PostgreSQL

Banco de dados para aplicações web modernas



Casa do Código

VINÍCIUS CARVALHO

SOBRE O GRUPO CAELUM

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 portfolio e à sua carreira.

A Casa do Código faz parte do Grupo Caelum, um grupo focado na educação e ensino de tecnologia, design e negócios.

Se você gosta de aprender, convidamos você a conhecer a Alura (www.alura.com.br) que é o braço de cursos online do Grupo. Acesse o site deles e veja as centenas de cursos disponíveis para você fazer da sua casa também, no seu computador. Muitos instrutores da Alura são também autores aqui da Casa do Código.

O mesmo vale também para os cursos da Caelum (www.caelum.com.br) que é o lado de cursos presenciais, onde você pode aprender junto dos instrutores, em tempo real e usando toda a infraestrutura fornecida pela empresa. Veja também as opções disponíveis lá.

ISBN

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

EPUB: 978-85-5519-256-2

MOBI: 978-85-5519-257-9

Você pode discutir sobre este livro no Fórum da Casa do Código: http://forum.casadocodigo.com.br/.

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 frustação 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. Pois, 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, que o leitor 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 de um desenvolvedor.

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/viniciuscdes/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.

http://www.viniciuscdes.net

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 de 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 que eu tive durante todos esses anos de vida. Acredito que uma das grandes realizações de um professor é saber como estão os alunos 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 professor antigo, tento passar essa mensagem. Creio que sirva de incentivo para que eles continuem se dedicando e 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.

E não poderia de deixar de agradecer minha esposa, Thais, pelo incentivo em todos os meus projetos.

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.

Apaixonado pela busca pelo conhecimento, procura manter-se atualizado nas tendências de desenvolvimento de software, tecnologia e tem como meta aprender algo novo todos os dias.

Na sua carreira profissional, teve oportunidades de trabalhar como analista de suporte, desenvolvedor, gerente de projetos, consultor e como um empreendedor incansável, arriscando-se a ter seu próprio negócio. É apaixonado por tecnologia e informação. Vive em constante aprendizagem.

Teve chance de palestrar em congresso de software livre, como o VOL DAY, evento criado pela comunidade *Viva o Linux*; publicar artigos em diversos congressos no Brasil; e ministrar aulas de graduação no Centro Universitário Filadélfia (UniFil), faculdade referência em cursos de graduação e pós-graduação no Paraná, na qual se formou.

Lançou seu primeiro livro em 2015, sobre MySQL, que você pode encontrar em: https://www.casadocodigo.com.br/products/livro-banco-mysql.

Sua página pessoal é http://www.viniciuscdes.net. Lá você pode conferir seu currículo e outras informações.

Sumário

1 Introdução **1**

1.1 Banco de dados 1

1.2 PostgreSQL 3

1.3 O que dizem os desenvolvedores 10

1.4 Instalando e configurando 11

1.5 Para pensar! 20

2 Comece a desenvolver com o PostgreSQL **21**

2.1 PL/pgSQL 21

2.2 DataTypes: do básico ao avançado 22

2.3 Para pensar! 30

3 Nosso primeiro projeto **31**

3.1 Entendendo nossos dados 33

3.2 A estrutura das tabelas 34

3.3 Chaves primárias e chaves estrangeiras 36

3.4 Criando nossas tabelas 39

3.5 Constraints: integridade de seus dados 41

3.6 Criando sequências para as nossas tabelas 45

3.7 E os nossos registros? Já podemos inserir! 48

3.8 Consultando nossos registros 52

3.9 Para pensar! 55

4 Functions — Agilizando o dia a dia 56

4.1 Functions para poupar esforços 56
4.2 Utilizando a function 61
4.3 Functions sem return 62
4.4 Alterando functions 67
4.5 Excluindo functions 68
4.6 Vantagens da utilização das functions 68
4.7 Para pensar! 69

5 Funções, operadores e operações 70

5.1 Funções embutidas 70
5.2 Operadores lógicos 71
5.3 Operadores de comparação 74
5.4 Operadores e funções matemáticas 75
5.5 Funções de texto 79
5.6 Funções data/hora 82
5.7 Funções agregadoras 88
5.8 Consultas utilizando like 100
5.9 Para pensar! 105

