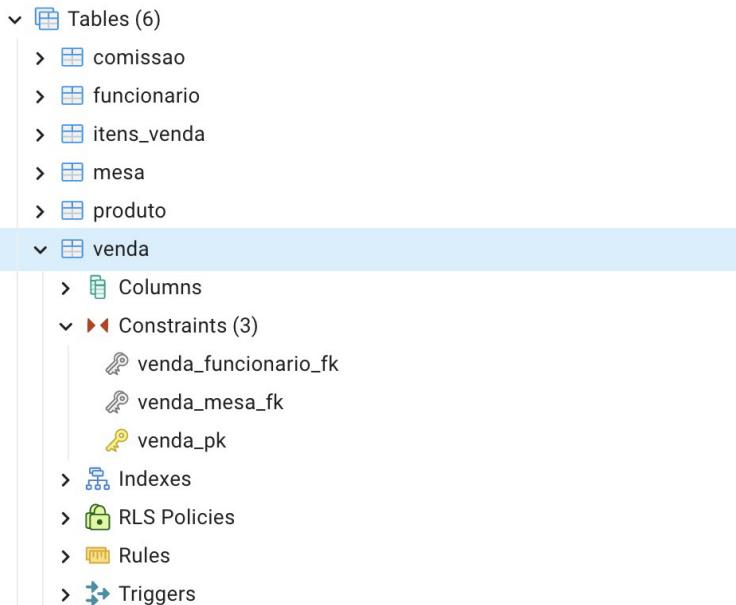


Figura 3.19: Lista de constraints da tabela venda.

Ou, no terminal, use o comando `\d nome_tabela`.

```
db_restaurant=# \d venda;
```

Observe que, além de listar as *constraints*, esse comando serve para listar os campos de uma tabela. Vamos utilizá-lo para visualizar as colunas da tabela `venda`. Como mostra a figura a seguir.

```

db_restaurant=# \d venda
              Table "public.venda"
   Column    |      Type       | Collation | Nullable | Default
---+-----+-----+-----+-----+
venda_id     | uuid          |           | not null |
venda_codigo | character varying(20) |           |           |
venda_valor  | double precision |           |           |
venda_total  | double precision |           |           |
venda_desconto | double precision |           |           |
venda_ativo  | boolean        |           |           |
venda_data_criacao | timestamp without time zone |           |           |
venda_data_atualizacao | timestamp without time zone |           |           |
funcionario_id | uuid          |           |           |
mesa_id      | integer        |           |           |
Indexes:
 "venda_pk" PRIMARY KEY, btree (venda_id)
Foreign-key constraints:
 "venda_funcionario_fk" FOREIGN KEY (funcionario_id) REFERENCES funcionario(funcionario_id)
 "venda_mesa_fk" FOREIGN KEY (mesa_id) REFERENCES mesa(mesa_id)
Referenced by:
 TABLE "itens_venda" CONSTRAINT "itens_venda_venda_fk" FOREIGN KEY (venda_id) REFERENCES venda(venda_id)

```

Figura 3.20: \d venda - listas dos campos da tabela e suas constraints.

## Constraint de PK e FK

Em vez de criarmos as *constraints* de PK e FK na criação da tabela, podemos criá-las separadamente. Vamos excluir uma tabela e criá-la novamente, só que agora criaremos as *constraints* de PK e FK posteriormente à sua criação.

Criar separado ou junto é uma questão de padrão de desenvolvimento. Como sempre falo, cada um tem o seu, o que for mais prático no dia a dia de cada desenvolvedor(a). Eu gosto de criar separado para ter um maior controle dos códigos de um projeto. Assim, consigo separar os códigos de uma determinada função em arquivos diferentes.

Para excluir uma tabela, use o comando:

```
DROP TABLE comissao;
```

Ou você pode clicar com o botão direito do mouse na tabela que deseja excluir e então em **Delete/Drop**.

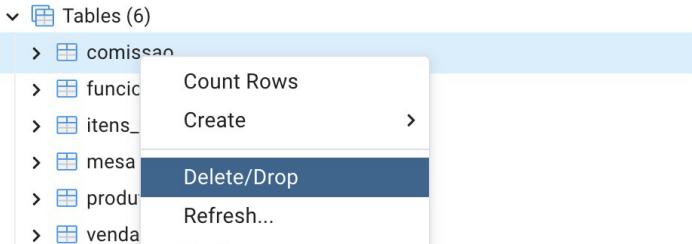


Figura 3.21: Drop table - excluindo uma tabela.

Agora crie novamente a tabela, sem as *constraints*.

```
CREATE TABLE IF NOT EXISTS comissao(
    comissao_id uuid NOT NULL,
    funcionario_id uuid,
    comissao_valor double precision,
    comissao_ativo boolean DEFAULT true,
    comissao_data_criacao timestamp without time zone,
    comissao_data_atualizacao timestamp without time zone);
```

A tabela será criada sem uma chave primária e sem chave estrangeira. Vamos criar as *constraints*. Primeiro a PK:

```
ALTER TABLE comissao ADD CONSTRAINT comissao_pk PRIMARY KEY (comissao_id);
```

Agora a FK, que referencia a tabela `funcionario` :

```
ALTER TABLE comissao
ADD CONSTRAINT comissao_funcionario_fk FOREIGN KEY (funcionario_id)
REFERENCES funcionario (funcionario_id) MATCH SIMPLE
ON UPDATE NO ACTION
ON DELETE NO ACTION;
```

Para verificarmos se tudo ocorreu bem, vamos listar os campos e as *constraints* no pgAdmin, como mostra a figura a seguir. Para visualizar os campos de uma tabela, basta expandir também o item `Columns` na navegação lateral.

The screenshot shows the Oracle SQL Developer interface with the 'Tables' node selected. Under 'Tables (6)', the 'comissao' table is expanded. It shows 6 columns: comissao\_id, funcionario\_id, comissao\_valor, comissao\_ativo, comissao\_data\_criacao, and comissao\_data\_atualizacao. It also shows 2 constraints: comissa\_funcionario\_fk (foreign key constraint) and comissao\_pk (primary key constraint). Other sections like Indexes, RLS Policies, Rules, and Triggers are listed but not expanded.

Figura 3.22: Listas dos campos da tabela e suas constraints.

Se em algum momento você precisar deletar uma *constraint*, algo que não aconselho, utilize o comando:

```
ALTER TABLE comissao DROP CONSTRAINT comissao_funcionario_fk;
```

Ou, através da ferramenta, com o clique direito na *constraint* que deseja excluir e então Delete/Drop .

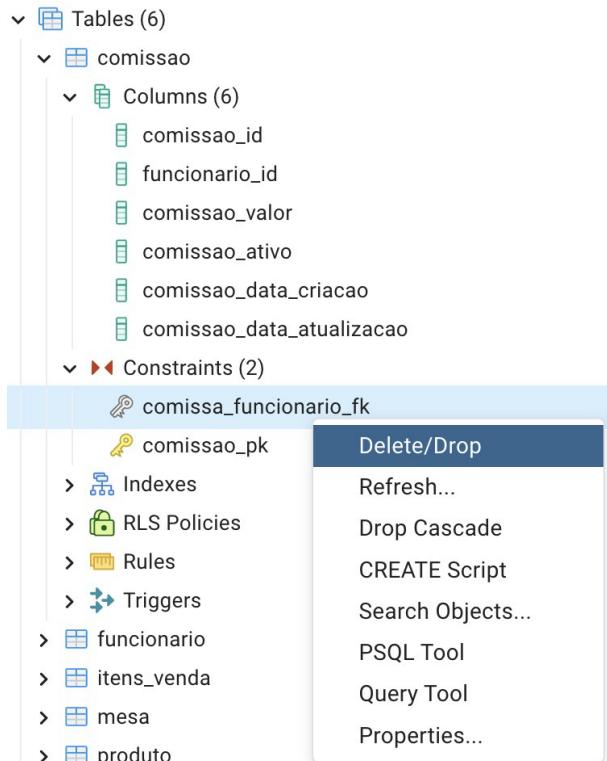


Figura 3.23: Excluindo uma constraint.

## CUIDADO!

Para você deletar uma `CONSTRAINT`, você deve ter certeza de qual é a sua finalidade e se ela não está mais sendo usada. Isso porque, na maioria das vezes, esse recurso é utilizado para fazer validações e criar maneiras de se manter a integridade dos dados.

## Constraints de validações

As *constraints* para validação de dados também são usadas para dar mais segurança para seus dados através de validações. Imagine a situação em que o valor de suas vendas está ficando negativo, algo que não pode acontecer. Como você controlará essa situação? Podemos criar uma *constraint* para validar se o valor total da venda é positivo. Assim, não será necessário criar uma validação na aplicação nem se preocupar com futuros imprevistos.

Vamos criar uma *constraint* que dirá para o nosso banco testar se o campo `venda_total` é maior que zero (positivo) a cada novo registro que estiver sendo inserido.

```
ALTER TABLE venda ADD CHECK (venda_total > 0);
```

Quando criamos nossas tabelas, em alguns campos usamos `not null`, isto é, o campo não pode ser nulo. Mas também podemos criar uma *constraint* que checará se o campo está recebendo um valor nulo na inserção de um novo registro. Sabendo disso, vamos criar uma *constraint* na tabela `funcionario` para não permitir que seja inserido nulo no campo `funcionario_nome`. Com isso, nosso banco vai verificar se tentamos inserir um valor nulo, assim como o `not null` faria.

```
ALTER TABLE funcionario ADD CHECK( funcionario_nome <> null);
```

## 3.7 CRIANDO SEQUÊNCIAS PARA AS NOSSAS TABELAS

### Para campos do tipo integer

Em todas as nossas tabelas, temos uma chave primária. Em

nosso padrão, adotei um campo chamado `nometabela_id` como padrão para as nossas chaves primárias. Na tabela `mesa`, temos uma particularidade onde a chave primária `mesa_id` é do tipo `integer` e esse campo deve conter uma sequência numérica única — nada mais lógico do que termos um mecanismo para gerar esse `mesa_id` único e automaticamente. Por isso, temos as chamadas `SEQUENCES`, que são sequências numéricas autoincrementadas.

Criando uma `SEQUENCE` para a tabela, o campo `mesa_id` vai se autoincrementar a cada inserção em cada tabela.

Vamos criar uma `SEQUENCE` utilizando o comando `CREATE SEQUENCE`. Para o nome de nossa `SEQUENCE`, usaremos o padrão `nometabela_nomecoluna_seq`. Esse padrão é como eu gosto de utilizar. Você pode (e eu aconselho) criar o seu também. Com esse padrão, o nome da `SEQUENCE` para a tabela `mesa` ficaria: `mesa_id_seq`. Então, vamos criar a `SEQUENCE` com os seguintes códigos:

```
CREATE SEQUENCE mesa_id_seq;
```

Depois da `SEQUENCE` criada, devemos vincular cada `SEQUENCE` com sua respectiva tabela, da seguinte maneira:

```
ALTER TABLE mesa ALTER column mesa_id SET DEFAULT NEXTVAL('mesa_id_seq');
```

Com todas as `SEQUENCES` criadas, não precisamos nos preocupar como vamos preencher cada campo. A vantagem da utilização das `SEQUENCES` é que, caso não quisermos mais usá-las, apenas deletamos a `SEQUENCE` e utilizamos outro meio para inserir a sequência da chave primária. No entanto, não aconselho fazer isso, pois, usando esse método de incremento numérico, tenho mais segurança e controle da função que está gerando e

gravando os números para mim.

Para deletar uma SEQUENCE , usamos o comando DROP SEQUENCE da seguinte maneira:

```
DROP SEQUENCE mesa_id_seq CASCADE;
```

Devemos utilizar o comando CASCADE no final do nosso comando para o banco também deletar o vínculo feito com a tabela mesa . Se fosse apenas uma SEQUENCE que não é utilizada em alguma tabela, não o usaríamos.

Vamos criar uma SEQUENCE que não usaremos e, depois, faremos um *drop*.

Criando a SEQUENCE :

```
CREATE SEQUENCE proximo_numero;
```

Por ela não estar vinculada com alguma tabela, o comando será:

```
DROP SEQUENCE proximo_numero;
```

## Para campos do tipo UUID

Para campos UUID, também podemos fixar um valor padrão para a chave primária ser gerada automaticamente, através do seguinte comando:

```
ALTER TABLE produto  
ALTER column produto_id SET DEFAULT gen_random_uuid();
```

Com isso, o campo produto\_id terá o mesmo funcionamento do campo mesa\_id . Ambos terão geração automática de valores únicos.

Outra alternativa seria deixar como valor padrão, como podemos observar na criação da tabela `funcionario`, onde temos

```
funcionario_id      UUID      NOT      NULL      DEFAULT  
gen_random_uuid() .
```

## Alterando tabelas

Nada mais comum do que a necessidade de fazermos alterações em nossas tabelas posteriormente à criação delas. Como alterações em uma tabela, podemos considerar: inserção de novos campos, exclusão de campos e alteração no tipo de um campo.

Vamos imaginar que surgiu a necessidade de inserirmos mais um campo na tabela `comissao`, o campo de `comissao_data_pagamento`, para informar a data em que a comissão foi ou será paga. Imagine que já começamos a inserir registros nessa tabela. Com isso, deletá-la está fora de cogitação. Mesmo que não tenhamos registros inseridos, deletar e criar a tabela novamente não é muito viável. Então, realizamos um comando para inserir um novo registro na tabela.

Vamos inserir o novo campo na tabela com o comando `ALTER TABLE...ADD COLUMN....`

```
ALTER TABLE comissao ADD COLUMN comissao_data_pagamento INT;
```

Ops! Observe que criamos um campo que será usado para armazenar data com o tipo `INT`, que é usado para armazenar números inteiros. Nessa situação, como não há ainda registros nessa nova coluna, temos duas escolhas: ou excluímos o campo e o criamos novamente, ou modificamos o seu tipo.

Para excluir uma coluna, utilizamos o comando `ALTER`

```
TABLE...DROP COLUMN .  
  
ALTER TABLE comissao DROP COLUMN comissao_data_pagamento;  
  
ALTER TABLE comissao ADD COLUMN comissao_data_pagamento TIMESTAMP  
;
```

Vamos também alterar a nossa tabela `MESA` e adicionar um tipo de campo `ENUM` para podermos indicar onde a mesa fica posicionada, se é na área externa ou interna do restaurante.

Primeiro precisamos criar o campo do tipo `ENUM`.

```
CREATE TYPE area_restaurante AS ENUM('INTERNA', 'EXTERNA');
```

E então adicionamos na tabela um novo campo `mesa_area_restaurante`, que passa a ser um campo do tipo `area_restaurante` que foi criado como `ENUM`.

```
ALTER TABLE mesa ADD COLUMN mesa_area_restaurante area_restaurant  
e;
```

## 3.8 E OS NOSSOS REGISTROS? JÁ PODEMOS INSERIR!

Com todas as nossas tabelas criadas, agora podemos começar a inserir registros em todas elas. Vamos inserir os registros que vamos utilizar em todo o livro. Fique à vontade para inserir os registros que desejar.

Na tabela de funcionário, você pode escolher nome de pessoas que você conhece. Na tabela de produtos, você pode utilizar produtos com que você tenha familiaridade ou pode utilizar os exemplos utilizados no código do livro mesmo.

## Inserindo registros

Para inserir registros, vamos utilizar o comando `INSERT INTO... VALUES...`. Usamos constantemente esse recurso de inserção manual, principalmente quando desejamos testar uma aplicação. Muitas vezes você ainda não terá a aplicação para testar se a modelagem dos dados está correta. Essa inserção funciona como um teste de mesa para a nossa modelagem. Sempre que criar um banco de dados, procure fazer a inserção manual de registros.

Vou inserir alguns registros em cada tabela que criamos.

```
INSERT INTO mesa (mesa_codigo,
    mesa_ativo,
    mesa_data_criacao,
    mesa_data_atualizacao,
    mesa_area_restaurante)
VALUES('00001',
    true,
    '01/01/2023',
    '01/01/2023'
    'INTERNA');

INSERT INTO mesa (mesa_codigo,
    mesa_ativo,
    mesa_data_criacao,
    mesa_data_atualizacao,
    mesa_area_restaurante)
VALUES('00002',
    true,
    '01/01/2023',
    '01/01/2023'
    'EXTERNA');
```

Observe que, no comando, nós suprimimos o campo `mesa_id` da tabela `mesa`. Isso porque nós criamos uma `SEQUENCE` para inserir esse valor de todas as tabelas. Vamos continuar com a inserção dos registros.

```
-- Inserindo dados na tabela funcionario

INSERT INTO funcionario(funcionario_codigo,
                        funcionario_nome,
                        funcionario_ativo,
                        funcionario_comissao,
                        funcionario_cargo,
                        funcionario_data_criacao,
                        funcionario_data_atualizacao)
VALUES('0001',
       'VINICIUS CARVALHO',
       true,
       5,
       'GERENTE',
       '01/01/2023',
       '01/01/2023');

INSERT INTO funcionario(funcionario_codigo,
                        funcionario_nome,
                        funcionario_ativo,
                        funcionario_comissao,
                        funcionario_cargo,
                        funcionario_data_criacao,
                        funcionario_data_atualizacao)
VALUES('0002',
       'SOUZA',
       true,
       2,
       'GARÇOM',
       '01/01/2023',
       '01/01/2023');

-- Inserindo dados na tabela produto

INSERT INTO produto(produto_codigo,
                    produto_nome,
                    produto_valor,
                    produto_ativo,
                    produto_data_criacao,
                    produto_data_atualizacao)
VALUES('001',
       'REFRIGERANTE',
       10,
       true,
```

```
'01/01/2023',
'01/01/2023');

INSERT INTO produto(produto_codigo,
                    produto_nome,
                    produto_valor,
                    produto_ativo,
                    produto_data_criacao,
                    produto_data_atualizacao)
VALUES('002',
       'AGUA',
       3,
       true,
       '01/01/2023',
       '01/01/2023');

INSERT INTO produto(produto_codigo,
                    produto_nome,
                    produto_valor,
                    produto_ativo,
                    produto_data_criacao,
                    produto_data_atualizacao)
VALUES('003',
       'PASTEL',
       7,
       true,
       '01/01/2023',
       '01/01/2023');
```

Não vamos inserir registro na tabela `comissao`, pois nela criaremos um processo que vai populá-la automaticamente. Por se tratar de uma tabela que vai gerar as comissões dos funcionários, o melhor é criar um processo que faça isso sozinho. No capítulo seguinte, vamos trabalhar com esse processo.

Também não vamos inserir neste momento os registros nas tabelas `venda` e `itens_venda`, pois note que nelas temos chaves estrangeiras que não conhecemos. Por exemplo, na tabela `venda`, temos os campos `funcionario_id` e `mesa_id` que não conhecemos, já que foram gerados automaticamente. No próximo

tópico, vamos aprender como consultar e buscar esses dados.

Observação: Note que estamos utilizando a função `gen_random_uuid()` no lugar do valor para as chaves primárias das tabelas que possuem chave primária do tipo UUID e não possuem como valor `DEFAULT` a geração do código UUID.

### 3.9 CONSULTANDO NOSSOS REGISTROS

Com registros em nosso banco, o mais lógico é criarmos consultas para visualizá-los. O comando para criarmos consultas é o `SELECT... FROM...`.

Vamos consultar os registros da tabela `mesa`. Para isso, usaremos o `SELECT` e o `FROM`, responsáveis por dizer para o banco de dados qual é tabela que desejamos consultar. Nossa comando e o resultado ficarão da seguinte maneira:

```
SELECT * FROM mesa;
```

The screenshot shows a PostgreSQL terminal window. At the top, there are tabs for 'Query' and 'Query History'. Below the tabs, the command `SELECT * FROM mesa;` is entered. Underneath the command, the results are displayed in a table format. The table has a header row with column names: `mesa_id [PK] integer`, `mesa_codigo character varying (20)`, `mesa_ativo boolean`, `mesa_data_criacao timestamp without time zone`, `mesa_data_atualizacao timestamp without time zone`, and `mesa_area_restaurante area_restaurante`. There are two data rows: Row 1 has values 1, 00001, true, 2023-01-01 00:00:00, 2023-01-01 00:00:00, and INTERNA; Row 2 has values 2, 00002, true, 2023-01-01 00:00:00, 2023-01-01 00:00:00, and EXTERNA. At the bottom of the terminal, it says 'Total rows: 2 of 2' and 'Query complete 00:00:00.042'.

	mesa_id [PK] integer	mesa_codigo character varying (20)	mesa_ativo boolean	mesa_data_criacao timestamp without time zone	mesa_data_atualizacao timestamp without time zone	mesa_area_restaurante area_restaurante
1	1	00001	true	2023-01-01 00:00:00	2023-01-01 00:00:00	INTERNA
2	2	00002	true	2023-01-01 00:00:00	2023-01-01 00:00:00	EXTERNA

Figura 3.24: Selecionando todas as mesas.

Observe que não foi necessário informar nenhum campo em nosso comando para fazer a consulta, apenas usamos o \* (asterisco); assim, o banco entendeu que era para buscar todos os campos da tabela informada logo após o FROM . Mas se desejarmos selecionar apenas algumas colunas, o nosso comando ficaria da seguinte maneira:

```
SELECT mesa_codigo, mesa_data_criacao FROM mesa;
```

Nessas duas consultas, o resultado foi todos os registros, porém podemos ter a necessidade de buscar apenas um ou alguns registros. Para fazermos isso, devemos informar qual registro queremos buscar, o que é feito com o comando where . Vamos então consultar a mesa que possui o mesa\_codigo igual a dois.

```
SELECT * FROM mesa WHERE mesa_codigo = '00002';
```

Observe que o código 00002 foi colocado entre aspas simples. Isso porque, para fazer comparação de strings, devemos deixar entre aspas simples, tanto nas consultas como nas inserções e alteração de dados. Observe na inserção dos registros que campos de caracteres e datas estão com aspas simples, e os numéricos não estão.

Agora que sabemos consultar, vamos buscar os registros da tabela funcionario e mesa para inserir os registros de venda . Primeiro, vamos consultar os funcionários:

```
SELECT funcionario_id, funcionario_nome FROM funcionario;
```

Com o resultado, como mostra a figura a seguir, copie e cole os valores do campo funcionario\_id , pois vamos utilizá-los.

The screenshot shows a PostgreSQL query interface. At the top, there are tabs for 'Query' and 'Query History'. Below the tabs, a code editor contains the following SQL query:

```

1  SELECT funcionario_id, funcionario_nome FROM funcionario;
2

```

Below the code editor is a toolbar with icons for file operations like new, open, save, and refresh. Underneath the toolbar is a table preview area. The table has two columns: 'funcionario\_id' and 'funcionario\_nome'. The data is as follows:

funcionario_id	funcionario_nome
[PK] uuid 4810ff6d-2bff-4b2d-8789-893bc9e50d32	VINICIUS CARVALHO
d44458ed-877d-4251-9a53-24ef06a248c5	SOUZA

Figura 3.25: Selecionando todos os funcionários.

E, agora, vamos buscar as mesas.

```
SELECT * FROM mesa;
```

The screenshot shows a PostgreSQL query interface. At the top, there are tabs for 'Query' and 'Query History'. Below the tabs, a code editor contains the following SQL query:

```

1  SELECT * FROM mesa;

```

Below the code editor is a toolbar with icons for file operations like new, open, save, and refresh. Underneath the toolbar is a table preview area. The table has seven columns: 'mesa\_id', 'mesa\_codigo', 'mesa\_ativo', 'mesa\_data\_criacao', 'mesa\_data\_atualizacao', 'mesa\_area\_restaurante', and 'area\_restaurante'. The data is as follows:

mesa_id	mesa_codigo	mesa_ativo	mesa_data_criacao	mesa_data_atualizacao	mesa_area_restaurante	area_restaurante
1	00001	true	2023-01-01 00:00:00	2023-01-01 00:00:00	INTERNA	
2	00002	true	2023-01-01 00:00:00	2023-01-01 00:00:00	EXTERNA	

Figura 3.26: Selecionando todas as mesas.

Agora temos todos os dados necessários para montar o `INSERT INTO venda . . .`. Crie uma venda para cada funcionário e utilize uma mesa em cada inserção. Com isso, nosso código ficará assim:

```
-- Inserindo dados na tabela venda

insert into venda(venda_id,
                  venda_codigo,
```

```

venda_valor,
venda_total,
venda_desconto,
venda_ativo,
venda_data_criacao,
venda_data_atualizacao,
funcionario_id,
mesa_id)
values(gen_random_uuid(),
'0001',
'20',
'20',
'0',
true,
'01/01/2023',
'01/01/2023',
'4810ff6d-2bff-4b2d-8789-893bc9e50d32',
1);

insert into venda(venda_id,
venda_codigo,
venda_valor,
venda_total,
venda_desconto,
venda_ativo,
venda_data_criacao,
venda_data_atualizacao,
funcionario_id,
mesa_id)
values(gen_random_uuid(),
'0001',
'20',
'20',
'0',
true,
'01/01/2023',
'01/01/2023',
'd44458ed-877d-4251-9a53-24ef06a248c5',
2);

```

Agora que temos as vendas, podemos inserir os registros da tabela `itens_venda`. Primeiro, consultamos todas as vendas.

```
SELECT * FROM venda;
```

Como resultado, temos:

The screenshot shows a PostgreSQL query interface. At the top, there are tabs for 'Query' (which is selected) and 'Query History'. Below that is a code editor containing the SQL command: '1 SELECT \* FROM venda;'. Underneath the code editor are tabs for 'Data output', 'Messages', and 'Notifications'. Below these tabs is a toolbar with several icons: a plus sign, a file icon, a downward arrow, a clipboard icon, a trash can icon, a database icon, a download icon, and a refresh icon. The main area displays the results of the query in a table format. The table has two columns: 'venda\_id' and 'venda\_codigo'. There are two rows of data: Row 1 has 'venda\_id' as 'fa9010d8-ef7e-4191-bb4b-7e8d4ac501cd' and 'venda\_codigo' as '0001'; Row 2 has 'venda\_id' as 'da80d450-235a-4c02-af3b-e3adc43cd14f' and 'venda\_codigo' as '0002'. The 'venda\_id' column is marked as a primary key (PK) and is of type 'uuid'. The 'venda\_codigo' column is of type 'character varying (20)'.

	venda_id [PK] uuid	venda_codigo character varying (20)
1	fa9010d8-ef7e-4191-bb4b-7e8d4ac501cd	0001
2	da80d450-235a-4c02-af3b-e3adc43cd14f	0002

Figura 3.27: Selecionando todos as vendas.