6 Banco de dados rápido nos gatilhos 107

6.1 Triggers — Gatilhos para agilizar tarefas 107
6.2 Triggers: insert, update e delete 108
6.3 Desabilitando, habilitando e deletando uma trigger 117
6.4 Para pensar! 120

7 Turbinando as consultas com joins e views 121

7.1 Subconsultas 121
7.2 Consultas entre duas ou mais tabelas através das joins 123

7.3 Views 129
7.4 Para pensar! 133

8 Administração do banco e outros tópicos 135
8.1 Administrador de banco de dados vs. desenvolvedor 135
8.2 Comandos úteis 136
8.3 Backups 138
8.4 Índices e performance das consultas 143
8.5 Para pensar! 150

9 Tipos de dados especiais 152
9.1 Tipos de campos especiais 152
9.2 Campos array 153
9.3 Campos do JSON 157
9.4 Para pensar! 162

10 Exercícios de concurso 163
10.1 Concursos pelo Brasil 163
10.2 Exercícios 163

11 Gabarito 188
11.1 Para pensar! 189

12 Apêndice A — Profissão DBA 190
12.1 Comandos básicos e úteis 193
12.2 Trabalhando com pgAdmin 193
12.3 Para pensar e agradecer! 203

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.

O entendimento dos seus registros da empresa é crucial para a formulação de consultas e perguntas para o negócio. Isso é possível se a empresa tem o suporte de um bom banco de dados para essas questões.

Umas das palavras que mais temos ouvido nos últimos 3 anos é o tal do **big data**, que em poucas palavras é: extrair e, de alguma maneira, relacionar a maior quantidade de dados sobre o seu 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.

Big data é um assunto tão extenso quando o PostgreSQL. Se

você se interessar por análise de dados, vale a pena buscar se informar sobre como implantar e gerenciar softwares para extrair dados de bancos e implantar o big data em sua empresa ou para seus clientes.

Quando digo dados, estou querendo dizer todas as informações e registros gravados em um banco de dados da empresa, seja esse banco conectado a um ERP, CRM 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 outros desenvolvedores 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 esta etapa dirá muito sobre a qualidade do seu sistema no futuro. É claro, conforme 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 descriminado 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:** PostgreSQL é totalmente transacional, incluindo mudanças estruturais destrutivas. Isto 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:** 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:** PostgreSQL é muito amigável tanto para o desenvolvimento de software quanto para administração de banco de dados. Todos 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:** 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 demonstração de explicar. Logo, você pode encontrar e corrigir problemas de desempenho se eles surgirem. PostgreSQL pode lidar com mais armazenamento de dados e gerenciamento de necessidades com facilidade, e é uma excelente ferramenta para aprender também.

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 a 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 4TB de dados.

Resumindo, o que você precisar, não ultrapassando os limites apresentados na lista a seguir, o PostgreSQL poderá lhe oferecer com excelência de um 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 as perguntas 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, aplicada com rigor.

Imagine uma tabela com alguns dados contidas 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.

Estas tabelas do banco de dados possui 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 devem 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 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 a replicação e armazenando XML, JSON, cordas matérias, e usando o Ltree e extensões do PostgreSQL hstore, você pode obter muitos benefícios de seu sistema NoSQL. Você vai descobrir que poderá utilizar em todos seus projetos, tanto online como offline. Você 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. Eu separei três. Ao pesquisar em um estudo diferente, você pode encontrar que alguns comandos citados por mim em uma categoria talvez estejam em outra. 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!

PostgreSQL na Web vs. PostgreSQL offline

Mesmo com a ascensão das linguagens de programação para web e mundo mobile, ainda existem vários desenvolvedores que desenvolvem sistemas offline ou ainda dão manutenção em sistemas legados. Essa é uma questão que dependerá do projeto. O PostgreSQL vai cumprir seu papel da melhor forma independentemente da plataforma, seja ela *online* ou *offline*.

Isto é algo que você não deve se preocupar. Algo que é independente da plataforma é a capacidade de processamento da máquina em que você rodará o seu servidor de banco de dados.

Deve-se pesquisar de qual máquina você precisará para rodar a sua aplicação sem ter problemas.