Temos esses dois registros. Vale lembrar que, em seu banco de dados, o `venda_id` será diferente, pois é um código gerado automaticamente. Salve esse código, pois antes precisamos buscar os códigos dos produtos.

```
SELECT produto_id, produto_nome FROM produto;
```

Como resultado, temos os itens, como mostra a figura a seguir. Nós vamos utilizar apenas os produtos 'REFRIGERANTE' e 'AGUA' para inserir na tabela `itens_venda` e relacioná-los a uma venda.

The screenshot shows a PostgreSQL query editor interface. At the top, there are tabs for 'Query' and 'Query History'. Below the tabs, a code editor contains the following SQL query:

```
1  SELECT produto_id, produto_nome FROM produto;
```

Below the code editor is a toolbar with various icons. Underneath the toolbar is a table showing the results of the query. The table has two columns: 'produto\_id' and 'produto\_nome'. The data is as follows:

	produto_id	produto_nome
1	e8f1f733-09ff-4992-9b60-e0321266b894	REFRIGERANTE
2	86b46f1e-5056-4562-bffc-85544cc6ea9b	AGUA
3	cae592dd-1a7d-4fdc-bc4e-0b793fb833fa6	PASTEL

Figura 3.28: Selecionando todos os produtos.

Assim, os nossos códigos ficarão da seguinte maneira:

```
-- Utilizando o produto REFRIGERANTE e primeira venda

insert into itens_venda(itens_venda_id,
                      produto_id,
                      venda_id,
                      itens_venda_valor,
                      itens_venda_quantidade,
                      itens_venda_total,
                      itens_venda_comissionado,
                      itens_venda_data_criacao,
                      itens_venda_data_atualizacao)
values(gen_random_uuid(),
      'e8f1f733-09ff-4992-9b60-e0321266b894',
      'fa9010d8-ef7e-4191-bb4b-7e8d4ac501cd',
      10,
      2,
      20,
      '01/01/2023',
      '01/01/2023');
```

```
-- Utilizando o produto AGUA e segunda venda
```

```
insert into itens_venda(itens_venda_id,
                      produto_id,
                      venda_id,
```

```
    itens_venda_valor,
    itens_venda_quantidade,
    itens_venda_total,
    itens_venda_comissionado,
    itens_venda_data_criacao,
    itens_venda_data_atualizacao)
values(gen_random_uuid(),
       '86b46f1e-5056-4562-bffc-85544cc6ea9b',
       'da80d450-235a-4c02-af3b-e3adc43cd14f',
       7,
       3,
       21,
       '01/01/2023',
       '01/01/2023');
```

## Atualizando registros

Fizemos a inserção manual dos registros para conseguirmos trabalhar durante o desenvolvimento do projeto que estamos criando no livro. Se desejarmos apenas modificar um registro em uma tabela em vez de inserir um novo e excluir algum outro, usamos o comando `UPDATE`.

Como já sabemos qual o `produto_id` do produto AGUA , conforme a imagem anterior, aqui no meu banco, quando consultamos todos os produtos, o código é `86b46f1e-5056-4562-bffc-85544cc6ea9b` (no seu banco de dados, esse código será diferente). Vamos alterar o valor para ele. Agora ele passará a custar 4. Então podemos fazer a atualização de valor.

```
UPDATE produto SET produto_valor = 4
WHERE produto_id = '86b46f1e-5056-4562-bffc-85544cc6ea9b';
```

Observe que utilizamos também o comando `WHERE` para indicar qual o produto que queríamos fazer a alteração. Se não usássemos o `WHERE` no comando, seria atualizado o campo `produto_valor` para todos os registros da tabela. Vamos fazer

uma alteração sem `WHERE` , e atualizar o `produto_data_criacao` de todos os produtos.

```
UPDATE produto SET produto_data_criacao = '02/10/2023';
```

Utilize o `SELECT` para visualizar todas as datas alteradas na tabela `PRODUTOS` .

```
SELECT produto_data_criacao FROM produto;
```

## Excluindo registros

Inserimos, consultamos e alteramos nossos registros. Nada mais normal haver situações nas quais será necessária a exclusão de registros. Para isso, temos o comando `DELETE` .

Vamos excluir uma mesa que não estamos usando atualmente.

```
DELETE FROM mesa WHERE mesa_id = 2;
```

Muita atenção ao realizar comandos `DELETE` no PostgreSQL, pois você não vai recuperar o seu registro perdido. Se não colocar o `WHERE` no comando de `DELETE` , você está enviando a informação para deletar todos os registros da tabela.

Em nosso comando, deixei especificado que era para deletar apenas o registro com `mesa_id = 2` . O `DELETE` sem `WHERE` é muito famoso por pregar peças em desenvolvedores(as) desatentos(as).

## 3.10 PARA PENSAR!

Observe que apenas inserimos poucos registros em nossas tabelas. Para colocar em prática o que vimos neste capítulo, insira

mais registros nelas. Insira pelo menos mais três em cada tabela.

Tente escrever os códigos. Depois que você tiver aprendido e fixado como utilizar cada comando, então poderá usar o `CTRL+C` e `CTRL+V`.

Temos as tabelas e sabemos como alterar suas estruturas. Temos registros, sabemos manipulá-los, inseri-los e alterá-los. Já temos o básico para conseguirmos trabalhar com o PostgreSQL. Já podemos partir para assuntos mais complexos. Treine um pouco o que já aprendemos e *#PartiuPróximoCapítulo*.

## CAPÍTULO 4

# FUNCTIONS — AGILIZANDO O DIA A DIA

*"Você não precisa ser um gênio ou visionário — nem mesmo graduado em uma faculdade sobre qualquer assunto — para ser bem-sucedido. Você precisa apenas de estrutura e um sonho."* — Michael Dell

## 4.1 FUNCTIONS PARA POUPAR ESFORÇOS

Funções são um conjunto de procedimentos escritos em SQL que são armazenados no banco de dados a fim de executar uma determinada função. Para mim, *functions* são procedimentos armazenados no banco de dados que servem para agilizar o dia a dia e otimizar seus códigos. Nestas podemos escrever instruções para realizar operações como: consultar e retornar valores, realizar cálculos e retornar ou não valores, chamar outros procedimentos, entre outras.

É comum no cotidiano da pessoa desenvolvedora criar algumas consultas para retornar dados básicos de uma tabela. Por exemplo, se eu preciso consultar o nome de um funcionário e eu só tenho o seu `funcionario_id`, eu crio uma `FUNCTION` na qual passo o `funcionario_id` como parâmetro e, como retorno, tenho o

nome do funcionário e qualquer outra informação que desejo. Eu posso utilizar essa FUNCTION em qualquer processo que desejar ou em consultas mais complexas. Vamos exemplificar para ficar mais claro.

Criaremos uma FUNCTION que retorne o nome do funcionário, concatenando o campo `funcionario_ativo`. Para isso, vamos levar em consideração a seguinte tabela de possíveis valores para o campo `funcionario_ativo`.

Sigla	Descrição
True	Ativo
False	Inativo

Em nossa FUNCTION, passaremos como parâmetro o `funcionario_id` e verificaremos a situação. Em seguida, vamos concatenar a descrição da situação com o seu nome.

No código, vamos passar como parâmetro o `funcionario_id` do funcionário e, depois, teremos uma consulta para buscar o funcionário; como retorno, teremos o nome e a situação dele. Na sequência, verificamos o tipo da situação e fazemos a concatenação do nome e da descrição da situação. Vamos ao código.

```
CREATE OR REPLACE FUNCTION
retorna_nome_funcionario(func_id uuid)
RETURNS TEXT AS
$$
DECLARE
    nome TEXT;
    ativo BOOLEAN;
BEGIN
    SELECT funcionario_nome,
        funcionario_ativo
```

```
        INTO nome, ativo
        FROM funcionario
        WHERE funcionario_id = func_id;

        IF ativo = true THEN
            RETURN nome || ' - Funcionario Ativo';
        ELSE
            RETURN nome || ' - Funcionario Inativo';
        END IF;
    END
$$
LANGUAGE PLPGSQL;
```

Em nosso código, temos alguns símbolos diferentes dos quais já vimos. Não se preocupe, pois agora na sequência vou explicá-los.

## Cifrão duplo (\$\$)

Você percebeu algo diferente em nosso código? Os \$\$ são usados para limitar o corpo da função e para o banco de dados entender que tudo o que está dentro dos limites do cifrão duplo é código de uma única função.

Cifrão não é parte do padrão SQL, mas muitas vezes é uma forma mais conveniente para escrever strings literais complicadas do que a sintaxe simples compatível com o padrão SQL. Observe que, além de nosso código ser grande, tivemos de quebrar em algumas linhas para conseguirmos entendê-lo. O cifrão está dizendo para o banco que todo o código contido entre eles pertence ao mesmo código.

Se pesquisar na internet exemplos de FUNCTIONS , você pode encontrar o seguinte:

```
$palavra_qualquer$
BEGIN
```

```
Instruções da FUNCTION();  
  
END  
$palavra_qualquer$
```

Observe que, em vez de utilizar apenas `$$`, escrevemos `$palavra_qualquer$`. O cifrão é apenas um limitador e, por isso, você pode usar uma outra palavra para limitar seu código.

## DECLARE

Em processos de `FUNCTION`, temos a necessidade de utilizar variáveis para armazenar informações temporariamente. Por isso, precisamos fazer a declaração das variáveis, colocando o seu nome e o seu tipo. Todas as variáveis que você precisar utilizar deverão ser declaradas logo abaixo do `DECLARE`.

Em nosso código, nós fizemos a declaração da seguinte maneira:

```
DECLARE  
    nome TEXT;  
    ativo BOOLEAN;
```

## Language PLPGSQL

No final do nosso bloco de instruções, devemos colocar `LANGUAGE PLPGSQL`, que é a linguagem que usamos para escrever nossa `FUNCTION`. Ela, linguagem utilizada pelo PostgreSQL, está para o PostgreSQL, assim como o PL/SQL está para o Oracle.

Nós colocamos no final do código, pois temos de informar que estamos usando a linguagem `PLPGSQL`. Isso porque, no PostgreSQL, podemos utilizar outra linguagem de programação,

como a linguagem C. No entanto, esse assunto não será abordado neste livro, por se tratar de algo que não é muito utilizado no mercado.

## IF e ELSE

Em nossa FUNCTION , também utilizamos as declarações condicionais IF... THEN... ELSE . Se você ainda não está familiarizado(a) com alguma linguagem de programação, nós usamos essas condições para testar uma condição e verificar se é verdadeira.

Em nosso exemplo, testamos se o funcionario\_ativo do funcionário era igual a true , e escrevemos a instrução para que, se ele fosse igual, concatenasse as palavras Funcionario Ativo com o seu nome. Caso contrário ( ELSE ), ele concatenaria as palavras Funcionario Inativo .

Há a possibilidade de testarmos quantas condições quisermos. Basta utilizar as outras declarações do IF . Vamos pegar apenas o corpo de nossa FUNCTION e reescrevê-lo colocando outras condições.

Vamos testar se o funcionario\_ativo é igual a true , false , vazio ou diferente das três condições.

```
$$  
BEGIN  
  
  IF situacao = true THEN  
    'Funcionario Ativo';  
  
  ELSIF situacao = false THEN
```

```
'Funcionario Inativo'

ELSIF situacao IS NULL THEN

    'Funcionario Sem status'

ELSE

    'Funcionario com status diferente de true e false'

END IF;

END
$$
```

Veja que, após o primeiro `IF`, usamos `ELSIF` para verificar a condição seguinte. Só quando eu não quero mais verificar nenhuma condição, eu utilizo o `ELSE`.

Sempre que concluir a declaração de um `IF`, não esqueça de escrever a sua finalização, o `END IF`. Só assim o PostgreSQL vai entender que você está finalizando aquele bloco condicional.

## 4.2 UTILIZANDO A FUNCTION

Após termos criado e entendido como criar uma function, vamos aprender como usá-la. A function criada, a `retorna_nome_funcionario`, tem um retorno, certo? Sim, pois a criamos para buscar o nome de um funcionário. E como ela possui um retorno, que é o nome do funcionário e sua situação, devemos utilizá-la em uma consulta. Portanto, vamos criar uma consulta.

Como parâmetro, devemos passar o `funcionario_id` do funcionário. Sabendo disso, vamos passar como parâmetro o `funcionario_id = '4810ff6d-2bff-4b2d-8789-893bc9e50d32'`. Vale lembrar que, por esse código ser único e

gerado automaticamente, você precisa primeiro fazer uma consulta para buscar o código do UUID no seu banco de dados, pois será diferente do código que estou utilizando.

```
postgresql=> SELECT retorna_nome_funcionario('4810ff6d-2bff-4b2d-8789-893bc9e50d32');
```

O resultado da nossa consulta será:

The screenshot shows the pgAdmin 4 interface. At the top, there's a toolbar with various icons. Below it, the connection bar shows 'db\_restaurant/postgres@ServerPostgre'. Underneath the toolbar is a menu bar with 'Query' and 'Query History' selected. The main area contains a code editor with the following SQL query:

```
1  SELECT retorna_nome_funcionario('4810ff6d-2bff-4b2d-8789-893bc9e50d32');
```

Below the code editor, there are tabs for 'Data output', 'Messages', and 'Notifications'. The 'Data output' tab is active and displays the result of the query:

	retorna_nome_funcionario
1	VINICIUS CARVALHO - Funcionario Ativo

Figura 4.1: Resultado da função retorna\_nome\_funcionario.

Com o tempo, você vai criando **FUNCTIONS** automaticamente e conseguindo enxergar onde poderá utilizá-las. É muito comum no começo, quando estamos aprendendo alguma linguagem ou tecnologia, não conseguirmos saber onde e quando usar. Mas isso só a prática lhe dirá.

Para fixar ainda mais, vamos criar outra **FUNCTION** que vai retornar o valor da porcentagem de comissão de cada funcionário, pois vamos utilizá-la mais à frente para criar um processo que calculará um valor de comissão de cada venda.

Novamente vamos passar como parâmetro o `funcionario_id` do funcionário e, como retorno, pegaremos o valor do campo `funcionario_comissao`.

```
CREATE OR REPLACE FUNCTION
rt_valor_comissao(func_id UUID)
RETURNS DOUBLE PRECISION AS
$$
DECLARE
    valor_comissao DOUBLE PRECISION;
BEGIN
    SELECT funcionario_comissao
        INTO valor_comissao
        FROM funcionario
       WHERE funcionario_id = func_id;

    RETURN valor_comissao;
END
$$
LANGUAGE PLPGSQL;
```

Vamos testar nossa FUNCTION .

```
select rt_valor_comissao('4810ff6d-2bff-4b2d-8789-893bc9e50d32');
```

Como resultado, temos:

The screenshot shows the pgAdmin 4 interface. At the top, there's a toolbar with various icons for database management. Below it is a menu bar with 'Query' and 'Query History'. The main area contains a single line of SQL code: 'SELECT rt\_valor\_comissao('4810ff6d-2bff-4b2d-8789-893bc9e50d32');'. Underneath the code, the results pane shows a table with one row. The table has a single column labeled 'rt\_valor\_comissao' with the type 'double precision'. The value in the first row is '5'. Below the results are tabs for 'Data output', 'Messages', and 'Notifications', along with a toolbar for managing the results.

	rt_valor_comissao
1	5

Figura 4.2: Resultado da função rt\_valor\_comissao.

## 4.3 FUNCTIONS SEM RETURN

No PostgreSQL, temos as functions que possuem algum retorno e podemos utilizá-lo por meio das consultas. Temos também as que não possuem retorno. Elas são usadas para executar determinados processos internamente no banco de dados.

Usamos esse tipo de function quando desejamos realizar um processamento que envolve vários registros. Para não precisarmos executar cada consulta ou processo isoladamente, colocamos várias instruções dentro da function e falamos para o banco de dados executar essas instruções quando necessário.

Vamos criar agora uma FUNCTION que fará o cálculo de comissionamento de todos os funcionários das vendas que tiveram em algum período. Aproveite esse exemplo para exercitar a inserção de valores nas tabelas e popule a tabela vendas e itens\_venda , para que você tenha vários registros para trabalhar,

além dos que inserimos anteriormente. Agora, mãos no teclado e vamos à nossa FUNCTION .

Para esse cenário, vamos supor que a sua empresa realiza o cálculo de comissionamento levando em consideração um período de vendas. Nele vamos passar por parâmetro uma data inicial e uma final, para o nosso processo buscar todas as vendas que foram feitas nesse intervalo.

A nossa consulta também vai levar em consideração apenas as vendas cujo campo venda\_comissionado seja igual a false . Isso porque, após calcular a comissão da venda, vamos fazer um update nesse campo para true , para indicar que a venda já foi comissionada, e não corra o risco de ser comissionada novamente. O nosso processo vai pegar todas as vendas que foram realizadas e todos os funcionários, calcular suas comissões e inserir na tabela comissao .

```
CREATE OR REPLACE FUNCTION
calc_comissao(data_ini TIMESTAMP,
               data_fim TIMESTAMP)
RETURNS VOID AS $$

DECLARE

-- Declaração das variáveis que vamos
-- utilizar. Já na declaração elas
-- recebem o valor zero, pois assim
-- garanto que elas estarão zeradas
-- quando for utilizá-las

    total_comissao  DOUBLE PRECISION := 0;
    porc_comissao   DOUBLE PRECISION := 0;

-- Declarando uma variável para armazenar
-- os registros dos loops
    reg RECORD;

-- Cursor para buscar a % de comissão do funcionário
```

```

cr_porce CURSOR (func_id UUID) IS
    SELECT rt_valor_comissao(func_id);

BEGIN

-- Realiza um loop e busca todas as vendas
-- no período informado

FOR reg IN(
    SELECT venda_id,
        funcionario_id,
        venda_total
    FROM venda
    WHERE venda_data_criacao  >= data_ini
        AND venda_data_criacao <= data_fim
        AND venda_comissionado = false)LOOP

-- Abertura, utilização e fechamento do cursor

OPEN cr_porce(reg.funcionario_id);
FETCH cr_porce INTO porc_comissao;
CLOSE cr_porce;

total_comissao := (reg.venda_total *
porc_comissao)/100;

-- Insere na tabela de comissões o valor
-- que o funcionário vai receber de comissão
-- daquela venda

INSERT INTO comissao( comissao_id,
    funcionario_id,
    comissao_valor,
    comissao_ativo,
    comissao_data_criacao,
    comissao_data_atualizacao )
VALUES( gen_random_uuid(),
    reg.funcionario_id,
    total_comissao,
    true,
    NOW(),
    NOW() );

```

```

-- Update no campo venda_comissionado
-- para que ela não seja mais comissionada

    UPDATE venda SET venda_comissionado = true
    WHERE venda_id = reg.venda_id;

-- Devemos zerar as variáveis para reutilizá-las

    total_comissao := 0;
    porc_comissao := 0;

-- Término do loop

END LOOP;

END
$$ LANGUAGE PLPGSQL;

```

Para executar esse processo, usaremos o seguinte comando:

```

SELECT calc_comissao('01/01/2023 00:00:00', '01/01/2023 00:00:00')
;
```

Agora vamos ver os registros na tabela `comissao`. Vamos consultar as comissões do funcionário `funcionario_id = '4810ff6d-2bff-4b2d-8789-893bc9e50d32'`.

```

postgres=> SELECT comissao_valor,
                  comissao_data_criacao
            FROM comissao
           WHERE funcionario_id = '4810ff6d-2bff-4b2d-8789-893
bc9e50d32';

```

Com isso, temos cada comissão gerada de todas as vendas do funcionário.

The screenshot shows the pgAdmin interface with a query editor and a results table. The query is:

```
1 SELECT comissao_valor,
2        comissao_data_criacao
3   FROM comissao
4 WHERE funcionario_id = '4810ff6d-2bff-4b2d-8789-893bc9e50d32'
5
```

The results table has two columns: 'comissao\_valor' (double precision) and 'comissao\_data\_criacao' (timestamp without time zone). One row is shown with the value '1' in both columns.

Figura 4.3: Comissões geradas do funcionário `funcionario_id = '4810ff6d-2bff-4b2d-8789-893bc9e50d32'`.

Assim como vimos alguns elementos novos quando em nossa primeira `FUNCTION`, nesta última, podemos observar alguns elementos com que ainda não tínhamos trabalhado.

## Cursor

Um pouco mais de código, não é mesmo? E eu utilizei um conceito novo. Declarei um cursor `cr_porce`, usado para armazenar uma consulta e utilizado onde o desejamos no código. Em vez de colocarmos o `SELECT` ao meio do código, nós declaramos um cursor com essa consulta e o utilizamos onde desejamos na `FUNCTION`. Com isso, separamos o processo das consultas auxiliares, e o código fica mais limpo.

Mas cuidado com o excesso de cursores, pois um pouco de memória do servidor é consumida em cada abertura de um cursor. Para usá-lo, não esqueça de abri-lo e fechá-lo.

O `FETCH` é utilizado para pegar o valor da consulta e jogá-lo para uma variável. Não tem um número exato mínimo ou máximo de cursores que devemos usar. Com a prática, você aprenderá a otimizar seu código e observar o que deixa o código rápido ou lento.

## FOR... LOOP... END LOOP

A instrução `LOOP` define um laço incondicional, repetido indefinidamente até ser terminado. O `FOR` cria um laço que interage em um intervalo de valores inteiros. A variável `reg` é definida automaticamente como sendo do tipo `integer`, e somente existe dentro do laço.

Em nosso exemplo, utilizamos o `FOR...` `LOOP` para varrer todas as vendas que foram realizadas em um determinado período e fazer os cálculos de comissionamento.

## 4.4 ALTERANDO FUNCTIONS

É muito comum alterações nas regras de negócios que impactam nos códigos de seus projetos, e você acaba tendo de modificar e criar novamente as `FUNCTIONS`. Observe na sintaxe de criação onde escrevemos `CREATE OR REPLACE`. Isso significa que, ao inserir o código no terminal de comando do PostgreSQL, o gerenciador vai criar ou substituir o procedimento. Isso quer dizer que, se você fizer qualquer alteração nos códigos, é só inserir novamente que o SGBD vai atualizar o procedimento em questão.

Há a possibilidade da alteração de uma `FUNCTION`, mas eu não recomendo, pois isso pode impactar e ocasionar erros em outras

partes de seu sistema nas quais ela possa estar sendo utilizada. Prefira criar uma nova `FUNCTION` com o nome que deseja e, se necessário, exclua a que não desejar mais, e tenha certeza de que ela não está sendo utilizada em nenhum trecho.

## 4.5 EXCLUINDO FUNCTIONS

Para excluir uma function, usaremos o comando `DROP FUNCTION`... Vamos excluir a `calc_comissao` com o comando:

```
postgresql=> DROP FUNCTION calc_comissao();
```

### ATENÇÃO!

Ao excluir uma function, tenha certeza de que ela não está sendo usada em nenhum processo em seu projeto.

## 4.6 VANTAGENS DA UTILIZAÇÃO DAS FUNCTIONS

A principal vantagem da utilização desse procedimento é a reutilização em qualquer lugar no projeto de banco de dados. Você escreve uma vez e reutiliza onde desejar.

O conceito da programação das regras de negócio no banco de dados (isto é, deixar a parte de cálculos e regras em procedimentos armazenados no banco) é algo que gera muitas discussões. Isso por causa de linguagens de programação que fazem isso bem, como Java, Ruby on Rails, entre outras. Mas isso vai depender do cenário

em que você estiver trabalhando e também da tecnologia.

Não há dúvidas de que o processamento direto no banco é muito rápido. Porém, isso também dependerá de seu código ser consistente. E como é comum desenvolvedores(as) fazerem a refatoração de suas aplicações, no banco de dados não é diferente. Conforme as consultas vão se tornando complexas, vai se exigindo mais do banco de dados. Por isso, elas devem estar em constante refatoração.

## 4.7 PARA PENSAR!

Otimizar e automatizar. Algo que vimos bastante neste capítulo. São duas palavras muito presentes na vida da programadora e do programador. Estamos sempre otimizando e procurando uma maneira de melhorar códigos e processos. Tenha isso em mente e sempre terá excelentes códigos em seus projetos.

Agora, se estiver trabalhando em algum projeto que não seja este do livro, pense em como otimizar seus códigos por meio de functions! Se não tiver nenhum código, crie novas funções para as outras tabelas que não fizemos.