1.3 O QUE DIZEM OS DESENVOLVEDORES

Fiz uma pesquisa entre desenvolvedores que usam o PostgreSQL no dia a dia perguntando a eles sobre os desafios de seus projetos, como utilizam este banco de dados e como ele os ajuda. Com o resultado desta pesquisa, fiz uma compilação que você confere na sequência. Ao todo, foram 225 respostas.

- **Você está satisfeito com o desempenho do PostgreSQL pelo que você espera de um SGBD?**
  - *90% SIM*
  - *10% NÃO*

- **Em qual plataforma do seu sistema você utiliza o PostgreSQL?**
  - *8% Desktop*
  - *92% Web*

- **Se você respondeu web na pergunta anterior, qual servidor de hospedagem você utiliza?**
  - *22% DIGITAL OCEAN*
  - *42% HEROKU*
  - *26% AMAZON*
  - *3% LINODE*
  - *2% LOCAWEB*
  - *5% OUTROS*

- **Qual o sistema operacional do seu servidor de banco de dados?**
  - *85% Linux*
  - *15% Windows*

- **O que o levou a escolher o PostgreSQL como seu gerenciador de banco de dados?**
  - *22% Variedade de funções*
  - *5% Facilidade de desenvolvimento*
  - *30% Desempenho*
  - *2% Compatibilidade*
  - *33% O melhor SGBD open source*
  - *8% Outros*

Pelos números apresentados, podemos dizer que os desenvolvedores estão satisfeitos 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.

1.4 INSTALANDO E CONFIGURANDO

Durante o livro, para desenvolvermos o nosso projeto, vamos utilizar a versão 9.6 do PostgreSQL, que é a última lançada. Descreverei como você poderá instalar nos 3 principais sistemas operacionais mais usados: Linux, Mac OS e Windows.

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

Instalando no Mac OS

Para fazer essa instalação, estou usando o Mac OS X 10.10 Yosemite. Você pode usar a versão disponível no site, ou utilizar uma versão mais simples e fácil. Para MAC OS, existe um aplicativo chamado *PostgreApp*, que é a maneira mais simples de usar o PostgreSQL no MAC OS. Ele roda como um serviço. Você baixa e executa-o. Muito simples.

Primeiro, vá até o site http://http://postgresapp.com/ e faça o download. Quando ele estiver baixado, abra-o. Ao abrir, você verá que, ao lado do relógio, aparecerá uma imagem de um elefante que mostrará o serviço do PostgreSQL rodando, como mostra a figura na sequência.

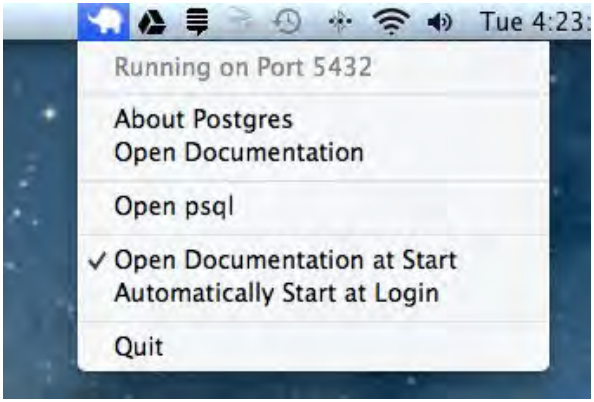

Figura 1.1: Serviço do PostgreSQL rodando

Quando você abre, também abrirá uma tela inicial do Postgre App. Ela possui o botão **open psql** para você iniciar o *console* para começar a criar e manipular seu banco de dados.



Figura 1.2: Tela de inicio do Postgre App

Será no *console* da aplicação onde escreveremos nossos *scripts* e comandos de criação e manipulação do nosso futuro banco de dados. Mais à frente, mostrarei esses comandos.

Instalando no Linux

Na página de download, você poderá encontrar versões disponíveis para as distribuições do Linux, como: Red Hat, Debian, Ubuntu, Suse e para versões genéricas. Para realizar a instalação, vou utilizar o Ubuntu, versão 14.04.

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/ACCC4
CF8.asc | sudo apt-key add -
```

Para manter sempre atualizado 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-9.6;
```

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. Ele abrirá a tela seguinte. Em seguida, clique em *Next*.

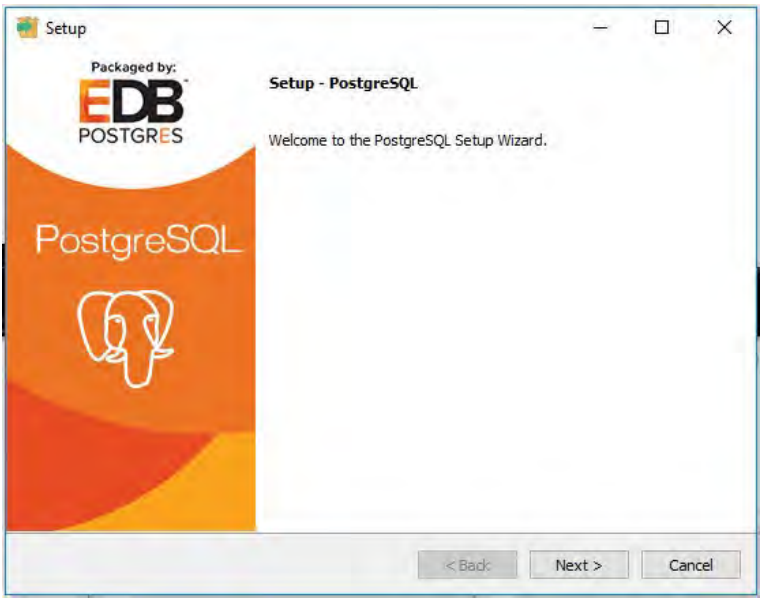

Figura 1.3: Instalação no Windows — Passo 1

Escolha a pasta onde será instalado e novamente *Next*.

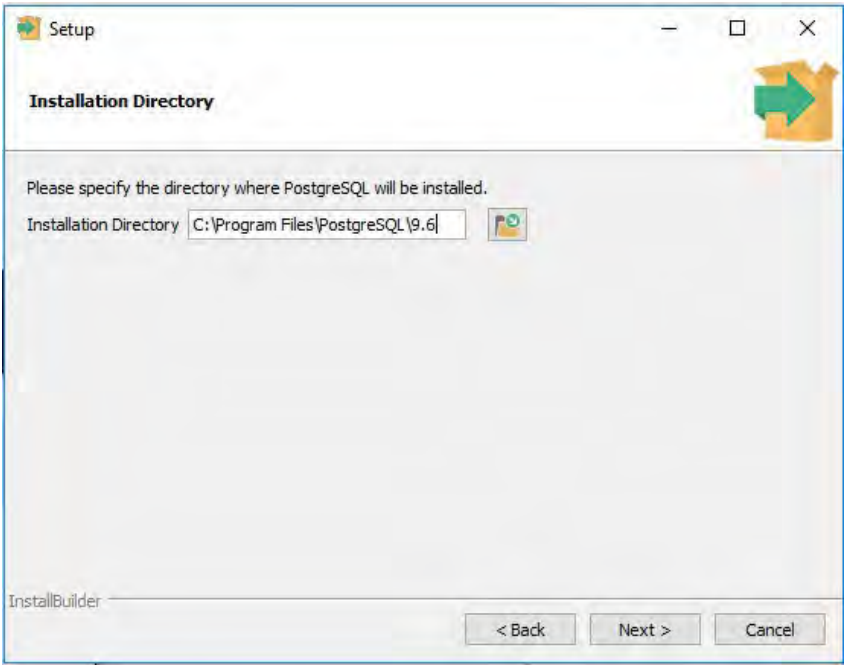

Figura 1.4: Instalação no Windows — Passo 2

Agora escolha a pasta na qual seus arquivos de dados ficarão.