Gostou de criar functions que podemos reutilizar? E de criar processos para automatizar? Vamos trabalhar mais um pouco com funções, mas agora com as embutidas no SGBD. Elas são muito úteis. #PartiuConsultar

## CAPÍTULO 5

# FUNÇÕES, OPERADORES E OPERAÇÕES

*"Mova-se rapidamente e quebre as coisas. A menos que você não esteja quebrando coisas, você não está se movendo rápido o suficiente." — Mark Zuckerberg*

## 5.1 FUNÇÕES EMBUTIDAS

Nativamente, o PostgreSQL, assim como outros bancos, como MySQL e Oracle, possui funções e operadores para os tipos de dados nativos que fazem determinadas tarefas e estão armazenadas nele. Tarefas de que rotineiramente necessitamos quando estamos desenvolvendo software; por exemplo, calcular a quantidade de caracteres em uma `string` ou o valor máximo armazenado em uma coluna de uma determinada tabela.

Esses são exemplos de funções que estão embutidas no banco de dados e podemos utilizar em nossas consultas e processos. São muito úteis, uma vez que são operações simples que seriam muito trabalhosas para fazer na mão. Separei este capítulo para mostrar essas funções. Elas estão divididas em grupos: as funções numéricas, utilizadas para manipularmos números e extrair informações deles; as funções para trabalharmos com cadeias de

caracteres, utilizadas para criar funções com texto; e as funções para trabalhamos com datas e horários e extrair informações desses tipos de dados.

Uma preocupação, levada em consideração nos mais variados SGBD ao usarem esse padrão, é a possibilidade da portabilidade. Isso torna as funcionalidades do PostgreSQL compatíveis e consistentes entre as várias implementações em outros bancos.

Antes de apresentar cada grupo de função, serão apresentados os operadores de grupo. Por exemplo, antes de apresentar as funções matemáticas, serão apresentados os operadores matemáticos, e assim sucessivamente. Mas, antes de apresentar os operadores possuidores de funções, temos de conhecer os operadores lógicos e os de comparação.

## 5.2 OPERADORES LÓGICOS

Até o momento em que aprendemos a trabalhar com `functions`, não tínhamos feito consultas usando outros operadores além do `WHERE`. Quando criamos a function `calc_comissao`, utilizamos o operador lógico `AND`, como mostra a consulta a seguir:

```
SELECT venda_id,  
       funcionario_id,  
       venda_total  
  FROM venda  
 WHERE venda_data_criacao >= 'data_ini'  
   AND venda_data_criacao <= 'data_fim'  
   AND venda_ativo = false
```

Os operadores lógicos que o PostgreSQL possui são os três habituais. Se você já começou ao menos a estudar lógica de

programação, já deve ter esbarrado neles.

Operador	Descrição
AND	É utilizado quando queremos incluir duas ou mais condições em nossa operação. Os registros recuperados em uma declaração que une duas condições com esse operador deverão suprir as duas ou mais condições.
OR	É utilizado quando queremos combinar duas ou mais condições em nossa operação. Os registros recuperados em uma declaração que une duas condições com esse operador deverão suprir uma das duas condições.
NOT	É utilizado quando não queremos que uma das condições seja cumprida. Os registros recuperados em uma declaração que exclui uma condição não deverão trazer aqueles que cumprem a condição que se está testando.

Vamos consultar para exemplificar os outros operadores que ainda não usamos. Para isso, inseriremos um novo produto.

```
INSERT INTO produto(produto_codigo,
                    produto_nome,
                    produto_valor,
                    produto_ativo,
                    produto_data_criacao,
                    produto_data_atualizacao)
VALUES('2832',
       'SUCO DE LIMÃO',
       15,
       false,
       '02/02/2023',
       '02/02/2023');
```

Agora vamos consultar os produtos.

```
SELECT *
FROM produto
WHERE produto_ativo = true
AND produto_ativo = false;
```

O resultado de nossa consulta não trará nenhum registro. Isso acontece pois não temos nenhum produto que está ativo E cancelado ao mesmo tempo. Agora, vamos modificá-la um pouco

e consultar os produtos que estão ativos **OU** os que estão cancelados.

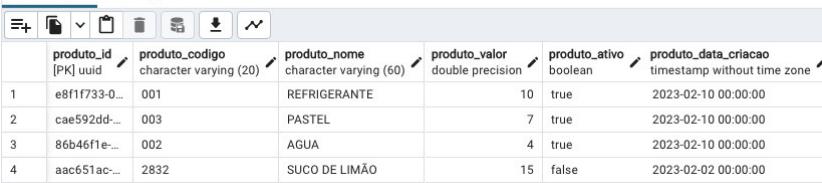
```
SELECT *
  FROM produto
 WHERE produto_ativo = true
   OR produto_ativo = false;
```

---

1	SELECT *
2	FROM produto
3	WHERE produto_ativo = true
4	OR produto_ativo = false;

---

Data output    Messages    Notifications



	produto_id [PK] uuid	produto_codigo character varying (20)	produto_nome character varying (60)	produto_valor double precision	produto_ativo boolean	produto_data_criacao timestamp without time zone
1	e8f1f733-0...	001	REFRIGERANTE	10	true	2023-02-10 00:00:00
2	cae592dd-...	003	PASTEL	7	true	2023-02-10 00:00:00
3	86b46f1e-...	002	AGUA	4	true	2023-02-10 00:00:00
4	aac651ac-...	2832	SUCO DE LIMÃO	15	false	2023-02-02 00:00:00

Figura 5.1: Consulta com o operador OR.

O resultado mudou um pouco, não é mesmo? Pois temos produtos que estão ativos e temos outros que estão cancelados.

Agora vamos criar uma consulta para buscar os produtos que não possuam o campo `produto_nome` igual a `SUCO DE LIMÃO`. Vamos ao nosso código. Nessa consulta, negamos a condição declarada, que é `produto_nome = 'SUCO DE LIMÃO'`.

```
SELECT *
  FROM produto
 WHERE NOT produto_nome = 'SUCO DE LIMÃO';
```

```

1   SELECT *
2     FROM produto
3 WHERE NOT produto_nome = 'SUCO DE LIMÃO';

```

Data output Messages Notifications

	produto_id [PK] uuid	produto_codigo character varying (20)	produto_nome character varying (60)	produto_valor double precision	produto_ativo boolean
1	e8f1f733-09ff-4992-9b60-e0321266b894	001	REFRIGERANTE	10	true
2	cae592dd-1a7d-4fdc-bc4e-0b793f833fa6	003	PASTEL	7	true
3	86b46f1e-5056-4562-bffc-85544cc6ea9b	002	AGUA	4	true

Figura 5.2: Consulta com o operador NOT

Além de utilizá-los separadamente, podemos criar consultas unindo-os. Vamos criar uma consulta para buscar os produtos que estão ativos, isto é, `produto_ativo = true`, OU que estejam cancelados, isto é, `produto_ativo = false`, e que ao mesmo tempo possuam data de criação igual a `02/02/2023`.

```

SELECT *
  FROM produto
 WHERE produto_ativo = true
    OR (produto_ativo = false
        AND produto_data_criacao = '02/02/2023');

```

```

1   SELECT *
2     FROM produto
3 WHERE produto_ativo = true
4     OR (produto_ativo = false
5          AND produto_data_criacao = '02/02/2023');

```

Data output Messages Notifications

	produto_id [PK] uuid	produto_codigo character varying (20)	produto_nome character varying (60)	produto_valor double precision	produto_ativo boolean
1	e8f1f733-09ff-4992-9b60-e0321266b894	001	REFRIGERANTE	10	true
2	cae592dd-1a7d-4fdc-bc4e-0b793f833fa6	003	PASTEL	7	true
3	86b46f1e-5056-4562-bffc-85544cc6ea9b	002	AGUA	4	true
4	aac651ac-ab8b-410c-92e7-d45c701e5a...	2832	SUCO DE LIMÃO		15 false

Figura 5.3: Consulta com os operadores AND e OR.

Veja que tivemos de satisfazer os dois operadores. Usamos os parênteses para isolar as condições, pois o OR precisou satisfazer duas condições diferentes.

## 5.3 OPERADORES DE COMPARAÇÃO

Em todo momento em que criamos consultas, utilizamos operadores de comparação. Sem eles, seria impossível realizá-las, já que não estamos querendo buscar todos os registros, mas sim querendo satisfazer alguma condição. E para satisfazer uma condição, é necessário utilizarmos os operadores de comparação. Nossos operadores são:

Operador	Descrição
<	Menor
>	Maior
<=	Menor ou igual
>=	Maior ou igual
=	Igual
<> ou !=	Diferente

Alguns deles já usamos em nossos códigos criados. O = (igual) já utilizamos a todo tempo. Em uma de nossas function , usamos os operadores <= e >= para testar a condição de datas, como mostra a consulta seguinte.

```
SELECT venda_id,
       funcionario_id,
       venda_total
  FROM venda
 WHERE venda_data_criacao >= 'data_ini'
   AND venda_data_criacao <= 'data_fim'
```

```
AND venda_ativo = false;
```

Se quiséssemos excluir os valores das variáveis `data_ini` e `data_fim`, utilizariamós apenas os operadores `>` e `<`, pois assim consultariamós apenas as datas entre esses dois valores, como mostra o código a seguir:

```
SELECT venda_id,  
       funcionario_id,  
       venda_total,  
       venda_data_criacao  
  FROM venda  
 WHERE venda_data_criacao >= '01/01/2023'  
   AND venda_data_criacao < '02/02/2023'  
   AND venda_ativo = false;
```

Nessa última consulta, as datas levadas em consideração seriam a `01/01/2023` e todas as menores que `02/02/2023`, excluindo portanto a data `02/02/2023`, pois utilizamos o operador `<` e não o operador `<=` (que iria incluir a data `02/02/2023` na consulta).

## 5.4 OPERADORES E FUNÇÕES MATEMÁTICAS

### Operadores matemáticos

São fornecidos operadores matemáticos para muitos tipos de dados do PostgreSQL. Para alguns operadores, veremos que existem funções que fazem o seu trabalho, como somar todos os valores de uma coluna em uma tabela, com a função `sum()`. Veremos mais à frente como será o seu funcionamento.

Operador	Descrição	Exemplo	Resultado
+	adição	select(2 + 3)	5
-	subtração	select(2 - 3)	-1
*	multiplicação	select(2 * 3)	6
/	divisão (divisão inteira trunca o resultado)	select(4 / 2)	2
%	módulo (resto)	select(5 % 4)	1
^	exponenciação	select(2.0 ^ 3.0)	8
@	valor absoluto	select(@ -5.0)	5
/	raiz quadrada	select( /25)	5
/	raiz cúbica	select(  /64)	4

Esses operadores serão úteis para escrever consultas nas quais há a necessidade de se realizar cálculos. Nas demais, veremos como fica mais fácil utilizar as funções. Mas ainda assim, não podemos ignorar os operadores matemáticos, pois constantemente vamos usá-los. Se você já terminou o ensino médio e achava que ficaria longe da matemática, desculpe-me por decepcioná-lo(a).

Para você utilizar qualquer um desses operadores, você pode fazer da seguinte maneira:

```
select(||/64);
```

The screenshot shows a PostgreSQL database client interface. At the top, there's a connection bar with the text "db\_restaurant/postgres@ServerPostgre". Below it is a toolbar with various icons for file operations like Open, Save, and Print, along with dropdown menus and a "No limit" button. The main area has tabs for "Query" and "Query History", with "Query" currently selected. A single line of SQL code, "1 select(1||64);", is visible. Below the query results, there are tabs for "Data output", "Messages", and "Notifications", with "Data output" being the active tab. The data output section shows a table with one row. The first column is labeled "1" and the second column is labeled "?column? double precision". There is a lock icon next to the column header. The value in the cell is "4".

Figura 5.4: Consulta com operador matemático.

## Funções matemáticas

Existem recursos disponíveis em cada tipo de linguagem de programação ou banco de dados que usamos diariamente. As funções, sejam elas matemáticas ou de caracteres, são algo que utilizamos nas mais diversas situações. É importante que você saiba que elas existem, mas não é necessário decorar cada uma delas. Sabendo que um recurso existe, você saberá onde deve procurá-lo.

Funções	Descrição	Exemplo	Resultado
$abs(variável)$	Para calcular o valor absoluto de uma variável.	<code>select abs(-5);</code>	5
$cbrt(variável)$	Para calcular o valor da raiz cúbica de uma variável.	<code>select cbrt(8);</code>	2
$ceil(variável)$	Para calcular o menor valor que seja maior ou igual à variável.	<code>select ceil(14.2);</code>	15
$ceiling(variável)$	O mesmo que o $ceil$ .	<code>select ceil(-51.1);</code>	-51
$degrees(variável)$	Utilizado para converter um valor de radianos para graus.	<code>select degrees(1)</code>	57.2957795130823
$div(variável x, variável y)$	Utilizado para fazer a divisão entre dois números.	<code>select div(8,4);</code>	2
$exp(variável)$	Utilizado para fazer cálculo de exponenciação.	<code>select exp(2);</code>	7.38905609893065
$floor(variável)$	Utilizado para encontrar o maior número inteiro não maior que a variável.	<code>select floor(-12.9);</code>	-13
$ln(variável)$	Calcular e mostrar o valor do logaritmo comum.	<code>select ln(2.0)</code>	0.693147180559945
$log(variável)$	Utilizado para calcular logaritmo na base 10.	<code>select log(10)</code>	2
$log(variável b,$	Logaritmo na		

<i>variável k)</i>	base b.	<i>select log(100.2);</i>	2.0008677215312267
<i>mod(variável x, variável y)</i>	Cálculo do resto da divisão de dois números.	<i>select mod(11,2);</i>	1
<i>pi()</i>	Retorna o valor de pi.	<i>select pi();</i>	3.14159265358979
<i>power(variável x , variável y)</i>	Faz o cálculo exponencial da variável x pela y.	<i>select power(9.0, 3.0);</i>	729
<i>radians(variável)</i>	Faz o cálculo de conversão para graus radianos.	<i>select radians(12);</i>	0.20943951023932
<i>round(variável)</i>	Faz o arredondamento da variável informada.	<i>select round(25.2);</i>	25
<i>round(variável x, variável y)</i>	Realiza o arredondamento para o número especificado depois da vírgula.	<i>select round(54.123,2);</i>	54.12
<i>sqrt(variável)</i>	Calcula a raiz quadrada de um número.	<i>select sqrt(9);</i>	3
<i>trunc(variável)</i>	Utilizado para separar o número inteiro dos decimais.	<i>select trunc(335.23);</i>	335
<i>trunc(variável x, variável y)</i>	Utilizado para separar uma quantidade específica de números decimais.	<i>select trunc(335.123,2);</i>	335.12

Há também algumas funções trigonométricas. São elas:

Function	Descrição	Exemplo	Resultado
$\text{acos}(\text{variável } x)$	Utilizado para calcular o inverso do cosseno.	<code>select acos(0);</code>	1.5707963267949
$\text{asin}(\text{variável } x)$	Utilizado para calcular o inverso do seno.	<code>select asin(0.2);</code>	0.201357920790331
$\text{atan}(\text{variável } x)$	Utilizado para calcular o inverso da tangente.	<code>select atan(2);</code>	1.10714871779409
$\text{cos}(\text{variável } x)$	Utilizado para calcular o valor do cosseno.	<code>select cos(3);</code>	-0.989992496600445
$\text{cot}(\text{variável } x)$	Utilizado para calcular o valor da cotangente.	<code>select cot(3);</code>	-7.01525255143453
$\text{sin}(\text{variável } x)$	Utilizado para calcular o valor do seno.	<code>select sin(3);</code>	0.141120008059867
$\text{tan}(\text{variável } x)$	Utilizado para calcular o valor da tangente.	<code>select tan(3);</code>	-0.142546543074278

## 5.5 FUNÇÕES DE TEXTO

Para trabalharmos com caracteres do tipo `string`, temos algumas funções específicas.

### Função `||` para concatenar strings

Utilizando as tabelas criadas no banco, vamos concatenar o código de um funcionário com seu nome. A consulta vai juntar o código e o nome em uma só coluna:

```
SELECT funcionario_codigo || funcionario_nome  
FROM funcionario  
WHERE funcionario_id = '4810ff6d-2bff-4b2d-8789-893bc9e50d32';
```

Fica estranho tudo junto, não é mesmo? Vamos colocar mais um item em nossa concatenação.

```
SELECT (funcionario_codigo || ' ' || funcionario_nome)
  FROM funcionario
 WHERE funcionario_id = '4810ff6d-2bff-4b2d-8789-893bc9e50d32';
```

Incluir um caractere de espaço no resultado ficou bem melhor. Não necessariamente temos de ter apenas caracteres do tipo `string`. Poderíamos ter um do tipo numérico no lugar do de espaço.

```
SELECT (funcionario_codigo ||8|| funcionario_nome)
  FROM funcionario
 WHERE funcionario_id = '4810ff6d-2bff-4b2d-8789-893bc9e50d32';
```

## Contando os caracteres de uma string com `char_length(string)` ou `length(string)`

Você já deve ter se deparado com sites que possuem formulário de cadastro nos quais a senha ou o nome de usuário deve possuir um número mínimo de caracteres. Podemos fazer essa contagem de caracteres através de uma função que o SGBD possui. Vamos contar os caracteres do nome de um funcionário.

```
SELECT CHAR_LENGTH(funcionario_nome)
  FROM funcionario
 WHERE funcionario_id = '4810ff6d-2bff-4b2d-8789-893bc9e50d32';
```

Como uma função dessas, você já pode verificar se um determinado campo já está sendo preenchido corretamente.

## Transformando letras minúsculas em maiúsculas com upper(string)

Em determinados sistemas, é muito comum as informações que estão armazenadas no banco de dados estarem todas maiúsculas, tanto por uma questão de padronização de quem está desenvolvendo o sistema, como por uma questão de estética, quando forem mostradas para os usuários. Essa função pode lhe ajudar nessa tarefa.

Vamos selecionar todos os nomes de funcionários. Se algum registro estiver salvo no banco com letras minúsculas, será mostrado como maiúscula.

```
SELECT UPPER(funcionario_nome)
      FROM funcionario;
```

Como nenhum exemplo de inserção de registro no banco foi feito com letras minúsculas, para você ver melhor o efeito de alteração das letras, utilize a seguinte consulta:

```
SELECT UPPER('livro postgresql');
```

E para deixar apenas as primeiras letras de cada palavra em maiúsculo, podemos usar o comando `initcap`. Como resultado, teremos `Livro Postgresql`.

```
SELECT INITCAP('livro postgresql');
```

## Transformando maiúsculas em minúsculas com lower(string)

Assim como temos uma função para transformar letras minúsculas em maiúsculas, temos uma função que faz o contrário: transforma maiúsculas em minúsculas.

Vamos deixar todos os nomes de funcionários em letras minúsculas.

```
SELECT LOWER(funcionario_nome)
      FROM funcionario;
```

## Substituindo string com overlay() e extraíndo com substring()

Se você se deparar em alguma situação em seu sistema na qual precisa ocultar ou substituir alguma parte de uma string , você pode utilizar esta função. É muito comum em alguns sites ocultarem uma parte de seu nome ou e-mail por uma questão de segurança.

Utilizando novamente o nome de um funcionário que está em nosso banco de dados, vamos substituir uma parte de seu nome com caracteres 000000 . Para isso, temos de informar a partir de qual caractere e até qual caractere a substituição deve ocorrer.

```
SELECT OVERLAY(funcionario_nome placing '000000'
                           from 3 for 5)
      FROM funcionario
     WHERE funcionario_id = '4810ff6d-2bff-4b2d-8789-893bc9e50d32';
```

Fizemos a substituição de uma parte da string . Agora vamos realizar a extração desse mesmo trecho da string . A nossa consulta vai ficar só um pouco diferente. Mão no código.

```
SELECT SUBSTRING(funcionario_nome FROM 3 for 5)
      FROM funcionario
     WHERE funcionario_id = '4810ff6d-2bff-4b2d-8789-893bc9e50d32';
```

## Localizando uma string position()

Quando estamos fazendo aquela pesquisa da Wikipédia e estamos com preguiça de procurar um determinado termo, nada melhor que utilizar o bom e velho `CTRL + F` e buscar a palavra dentro de um texto. Essa função tem o objetivo de identificar em qual posição se encontra um caractere ou se inicia uma `string`.

Utilizando o nome do funcionário com `funcionario_id = '4810ff6d-2bff-4b2d-8789-893bc9e50d32'` cujo nome é `VINICIUS`, vamos localizar em qual posição se encontra a string `CIUS`.

```
SELECT POSITION('CIUS' IN funcionario_nome)
  FROM funcionario
 WHERE funcionario_id = '4810ff6d-2bff-4b2d-8789-893bc9e50d32';
```

## 5.6 FUNÇÕES DATA/HORA

Se você já trabalha (ou não) com programação, vai descobrir que trabalhar com data e hora em seu sistema dá um pouco de trabalho. Não importa a linguagem, de cada três pessoas desenvolvedoras, duas já tiveram problemas nesse quesito.

As funções para data/hora dão uma grande ajuda, pois já existem algumas prontas para tarefas, como calcular a quantidade de dias que há entre duas datas ou a quantidade de anos de uma determinada pessoa, apenas informando uma data de entrada como parâmetro.

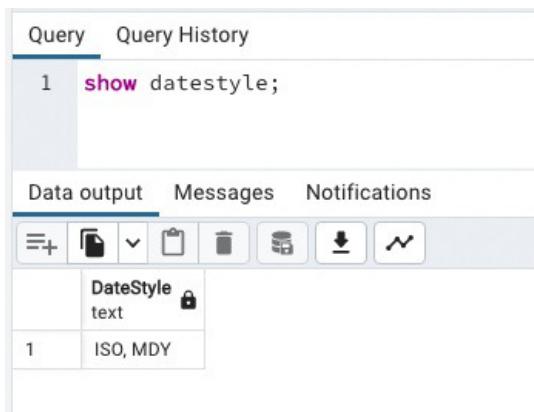
## DICA

O formato de data que usamos aqui no Brasil é Dia/Mês/Ano, diferente do de outros países. A maioria dos gerenciadores de banco de dados utilizam como padrão o formato Mês/Dia/Ano, ou Ano/Mês/Dia. E para você saber em qual formato está o seu banco de dados e mudar para o desejado, vamos fazer o seguinte.

Primeiro, vamos utilizar um comando para descobrir o formato de data atual do banco. Nós consultaremos a variável do banco de dados que armazena essa informação, que é o `datestyle`.

```
show datestyle;
```

Como resultado, descobri que o meu banco de dados está no formato Mês/Dia/Ano.



The screenshot shows a SQL query window with the following interface elements:

- Top navigation bar: "Query" (selected) and "Query History".
- Query editor area:
  - Text input field: "1 show datestyle;"
- Bottom navigation bar: "Data output" (selected), "Messages", and "Notifications".
- Toolbar icons: New query, Open, Save, Copy, Paste, Delete, Import, Export, Refresh, and Help.
- Data grid area:

DateStyle	text
1	ISO, MDY

Figura 5.5: Formato de datas atual do banco de dados.

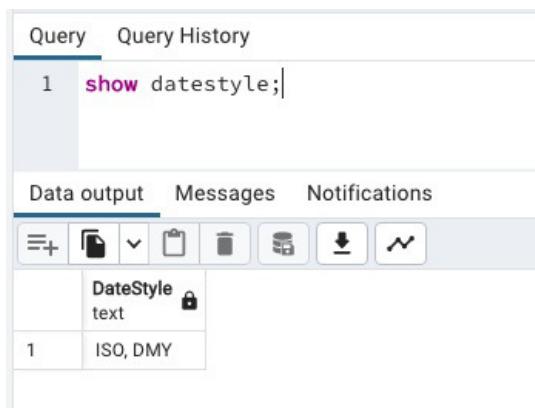
Sabendo disso, vamos alterar o formato do nosso banco para o que queremos, que é o formato Dia/Mês/Ano. Para isso, faremos um comando que alterará a variável `datestyle`.

```
alter database postgres set datestyle to iso, dmy;
```

E se você já estiver logado(a) no banco de dados, você pode executá-lo para aplicar essa alteração na sessão em que estiver logado.

```
set datestyle to iso, dmy;
```

Se consultarmos novamente o formato do banco, vamos obter como resultado:



The screenshot shows a PostgreSQL query interface. In the 'Query' tab, the command `show datestyle;` is entered. In the 'Data output' tab, the results are displayed in a table:

	DateStyle
1	text
1	ISO, DMY

Figura 5.6: Formato de datas atual do banco de dados.

Com essa alteração, podemos prosseguir com o projeto e utilizar datas no formato que conhecemos durante o nosso projeto.

## Funções de idade

Para criar uma função manualmente que traga quantos anos, meses e dias você tem é algo muito trabalhoso. O PostgreSQL e outros banco de dados sabem disso, e sabem que é uma função muito usada. O SGBD já possui uma função que realiza essa tarefa, a `age()`.

Vamos calcular agora a minha idade. Você pode usar a sua data de aniversário para saber a sua idade exata. Mão no teclado e vamos ao nosso código.

```
SELECT AGE(timestamp '11/04/1988');
```

The screenshot shows a PostgreSQL query editor interface. At the top, there are tabs for 'Query' (which is selected) and 'Query History'. Below the tabs, a single line of SQL code is shown: '1 SELECT AGE(timestamp '11/04/1988');'. Underneath the code, there are three tabs: 'Data output' (selected), 'Messages', and 'Notifications'. The 'Data output' tab displays the result of the query in a table. The table has two columns: 'age' and 'interval'. There is one row with the value '34 years 6 mons 6 days'. To the right of the 'age' column, there is a small lock icon.

	age interval
1	34 years 6 mons 6 days

Figura 5.7: Calculando a idade.