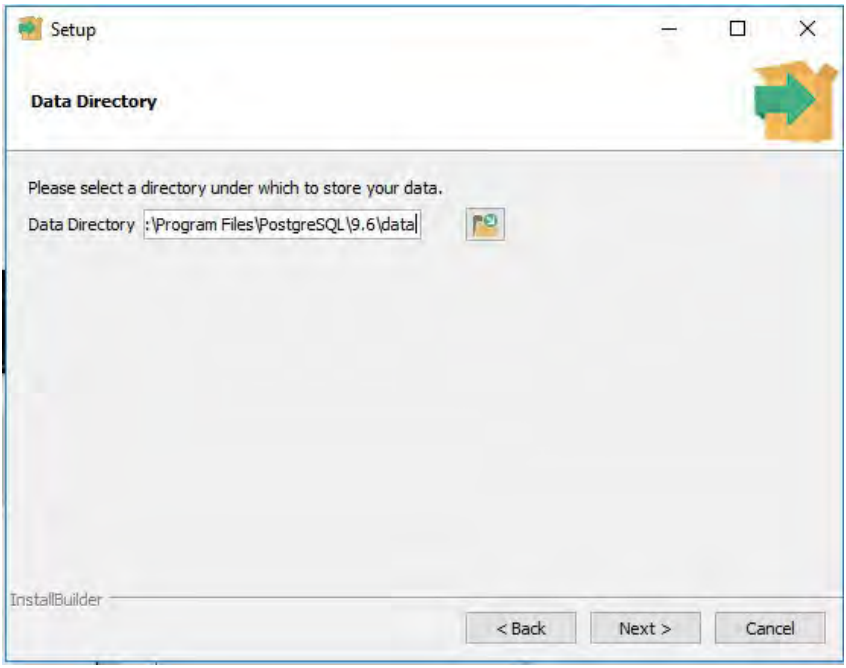

Figura 1.5: Instalação no Windows — Passo 3

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

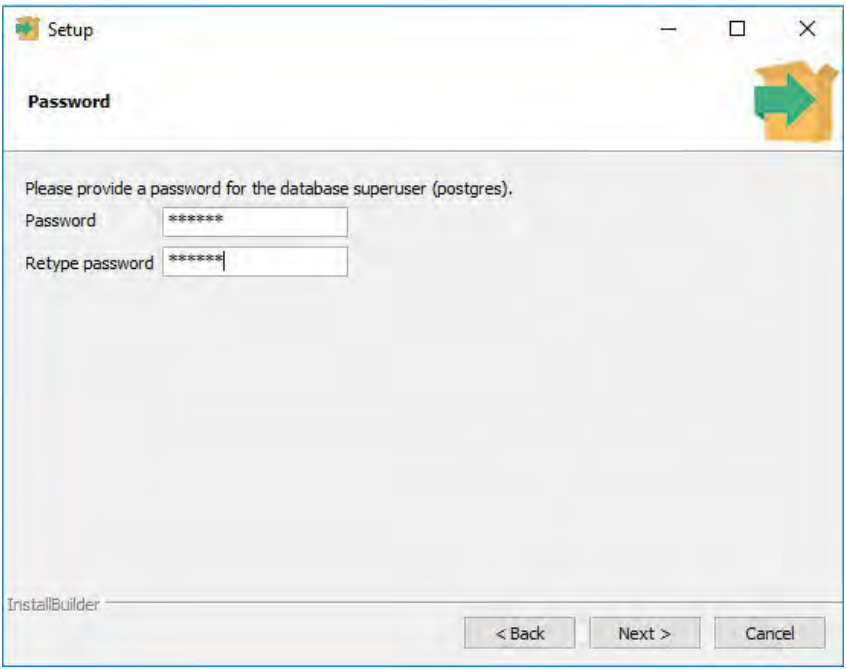

Figura 1.6: Instalação no Windows — Passo 4

Na tela seguinte, terá um *input box* com a porta de acesso de 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*.