É possível também calcular a idade com base em duas datas da seguinte maneira:

```
SELECT AGE(timestamp '11/04/2000', timestamp '11/04/1988');
```

The screenshot shows a PostgreSQL query editor interface. At the top, there are tabs for 'Query' (which is selected) and 'Query History'. Below the tabs, a query is displayed: '1 SELECT AGE(timestamp '11/04/2000', timestamp '11/04/1988');'. Underneath the query, there are three tabs: 'Data output', 'Messages', and 'Notifications'. The 'Data output' tab is selected and shows a single row of results. The result table has two columns: 'age' and 'interval'. The 'age' column contains the value '12 years'. There is also a small lock icon next to the 'age' column header. Below the table, the number '1' indicates the count of rows.

Figura 5.8: Calculando a idade entre duas datas.

## Funções para consultar data, hora e data/hora atuais

A todo momento estamos olhando no relógio para saber o horário. E no desenvolvimento de software, por uma infinidade de motivos, temos de consultar a data e hora para criar validações em nosso sistema.

Imagine que você precise bloquear o acesso dos usuários depois que a conta deles esteja vencida. Para isso, você deverá conferir a data de vencimento delas com a data atual. E muitas vezes, deverá também levar em consideração o horário em que está fazendo a consulta para realizar o bloqueio.

Para a nossa salvação, o PostgreSQL fornece algumas maneiras para consultarmos data, hora e data/hora atuais. Essas funções estão relacionadas na tabela a seguir.

Função	Descrição	Exemplo
<code>clock_timestamp()</code>	Data e hora atual	<code>select clock_timestamp();</code>
<code>current_date</code>	Data atual	<code>select current_date;</code>
<code>current_time</code>	Hora atual	<code>select current_time;</code>
<code>current_timestamp</code>	Data e hora atual	<code>select current_timestamp;</code>
<code>localtime</code>	Hora atual	<code>select localtime;</code>
<code>localtimestamp</code>	Data e hora atual	<code>select localtimestamp;</code>
<code>now()</code>	Data e hora atual	<code>select now();</code>
<code>statement_timestamp()</code>	Data e hora atual	<code>select statement_timestamp();</code>
<code>timeofday()</code>	Data e hora atual no formato de texto	<code>select timeofday();</code>

Muitas dessas funções de datas e horários podem ser utilizadas para a mesma finalidade. Escolha a que desejar!

Para trabalharmos com data e horário, também temos à nossa disposição as funções:

Função	Descrição	Exemplo
<code>date_part('day', timestamp)</code>	Extrai o dia da data/hora informada.	<code>select date_part('day', timestamp '04/11/1988 20:38:40');</code>
<code>date_part('month', timestamp)</code>	Extrai o mês da data/hora informada.	<code>select date_part('month', timestamp '04/11/1988 20:38:40');</code>
<code>date_part('year', timestamp)</code>	Extrai o ano da data/hora informada.	<code>select date_part('year', timestamp '04/11/1988 20:38:40');</code>
<code>date_part('hour',</code>	Extrai a hora da data/hora informada.	<code>select date_part('hour', timestamp '04/11/1988</code>

<code>timestamp)</code>		<code>20:38:40');</code>
<code>date_part('minute', timestamp)</code>	Extrai os minutos da data/hora informada.	<code>select date_part('minute', timestamp '04/11/1988 20:38:40');</code>
<code>date_part('second', timestamp)</code>	Extrai os segundos da data/hora informada.	<code>select date_part('second', timestamp '04/11/1988 20:38:40');</code>
<code>justify_days(intervalo)</code>	Conta a quantidade de meses em um intervalo de dias.	<code>select justify_days(interval '43 days');</code>
<code>justify_hours(intervalo)</code>	Conta a quantidade de dias em um intervalo de horas.	<code>select justify_hours(interval '32 hours');</code>
<code>justify_interval(intervalo)</code>	Calcula a quantidade de meses, dias ou horas, subtraindo meses com horas.	<code>select justify_interval(interval '2 mon - 25 hours');</code>
<code>justify_interval(intervalo)</code>	Calcula a quantidade de meses, dias ou horas, subtraindo meses com dias.	<code>select justify_interval(interval '2 mon - 14 days');</code>
<code>justify_interval(intervalo)</code>	Calcula a quantidade de meses, dias ou horas, subtraindo dias com horas.	<code>select justify_interval(interval '3 days - 8 hour');</code>
<code>justify_interval(intervalo)</code>	Calcula a quantidade de meses, dias ou horas, somando meses com horas.	<code>select justify_interval(interval '4 mon - 28 hour');</code>

Uma outra função muito interessante é a `extract`. Com ela, é possível extrair diversas informações de uma variável de data/hora, data ou somente hora. Ela sempre vai retornar um resultado do tipo `double`. As informações mais relevantes que essa função pode extrair são:

Função	Descrição	Exemplo
<i>century</i>	Para extrair o século de uma determinada data.	<i>select extract (century from timestamp '04/11/1988 12:21:13');</i>
<i>day</i>	Para extrair o dia de uma determinada data.	<i>select extract (day from timestamp '04/11/1988 12:21:13');</i>
<i>decade</i>	Para a extraír a década de uma determinada data.	<i>select extract (decade from timestamp '04/11/1988 12:21:13');</i>
<i>doy</i>	Para extraír o dia do ano de uma determinada data.	<i>select extract (doy from timestamp '04/11/1988 12:21:13');</i>
<i>hour</i>	Para extraír a hora de um determinado horário.	<i>select extract (hour from timestamp '04/11/1988 12:21:13');</i>
<i>year</i>	Para extraír o ano de uma determinada data.	<i>select extract (year from timestamp '04/11/1988 12:21:13');</i>
<i>minute</i>	Para extraír os minutos de um determinado horário.	<i>select extract (minute from timestamp '04/11/1988 12:21:13');</i>
<i>month</i>	Para extraír o mês de uma determinada data.	<i>select extract (month from timestamp '04/11/1988 12:21:13');</i>
<i>second</i>	Para extraír o valor dos segundos de um determinado horário.	<i>select extract (second from timestamp '04/11/1988 12:21:13');</i>

Para utilizar o `extract` em uma das tabelas do nosso projeto, o código será:

```
SELECT EXTRACT(YEAR FROM data_criacao)
  FROM funcionario
 WHERE funcionario_id = '4810ff6d-2bff-4b2d-8789-893bc9e50d32';
```

Essas funções que extraem informações de uma determinada data e horário são muito úteis quando estamos desenvolvendo uma aplicação e precisamos exibir ao usuário alguma informação baseada em datas do sistema. Para não precisarmos programar os cálculos, o banco de dados já tem essas funções para nos auxiliar.

## 5.7 FUNÇÕES AGREGADORAS

As funções agregadoras são, sem sombra de dúvidas, as mais importantes e as mais usadas nas consultas do dia a dia da pessoa programadora. Isso porque é com elas que fazemos cálculos e extraímos informações importantes dos nossos dados.

### Contando nossos registros com count(\*)

Sem olhar no banco de dados, você sabe quantos registros já temos na tabela `funcionario`? Conforme a quantidade de registros no banco de dados vai aumentando, é claro que não conseguimos acompanhar e saber quantos uma tabela contém. Para nos auxiliar nessa tarefa, temos uma função muito bacana, a `count(*)`.

Vamos contar quantos registros a nossa tabela possui. Mão no teclado e vamos escrever essa consulta.

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

The screenshot shows a database query interface. At the top, there are tabs for 'Query' and 'Query History'. Below the tabs, the query is written:

```
1 SELECT COUNT(*)  
2 FROM funcionario;
```

Below the query area, there are tabs for 'Data output', 'Messages', and 'Notifications'. Under 'Data output', there is a toolbar with various icons. A table is displayed with the following data:

	count	bigint
1	2	

Figura 5.9: Contando os registros da tabela funcionários.

### DICA!

Quando a tabela não possui muitos registros nem muitos campos, podemos usar o \* (asterisco) entre parênteses. No entanto, quando uma tabela possui muitos registros e campos, procure utilizar um campo entre parênteses, de preferência a chave primária da tabela.

Quando o \* é usado, o banco utiliza todos os campos para fazer a contagem. Isso faz com que ele fique lento e a consulta demore. Quando indicamos um campo, a consulta ganha performance. Ela ficaria da seguinte maneira: select count(funcionario\_id) from funcionario .

## Somando as colunas utilizando sum()

O foco dos sistemas que estamos desenvolvendo durante o livro são vendas. Então, saber a soma de todas as vendas é um dos objetivos quando se tem um sistema de vendas. Vamos calcular o total de vendas que fizemos até agora.

```
SELECT SUM(venda_total)  
FROM venda;
```

Levando em consideração que você não tenha inserido nenhuma outra venda, além das que inserimos no começo do livro, como resultado você também terá:

The screenshot shows a MySQL Workbench interface. The 'Query' tab is selected, displaying the SQL code: 'SELECT SUM(venda\_total) FROM venda;'. Below the code, the 'Data output' tab is selected, showing a single row of results:

	sum
1	40

Below the table are standard export and refresh icons.

Figura 5.10: Somando nossas vendas.

Se você inseriu mais registros, não tem problema. Você está de parabéns, isso mostra que você está praticando bem os assuntos de que estamos tratando.

## Calculando a média dos valores com avg()

Agora que já possuímos o valor total de todas as vendas, vamos calcular a média de preço dos produtos que comercializamos. Para essa tarefa, usaremos a função `avg()`.

```
SELECT AVG(produto_valor)
FROM produto;
```

The screenshot shows a MySQL Workbench interface. The 'Query' tab is selected, displaying the SQL code: 'SELECT AVG(produto\_valor) FROM produto;'. Below the code, the 'Data output' tab is selected, showing a single row of results. The results table has one column labeled 'avg' with the value '9'. The table also indicates it is of type 'double precision' and has a lock icon.

	avg
1	9

Figura 5.11: Calculando o valor médio dos produtos.

## Valores máximos e mínimos de uma coluna com max() e min()

Temos a média dos valores dos produtos. A média varia com os maiores e menores preços dos produtos. E para saber qual o produto de maior valor e o de menor? Também temos uma função para nos auxiliar nessa tarefa. É a função `MAX()` e `MIN()`, para encontrarmos o maior e o menor valor, respectivamente. Então, mãos no teclado e vamos lá!

```
SELECT MAX(produto_valor), MIN(produto_valor)
  FROM produto;
```

The screenshot shows a PostgreSQL client interface. At the top, there's a connection bar with the text "db\_restaurant/postgres@ServerPostgre". Below it is a toolbar with various icons for database management. The main area is titled "Query" and contains the following SQL code:

```
1 SELECT MAX(produto_valor), MIN(produto_valor)
2 FROM produto;
```

Below the query results, there are tabs for "Data output", "Messages", and "Notifications". Under "Data output", there is a table with two columns: "max double precision" and "min double precision". The data row shows values 15 and 4 respectively.

	max double precision	min double precision
1	15	4

Figura 5.12: Valor máximo e mínimo

## Agrupando registros iguais com group by

Descobrimos algumas informações interessantes sobre os nossos dados. Descobrimos o total de vendas, o valor médio de nossos produtos e o maior e o menor valor de um produto que temos em nosso banco. Agora vamos criar uma consulta para buscarmos a venda de cada produto vendido. Usaremos uma função que já conhecemos, a `SUM()`, e conheceremos uma nova função agregadora, a `group by`.

Vamos utilizá-la para agrupar os registros iguais para criar algum tipo de totalizador, que no nosso caso será a soma das vendas de cada produto. Vamos à nossa consulta, mas, antes de criarmos, vamos inserir mais alguns registros nas tabelas `venda` e `itens_venda` para termos mais dados para testarmos.

```
INSERT INTO venda ( venda_id,
```

```

funcionario_id,
mesa_id,
venda_codigo,
venda_valor,
venda_total,
venda_desconto,
venda_ativo,
venda_comissionado,
venda_data_criacao,
venda_data_atualizacao)
VALUES (gen_random_uuid(),
'd44458ed-877d-4251-9a53-24ef06a248c5',
1,
'10201',
'51',
'51',
'0',
true,
false,
'01/01/2023',
'01/01/2023');

INSERT INTO itens_venda(itens_venda_id,
produto_id,
venda_id,
itens_venda_valor,
itens_venda_quantidade,
itens_venda_total,
itens_venda_data_criacao,
itens_venda_data_atualizacao)
VALUES (gen_random_uuid(),
'aac651ac-ab8b-410c-92e7-d45c701e5a22',
'f14abea0-d9e5-469e-9557-e52a5c79e151',
15,
2,
30,
'01/01/2023',
'01/01/2023');

INSERT INTO itens_venda(itens_venda_id,
produto_id,
venda_id,
itens_venda_valor,
itens_venda_quantidade,
itens_venda_total,

```

---

```

        itens_venda_data_criacao,
        itens_venda_data_atualizacao)
VALUES (gen_random_uuid(),
        '86b46f1e-5056-4562-bffc-85544cc6ea9b',
        'f14abea0-d9e5-469e-9557-e52a5c79e151',
        7,
        3,
        21,
        '01/01/2023',
        '01/01/2023');

INSERT INTO venda ( venda_id,
                    funcionario_id,
                    mesa_id,
                    venda_codigo,
                    venda_valor,
                    venda_total,
                    venda_desconto,
                    venda_ativo,
                    venda_comissionado,
                    venda_data_criacao,
                    venda_data_atualizacao)
VALUES (gen_random_uuid(),
        'd44458ed-877d-4251-9a53-24ef06a248c5',
        1,
        '10202',
        '20',
        '20',
        '0',
        true,
        false,
        '01/01/2023',
        '01/01/2023');

INSERT INTO itens_venda(itens_venda_id,
                      produto_id,
                      venda_id,
                      itens_venda_valor,
                      itens_venda_quantidade,
                      itens_venda_total,
                      itens_venda_data_criacao,
                      itens_venda_data_atualizacao)
VALUES (gen_random_uuid(),
        'e8f1f733-09ff-4992-9b60-e0321266b894',
        '78ea20e5-a5ff-48a5-b6e2-5254cbf2653d',
        
```

```

        10,
        2,
        20,
        '01/01/2023',
        '01/01/2023');

INSERT INTO venda ( venda_id,
                    funcionario_id,
                    mesa_id,
                    venda_codigo,
                    venda_valor,
                    venda_total,
                    venda_desconto,
                    venda_ativo,
                    venda_comissionado,
                    venda_data_criacao,
                    venda_data_atualizacao)
VALUES (gen_random_uuid(),
        'd44458ed-877d-4251-9a53-24ef06a248c5',
        1,
        '10002',
        '45',
        '45',
        '0',
        true,
        false,
        '01/01/2023',
        '01/01/2023');

INSERT INTO itens_venda(itens_venda_id,
                      produto_id,
                      venda_id,
                      itens_venda_valor,
                      itens_venda_quantidade,
                      itens_venda_total,
                      itens_venda_data_criacao,
                      itens_venda_data_atualizacao)
VALUES (gen_random_uuid(),
        'aac651ac-ab8b-410c-92e7-d45c701e5a22',
        '04187a8a-a1e3-4ae2-b869-23928f2de70e',
        15,
        3,
        45,
        '01/01/2023',
        '01/01/2023');

```

Agora que temos vários registros, montaremos a consulta. Vamos buscar os produtos e somar o total vendido de cada item. Conseguiremos fazer isso se agruparmos todos os registros iguais no campo `produto_id`. Por isso, vamos utilizar a função de agrupamento.

```
SELECT produto_id , SUM(itens_venda_total)
  FROM itens_venda
GROUP BY produto_id;
```

Como resultado da nossa consulta, temos:

The screenshot shows the MySQL Workbench interface. In the 'Query' tab, the following SQL code is displayed:

```
1 SELECT produto_id , SUM(itens_venda_total)
2      FROM itens_venda
3 GROUP BY produto_id;
```

In the 'Data output' tab, the results of the query are shown in a table:

	produto_id uuid	sum double precision
1	86b46f1e-5056-4562-bffc-85544cc6ea9b	42
2	aac651ac-ab8b-410c-92e7-d45c701e5a22	75
3	e8f1f733-09ff-4992-9b60-e0321266b894	40

Figura 5.13: Valor vendido de cada produto.

Não ficou muito legal a apresentação, pois só temos o `produto_id` de cada produto. Vamos utilizar um recurso que já aprendemos para mostrar o nome do produto. Logo, criaremos uma função que busca o seu nome. Mão no teclado e vamos colocar em prática um recurso já aprendido.

`CREATE OR REPLACE FUNCTION`

```

retorna_nome_produto(prod_id UUID)
RETURNS TEXT AS
$$
DECLARE
    nome      TEXT;
BEGIN
    SELECT produto_nome
        INTO nome
        FROM produto
       WHERE produto_id = prod_id;

    RETURN nome;
END
$$
LANGUAGE PLPGSQL;

```

Agora que temos uma função que retorna o nome do produto, vamos reescrever a consulta que fizemos com os totais de cada produto vendido.

```

SELECT retorna_nome_produto(produto_id) , sum(itens_venda_total)
)
FROM itens_venda
GROUP BY produto_id;

```

The screenshot shows a PostgreSQL query editor interface. At the top, there's a 'Query' tab and a 'Query History' section containing the function definition. Below it is a 'Data output' tab, which is currently active, showing the results of the executed query. The results are presented in a table with three rows:

	retorna_nome_produto	sum
	text	double precision
1	AGUA	42
2	SUCO DE LIMÃO	75
3	REFRIGERANTE	40

Figura 5.14: Valor vendido de cada produto e uma function para retornar o nome do produto.

Muito melhor, não é mesmo? Agora conseguimos saber qual produto está listado. Mas ainda podemos melhorar um pouco mais a nossa consulta. Vamos ordenar o nosso resultado e apelidar as colunas. Ficou meio bagunçada essa última consulta, pois, se você ver o nome da coluna no seu resultado, o nome é `sum`. Se você estivesse vendo pela primeira vez esse resultado, você não saberia dizer do que se trata. Então, vamos melhorar o nosso código.

```
SELECT retorna_nome_produto(produto_id) PRODUTO,
       sum(itens_venda_total) VL_TOTAL_PRODUTO
  FROM itens_venda
 GROUP BY produto_id
 ORDER BY vl_total_produto, produto;
```

The screenshot shows a MySQL query editor interface. At the top, there are tabs for 'Query' and 'Query History'. Below the tabs is the SQL code. Underneath the code, there are tabs for 'Data output', 'Messages', and 'Notifications'. The 'Data output' tab is selected, showing a table with three rows of data. The table has two columns: 'produto' (type text) and 'vl\_total\_produto' (type double precision). The data is as follows:

	produto	vl_total_produto
1	REFRIGERANTE	40
2	AGUA	42
3	SUCO DE LIMAO	75

Figura 5.15: Valor vendido de cada produto, uma function para retornar o nome do produto, ordenado e apelidado.

Observe como conseguimos evoluir o nosso código. Para apelidar uma coluna, basta colocar um nome qualquer na frente dela. Quando você executar a consulta, o que será apresentado como resultado é o seu apelido.

A ordenação é feita usando o comando `order by`. Na

consulta, informamos que gostaríamos que o resultado fosse ordenado primeiro pelo campo `vl_total_produto` e depois pela coluna `produto`. Por padrão, o PostgreSQL ordena de forma ascendente. Poderíamos ter solicitado uma ordenação de forma descendente. O código ficaria da seguinte maneira:

```
SELECT retorna_nome_produto(produto_id) PRODUTO,
       sum(itens_venda_total) VL_TOTAL_PRODUTO
  FROM itens_venda
 GROUP BY produto_id
 ORDER BY vl_total_produto DESC, produto;
```

Para trabalhar juntamente com a cláusula `group by()`, temos o `having count()`, que vai eliminar as linhas de um agrupamento que você não deseja que seja exibido. Vamos supor que surgiu a necessidade de extrairmos de nossos projetos a quantidade vendida de cada produto. Utilizando somente o `group by()`, podemos fazer uma consulta que nos retornará o nome do produto e contará quantas vezes ele foi vendido. Mão no teclado e vamos criar a seguinte consulta:

```
SELECT retorna_nome_produto(produto_id),
       COUNT(itens_venda_id) QTDE
  FROM itens_venda
 GROUP BY produto_id;
```

Como resultado, teremos:

The screenshot shows a database query interface. At the top, there's a 'Query' tab and a 'Query History' tab. Below that is a code editor containing the following SQL query:

```
1  SELECT retorna_nome_produto(produto_id),  
2      COUNT(itens_venda_id) QTDE  
3  FROM itens_venda  
4  GROUP BY produto_id;
```

Below the code editor is a toolbar with icons for file operations like new, open, save, etc. Underneath the toolbar is a table with three rows of data. The table has two columns: 'retorna\_nome\_produto' (text type) and 'qtde' (bigint type). The data is as follows:

	retorna_nome_produto	qtde
1	AGUA	2
2	SUCO DE LIMÃO	2
3	REFRIGERANTE	2

Figura 5.16: Número de vendas por produto.

Agora que sabemos a quantidade vendida de cada produto, vamos filtrar nela apenas os produtos que tiveram vendas iguais ou superiores a 2. Agora é que entra a cláusula `having count()`. Como usamos o `count()` anteriormente para contar os itens, vamos utilizar o `having count()` em nossa consulta para indicar que queremos apenas os produtos que possuam contagem maior ou igual a 2, e ordenar pela quantidade contada. Vamos ao nosso código.

```
SELECT retorna_nome_produto(produto_id) produto,  
       COUNT(itens_venda_id) qtde  
  FROM itens_venda  
GROUP BY produto_id  
 HAVING COUNT(produto_id) >= 2  
ORDER BY qtde;
```

E como resultado de nossa consulta, teremos:

The screenshot shows a MySQL Workbench interface. In the top left, under the 'Query' tab, is the following SQL code:

```
1  SELECT retorna_nome_produto(produto_id) produto,
2         COUNT(itens_venda_id) qtde
3     FROM itens_venda
4    GROUP BY produto_id
5   HAVING COUNT(produto_id) >= 2
6 ORDER BY qtde;
```

Below the code, under the 'Data output' tab, is a table showing the results:

	produto text	qtde bigint
1	AGUA	2
2	SUCO DE LIMÃO	2
3	REFRIGERANTE	2

Figura 5.17: Having count como forma de filtrar as linhas apresentadas.

Observe que, nessa última consulta, utilizamos o `count()` , `group by()` , `order by` e o `having count()` . Cada vez mais nossas consultas estão ficando mais completas. Agora você poderá usar essas funções conforme a necessidade no dia a dia.

## Funções de formatação

No cotidiano do desenvolvimento de software, podem surgir diversas necessidades, muitas vezes pela regra de negócio em que estamos trabalhando, e muitas vezes precisamos adaptar os nossos dados para os processos funcionarem corretamente. Conforme nosso sistema cresce, não conseguimos definir com exatidão quais os processos que teremos no futuro e quais os tipos de dados com que vamos trabalhar.

Poderemos ter processos nos quais precisaremos converter

dados de texto para o tipo data e hora, tipos numéricos para tipo de texto, registro de tempo para texto, entre outros. Para esses casos, temos funções que fazem essa conversão e tornam nossa vida muito mais feliz. Vamos à lista:

Função	Descrição	Exemplo	Resultado
<code>to_char(hora, formato)</code>	Converte tipo hora para texto.	<code>select to_char(current_timestamp, 'HH12:MI:SS');</code>	07:11:45
<code>to_char(data, formato)</code>	Converte tipo data para texto.	<code>select to_char(current_date, 'DD/MM/YYYY');</code>	23/03/2016
<code>to_char(data, formato)</code>	Converte tipo data/hora para texto.	<code>select to_char(current_timestamp, 'DD/MM/YYYY HH12:MI:SS');</code>	23/03/2016 07:13:46
<code>to_char(número inteiro, quantidade de caracteres do primeiro número substituída por '9')</code>	Converte número do tipo inteiro para texto.	<code>select to_char(23232,'99999');</code>	23232
<code>to_char(número double/real, '999D9')</code>	Converte número do tipo double/real para texto.	<code>select to_char(125.8::real, '999D9');</code>	125,8
<code>to_date(texto, formato)</code>	Converte texto para data.	<code>select to_date('04 Nov 1988', 'DD Mon YYYY');</code>	1988-11-04
<code>to_number(texto, formato)</code>	Converte texto para dado numérico.	<code>select to_number('5215.3', '99G999D9S');</code>	52153
<code>to_timestamp(text, text)</code>	Converte tipo data/hora com fuso horário para texto.	<code>select to_timestamp('04 Nov 1988', 'DD Mon YYYY');</code>	1988-11-04 00:00:00-02

## 5.8 CONSULTAS UTILIZANDO LIKE

Até agora, sempre utilizamos o sinal de = (igual) para verificar uma condição e, na maioria das vezes, usamos números para fazer essa comparação. Mas ainda não havíamos feito uma verificação de consulta com um campo que possuísse registro de algum nome ou com mais de uma palavra. Por exemplo, como no campo `funcionario_nome`, em que temos o nome e sobrenome.

Vamos inserir mais alguns registros para conseguirmos entender melhor a utilização do `like`.

```
INSERT INTO FUNCIONARIO(funcionario_id,
                        funcionario_codigo,
                        funcionario_nome,
                        funcionario_ativo,
                        funcionario_comissao,
                        funcionario_cargo,
                        funcionario_data_criacao)
VALUES(gen_random_uuid(),
       '0100',
       'VINICIUS SOUZA',
       true,
       2,
       'GARÇOM',
       '01/03/2023');
```

```
INSERT INTO FUNCIONARIO(funcionario_id,
                        funcionario_codigo,
                        funcionario_nome,
                        funcionario_ativo,
                        funcionario_comissao,
                        funcionario_cargo,
                        funcionario_data_criacao)
VALUES(gen_random_uuid(),
       '0101',
       'VINICIUS SOUZA MOLIN',
       true,
       2,
```

```
'GARÇOM',
'01/03/2023');

INSERT INTO FUNCIONARIO(funcionario_id,
funcionario_codigo,
funcionario_nome,
funcionario_ativo,
funcionario_comissao,
funcionario_cargo,
funcionario_data_criacao)
VALUES(gen_random_uuid(),
'0102',
'VINICIUS RANKEL C',
true,
2,
'GARÇOM',
'01/03/2023');

INSERT INTO FUNCIONARIO(funcionario_id,
funcionario_codigo,
funcionario_nome,
funcionario_ativo,
funcionario_comissao,
funcionario_cargo,
funcionario_data_criacao)
VALUES(gen_random_uuid(),
'0103',
'BATISTA SOUZA LUIZ',
true,
2,
'GARÇOM',
'01/03/2023');

INSERT INTO FUNCIONARIO(funcionario_id,
funcionario_codigo,
funcionario_nome,
funcionario_ativo,
funcionario_comissao,
funcionario_cargo,
funcionario_data_criacao)
VALUES(gen_random_uuid(),
'0104',
```

```

        'ALBERTO SOUZA CARDOSO',
        true,
        2,
        'GARÇOM',
        '01/03/2023');

INSERT INTO FUNCIONARIO(funcionario_id,
                        funcionario_codigo,
                        funcionario_nome,
                        funcionario_ativo,
                        funcionario_comissao,
                        funcionario_cargo,
                        funcionario_data_criacao)
VALUES(gen_random_uuid(),
       '0105',
       'CARLOS GABRIEL ALMEIDA',
       true,
       2,
       'GARÇOM',
       '01/03/2023');

INSERT INTO FUNCIONARIO(funcionario_id,
                        funcionario_codigo,
                        funcionario_nome,
                        funcionario_ativo,
                        funcionario_comissao,
                        funcionario_cargo,
                        funcionario_data_criacao)
VALUES(gen_random_uuid(),
       '0106',
       'RENAN SIMOES SOUZA',
       true,
       2,
       'GARÇOM',
       '01/03/2023');

```

Se você não inseriu nenhum outro registro no banco, apenas os exemplos que estão no livro, caso consulte o campo `funcionario_nome` na tabela de funcionários, o resultado será:

The screenshot shows a MySQL Workbench interface. At the top, there's a navigation bar with 'Query' and 'Query History'. Below it is a code editor with the following SQL query:

```
1 select funcionario_nome from funcionario
```

Under the code editor, there are tabs for 'Data output', 'Messages', and 'Notifications'. Below these tabs is a toolbar with icons for copy, paste, refresh, and export. The main area displays the results of the query in a table:

	funcionario_nome
1	SOUZA
2	VINICIUS CARVALHO
3	VINICIUS SOUZA
4	VINICIUS SOUZA MOLIN
5	VINICIUS RANKEL C
6	BATISTA SOUZA LUIZ
7	ALBERTO SOUZA CARDOSO
8	CARLOS GABRIEL ALMEIDA
9	RENAN SIMOES SOUZA

Figura 5.18: Consultando com like.

Agora imagine se tivéssemos uma lista com mais de mil nomes de funcionários e quiséssemos consultar apenas aqueles cujo primeiro nome seja igual a `VINICIUS`. Seria um trabalho grande selecionar todos e procurar os códigos desses registros.

Com o `like`, podemos criar uma consulta que buscará todos os registros que iniciam com `VINICIUS` e não levará em consideração a continuação do nome. Mão no teclado e vamos ao nosso código.

```
SELECT funcionario_nome  
FROM funcionario  
WHERE funcionario_nome LIKE 'VINICIUS%';
```

Veja no resultado a seguir que o banco trouxe todos os registros que iniciavam com `VINICIUS`, ignorando todo o restante do nome em cada registro. Isso ocorreu porque, no final

da string , inserimos o caractere % . Quando não sabemos uma parte da string , usamos esse caractere para dizer para o banco de dados que não sabemos o que está escrito depois de VINICIUS , por isso, o banco de dados deverá retornar todos os resultados que contêm VINICIUS .

The screenshot shows a database query interface with the following details:

- Query History:** Shows the query: `SELECT funcionario_nome  
FROM funcionario  
WHERE funcionario_nome like 'VINICIUS%';`
- Data output:** Shows the results of the query in a table format.
- Table Headers:** `funcionario_nome` (character varying (100))
- Table Data:**

	funcionario_nome
1	VINICIUS CARVALHO
2	VINICIUS SOUZA
3	VINICIUS SOUZA MOLIN
4	VINICIUS RANKEL C

Figura 5.19: Resultado da consulta com like.

Também podem ocorrer situações em que não sabemos onde o termo que queremos consultar se encontra. Vamos supor que precisamos consultar todos os nomes que possuem a palavra SOUZA . Não sabemos se está no início, no meio ou no final do nome. Apenas sabemos que há nomes que contêm a palavra SOUZA . Novamente, usaremos o sinal % , só que agora vamos colocar duas vezes. Vamos ao nosso código.

```
SELECT funcionario_nome  
      FROM funcionario  
     WHERE funcionario_nome LIKE '%SOUZA%';
```

The screenshot shows a MySQL Workbench interface. In the 'Query' tab, there is a code editor with the following SQL query:

```
1 SELECT funcionario_nome
2 FROM funcionario
3 WHERE funcionario_nome LIKE '%SOUZA%';
```

Below the code editor, there are tabs for 'Data output', 'Messages', and 'Notifications'. The 'Data output' tab is selected, showing a table with the results of the query. The table has one column labeled 'funcionario\_nome' and six rows of data:

	funcionario_nome
1	SOUZA
2	VINICIUS SOUZA
3	VINICIUS SOUZA MOLIN
4	BATISTA SOUZA LUIZ
5	ALBERTO SOUZA CAR...
6	RENAN SIMOES SOUZA

Figura 5.20: Consultando com like no meio.

Nessa última consulta, não sabíamos onde a palavra ou o termo se encontrava. Apenas sabíamos que existia em algum lugar. Foi isso que fizemos, instruímos o banco de dados para buscar todos os registros que continham a palavra `SOUZA`.

Diferentemente do `like`, quando usamos o sinal de `=`, temos de saber com exatidão o conteúdo do campo. Se não, o banco de dados não vai encontrar resultados na consulta. Você pode até tentar utilizar o sinal `%` com o sinal de igual, só que não terá resultado. Faça a consulta:

```
SELECT funcionario_nome
  FROM funcionario
 WHERE funcionario_nome = 'VINICIUS%';
```

E verá que não trará nenhum registro. Agora utilize a seguinte consulta:

```
SELECT funcionario_nome
```

```
FROM funcionario  
WHERE funcionario_nome = 'VINICIUS SOUZA';
```

Agora deu certo, pois informamos com exatidão a string que o banco deveria procurar. Também não é obrigatório usar o sinal % nas consultas com o like . Você pode utilizá-lo da mesma maneira que o sinal = .

## 5.9 PARA PENSAR!

Conseguimos produzir muitas linhas de código neste capítulo, e o mais legal foi que, em algumas partes, conseguimos inserir assuntos e recursos que aprendemos nos capítulos anteriores. É assim que vamos fixando os assuntos aprendidos. Pegue essas funções que aprendemos e tente aplicar nas outras tabelas que foram criadas no banco de dados. Crie vários outros registros, se necessário.

Lembra dos nossos processos que fazem alguns cálculos? E se pudéssemos executá-los a partir de ações que acontecem em nosso banco de dados de uma forma automática? Seria muito legal, certo? É exatamente isso o que nos espera no próximo capítulo. Vá tomar um café e, em seguida, vire a página e vamos para o próximo capítulo!

## CAPÍTULO 6

# BANCO DE DADOS RÁPIDO NOS GATILHOS

*"Nós somos aquilo que fazemos com frequência. Excelência, então, não é um ato, mas um hábito." — Aristóteles*

## 6.1 TRIGGERS — GATILHOS PARA AGILIZAR TAREFAS

Segundo a documentação oficial do PostgreSQL 9.4, uma trigger é uma instrução ao banco de dados que deve automaticamente executar uma função específica quando uma operação específica for feita. Elas podem ser para tabelas, views e chaves estrangeiras.

Podemos dizer que *triggers*, ou gatilhos, são procedimentos armazenados no banco de dados que utilizamos para disparar ações automaticamente ou realizar uma tarefa automática, como gravar logs de alterações de uma tabela. Podemos pedir para o banco de dados gravar em uma tabela de logs todas as alterações que houver em determinada tabela.

Vamos imaginar uma máquina de lavar roupa, dessas automáticas que lavam, enxaguam e secam. Você apenas a

programa uma vez e ela faz uma operação assim que a outra termina. É exatamente assim que as triggers funcionam: disparam automaticamente uma função quando uma outra termina.

Existem três eventos em que usamos as triggers: na inserção (*insert*), na alteração (*update*) e na exclusão de registros (*delete*). Cada um desses eventos pode ocorrer em dois momentos: antes da execução do evento (*before*) ou depois da execução do evento (*after*). Algo muito legal é que podemos incluir mais de um momento para executar uma trigger. Isso facilita para não termos de repetir o mesmo código várias vezes.

Imagine o cenário do cálculo da comissão do vendedor. A trigger dispararia no evento *insert*, ao inserir o registro de venda, e depois (*after*) do registro da venda ter sido inserido. Simples, não é mesmo? Se ficar na dúvida, lembre-se de que, para inserir a trigger, você precisa escolher em qual evento ela vai disparar e em que momento deve acontecer.

Se ainda está com dúvida, não se preocupe. Vamos exemplificar cada evento e todos os momentos que podemos usar os gatilhos. Para isso, usaremos como base as tabelas que criamos e os registros que inserimos. Vamos criar cenários reais, tudo baseado em nosso projeto.

## 6.2 TRIGGERS INSERT, UPDATE E DELETE

Vamos criar uma trigger que disparará uma função que vai gravar os registros que estão sendo alterados. Vamos pedir para armazenar o registro antigo e os novos.

Para isso, precisaremos criar alguns objetos que aprendemos anteriormente. Criaremos uma tabela para armazenar os logs da tabela de produtos. Nela, vamos incluir todos os mesmos campos da tabela de produtos duas vezes, pois na mesma linha gravaremos o valor antigo do campo e o novo valor. Além da tabela, vamos precisar criar uma função que vai fazer o processo de inserção dos registros nessa nova tabela. Diferente dos tipos de funções que já criamos, essa terá o retorno do tipo trigger.

Em PostgreSQL, um gatilho (trigger) pode executar qualquer função definida pelo usuário em uma de suas linguagens procedurais Java, C, Perl, Python ou TCL, além da linguagem SQL. Em MySQL, gatilhos são ativados por comandos SQL, mas não por APIs, já que estas não transmitem comandos SQL ao servidor MySQL.

A nossa trigger vai se chamar `tri_log_produtos` e será disparada após haver uma inserção, uma alteração ou exclusão de registro na tabela `produtos`. Então, mãos no teclado e vamos ao código.

Primeiro, vamos criar a nossa tabela. A tabela se chamará `log_produto`. Nos campos em que serão armazenados os valores antigos, vamos colocar um sufixo `_old` ao final do nome; e para os campos nos quais serão armazenados os valores novos, colocaremos o sufixo `_new` ao final do nome. Essa é uma convenção que eu, particularmente, gosto de usar. Você pode criar a sua. :)

Vou criar uma coluna chamada `alteracao` em nossa tabela. Ela vai armazenar o tipo de operação que foi feita na tabela, ou seja, gravará se foi uma inserção, uma alteração ou uma exclusão

de registros. Isso é possível pois, durante a execução da function que retorna um tipo trigger, são criadas variáveis na memória que armazena os valores antigos e novos, além de armazenar o tipo da operação executada.

A variável que vai nos retornar o tipo da ação é a `TG_OP`, e os valores antigos e novos são as variáveis `old.nome_coluna` e `new.nome_coluna`, respectivamente.

```
CREATE TABLE IF NOT EXISTS log_produto(
    log_produto_id          INT NOT NULL PRIMARY KEY,
    log_produto_data_alteracao TIMESTAMP,
    log_produto_alteracao    VARCHAR(10),
    produto_id_old           UUID,
    produto_codigo_old        VARCHAR(20),
    produto_nome_old          VARCHAR(60),
    produto_valor_old         DOUBLE PRECISION,
    produto_ativo_old         BOOLEAN,
    produto_data_criacao_old TIMESTAMP,
    produto_data_alteracao_old TIMESTAMP,
    produto_id_new            UUID,
    produto_codigo_new         VARCHAR(20),
    produto_nome_new           VARCHAR(60),
    produto_valor_new          DOUBLE PRECISION,
    produto_ativo_new          BOOLEAN,
    produto_data_criacao_new  TIMESTAMP,
    produto_data_alteracao_new TIMESTAMP);
```

Vamos criar uma sequence para a nossa tabela de logs.

```
CREATE SEQUENCE log_produto_id_seq;
```

Agora vamos vincular a sequence à coluna `log_produto_id` da tabela `log_produto`.

```
ALTER TABLE log_produto
    ALTER COLUMN log_produto_id SET DEFAULT
        nextval('log_produto_id_seq');
```

Pronto! A tabela está pronta para receber os logs de alterações

da tabela `produto`. Vamos agora criar a função e o nosso gatilho.

Primeiro, vamos criar a function, mas antes devemos fazer uma análise da situação. Nós usaremos a mesma trigger para o `insert`, `update` e `delete`, correto? Sim. E queremos armazenar o valor antigo do campo e o valor novo. Mas, se analisarmos, nós temos o valor anterior e o novo para um determinado campo (por exemplo, quando fazemos um `update`); se vamos inserir um registro, não temos um valor antigo, apenas o novo. E se fizermos um `delete`, não teremos um valor novo, apenas o antigo, já que o registro deixa de existir.

Pois bem, pensando nisso, vamos fazer um tratamento utilizando os condicionais que aprendemos, o `if` e o `else`. Se não fizermos o tratamento ao consultar as variáveis criadas em tempo de execução, o SGBD não encontrará o registro e ocorrerá um erro, dizendo que a variável não possui valor. Mão no teclado e vamos ao código.

```
CREATE OR REPLACE FUNCTION gera_log_produto()
RETURNS TRIGGER AS
$$
BEGIN
    IF TG_OP = 'INSERT' THEN
        INSERT INTO log_produto (
            log_produto_data_alteracao,
            log_produto_alteracao,
            produto_id_new,
            produto_codigo_new,
            produto_nome_new,
            produto_valor_new,
            produto_ativo_new,
            produto_data_criacao_new,
            produto_data_alteracao_new)
```

```

VALUES(
    NOW(),
    TG_OP,
    NEW.producto_id,
    NEW.producto_codigo,
    NEW.producto_nome,
    NEW.producto_valor,
    NEW.producto_ativo,
    NEW.producto_data_criacao,
    NEW.producto_data_atualizacao);

RETURN NEW;

ELSIF TG_OP = 'UPDATE' THEN

    INSERT INTO log_produto(
        log_produto_data_alteracao,
        log_produto_alteracao,
        produto_id_old,
        produto_codigo_old,
        produto_nome_old,
        produto_valor_old,
        produto_ativo_old,
        produto_data_criacao_old,
        produto_data_alteracao_old,
        produto_id_new,
        produto_codigo_new,
        produto_nome_new,
        produto_valor_new,
        produto_ativo_new,
        produto_data_criacao_new,
        produto_data_alteracao_new)
VALUES(
    NOW(),
    TG_OP,
    OLD.producto_id,
    OLD.producto_codigo,
    OLD.producto_nome,
    OLD.producto_valor,
    OLD.producto_ativo,
    OLD.producto_data_criacao,
    OLD.producto_data_atualizacao,
    NEW.producto_id,
    NEW.producto_codigo,
    NEW.producto_nome,

```

```

        NEW.producto_valor,
        NEW.producto_ativo,
        NEW.producto_data_criacao,
        NEW.producto_data_atualizacao);

    RETURN NEW;

ELSIF TG_OP = 'DELETE' THEN

    INSERT INTO log_produto (
        log_produto_data_alteracao,
        log_produto_alteracao,
        produto_id_new,
        produto_codigo_new,
        produto_nome_new,
        produto_valor_new,
        produto_ativo_new,
        produto_data_criacao_new,
        produto_data_alteracao_new)
VALUES(
        NOW(),
        TG_OP,
        OLD.producto_id,
        OLD.producto_codigo,
        OLD.producto_nome,
        OLD.producto_valor,
        OLD.producto_ativo,
        OLD.producto_data_criacao,
        OLD.producto_data_atualizacao);

    RETURN NEW;
END IF;
END;
$$
LANGUAGE 'plpgsql';

```

Com essa function, o banco de dados vai verificar a operação que foi executada e, com isso, fazer o insert na tabela de log, inserindo a operação que foi executada.

E agora, finalmente, criaremos a trigger .

```
CREATE TRIGGER tri_log_produto
```

```
AFTER INSERT OR UPDATE OR DELETE ON produto
FOR EACH ROW EXECUTE
PROCEDURE gera_log_produto();
```

Para sabermos que a trigger está criada corretamente na tabela que desejamos, podemos listar os objetos dessa tabela.

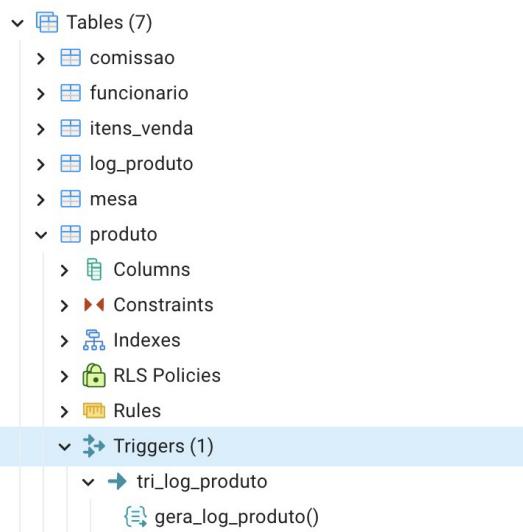


Figura 6.1: Os itens da tabela de produtos.

Agora que criamos todo o fluxo para gravarmos os logs da tabela produtos, vamos realizar alterações nela que possibilitem a visualização dos registros nessa nova tabela.

Vamos inserir três novos produtos. Fique à vontade para criar quantos registros você quiser. Quanto mais, melhor!

```
INSERT INTO produto
VALUES (gen_random_uuid(),
        '1512',
        'LASANHA',
        46,
```

```
true,  
'01/01/2023',  
'01/01/2023');
```

```
INSERT INTO produto  
VALUES (gen_random_uuid(),  
'1613',  
'PANQUECA',  
38,  
true,  
'01/01/2023',  
'01/01/2023');
```

```
INSERT INTO produto  
VALUES (gen_random_uuid(),  
'733',  
'CHURRASCO',  
72,  
true,  
'01/01/2023',  
'01/01/2023');
```

A inserção de dados na tabela produtos já deve ter gerado logs nela. Vamos verificar. Como sabemos as colunas que tiveram inserção de dados, vamos consultar apenas elas.

```
SELECT log_produto_alteracao  
,log_produto_data_alteracao  
,produto_id_new  
,produto_codigo_new  
,produto_nome_new  
,produto_valor_new  
,produto_ativo_new  
,produto_data_criacao_new  
,produto_data_alteracao_new  
FROM log_produto;
```

```

db_restaurant/postgres@ServerPostgre
No limit
Query History
Query
SELECT log_produto_alteracao
     ,log_produto_data_alteracao
     ,produto_id_new
     ,produto_codigo_new
     ,produto_nome_new
     ,produto_valor_new
     ,produto_ativo_new
     ,produto_data_criacao_new
     ,produto_data_alteracao_new
  FROM log_produto;

```

	log_produto_alteracao character varying (10)	log_produto_data_alteracao timestamp without time zone	produto_id_new uuid	produto_codigo_new character varying (20)	produto_nome_new character varying (60)
1	INSERT	2022-10-17 18:53:16.438127	2f6f62af-5f10-4...	1512	LASANHA
2	INSERT	2022-10-17 18:53:50.586175	904c2294-7fef...	1613	PANQUECA
3	INSERT	2022-10-17 18:53:52.565969	3054f5e7-81bc...	733	CHURRASCO

Figura 6.2: Os primeiros logs.

Observe que o campo `alteracao` gravou corretamente o tipo da operação que realizamos, um `insert` na tabela. Mas agora vamos fazer uma alteração na tabela `produto` e atualizar o preço do 'CHURRASCO' .

```
UPDATE produto SET produto_valor = 99
WHERE produto_nome = 'CHURRASCO';
```

Como sabemos que o `update` realiza uma inserção em todos os campos da tabela de logs, vamos realizar uma consulta completa na tabela.

```
SELECT *
  FROM log_produto;
```

The screenshot shows a pgAdmin interface with a query window and a results table.

```

db_restaurant/postgres@ServerPostgre
Query History
Query
1 SELECT *
2 FROM log_produto;

```

Data output

	log_produto_id	log_produto_data_alteracao	log_produto_alteracao	produto_id_old	produto_codigo_old	produto_nome_old
1	1	2022-10-17 18:53:16.438127	INSERT	[null]	[null]	[null]
2	2	2022-10-17 18:53:50.586175	INSERT	[null]	[null]	[null]
3	3	2022-10-17 18:53:52.565969	INSERT	[null]	[null]	[null]
4	4	2022-10-17 18:57:53.468896	UPDATE	3054f5e7-81bc...	733	CHURRASCO

Figura 6.3: Logs de inserção (insert) e alteração (update).

Agora que fizemos o `insert` e o `update`, só nos restou deletar um registro para visualizarmos como ficarão os logs da tabela de produtos. Assim, vamos deletar a 'PANQUECA' do nosso sistema.

```
DELETE FROM produto WHERE produto_nome = 'PANQUECA';
```

Executando a mesma consulta:

```
SELECT *
FROM log_produto;
```

The screenshot shows a pgAdmin interface with a query window and a results table.

```

Query History
Query
1 SELECT *
2 FROM log_produto;

```

Data output

	produto_data_alteracao_old	produto_id_new	produto_codigo_new	produto_nome_new	produto_valor_new
1	[null]	2f6f62af-5f10-4...	1512	LASANHA	46
2	[null]	904c2294-7fef-...	1613	PANQUECA	38
3	[null]	3054f5e7-81bc...	733	CHURRASCO	72
4	2023-01-01 00:00:00	3054f5e7-81bc...	733	CHURRASCO	99
5	[null]	904c2294-7fef...	1613	PANQUECA	38

Figura 6.4: Logs de inserção (insert), alteração (update) e exclusão (delete).

A partir de agora, todas as alterações realizadas na tabela de produtos serão gravadas nessa tabela de logs. Existem muitos outros modelos para armazenar logs de uma tabela. Este pode não ser o mais elegante, mas para iniciarmos um projeto já é o suficiente.

Se utilizarmos o comando `truncate` para excluir os registros de uma tabela, ele não vai disparar as triggers que estiverem configuradas para disparar `on delete`, pois o `truncate` ignora qualquer trigger .

## 6.3 DESABILITANDO, HABILITANDO E DELETANDO UMA TRIGGER

As regras de sistemas estão sempre mudando, e nós sempre devemos nos adequar a elas. É muito comum, por mudança de regra ou até por uma necessidade de manutenção do sistema, surgir a necessidade de não usar uma trigger por um período sem precisarmos excluí-la. Por isso, temos as possibilidades de desabilitar e habilitar uma trigger, além de deletá-la.

Vamos desabilitar a trigger que criamos:

```
ALTER TABLE produto  
DISABLE trigger tri_log_produto;
```

Vamos inserir um registro em nossa tabela para verificar se a trigger está realmente desabilitada.

```
INSERT INTO produto(produto_id,  
produto_codigo,
```

```

        produto_nome,
        produto_valor,
        produto_ativo,
        produto_data_criacao,
        produto_data_atualizacao)
VALUES (gen_random_uuid(),
        '912',
        'SORVETE',
        6,
        true,
        '01/01/2023',
        '01/01/2023');

```

Agora, se consultarmos a nossa tabela de logs, verificaremos que não houve inserção de registros na tabela de produtos.

```

SELECT *
FROM log_produto;

```

	produto_data_alteracao_old timestamp without time zone	produto_id_new uuid	produto_codigo_new character varying (20)	produto_nome_new character varying (60)	produto_valor_new double precision
1	[null]	2f6f62af-5f10-4...	1512	LASANHA	46
2	[null]	904c2294-7fef...	1613	PANQUECA	38
3	[null]	3054f5e7-81bc...	733	CHURRASCO	72
4	2023-01-01 00:00:00	3054f5e7-81bc...	733	CHURRASCO	99
5	[null]	904c2294-7fef...	1613	PANQUECA	38

Figura 6.5: O banco não registrou a inserção, pois a trigger está desabilitada.

Muito cuidado ao deixar uma trigger desabilitada, porque, se ela for importante, como a que salva o log de uma determinada tabela, podemos perder o rastreamento das alterações.