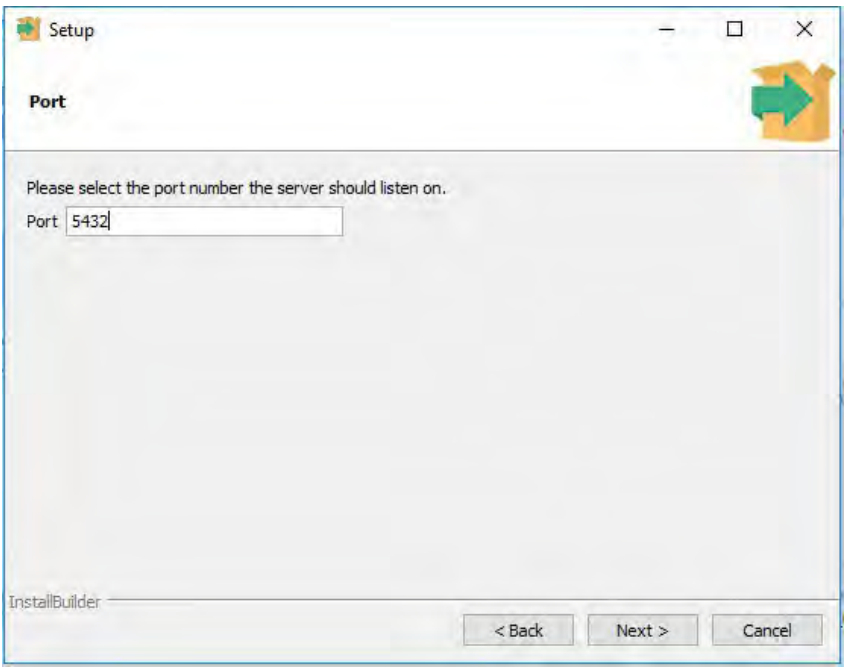

Figura 1.7: Instalação no Windows — Passo 5

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 isto, mostrarei mais à frente. No decorrer do livro, também darei exemplos de ferramentas para manipulação de dados, entre outras ferramentas para utilizar em mais de um sistema operacional.

Durante o desenvolvimento do projeto deste livro, usarei um Linux como meu sistema operacional principal para desenvolvimento. Se houver algum comando que fizer que é diferente em outros sistemas, mostrarei ambos, não se preocupe. Você pode programar com o sistema operacional que mais lhe agradar.

1.5 PARA PENSAR!

Se você vai desenvolver um sistema que várias pessoas vão utilizar, você precisa instalar o PostgreSQL em um servidor. A maioria dos desenvolvedores 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 neste 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. Darei 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 seu 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 e 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 ser 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 complexos 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 utilizar 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 DATATYPES: 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 problema tentar inserir um número decimal em campo que apenas suporta números inteiro.

Outro problema que é muito comum entre alguns desenvolvedores 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 alfa-numéricos. 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 campo `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, demonstrarei como usá-las.

Campos do tipo numéricos

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 numéricos, 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 precisar 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*. Este tipo de campo armazena números reais e com até 6

dígitos decimais.

- `double` : com a capacidade de armazenamento de 8 *bytes*. Pode armazenar até 15 dígitos nas casas decimais.
- `smallserial` : com a capacidade de armazenamento de 2 *bytes*. Este 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*. 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 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 ia inserindo 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 está errado.

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 cria 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!

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 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 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` .

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

- `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 este 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 deste tipo de campo é como a dos outros tipos. Mais adiante, quando formos desenvolver exemplos deste tipo de campo, veremos na prática a sua utilização e como podemos nos beneficiar.

Tipo JSON

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

Na versão 9.4, foi adicionado 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 colunas de uma tabela para ser definido 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, segue algumas maneiras de fazer a declaração deste 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 que 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.

Tipo personalizados

Também é possível criar tipos de dados personalizados, pelo

comando `create type` , no qual podemos criar um novo tipo de campo. Ele pode ser composto por vários campos, ou pode ser uma lista de valores.

2.3 PARA PENSAR!

Conhecemos diversos tipos de campos e 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 baseado 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 a descrição dele, e 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 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 esta 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 *scritps*.