Agora, vamos habilitar a trigger novamente:

```
ALTER TABLE produto
    ENABLE trigger tri_log_produto;
```

Vamos alterar o preço do 'SORVETE' .

```
UPDATE produto SET produto_valor = 10
WHERE produto_nome = 'SORVETE';
```

Consultando novamente:

```
SELECT *
FROM log_produto;
```

Data output						
	log_produto_id [PK] integer	log_produto_data_alteracao timestamp without time zone	log_produto_alteracao character varying (10)	produto_id_old uuid	produto_codigo_old character varying (20)	produto_nome_old character varying (60)
1	1	2022-10-17 18:53:16.438127	INSERT	[null]	[null]	[null]
2	2	2022-10-17 18:53:50.586175	INSERT	[null]	[null]	[null]
3	3	2022-10-17 18:53:52.565969	INSERT	[null]	[null]	[null]
4	4	2022-10-17 18:57:53.468896	UPDATE	3054f5e7-81bc-47dd-98d5-43c1ae4d6...	733	CHURBASCO
5	5	2022-10-17 18:59:26.004357	DELETE	[null]	[null]	[null]
6	6	2022-10-18 11:22:12.332598	UPDATE	a9c39a53-cb11-4b92-a645-321546f51...	912	SORVETE

Figura 6.6: Trigger voltando a gravar as alterações.

Nossa trigger voltou a funcionar. Agora, se você desejar realmente deletar a trigger, você pode usar o seguinte comando:

```
DROP TRIGGER tri_log_produto ON produto;
```

Se verificarmos os itens da tabela, veremos que ela não consta mais.

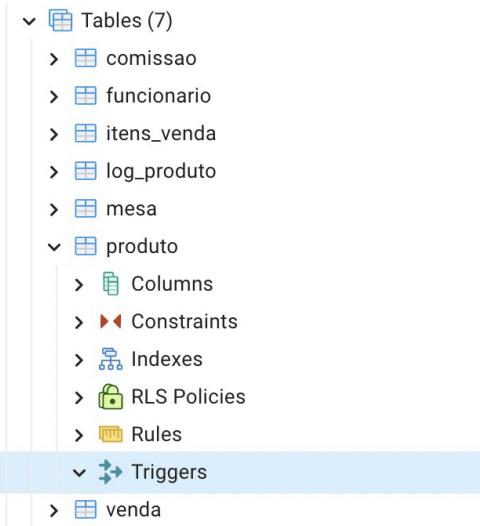


Figura 6.7: Ausência da trigger.

Para excluir uma trigger, tenha certeza de que ela não será mais necessária, uma vez que a exclusão pode prejudicar desde processos simples, como a gravação de logs, até mesmo cálculos que são executados por esses gatilhos. Por exemplo, imagine que você tem uma trigger que grava os logs de todas as transações do banco de dados. Se você a excluir, o processo vai parar de ser executado e, consequentemente, os logs vão parar de ser armazenados.

## 6.4 PARA PENSAR!

Logs são muito importantes para um sistema, principalmente para a segurança, já que é uma das maneiras de saber quem alterou algo no sistema e até mesmo de restaurar registros deletados indevidamente. Você agora pode criar uma tabela de log para cada

tabela que criamos em nosso projeto. É uma maneira de você colocar em prática o que aprendeu neste capítulo.

Além de logs, é possível automatizar outros processos no sistema, mas isso vai da sua imaginação. Poderíamos, por exemplo, criar uma trigger que faça os cálculos de comissão dos vendedores após a inserção de um registro. Podemos também criar triggers para validação de informações que estão sendo inseridas utilizando as variáveis criadas em tempo de execução.

Agora você já sabe criar e deletar triggers. Delete as que criamos e tente criá-las sem olhar o código que fizemos. No começo, pode colar um pouco — mas só um pouco! ;)

Os gatilhos nos ajudam bastante, pois disparam processos automaticamente após eventos que ocorrem em determinada tabela. Já conseguimos fazer diversas coisas no banco de dados. Mas, até agora, criamos apenas consultas simples. Já está na hora de criarmos algumas consultas mais complexas.

*#PartiuPróximoCapítulo*

## CAPÍTULO 7

# TURBINANDO AS CONSULTAS COM JOINS E VIEWS

*"Eu sempre fiz alguma coisa para a qual eu não estava muito pronta. Acho que é assim que você cresce: quando há aquele momento de 'Uau, eu não tenho tanta certeza de que posso fazer isso'. E aí você insiste nesses momentos; é aí que você tem um progresso." — Marissa Mayer*

## 7.1 SUBCONSULTAS

Existem algumas formas de se fazer consultas para extrair dados de uma ou mais consultas, seja por meio de consultas simples ou de funções que retornam dados. Entretanto, muitas vezes, precisamos criar ligações entre as tabelas em forma de dependência.

Por exemplo, se quisermos criar um `select` para consultar os funcionários relacionados com uma ou mais vendas, já saberíamos criar as duas consultas: uma para buscar os funcionários e outra para buscar as vendas. Mas com essas duas consultas separadas não conseguimos extrair valor para o nosso projeto, uma vez que

precisaríamos colocar o resultado delas em uma planilha, a fim de fazer algum relacionamento — e isso está fora de cogitação! :)

Uma maneira de relacionar duas tabelas é por meio de uma subconsulta, na qual buscaremos os funcionários que possuem vínculo com uma ou mais vendas.

```
SELECT funcionario_nome  
      FROM funcionario  
 WHERE funcionario_id IN (SELECT funcionario_id  
                           FROM venda);
```

The screenshot shows a PostgreSQL query editor interface. At the top, there are tabs for 'Query' and 'Query History'. Below them, a code editor displays the following SQL query:

```
1  SELECT funcionario_nome  
2    FROM funcionario  
3   WHERE funcionario_id IN (SELECT funcionario_id  
4                                FROM venda);
```

Below the code editor, there are tabs for 'Data output', 'Messages', and 'Notifications'. Under 'Data output', there is a table with two rows of data:

	funcionario_nome
1	SOUZA
2	VINICIUS CARVALHO

Figura 7.1: Consulta com subconsulta.

As subconsultas são úteis em diversas situações; no entanto, não são tão performáticas, uma vez que, se não passarmos um parâmetro para ela, ela faz uma busca completa na subtabela para passar um parâmetro para a consulta externa, fazendo com que a consulta seja lenta. Isso foi mostrado no exemplo, no qual não foi passado um parâmetro para a subconsulta e o banco teve de realizar uma busca em toda a tabela de vendas para retornar os funcionários.

Poderíamos realizar a mesma consulta apenas buscando as vendas realizadas em 2016; assim, restringiríamos um pouco a busca e melhoraríamos a sua performance, ainda não com o cenário ideal. Vamos utilizar uma função ensinada neste livro para extrair apenas o ano da data atual e passar como parâmetro.

```
SELECT funcionario_nome
  FROM funcionario
 WHERE funcionario_id IN (SELECT funcionario_id
                            FROM venda
                           WHERE date_part('year', venda_data_criacao) = '2023');
```

The screenshot shows a PostgreSQL query editor interface. At the top, there are tabs for 'Query' and 'Query History'. Below the tabs is the SQL query:

```
1 SELECT funcionario_nome
2   FROM funcionario
3 WHERE funcionario_id IN (SELECT funcionario_id
4                            FROM venda
5                           WHERE date_part('year', venda_data_criacao) = '2023');
```

Below the query area, there are tabs for 'Data output', 'Messages', and 'Notifications'. Under 'Data output', there is a table with two rows:

	funcionario_nome
1	SOUZA
2	VINICIUS CARVALHO

Figura 7.2: Consulta com subconsulta e parâmetros.

Observe que foram trazidos alguns funcionários repetidos como resultado. Isso acontece pois estamos retornando cada venda que o funcionário teve e pedindo para selecionar o nome dele. Como o nosso interesse é apenas exibir o nome dos funcionários, podemos pedir para o banco exibir os nomes distintos. Fazemos isso usando o `DISTINCT` antes da coluna que estamos selecionando. Além disso, vamos ordenar em ordem alfabética — isso é fácil, pois já aprendemos.

```
SELECT DISTINCT funcionario_nome
    FROM funcionario
   WHERE funcionario_id IN (SELECT funcionario_id
                             FROM venda)
        ORDER BY funcionario_nome;
```

The screenshot shows a PostgreSQL query editor interface. At the top, there are tabs for 'Query' (which is selected) and 'Query History'. Below the tabs is the SQL query:

```
1 SELECT DISTINCT funcionario_nome
2      FROM funcionario
3     WHERE funcionario_id IN (SELECT funcionario_id
4                                FROM venda)
5        ORDER BY funcionario_nome;
```

Below the query area are three tabs: 'Data output' (selected), 'Messages', and 'Notifications'. Under 'Data output', there is a table with one column 'funcionario\_nome' and two rows: 'SOUZA' and 'VINICIUS CARVALHO'. Above the table are several icons for file operations like copy, paste, and download.

funcionario_nome
character varying (100)
1 SOUZA
2 VINICIUS CARVALHO

Figura 7.3: Consulta com subconsulta agrupando registros sem repetição.

Uma maneira mais performática para realizar o relacionamento entre tabelas é pelas *joins*, ou no bom e velho português, *junções*.

## 7.2 CONSULTAS ENTRE DUAS OU MAIS TABELAS ATRAVÉS DAS JOINS

Criando *joins*, não precisamos realizar subconsultas e fazer o relacionamento direto entre as tabelas. Vamos recriar a mesma consulta, mas desta vez usaremos uma *join* para construí-la.

```
SELECT DISTINCT funcionario_nome
    FROM funcionario, venda
   WHERE funcionario.id = venda.funcionario_id
        ORDER BY funcionario_nome;
```

The screenshot shows a database interface with a query editor and a results viewer. The query editor contains the following SQL code:

```
1 SELECT DISTINCT funcionario.nome
2      FROM funcionario, venda
3     WHERE funcionario.funcionario_id = venda.funcionario_id
4 ORDER BY funcionario.nome;
```

The results viewer shows the output of the query:

	funcionario.nome
1	SOUZA
2	VINICIUS CARVALHO

Figura 7.4: Consulta com join.

Veja que fizemos uma consulta em duas tabelas simultaneamente e instruímos ao banco para comparar o `id` do funcionário de ambas as tabelas da consulta. Fazemos isso para manter a integridade das buscas, uma vez que, se não fizermos esse relacionamento, o SGBD se perde. Vamos fazer a seguinte consulta:

```
SELECT DISTINCT funcionario.nome
              FROM funcionario, venda
            ORDER BY funcionario.nome;
```

The screenshot shows a database query interface. At the top, there's a 'Query History' tab and a query editor with the following SQL code:

```
1 SELECT DISTINCT funcionario_nome
2      FROM funcionario, venda
3     ORDER BY funcionario_nome;
```

Below the query editor, there are tabs for 'Data output', 'Messages', and 'Notifications'. The 'Data output' tab is selected, displaying a table with the results of the query:

	funcionario_nome
1	ALBERTO SOUZA CARDOSO
2	BATISTA SOUZA LUIZ
3	CARLOS GABRIEL ALMEIDA
4	RENAN SIMOES SOUZA
5	SOUZA
6	VINICIUS CARVALHO
7	VINICIUS RANKEL C
8	VINICIUS SOUZA
9	VINICIUS SOUZA MOLIN

Figura 7.5: Consulta com join distinguindo os registros e os ordenando.

Quando houver relacionamento entre tabelas, não podemos esquecer a igualdade entre os campos que as relacionam.

Essa é a forma mais popular para se escrever uma `join` e o jeito mais simples para ler as consultas. Eu, particularmente, prefiro escrevê-las dessa maneira. Você verá nos exemplos a seguir como escrever `joins` de maneiras mais tradicionais. Ambos os jeitos estão corretos.

Alguns dizem que, da maneira tradicional, há um ganho de performance, mas muitos autores divergem a respeito disso. E se houver ganho de performance, esse ganho será mínimo. Como eu sempre comento sobre programação, cada um deve avaliar o contexto em que está trabalhando e escolher a maneira que melhor se encaixa.

## Inner join

Utilizando a sintaxe `inner join`, teremos a mesma consulta realizada anteriormente, com a diferença no modo de sua escrita. Vamos repetir a última consulta, só que desta vez usando a sintaxe do `inner join`.

```
SELECT DISTINCT funcionario_nome  
    FROM funcionario  
    INNER JOIN venda  
        ON (funcionario.funcionario_id = venda.funcionario_i  
d)  
            ORDER BY funcionario_nome;
```

The screenshot shows a PostgreSQL query editor interface. At the top, there are tabs for 'Query' (which is selected) and 'Query History'. Below the tabs is a code editor containing the SQL query shown above. Underneath the code editor is a toolbar with icons for file operations like new, open, save, and copy/paste. The main area displays the results of the query. A table header row shows the column name 'funcionario\_nome' and its type 'character varying (100)'. Below this, two rows of data are listed: '1 SOUZA' and '2 VINICIUS CARVALHO'.

	funcionario_nome
1	SOUZA
2	VINICIUS CARVALHO

Figura 7.6: Consulta utilizando inner join.

Observe que obtivemos o mesmo resultado. Tivemos uma diferença apenas na sintaxe, e podemos simplificar ainda mais. Podemos escrever apenas `join` e teremos o mesmo resultado.

```
SELECT DISTINCT funcionario_nome  
    FROM funcionario  
    JOIN venda  
        ON (funcionario.funcionario_id = venda.funcionario_i  
d)  
            ORDER BY funcionario_nome;
```

The screenshot shows a PostgreSQL query editor interface. At the top, there's a tab bar with 'Query' (which is selected) and 'Query History'. Below the tabs is a code editor containing the following SQL query:

```
1 SELECT DISTINCT funcionario_nome
2      FROM funcionario
3      JOIN venda
4        ON (funcionario.funcionario_id = venda.funcionario_id)
5   ORDER BY funcionario_nome;
```

Below the code editor is a navigation bar with tabs: 'Data output' (selected), 'Messages', and 'Notifications'. Underneath the navigation bar is a toolbar with icons for copy, paste, refresh, and download. The main area displays the results of the query in a table:

	funcionario_nome
1	SOUZA
2	VINICIUS CARVALHO

Figura 7.7: Consulta utilizando join.

As duas maneiras estão corretas e correspondem à mesma consulta. Apenas teremos diferença no resultado quando utilizamos o `outer join`, que veremos a seguir.

## Outer join

Diferentemente do `inner join`, o `outer join` possui dois tipos: temos o `left outer join` e o `right outer join`. Em ambos os casos, o SGBD vai retornar todos os campos da tabela à esquerda (quando utilizado o `left outer join`), ou da tabela à direita (quando usado o `right outer join`).

### Left outer join ou left join

Utilizando o `left outer join`, o SGBD realizará uma junção interna e, para cada linha listada da primeira tabela que não satisfizer a condição de relacionamento com a segunda tabela, vai ser adicionada uma linha juntada com valores nulos nas colunas da segunda tabela. Com isso, o resultado de nossa consulta possuirá, no mínimo, uma linha para cada linha da primeira tabela.

Vamos criar uma consulta baseada na usada anteriormente para ficar mais claro de visualizarmos esse cenário.

```
SELECT funcionario_nome, v.venda_id
      FROM funcionario f
LEFT JOIN venda v
        ON f.funcionario_id = v.funcionario_id
ORDER BY funcionario_nome DESC;
```

The screenshot shows a PostgreSQL query interface. At the top, there's a 'Query History' section with the same SQL code as above. Below it is a 'Data output' tab, which is selected, followed by 'Messages' and 'Notifications'. Under 'Data output', there's a toolbar with icons for new query, copy, paste, refresh, download, and search. The main area displays a table with two columns: 'funcionario\_nome' and 'venda\_id'. The 'venda\_id' column is highlighted with a blue border. The data consists of 12 rows, each containing a name from the 'funcionario' table and either a null value or a UUID from the 'venda' table.

	funcionario_nome character varying (100)	venda_id uuid
1	VINICIUS SOUZA MOLIN	[null]
2	VINICIUS SOUZA	[null]
3	VINICIUS RANKEL C	[null]
4	VINICIUS CARVALHO	fa9010d8-ef7e-4191-bb4b-7e8d4ac501cd
5	SOUZA	04187a8a-a1e3-4ae2-b869-23928f2de70e
6	SOUZA	78ea20e5-a5ff-48a5-b6e2-5254cbf2653d
7	SOUZA	f14abea0-d9e5-469e-9557-e52a5c79e151
8	SOUZA	da80d450-235a-4c02-af3b-e3adc43cd14f
9	RENAN SIMOES SOUZA	[null]
10	CARLOS GABRIEL ALMEIDA	[null]
11	BATISTA SOUZA LUIZ	[null]
12	ALBERTO SOUZA CARDOSO	[null]

Figura 7.8: Consulta utilizando left outer join ou left join.

Analisando o resultado de nossa consulta, podemos observar que ela trouxe todos os registros da tabela `funcionario` para as linhas que satisfazem a igualdade, e trouxe o valor na coluna `v.venda_id`, que é o identificador da venda. E para os funcionários que não possuem nenhum registro na tabela `venda`,

a consulta trouxe em branco.

### Right outer join ou right join

Agora vamos fazer o contrário e buscar as vendas fazendo uma junção externa com a tabela de funcionários. O SGBD vai utilizar o mesmo critério para apresentar a busca, só que desta vez buscará todas as vendas e verificará qual possui vínculo com o funcionário.

```
SELECT v.venda_id, v.venda_total, funcionario_nome  
      FROM vendas v  
RIGHT JOIN funcionarios f  
        ON v.funcionario_id = f.id  
ORDER BY v.venda_total;
```

The screenshot shows a database query interface with two tabs: 'Query' and 'Query History'. The 'Query' tab contains the SQL code for a right outer join. The 'Data output' tab is selected, displaying the results of the query in a table format.

venda_id	uuid	venda_total	funcionario_nome
1	78ea20e5-a5ff-48a5-b6e2-5254cbf265...	20	SOUZA
2	da80d450-235a-4c02-af3b-e3adc43cd...	20	SOUZA
3	fa9010d8-ef7e-4191-bb4b-7e8d4ac501...	20	VINICIUS CARVALHO
4	04187a8a-a1e3-4ae2-b869-23928f2de...	45	SOUZA
5	f1abaea0-d9e5-469e-9557-e52a5c79e1...	51	SOUZA
6	[null]	[null]	CARLOS GABRIEL ALMEI...
7	[null]	[null]	BATISTA SOUZA LUIZ
8	[null]	[null]	RENAN SIMOES SOUZA
9	[null]	[null]	VINICIUS RANKEL C
10	[null]	[null]	VINICIUS SOUZA
11	[null]	[null]	ALBERTO SOUZA CARDO...
12	[null]	[null]	VINICIUS SOUZA MOLIN

Figura 7.9: Consulta utilizando right outer join ou right join.

Se você ainda não trabalha com programação, não se preocupe em decorar cada um deles, pois você acaba absorvendo a construção das consultas de forma natural. E por mais que tenha todos esses tipos de `joins` , acabamos utilizando mais a primeira forma de escrita, uma vez que a leitura das consultas fica mais simples e consegue-se o mesmo resultado que com os outros tipos de junções.

## 7.3 VIEWS

Você percebeu como pode ser comum fazermos uma mesma consulta diversas vezes? Pense como o(a) cliente que está contratando você para desenvolver o sistema: pode ser muito comum em seu dia a dia querer consultar os funcionários que estão vinculados às vendas, ou saber qual o seu produto mais vendido.

Sabendo disso, melhor do que criarmos uma mesma consulta em diversos lugares do sistema ou diversas vezes, é criar uma visão estática da consulta no banco de dados, em forma de um objeto. Assim, sempre que quisermos o resultado que ela fornece, nós fazemos a consulta em cima da `view` , e não diretamente com as tabelas.

Vamos criar uma visão que nos trará os produtos mais vendidos no dia por ordem alfabética e por ordem de maior venda. Para trazer as vendas que tivemos no dia, usaremos a função `current_date` , mas como provavelmente não temos vendas no dia em que você estará fazendo essa consulta, vou colocar nela a data que coloquei nos `scripts` de inserção de dados. Agora fique à vontade em inserir dados e substituir a data da consulta pela

função que retorna a data atual.

```
CREATE OR REPLACE VIEW vendas_do_dia AS
SELECT DISTINCT produto_nome
, SUM(venda.venda_total)
FROM produto, itens_venda, venda
WHERE produto.produto_id = itens_venda.produto_id
AND venda.venda_id = itens_venda.venda_id
AND venda.venda_data_criacao = '01/01/2023'
GROUP BY produto_nome;
```

Agora que temos essa visão criada em nosso banco de dados, podemos consultá-la quando desejarmos, da seguinte maneira:

```
SELECT * FROM vendas_do_dia;
```

The screenshot shows a database interface with two tabs: 'Query' (selected) and 'Query History'. In the 'Query' tab, there is a single line of SQL code: 'SELECT \* FROM vendas\_do\_dia;'. Below the interface, the results of the query are displayed in a table titled 'Data output'.

	produto_nome	sum
1	AGUA	71
2	REFRIGERANTE	40
3	SUCO DE LIMAO	96

Figura 7.10: Resultado da consulta da view.

Podemos ainda fazer consultas em nossa view com outras cláusulas, como buscar se um determinado produto foi vendido no dia inserido na visão.

```
SELECT *
FROM vendas_do_dia
```

```

WHERE produto_nome = 'AGUA';

```

Query    Query History

```

1  SELECT *
2    FROM vendas_do_dia
3 WHERE produto_nome = 'AGUA';

```

Data output    Messages    Notifications

	produto_nome	sum
	character varying (60)	double precision
1	AGUA	71

Figura 7.11: Adicionando cláusula em consulta com views.

Podemos também criar uma visão que traria diversos campos e você poderia passar outros parâmetros.

```

CREATE OR REPLACE VIEW produto_venda AS
  SELECT produto.producto_id PRODUTO_ID
        , produto.producto_nome PRODUTO_NOME
        , venda.venda_id VENDA_ID
        , itens_venda.itens_venda_id ITEM_ID
        , itens_venda.itens_venda_valor ITEM_VALOR
        , venda.venda_data_criacao DATA_CRIACAO
   FROM produto, venda, itens_venda
 WHERE venda.venda_id = itens_venda.venda_id
   AND produto.producto_id = itens_venda.producto_id
 ORDER BY data_criacao DESC;

```

Agora podemos consultar qualquer uma das colunas que estão contidas na view e também realizar comparações com elas.

```

SELECT *
  FROM produto_venda;

```

Query    Query History

```
1 SELECT *
2   FROM produto_venda;
```

Data output    Messages    Notifications

	produto_id uuid	produto_nome character varying (60)	venda_id uuid	item_id uuid	item_valor double precision	data_criacao timestamp without time zone
1	86b46f1e...	AGUA	da80d450...	6422b30...	7	2023-01-01 00:00:00
2	e8f1f733-0...	REFRIGERANTE	fa9010d8...	9b481f66...	10	2023-01-01 00:00:00
3	aac651ac...	SUCO DE LIMÃO	f14abea0...	0c3be8b8...	15	2023-01-01 00:00:00
4	86b46f1e...	AGUA	f14abea0...	010fc78d...	7	2023-01-01 00:00:00
5	e8f1f733-0...	REFRIGERANTE	78ea20e5...	b9a544cf...	10	2023-01-01 00:00:00
6	aac651ac...	SUCO DE LIMÃO	04187a8a...	80f5cfa8...	15	2023-01-01 00:00:00

Figura 7.12: Uma view com diversos campos.

```
SELECT produto_nome
  FROM produto_venda
 WHERE data_criacao = '01/01/2023';
```

Query    Query History

```
1 SELECT produto_nome
2   FROM produto_venda
3 WHERE data_criacao = '01/01/2023';
```

Data output    Messages    Notifications

	produto_nome character varying (60)
1	AGUA
2	REFRIGERANTE
3	SUCO DE LIMÃO
4	AGUA
5	REFRIGERANTE
6	SUCO DE LIMÃO

Figura 7.13: Selecionando apenas um campo da view com uma condição.

Todas as views que criamos foram com colunas. Será que podemos criar visões de uma tabela inteira? Claro que sim! Vamos criar uma visão da nossa tabela de produtos.

```
CREATE OR REPLACE VIEW produtos_estoque AS
```

```
SELECT * FROM produto;
```

Agora podemos consultar qualquer campo da tabela, só que agora pela `view`.

```
SELECT produto_nome  
FROM produtos_estoque;
```

The screenshot shows a database interface with a query editor and a results viewer. The query editor contains the following SQL code:

```
1 SELECT produto_nome  
2   FROM produtos_estoque;
```

The results viewer displays the data output in a table format. The table has one column labeled "produto\_nome" and seven rows of data:

	produto_nome
1	REFRIGERANTE
2	PASTEL
3	AGUA
4	SUCO DE LIMÃO
5	LASANHA
6	CHURRASCO
7	SORVETE

Figura 7.14: View de uma tabela inteira.

Você deve estar se perguntando qual seria a vantagem ou o sentido de se fazer isso, não é mesmo? Além dos motivos que citei anteriormente, é muito comum termos de fornecer acesso à nossa base de dados para outras pessoas. Em vez de fornecermos acesso a toda a base de dados, fornecemos apenas a algumas `views` por meio dos direitos de usuários. É uma maneira de permitir acesso a apenas alguns campos e tabelas da maneira que desejar.

## 7.4 PARA PENSAR!

Ampliamos um pouco mais o nosso leque de opções para trabalhar com consultas. Esta é a principal função de um SGBD: extrair dados. Estamos saindo deste capítulo sabendo realizar consultas com várias tabelas juntas.

Imagine as possibilidades de integrar as views com functions , ou as joins naquelas functions que utilizamos para retornar valores. Nunca me canso de falar que é a prática que faz uma boa pessoa programadora. Insira mais registros em sua base de dados e faça consultas à vontade. Crie views para cada tabela que nós temos. Pratique!