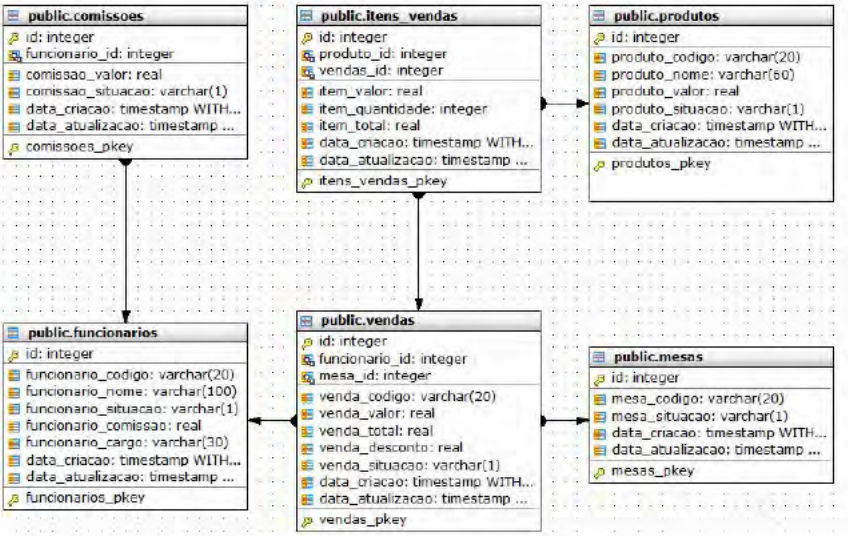

Figura 3.1: Diagrama de entidade e relacionamento

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

- `MESAS` : para cadastramos a quantidade de mesas que queremos que tenha no restaurante e seu respectivo código de identificação, pois assim podemos ampliar a quantidade 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.

- `VENDAS` : cada pedido de uma mesa será relacionado a uma venda, podendo esta estar relacionada ou não com um funcionário ou com 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_VENDAS` : 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.

- `PRODUTOS` : cada produto disponível para venda no restaurante deverá ser cadastrado na tabela de produtos. Vamos fazer um cadastro básico de produtos.

- `FUNCIONARIOS` : esta tabela será para cadastrar os funcionários do restaurante, na qual poderemos cadastrar os garçons, gerentes, atendentes etc. Vamos distingui-los por um campo de tipo, assim poderemos incluí-los nas vendas que estes atendem, e assim conseguir 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 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 da 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 seu projeto.

Estes cuidados estão diretamente ligados ao desempenho e 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 a desenvolver. 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 os desenvolvedores. 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 quão importante é esta etapa no

processo de desenvolvimento de um aplicativo. Mas o que são tabelas? E como são suas estruturas?

Segundo 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 conterá 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 é 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, na qual terá todos os tipos e, em nossa tabela de camisetas, apenas fazer uma referência a esta. Veja a figura a seguir.



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

Tabela: TIPO DE TECIDO

ID	DESCRIÇÃO TECIDO
1	ALGODÃO
2	JEANS

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). Na sequência estará detalhado cada um 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* (ou *chave primária*).

Vamos exemplificar este 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 deste cenário, conseguiremos manter a integridade dos dados.

Você deve ter uma chave primária, pois será um registro que não sofrerá alteração e 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.

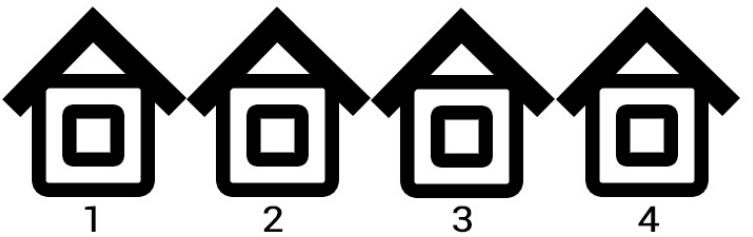

Figura 3.3: A identificação única das casas em uma rua

Em uma tabela de banco de dados, teríamos:



Figura 3.4: A identificação única dos registros em uma tabela

Dica!

Procure usar mecanismos de autoincremento em suas chaves primárias. Assim, você 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 neste quesito.

Chave estrangeira

As chaves estrangeiras, ou *foreign-keys*, 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 para sua tabela consistência e evitando erros.

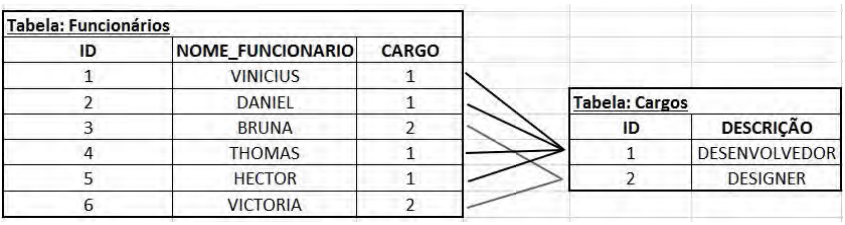

Tabela: Funcionários

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 CRIANDO NOSSAS TABELAS

Abra o terminal de comandos instalado na instalação do PostgresSQL para criarmos os objetos em nosso banco.