No próximo capítulo, vamos conhecer um pouco de administração de banco de dados, pois toda pessoa programadora deve conhecer um pouco de infraestrutura. Além de administração, aprenderemos um pouco sobre performance de nosso banco e de consultas.

## CAPÍTULO 8

# ADMINISTRAÇÃO DE BANCO DE DADOS E OUTROS TÓPICOS

*"Não faz sentido olhar para trás e pensar: devia ter feito isso ou aquilo, devia ter estado lá. Isso não importa. Vamos inventar o amanhã e parar de nos preocupar com o passado." — Steve Jobs*

## 8.1 ADMINISTRADOR(A) DE BANCO DE DADOS VS. DESENVOLVEDOR(A)

A pessoa responsável pela administração do banco de dados é conhecida como DBA (*Data Base Administrator*, ou, no bom e velho português, Administrador ou Administradora de Banco de Dados). Em muitas empresas, vocês podem se deparar com a própria pessoa desenvolvedora fazendo o papel de administradora do banco de dados. Isto é muito comum, principalmente em pequenas empresas que não têm um capital disponível para a contratação de um(a) profissional capacitado(a) para essa função.

A DBA é responsável por administrar aspectos de infraestrutura do banco, como performance, arquitetura, criação, importação e backup. Como nem sempre as empresas possuem

esse papel, essas e muitas outras funções ficam a cargo do(a) desenvolvedor(a).

Muitas pessoas divergem sobre até onde o(a) desenvolvedor(a) deve influenciar no banco de dados. Alguns dizem que ele(a) deveria apenas escrever os códigos que devem ser aplicados no banco de dados, e quem deveria aplicar no banco e fazer a validação dos códigos é o(a) administrador(a) do banco. Atualmente, como existem vários *frameworks* que criam automaticamente as tabelas e os código do banco de dados, acaba que, para o(a) administrador(a), fica a tarefa de gerenciar aspectos da performance, segurança e a estrutura do banco.

Se você não é e não pretende ser uma pessoa desenvolvedora, você deve se preocupar com aspectos da estrutura do servidor do banco de dados, como saber otimizar consultas e melhorar performance, pois é o que ocorre no dia a dia de uma pessoa administradora do banco de dados *cobra*, além de saber comandos para a configuração e otimização do servidor. E se você é um(a) desenvolvedor(a) e não tem disponível um(a) profissional para executar essas tarefas, é importante que você tenha algum conhecimento.

Agora se você é desenvolvedor(a) e tem a disponibilidade de um(a) administrador(a) para fazer o trabalho da administração do servidor de banco de dados, você pode se preocupar em apenas aprender os aspectos referentes a modelar o banco e a utilização da linguagem de programação. De qualquer forma, aprender alguns comandos de administração sempre será útil.

## 8.2 COMANDOS ÚTEIS

Para auxiliar você na utilização e administração do cotidiano, temos alguns comandos que são de extrema importância. Se você dominá-los, não terá a necessidade de utilizar uma ferramenta. Com os comandos em um terminal, você terá condição de dominar qualquer ferramenta visual de gerenciamento de banco de dados.

Antes de estar conectado ao banco de dados, os comandos a seguir podem lhe ajudar com algumas informações úteis, como os bancos de dados disponíveis ou as consultas que estão sendo realizadas. Os comandos são:

Comando	Finalidade
<code>sudo -i -u postgres psql -l</code>	Para listar os bancos de dados
<code>sudo -i -u postgres psql -U nomeusuario nomebanco</code>	Para conectar ao console psql no banco de dados
<code>sudo -i -u postgres psql banco -E</code>	Para mostrar internamente como cada consulta é realizada
<code>sudo -i -u postgres psql -version</code>	Para mostrar a versão do PostgreSQL

Quando você já estiver conectado, alguns comandos são ainda mais úteis:

Comando	Finalidade
\q	Para sair do console do banco
\c nomebanco nomeuser	Para alterar o usuário e o banco de dados
\dt+ nometabela	Lista os tipos de dados do PostgreSQL com detalhes
\cd	Para mudar para outro diretório
\d	Para listar as tabelas, índices, sequências ou views
\d nometabela	Mostra a estrutura da tabela
\dt	Lista tabelas
\di	Lista índices
\ds	Lista sequências
\dv	Lista views
\ds	Lista tabelas do sistema
\dn	Lista esquemas
\dp	Lista privilégios
\du	Lista usuários
\dg	Lista grupos
\l	Lista todos os bancos do servidor, juntamente com seus donos e codificações
\e	Abre o editor com a última consulta
\?	Ajuda com os comandos do psql
\h *	Para exibir ajuda de todos os comandos
\h comandosql	Terá ajuda específica sobre o comando SQL, como \h alter table
\H	Para ativar/desativar saída em HTML
\encoding	Exibe codificação atual

## 8.3 BACKUPS

Sempre que falamos de *backups* em uma roda de desenvolvedor, sempre tem alguém que já perdeu dados por ter esquecido de fazer uma cópia do banco em que trabalhava. É algo muito simples e importante, no entanto, é muito comum esquecermos de fazer.

Backup nada mais é do que fazer um clone do seu banco de dados e salvar em um arquivo. Se algo acontecer com o servidor que estiver rodando o seu banco, você terá uma cópia salva em algum lugar seguro. Quando eu digo um lugar seguro, leve a sério, pois muitos fazem backups e deixam no mesmo computador do servidor de banco atual. Se a máquina em que ele estiver instalado tiver algum problema, não só o banco de produção vai se perder, como o seu backup também. Daí de nada adiantará!

Então, quando fizer um backup, por gentileza, guarde-o em algum outro lugar que não seja a máquina em que o servidor atual se encontra.

Vamos agora entender como fazer backup do seu banco e como usá-lo posteriormente, além de conhecer as maneiras como o backup pode ser realizado. Para verificarmos que nosso arquivo de backup está correto, devemos importá-lo em algum banco de dados. Para isso, precisamos criar um banco e importar o arquivo exportado. Vamos criar o nosso novo banco de dados e aproveitar para criar também um novo usuário. Então, mãos no teclado.

```
CREATE USER nomedousuario SUPERUSER;
```

Alteramos a senha do novo usuário:

```
ALTER USER nomedousuario PASSWORD 'senha2'
```

Feito isso, vamos sair novamente do terminal e conectar com o usuário criado:

```
$> psql -U nomedousuario postgres -h localhost
```

E finalmente vamos criar o novo banco de dados, que chamarei de newbase :

```
CREATE DATABASE newbase;
```

Em vez de sair do terminal e entrar com outro usuário e banco de dados, no próprio terminal do PostgreSQL podemos fazer a troca utilizando o comando \c .

```
\c nomedousuario newbase;
```

Você pode observar que, no cursor do terminal, agora aparece 'newbase=# . Podemos listar os bancos de dados e visualizar o que criamos e o usuário. Na lista de comandos anterior, existe o comando \l , que lista os bancos de dados e seus respectivos usuários.

```
\l;
```

Se você não criou nenhum outro banco, você obterá o resultado a seguir:

```
db_restaurant=# CREATE DATABASE newbase;
CREATE DATABASE
db_restaurant=# \l
                                         List of databases
   Name    |  Owner   | Encoding | Collate | Ctype | Access privileges
---+-----+-----+-----+-----+-----+
db_restaurant | postgres | UTF8    | en_US.UTF-8 | en_US.UTF-8 |
newbase       | postgres | UTF8    | en_US.UTF-8 | en_US.UTF-8 |
postgres      | postgres | UTF8    | en_US.UTF-8 | en_US.UTF-8 |
template0     | postgres | UTF8    | en_US.UTF-8 | en_US.UTF-8 | =c/postgres      +
               |          |          |          |          | postgres=CTc/postgres
template1     | postgres | UTF8    | en_US.UTF-8 | en_US.UTF-8 | =c/postgres      +
               |          |          |          |          | postgres=CTc/postgres
(5 rows)

db_restaurant=#
```

Figura 8.1: Bancos de dados criados.

Observe a lista dos bancos criados e dos usuários que os criaram. Até agora estávamos usando o banco de dados db\_restaurant e o usuário postgres . Desse banco de dados, vamos exportar os objetos e registros.

## Exportação

Agora que temos um novo banco de dados para receber uma importação do banco que estamos trabalhando desde o início de nosso projeto, vamos exportar o banco de dados db\_restaurant . Na barra lateral, encontre o banco de dados, clique com o botão direito e depois em Backup....

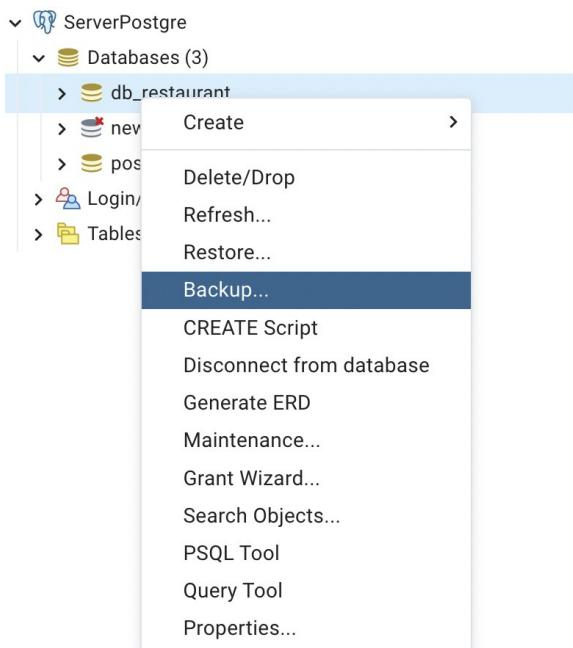


Figura 8.2: Exportando backup do banco de dados db\_restaurant.

Isso vai abrir a tela a seguir para você informar o local e o nome do arquivo de backup:

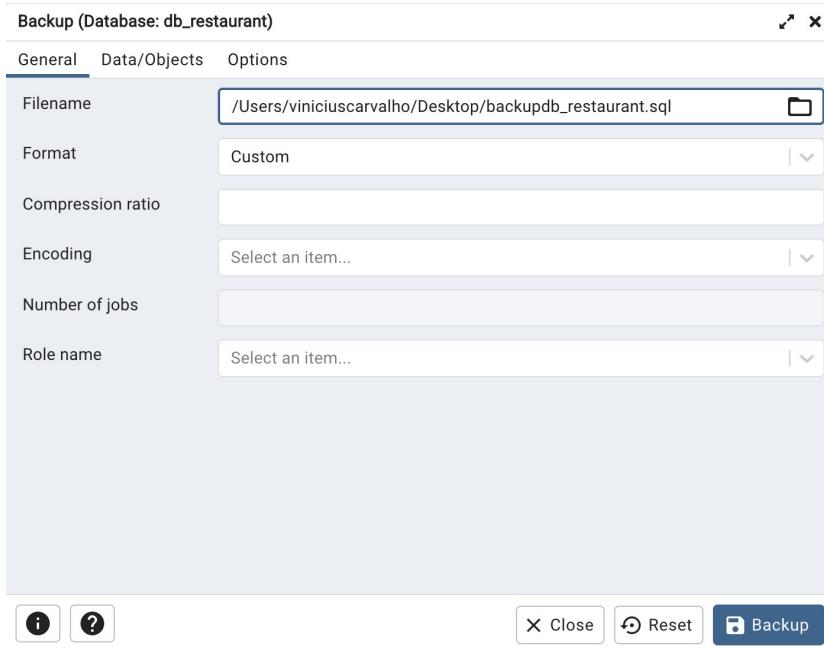


Figura 8.3: Exportando backup do banco de dados db\_restaurant.

Pronto! Muito simples, não é mesmo? A frequência com que você vai fazer o *backup* do seu banco de dados será você que decidirá. O aconselhável é que ele seja feito diariamente, pois se algo acontecer, você sempre terá um arquivo de recuperação pronto para ser usado.

## Importação

Restaurar um backup segue o mesmo processo. Vamos restaurar o backup naquele banco de dados que criamos

anteriormente chamado `newbase`. Selecione esse banco de dados, clique com botão direito do mouse e depois `Restore...`, conforme a imagem a seguir.

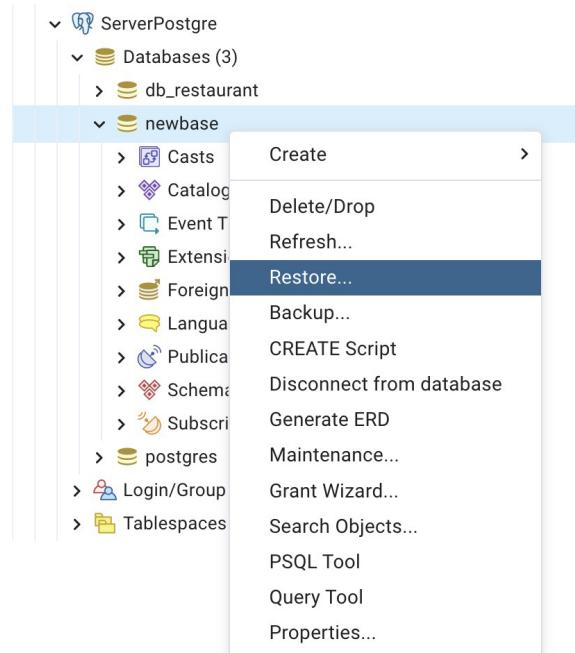


Figura 8.4: Importando os dados de um banco através de um arquivo de backup.

Selecione o arquivo de backup e click em `Restore` , conforme a imagem a seguir:

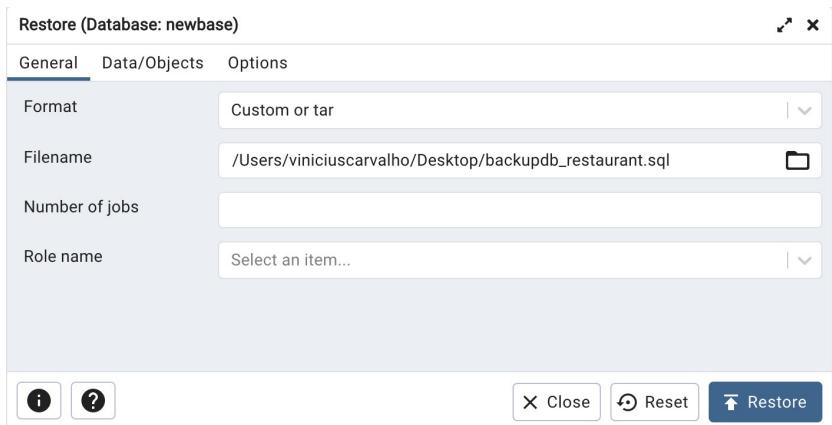


Figura 8.5: Selecionando o arquivo de backup.

Agora podemos conferir se a estrutura (tabelas) e os dados do banco foram restaurados com sucesso. Para isso, podemos verificar se as tabelas foram importadas.

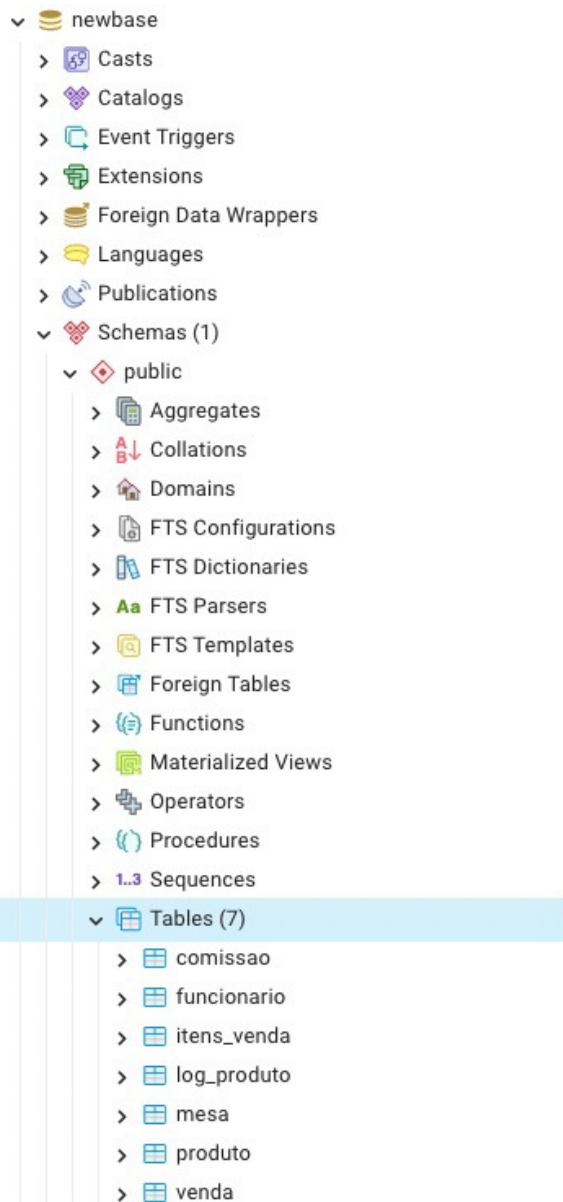


Figura 8.6: Estrutura importada.

Todas as tabelas criadas com sucesso, agora vamos verificar os registros com o comando:

```
SELECT *  
FROM funcionario;
```

The screenshot shows the pgAdmin interface with two panes. On the left is the Object Browser pane, which lists various database objects like Schemas, Tables, and Functions. On the right is the Query Results pane, which displays the output of the SELECT query.

	funcionario_id	funcionario_codigo	funcionario_nome	funcionario_ativo	funcionario_comissao	funcionario_cargo
1	d444548ed-877..	0002	SOUZA	true	2	GARÇOM
2	4810ff6d-2bf..	0001	VINICIUS CARVALHO	true	5	GERENTE
3	6667c05d-530..	0100	VINICIUS SOUZA	true	2	GARÇOM
4	c066457a-4b0..	0101	VINICIUS SOUZA MOLIN	true	2	GARÇOM
5	b99ca175-993..	0102	VINICIUS RANKEL C	true	2	GARÇOM
6	9601ab5f-562..	0103	BATISTA SOUZA LUIZ	true	2	GARÇOM
7	37947e1-189..	0104	ALBERTO SOUZA CAR..	true	2	GARÇOM
8	d398f0dd-d5..	0105	CARLOS GABRIEL ALM..	true	2	GARÇOM
9	6761115-2ff..	0106	RENAN SIMOES SOUZA	true	2	GARÇOM

Figura 8.7: Verificando os dados importados.

Agora temos certeza de que os dados foram exportados e importados com sucesso.

## Importação via planilha CSV

Utilizar planilhas para armazenar dados é a prática mais adotada por pequenas empresas em estágio inicial, por ser uma solução de baixo custo. Alguns estudos mostram que 50% das pequenas empresas armazenam seus dados em planilhas, e a importação de dados via arquivo CSV é tarefa frequente e importante no dia a dia de muitos administradores de banco de dados.

Para realizarmos a importação dos dados de uma planilha CVS para uma tabela específica do nosso banco de dados, vamos criar uma planilha com dados e importá-los para a tabela de funcionários. Abra uma planilha e, na primeira linha, adicione um registro para cada coluna da tabela.

	A	B	C	D	E	F	G
1	funcionario_codigo	funcionario_nome	funcionario_situacao	funcionario_comissao	funcionario_cargo	data_criacao	data_atualizacao
2	893	FUNCIONARIO ANTIGO	A		3 GARCOM	01/01/2023	01/01/2023
3	154	NOVO FUNCIONARIO	A		75 GARÇOM	01/01/2023	01/01/2023

Figura 8.8: Dados na planilha CSV.

Depois de ter criado e salvo a planilha com o nome `funcionarios.csv`, vamos usar o comando a seguir e fazer a importação dos dados. Observe no comando que vamos dizer para qual tabela copiaremos os dados e descrever para quais campos desta tabela os dados serão importados.

Note que, na planilha, as colunas devem ficar na mesma ordem em que as colunas no comando a seguir. No lugar do `/local_do_arquivo/`, substitua pelo caminho onde seu arquivo se encontra. O CSV é um arquivo que delimita os registros por `,`, por isso, no comando tem `DELIMITER ','`. Se você criou um arquivo CSV cujos registros possuem outro caractere separador, basta substituir o ponto e vírgula pelo seu caractere limitador.

```
COPY funcionario
(
    funcionario_codigo,
    funcionario_nome,
    funcionario_situacao,
    funcionario_comissao,
    funcionario_cargo,
    funcionario_data_criacao,
    funcionario_data_atualizacao
```

```
)  
FROM '/local_do_arquivo/funcionarios.csv'  
DELIMITER ';'   
CSV HEADER;
```

Exportar e importar dados é algo muito simples. Não tem desculpa para não os fazer. Então, não se esqueça: faça backup diariamente!

## 8.4 ÍNDICES E PERFORMANCE DAS CONSULTAS

A criação de índices é uma solução muito utilizada a fim de melhorar o desempenho das consultas no banco de dados. Segundo o próprio manual do PostgreSQL, o índice permite ao servidor de banco de dados encontrar e trazer linhas específicas muito mais rápido do que faria sem o índice. Ele consegue um melhor desempenho em uma consulta, pois o índice ordena os registros da coluna onde está criado, de forma que a consulta seja mais eficiente.

Porém, existe uma desvantagem ao optar pela criação de índices, algo que ocorre em qualquer banco de dados: as instruções de `insert`, `update` e `delete` de registros podem ficar mais lentas. Isso ocorre pois utilizar um desses comandos em uma tabela que possui índice provoca uma reorganização dos índices. Por isso, deve ser usado com cautela.

Vamos usar nossa tabela de funcionários. Atualmente, se executarmos a consulta:

```
SELECT *  
  FROM funcionario  
 WHERE funcionario_cargo = 'GARÇOM';
```

Como não temos um índice nessa tabela, o banco de dados vai percorrer toda ela, linha a linha, para encontrar todos os registros que correspondam à consulta. Imagine se tivéssemos mais muitos registros nessa tabela, e essa consulta nos retornasse diversos resultados.

Sem o índice, o banco de dados percorreria linha a linha para encontrar os que correspondem a `funcionario_cargo = 'GARÇOM'`. No entanto, se criarmos um índice na coluna `funcionario_cargo`, o banco de dados ordenaria os registros dessa tabela usando um método mais eficiente para localizar as linhas correspondentes.

Um exemplo que o manual do PostgreSQL cita é o método utilizado por alguns livros no qual os termos e os conceitos procurados frequentemente pelos leitores são reunidos em um índice alfabético colocado no final do livro. O leitor interessado pode percorrer o índice rapidamente e ir direto para a página desejada, em vez de ter de ler o livro por inteiro em busca do que está procurando.

Assim como é tarefa do autor prever os itens que os leitores provavelmente vão mais procurar, é tarefa do programador de banco de dados prever quais índices trarão benefícios. Sabendo disso, temos de entender qual será a utilidade do índice em uma determinada tabela para o projeto.

Procure criar índice para colunas que serão constantemente utilizadas para pesquisa em seu projeto. Outra dica é criar índice em colunas nas quais o resultado, na maioria das vezes, vai buscar mais de um registro, pois se for usado em colunas onde as consultas resultarão em apenas uma linha, o índice não será

eficiente, uma vez que o banco de dados terá dificuldade para ordenar os registros e buscá-los.

Voltando ao exemplo que estamos utilizando, vamos criar um índice na coluna cargo na tabela funcionario .

```
CREATE INDEX idx_cargo ON funcionario(funcionario_cargo);
```

Observe que, para o nome do índice idx\_cargo , usei um prefixo como padrão. Durante o nosso projeto, venho frisando a importância da utilização de padrões na criação de objeto no banco de dados, pois é algo muito importante para manutenções futuras e para manter a qualidade do nosso projeto. O uso de um padrão permite que alguém que não conhece o projeto, ao ver o código, consiga entender a que se refere uma determinada nomenclatura.

Se você observar que um índice não está sendo eficiente, você não só pode como deve excluí-lo, pois lembre-se de que ele pode prejudicar algumas execuções no banco. Sendo assim, para excluir um índice, você vai utilizar o comando a seguir:

```
DROP INDEX idx_cargo;
```

## Tipos de índices

Temos alguns tipos de índices no PostgreSQL. Cada um usa um algoritmo diferente para cada tipo de consulta. Por padrão, com o comando que utilizamos para a criação do índice na tabela de funcionários, o PostgreSQL usa o índice *B-tree*.

Ele é o mais adequado para as situações comuns de consultas. Vamos conhecer os principais tipos com que você poderá esbarrar no dia a dia.

## B-tree

O B-tree é o tipo padrão. Sempre que utilizamos o comando `create index`, estamos criando índice desse tipo. Os B-trees podem tratar consultas de igualdade e de faixa em dados que podem ser classificados em alguma ordem.

O SGBD, ao planejar as consultas, levará em consideração a utilização de um índice B-tree sempre que a coluna indexada estiver envolvida em uma comparação usando os operadores que já conhecemos. São eles:

- <
- <=
- =
- >=
- >
- between
- in

Vamos criar novamente o índice `idx_cargo`:

```
CREATE INDEX idx_cargo ON
    funcionario(funcionario_cargo);
```

## Hash

É um tipo de índice útil apenas com a utilização do operador de igualdade. Além de não oferecer transações de segurança, os índices hash do PostgreSQL não têm desempenho melhor do que os índices B-tree. Seu tamanho e o tempo de construção são muito piores. Por essas razões, desencoraja-se a utilização dos índices hash.

Para criarmos o índice do tipo hash, utiliza-se o comando:

```
CREATE INDEX idx_codigo ON
    funcionario USING HASH (funcionario_codigo);
```

## Concorrentes

Ao criar um índice, a tabela é bloqueada para inserção na tabela até que o índice seja construído. E se criamos um índice em uma tabela que possui um tamanho grande, ele pode levar muito tempo para ser criado, o que pode prejudicar o funcionamento de sua aplicação, pois pode bloquear ações de inserção, atualização e até exclusão de registro.

O PostgreSQL nos disponibiliza um tipo de índice para essas circunstâncias. Os índices concorrentes são bem úteis para essas situações, nas quais é necessário criarmos um índice em ambiente de produção que não pode ser interrompido.

Para criar esse tipo de índice, usamos o comando:

```
CREATE INDEX concurrently idx_nome ON
    funcionario USING btree (funcionario_nome);
```

## Multicolunas

Durante o nosso projeto, aprendemos a fazer consultas em mais de uma coluna de uma vez. Os índices de uma única coluna não serão úteis para melhorar a performance de consultas onde serão comparadas mais de uma coluna. Para isso, podemos criar um índice que seja utilizado para duas colunas ao mesmo tempo.

Para criar esse tipo de índice, usamos o comando:

```
CREATE INDEX idx_funcionario_comissao_codigo ON
    funcionario(funcionario_comissao, funcionario_codigo);
```

Esse tipo de índice seria útil para consultas do tipo:

```
SELECT *
  FROM funcionario
 WHERE funcionario_comissao > 4
   AND funcionario_codigo < '1000';
```

## Índices únicos

Anteriormente, aprendemos a criar constraints de chave primária e chave estrangeira. Também vimos que uma chave primária de tabela é um registro único. Se quisermos que qualquer outra coluna de uma determinada tabela tenha um valor único, podemos criar um índice que tornará o valor de uma coluna exclusivo.

Vamos transformar a coluna `funcionario_codigo` única. Para isso, usaremos o comando:

```
CREATE UNIQUE INDEX idx_unique_codigo ON
    funcionario(funcionario_codigo);
```

Para sabermos se esse índice está funcionando corretamente, vamos tentar inserir um novo registro na tabela `funcionario` e, no `insert` do registro, utilizar um código de funcionário que já existe na tabela. Primeiro, vamos fazer uma consulta para poder pegar um `funcionario_codigo` de um funcionário existente no banco.

```
SELECT funcionario_codigo
  FROM funcionario;
```

Em minha base, tenho um funcionário com o código `0001`. Se você vem seguindo o projeto e adicionando todos os registros sugeridos, você também deve tê-lo.

Agora vamos inserir um registro na tabela de funcionários e tentar utilizar esse mesmo código que já existe na base.

```
INSERT INTO funcionario(funcionario_codigo, funcionario_nome)
VALUES('0001', 'VINICIUS CARVALHO');
```

Ao clicarmos `enter`, será exibido o erro:

The screenshot shows a PostgreSQL query editor interface. The 'Query' tab is selected, displaying the following SQL code:

```
1 INSERT INTO funcionario(funcionario_codigo, funcionario_nome)
2      VALUES('0001', 'VINICIUS CARVALHO');
```

The 'Messages' tab is selected, showing the error output:

```
ERROR: duplicate key value violates unique constraint "idx_unique_codigo"
DETAIL: Key (funcionario_codigo)=(0001) already exists.
SQL state: 23505
```

Figura 8.9: Erro de unique key.

Utilize em colunas que ainda não possuam itens duplicados, ou antes de duplicar itens, pois se a coluna já possuir itens em duplicidade, esse erro também será exibido.

## Analyze

Com o índice criado em nossa tabela de funcionários, o banco de dados vai atualizar o índice automaticamente quando houver uma modificação. A otimização nas consultas é realizada quando ele julgar mais eficiente do que a busca linha a linha.

O PostgreSQL consegue julgar a forma mais eficiente de fazer as consultas por meio de estatísticas retiradas das tabelas. Além dessa ordenação que o próprio SGBD executa, é importante nós

executarmos o comando `analyze` periodicamente. Esse comando coleta estatísticas sobre o conteúdo das tabelas do banco de dados e armazena os resultados em uma tabela do sistema, a `pg_statistic`.

O SGBD usa essas estatísticas para ajudar a determinar qual a forma mais eficiente de executar as consultas em seu banco de dados. Se não passarmos nenhum parâmetro para o comando `analyze`, o SGBD analisará todas as tabelas do banco de dados que você estiver conectado. Mas podemos passar parâmetros para analisar determinados objetos.

Primeiro, vamos pedir para que sejam analisadas todas as tabelas do nosso banco:

```
analyze verbose;
```

Será exibido o resultado da análise das estatísticas de todas as tabelas. Agora, pediremos para que seja analisada a tabela de funcionários.

```
ANALYZE VERBOSE funcionario;
```

Nesse caso, apenas serão atualizadas as estatísticas da tabela de funcionários. Podemos ir além e pedir para executar a análise apenas da coluna `funcionario_cargo`.

```
analyze verbose funcionario(funcionario_cargo);
```

Entre os itens coletados pelo comando `analyze`, as listas de alguns valores mais comuns de cada coluna e um histograma mostrando a distribuição aproximada dos dados de cada coluna auxiliam a organização dos registros. Vale lembrar que as estatísticas podem mudar a cada execução do `analyze`, alterando,

assim, a forma com que o SGBD executa a otimização das consultas, mesmo que não tenha alteração de registros. Por isso, devemos fazer periodicamente a análise das tabelas.

## Reindexação

Mesmo realizando o `analyze` constantemente, o desempenho do índice pode ser pedido com o tempo e pode tornar o índice ineficiente. Se for percebida a perda do desempenho do índice, temos a opção de fazer a reindexação do índice.

Ela refará o processo de indexar os registros de onde ele foi criado. Essa ação fará com que o índice volte a performar melhor em seu SGBD. Para fazer essa reindexação, vamos usar o comando:

```
REINDEX TABLE funcionario;
```

Após esse comando, o SGBD vai reindexar os índices da tabela de funcionários.

## 8.5 PARA PENSAR!

Agora que você fez o backup do seu banco, tabelas e registros, e ele está salvo em algum lugar seguro (assim espero), quero que você tente reutilizar todos os métodos de backup e importação para outras tabelas que não fizemos. Se for preciso, crie outros bancos e faça a importação dos dados.

Lembra da frase do Aristóteles, em um dos capítulos anteriores, na qual ele diz que a repetição leva à perfeição? *#FicaADica*.

Outro ponto que vimos neste capítulo foi a importação via

CSV, muito útil quando você precisa fazer uma migração de uma base de dados para outra, ou até mesmo de um banco de dados diferente. Se você conhece alguém que quer sair das planilhas e utilizar um banco de dados, você já poderá ajudá-lo(a).

## CAPÍTULO 9

# TIPOS DE DADOS ESPECIAIS

*"Sempre entregue mais do que o esperado." — Larry Page*

## 9.1 TIPOS DE CAMPOS ESPECIAIS

Além dos tipos de campos que aprendemos anteriormente, o PostgreSQL possui alguns tipos de dados especiais que outros bancos de dados relacionais, como o MySQL, não possuem. A utilização de tipos de dados diferentes vai depender do projeto no qual você estiver trabalhando.

Sempre avalie o que você está desenvolvendo, pesquise por soluções e aplique aquela que melhor se encaixar em seu problema. Também não se prenda a apenas uma solução. Em desenvolvimento de software, muito dificilmente existirá apenas uma forma de resolver um problema. Não tenha preguiça de testar mais de uma solução.

Essa é a minha dica para você usar estes dois principais tipos de campos especiais do PostgreSQL. Não entrarei em detalhes de outros tipos especiais, pois muito dificilmente você vai utilizá-los em suas aplicações.

Você verá neste capítulo que os tipos de campos `array` e `JSON` terão uma boa aplicação em nosso projeto e poderão lhe ser úteis futuramente no desenvolvimento de aplicações web modernas.

## 9.2 CAMPOS ARRAY

Se você já conhece alguma linguagem de programação, já está familiarizado com o termo `array` e sua funcionalidade. Se você ainda não está familiarizado, um `array` é uma lista de objetos. Enquanto os tipos de campos que conhecemos até agora conseguem armazenar apenas um objeto, um `array` pode armazenar uma lista.

Provavelmente, você já deve ter se deparado com algum lugar exibindo uma lista de objetos como na figura a seguir, como se fosse uma lista de objetos que se referem à mesma coisa.

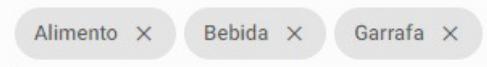


Figura 9.1: Utilização de listagem.

Você já consegue imaginar onde podemos utilizar esse campo em nosso projeto? Vamos supor que precisamos categorizar nossos produtos, mas não somente uma característica para cada um, mas sim uma lista de categoria para cada produto. Para isso, vamos criar um novo campo chamado `produto_categoria` do tipo `array` e entender como podemos usar esse campo.

A criação do tipo de campo `array` é parecida com o que aprendemos anteriormente, apenas adicionamos o elemento

colchetes ( [ ] ) à frente do tipo do campo que queremos criar. Se quiséssemos criar um campo do tipo `string` , normalmente faríamos `produto_categoria text` . No entanto, como queremos que ele seja do tipo `array` , devemos declará-lo como `produto_categoria TEXT[]` .

Os colchetes depois do campo dizem ao nosso banco de dados que este campo poderá armazenar uma string ou uma lista de strings. Mão no teclado para criamos o código para criação do nosso campo.

```
ALTER TABLE produto
    ADD COLUMN produto_categoria TEXT[];
```

Para utilizar o novo campo, vamos inserir um novo produto e aprender como devemos adicionar registro neste novo tipo de campo que acabamos de conhecer.

```
INSERT INTO produto (produto_id,
                     produto_codigo,
                     produto_nome,
                     produto_valor,
                     produto_ativo,
                     produto_data_criacao,
                     produto_data_atualizacao,
                     produto_categoria)
VALUES (gen_random_uuid(),
        '03251',
        'ESFIRRA',
        5,
        true,
        '01/01/2023',
        '01/01/2023',
        '{"CARNE", "SALGADO", "ASSADO" , "QUEIJO"}'
);
```

Observe que a inserção em campo do tipo `array` é um pouco diferente dos outros. Como se trata de uma lista de strings,

precisamos de alguma maneira limitar cada string, por isso devemos usar as aspas simples para indicar ao banco de dados que vamos inserir uma string, seguido do elemento chaves para indicar que se trata de uma lista de objetos. Depois, usamos as duplas para limitar cada string, e vírgula para separar cada item da lista, tendo a lista que formamos em nosso código: {"CARNE", "SALGADO", "ASSADO" , "QUEIJO"} .

Após fazer a inserção desse novo registro em nossa tabela, podemos fazer uma consulta para verificar como ficam os dados no banco de dados. Mão no teclado e vamos fazer uma consulta usando somente o campo que criamos agora.

```
SELECT produto_categoria  
      FROM produto  
     WHERE produto_nome LIKE 'ESFIRRA';
```

Como resultado, teremos:

The screenshot shows a MySQL Workbench interface. In the 'Query' tab, a SQL query is written:

```
1  SELECT produto_categoria  
2    FROM produto  
3   WHERE produto_nome LIKE 'ESFIRRA';
```

In the 'Data output' tab, the result of the query is displayed in a table:

produto_categoria
text[]
{CARNE,SALGADO,ASSADO,QUEIJO}

Figura 9.2: Inserindo uma lista de strings.

Além de fazermos uma consulta para verificar todos os elementos da lista, podemos criar uma consulta para extrair apenas

um item ou uma faixa de itens da lista que o campo array possui. Na inserção que fizemos, adicionamos uma lista com quatro elementos, como mostra a figura:

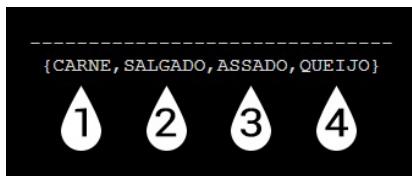


Figura 9.3: Elementos do nosso array.

Observando essa imagem, vemos todos os itens da lista inseridos no campo `produto_categoria`. E se em vez de selecionar todos os elementos da lista, quiséssemos consultar somente o segundo item da lista?

Baseando-se nessa figura dos itens, podemos deduzir que cada item se encontra em uma posição e, sabendo disso, o PostgreSQL nos permite fazer consultas nos itens de um array pela sua posição. Desta vez, em vez de selecionarmos todos os elementos da lista, consultaremos apenas o segundo elemento da lista. Vamos fazer isso passando como parâmetro o número 2, indicando para o banco de dados que desejamos que ele nos retorne o item que está na posição 2 da lista. Mão no teclado e vamos para o nosso código.

```
SELECT produto_categoria[2]
  FROM produto
 WHERE produto_nome LIKE 'ESFIRRA';
```

Simples, não é mesmo?! Em vez de colocarmos apenas o nome do campo, passamos entre colchetes a posição que queríamos. E como resultado, temos:

The screenshot shows a MySQL query editor interface. At the top, there are tabs for 'Query' (which is selected) and 'Query History'. Below the tabs is a code editor containing the following SQL query:

```
1 SELECT produto_categoria[2]
2   FROM produto
3 WHERE produto_nome LIKE 'ESFIRRA';
```

Below the code editor are three tabs: 'Data output', 'Messages', and 'Notifications'. The 'Data output' tab is selected. It displays the results of the query in a table format. The table has one column labeled 'produto\_categoria' and one row with the value 'text'. The row number is 1, and the value is 'SALGADO'.

Figura 9.4: Busca em uma posição do array.

Além de um consultar uma posição de um array , podemos consultar um intervalo de posições. Agora vamos criar uma consulta para buscar os itens da posição 2 até a posição 4 da lista.

```
SELECT produto_categoria[2:4]
      FROM produto
     WHERE produto_nome LIKE 'ESFIRRA';
```

Observe em nosso código que agora usamos dois pontos para separar a posição inicial da final. Como resultado, teremos:

The screenshot shows a MySQL query editor interface. At the top, there are tabs for 'Query' (selected) and 'Query History'. Below the tabs is a code editor containing the following SQL query:

```
1 SELECT produto_categoria[2:4]
2   FROM produto
3 WHERE produto_nome LIKE 'ESFIRRA';
```

Below the code editor are three tabs: 'Data output', 'Messages', and 'Notifications'. The 'Data output' tab is selected. It displays the results of the query in a table format. The table has one column labeled 'produto\_categoria' and one row with the value 'text[]'. The row number is 1, and the value is '{SALGADO,ASSADO,QUEIJO}'.

Figura 9.5: Busca em um intervalo de posições do array.

O tipo `array` pode ser muito útil dependendo do contexto de seu projeto. Para treinar, agora insira mais registros na tabela de produtos e atualize os registros existentes, adicionando informação no campo `produto_categoria`. Pratique!

O próximo tipo que conheceremos é o `JSON`. Antes da versão 9.2 do PostgreSQL, armazenar dados nesse formato era exclusividade dos bancos NoSQL. Estes são classificados como não relacionais, pois, diferentemente dos bancos MySQL e PostgreSQL, não possuem um esquema rígido no qual os relacionamentos entre as tabelas precisam acontecer. Se você quiser se aprofundar em banco de dados NoSQL, aconselho conhecer os livros da Casa do Código sobre o assunto — *NoSQL – Como armazenar os dados de uma aplicação moderna*, sobre NoSQL; e *MongoDB – Construa novas aplicações com novas tecnologias*, sobre MongoDB.

## 9.3 CAMPOS DO JSON

Até algum tempo atrás, o formato universal mais usado para troca de informações e dados era o `XML`, até que o `JSON` se popularizou e hoje em dia é o método mais utilizado para troca de informações. Seu significado é *JavaScript Object Notation*, mas, apesar do nome, ele pode ser manipulado por diversas linguagens de programação. São muitas as que dão suporte ao JSON.

Para usarmos esse tipo de campo em nosso projeto, vamos alterar novamente a tabela de produtos. Vamos inserir o campo `produto_estoque`. Ele vai armazenar informações sobre o estoque de cada produto.

Não tem segredo para a utilizar esse tipo de campo. Basta

apenas informar `produto_estoque` json . Então, vamos ao código.

```
ALTER TABLE produto  
ADD COLUMN produto_estoque JSON;
```

## Cenário simples

Agora vamos conhecer a estrutura que devemos inserir os registros nessa nova coluna. A estrutura básica de um objeto JSON é a seguinte:

```
{"ObjetoPai": "valor"}
```

E o objeto pai pode ter objetos filhos:

```
{  
  "ObjetoPai": {  
    "ObjetoFilho": "valor"  
  }  
}
```

Basicamente será isso que deveremos inserir em nosso código. Eu vou adicionar um objeto pai chamado `info_estoque` e os seguintes objetos filhos: `tem_estoque` , para indicar se o produto está disponível no estoque; `quantidade` , para indicar a quantidade disponível no estoque; e `ultima_compra` , para indicar a data da última compra. Da mesma maneira que descrevi, também informaremos um valor para cada objeto. Vamos ao nosso código.

```
INSERT INTO produto( produto_id,  
                     produto_codigo,  
                     produto_nome,  
                     produto_valor,  
                     produto_ativo,  
                     produto_data_criacao,
```

```
        produto_data_atualizacao,
        produto_categoria,
        produto_estoque)
VALUES( gen_random_uuid(),
        '6234',
        'COCA-COLA',
        6,
        true,
        '01/01/2023',
        '01/01/2023',
        '{"REFRIGERANTE",
         "LATA",
         "BEBIDA" ,
         "COLA"}',
        '{ "info_estoque":
            { "tem_estoque": "SIM",
              "quantidade": 17,
              "ultima_compra": "01/01/2023" }
        }'
    );

```

Observe que o código possui a mesma estrutura básica do JSON. Após a inserção desse registro, vamos criar uma consulta para visualizarmos como esse tipo de registro fica em nosso banco.

```
SELECT produto_estoque
  FROM produto
 WHERE produto_nome LIKE 'COCA-COLA';
```

E como resultado, teremos:

The screenshot shows the MySQL Workbench interface. In the 'Query' tab, there is a SQL query:

```
1 SELECT produto_estoque
2   FROM produto
3 WHERE produto_nome LIKE 'COCA-COLA';
```

In the 'Data output' tab, the results are displayed in JSON format:

produto_estoque
json

```
1 { "info_estoque": { "tem_estoque": "SIM", "quantidade": 17, "ultima_compra": "01/01/23" }}
```

Figura 9.6: Campo do tipo JSON.

Da mesma maneira que aprendemos que, em campos do tipo `array`, podemos selecionar o item que quisermos de uma lista, em campos do tipo JSON também temos a possibilidade de selecionar o conteúdo do objeto que desejarmos. Vamos montar uma consulta para nos retornar apenas o objeto filho `quantidade`.

Para isso, temos de fazer a busca através do objeto pai. Sabendo disso, teremos o nosso código:

```
SELECT produto_estoque->'info_estoque'->>'quantidade'
      AS quantidade
     FROM produto
    WHERE produto_nome LIKE 'COCA-COLA';
```

E como resultado, teremos o valor inserido no objeto filho `quantidade`.

## Operadores -> e ->>

Observe que, na última consulta, foi usado o operador `->>` e o banco de dados retornou o valor em formato de texto do objeto. Se

usássemos o operador `->` , teríamos como retorno um objeto JSON.

Vamos aplicar na prática e identificar a diferença. Montaremos uma consulta para buscar o valor do objeto `ultima_compra` , primeiro com o operador `->` e depois com o operador `->>` .

```
SELECT produto_estoque->'info_estoque'->'ultima_compra'  
      AS ultima_compra  
  FROM produto  
 WHERE produto_nome LIKE 'COCA-COLA';
```

Como retorno, teremos `"01/01/2023"` . Observe que o resultado foi o objeto da forma que foi inserido no banco de dados, entre aspas duplas. Agora vamos fazer a consulta com o `->>` .

```
SELECT produto_estoque->'info_estoque'->>'ultima_compra'  
      AS ultima_compra  
  FROM produto  
 WHERE produto_nome LIKE 'COCA-COLA';
```

Como retorno, teremos `01/01/2023` . Agora tivemos como retorno apenas o texto do objeto, sem as aspas. Conhecendo esses operadores, podemos passar para um exemplo um pouco mais complexo.

## Cenário complexo

Nessa primeira utilização de JSON, usamos um cenário simples, no qual tínhamos apenas um objeto pai e alguns filhos. Agora vamos inserir um JSON um pouco mais complexo. A única diferença é que este próximo JSON terá mais um objeto pai, que chamaremos de `ultima_venda` , em que será indicada a data da última venda do produto.

```

INSERT INTO produto( produto_id,
                     produto_codigo,
                     produto_nome,
                     produto_valor,
                     produto_ativo,
                     produto_data_criacao,
                     produto_data_atualizacao,
                     produto_categoria,
                     produto_estoque)
VALUES(gen_random_uuid(),
       '77978',
       'GATORADE',
       6,
       true,
       '01/01/2023',
       '01/01/2023',
       '{"ISOTONICO",
        "GARRAFA",
        "BEBIDA" }',
       '{ "info_estoque":
          { "tem_estoque": "SIM",
            "quantidade": 17,
            "ultima_compra": "01/01/2023" },
          "ultima_venda": "02/01/2023"
        }'
      );

```

Em seguida, após inserir esse novo registro, em vez de buscarmos um objeto específico como fizemos anteriormente, vamos utilizar os parâmetros de JSON na cláusula `where` para buscar o conteúdo do campo `produto_estoque`. Vamos criar uma consulta que nos retornará o JSON do campo `produto_estoque` que possui o valor do objeto `ultima_venda` igual a `02/01/2023`.

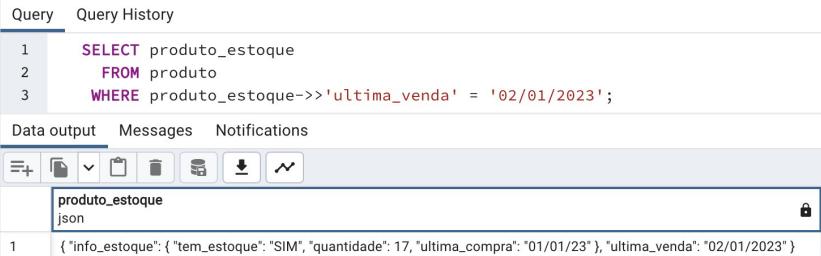
---

```

SELECT produto_estoque
  FROM produto
 WHERE produto_estoque->>'ultima_venda' = '02/01/2023';

```

Como resultado, teremos o conteúdo do campo `produto_estoque`.



The screenshot shows a PostgreSQL query editor interface. At the top, there are tabs for 'Query' (which is selected) and 'Query History'. Below the tabs is a code area containing three numbered lines of SQL:

```
1  SELECT produto_estoque
2    FROM produto
3   WHERE produto_estoque->>'ultima_venda' = '02/01/2023';
```

Below the code area are tabs for 'Data output', 'Messages', and 'Notifications'. Under 'Data output', there is a toolbar with icons for file operations like new, open, save, and copy. A dropdown menu shows 'produto\_estoque' and 'json'. The main data pane displays the result of the query as a single row of JSON data:

1	{"info_estoque": {"tem_estoque": "SIM", "quantidade": 17, "ultima_compra": "01/01/23"}, "ultima_venda": "02/01/2023"}
---	---

Figura 9.7: Consulta com JSON com mais de um objeto pai.

Observe como é muito simples trabalhar com JSON no PostgreSQL, mesmo tendo um ou mais objetos pais. Esse tipo de campo pode ser muito útil em seu projeto. Se você for desenvolver aplicações web, certamente você utilizará JSON e trocará objetos JSON com outras aplicações. Com esse tipo de campo em seu banco de dados, ficará mais fácil receber e armazenar dados.

## 9.4 PARA PENSAR!

Como já comentei anteriormente, tente sempre criar situações reais quando você estiver aprendendo algo novo, pois fica mais fácil absorver novos conceitos. É exatamente isso que eu tento fazer em cada capítulo, sempre utilizando o mesmo projeto e inserindo um contexto que pode acontecer de verdade.

Tente fazer isso sempre e você vai perceber que fica mais simples para aplicar novos conceitos em seu dia a dia. Agora que você aprendeu esses dois novos tipos de campos, aplique-os nas demais tabelas do projeto. Pense em situações que podem

acontecer no cotidiano da utilização do projeto que estamos construindo, e tente aplicar soluções usando esses novos campos. Pratique sempre e cada dia mais.

## CAPÍTULO 10

# CONCLUSÃO

*"Falhar é uma opção aqui. Se as coisas não estão dando errado, você não está inovando o suficiente." — Elon Musk*

## 10.1 PARA PENSAR E AGRADECER!

Parabéns por você ter chegado até aqui. Esse é o efeito do seu esforço e do seu comprometimento com a sua aprendizagem e crescimento como pessoa desenvolvedora — e como pessoa desenvolvedora, incluo todos os papéis que temos na área de desenvolvimento de software.

Muito obrigado por ter escolhido o meu livro para fazer parte da sua jornada. Fico muito honrado de ter feito parte da lista de autores que contribuíram de alguma maneira com o seu desenvolvimento. Espero que tenha gostado do conteúdo do livro e que o utilize em seu dia a dia. Qualquer dúvida ou dica, pode me adicionar nas redes sociais. Terei o prazer em lhe ajudar.

Nunca pare de estudar e se aperfeiçoar. É isso que vai lhe diferenciar no mercado de trabalho. Ótimos estudos!

*"Keep Coding!" — Vinícius Carvalho*