O `create table` é o comando usado para criar tabelas. Sabendo disso, vamos à criação delas.

```
# tabela para gravar registro das mesas

create table mesas (
     id               int not null primary key,
     mesa_codigo      varchar(20),
     mesa_situacao    varchar(1) default 'A',
     data_criacao     timestamp,
     data_atualizacao timestamp);

# tabela para gravar registro dos funcionários
create table funcionarios(
     id                    int not null primary key,
     funcionario_codigo    varchar(20),
     funcionario_nome      varchar(100),
```

```
     funcionario_situacao  varchar(1) default 'A',
     funcionario_comissao  real,
     funcionario_cargo     varchar(30),
     data_criacao          timestamp,
     data_atualizacao      timestamp);

# tabela para gravar registro das vendas
create table vendas(
     id               int not null primary key,
     funcionario_id   int references funcionarios (id),
     mesa_id          int references mesas(id),
     venda_codigo     varchar(20),
     venda_valor      real,
     venda_total      real,
     venda_desconto    real,
     venda_situacao   varchar(1) default 'A',
     data_criacao     timestamp,
     data_atualizacao timestamp);

# tabela para gravar registro dos produtos
create table produtos(
     id               int not null primary key,
     produto_codigo   varchar(20),
     produto_nome     varchar(60),
     produto_valor    real,
     produto_situacao varchar(1) default 'A',
     data_criacao     timestamp,
     data_atualizacao timestamp);

# tabela para gravar registro dos itens das vendas
create table itens_vendas(
     id                int not null primary key,
     produto_id        int not null references produtos(id),
     vendas_id int not null references vendas(id),
     item_valor        real,
     item_quantidade   int,
     item_total       real,
     data_criacao      timestamp,
     data_atualizacao  timestamp);

# tabela para gravar registro do cálculo das comissões
create table comissoes(
     id                int not null primary key,
     funcionario_id    int references funcionarios(id),
     comissao_valor    real,
     comissao_situacao varchar(1) default 'A',
     data_criacao      timestamp,
     data_atualizacao  timestamp);
```

Após criadas todas as nossas tabelas, para você ver se realmente estão criadas no banco de dados, utilize o comando `\dt` , como a figura a seguir. Uma lista com as tabelas será mostrada para você:

```
postgres=# \dt
            List of relations
 Schema |     Name      | Type  |  Owner
--------+---------------+-------+----------
 public | comissoes     | table | postgres
 public | funcionarios  | table | postgres
 public | itens_vendas  | table | postgres
 public | mesas         | table | postgres
 public | produtos      | table | postgres
 public | vendas        | table | postgres
(6 rows)
```

Figura 3.6: \d para listar as tabelas criadas no banco

3.5 CONSTRAINTS: INTEGRIDADE DE SEUS DADOS

Os tipos de dados são uma forma para limitar o tipo de dados que pode ser armazenado 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, estarei demonstrando os tipos de `constraints` e demonstrando os tipos de erros que podemos enfrentar.

As *contraints* 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, use o comando `\d nome_tabela` como mostra a figura a seguir. Observe que, além de listar as *constraints*, este comando serve para listar os campos de uma tabela. Vamos utilizá-lo para visualizar as colunas da tabela `vendas`.

```
postgres=# \d vendas;
```

E como resultado, teremos:

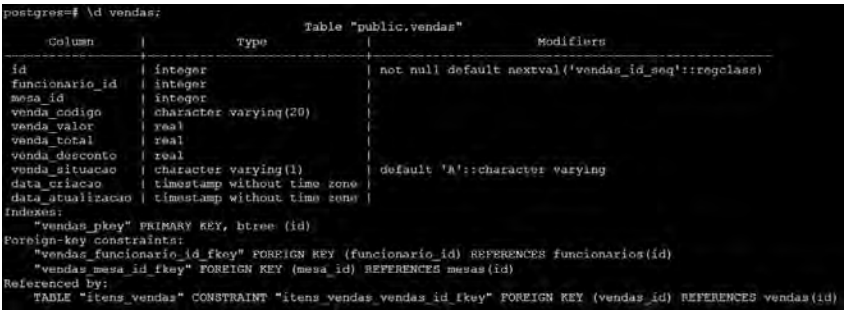

Figura 3.7: \d \d vendas - listas dos campos da tabela e suas constraints

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. 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 separados.

Para excluir uma tabela, use o comando:

```
postgres=> drop table comissoes;
```

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

```
create table comissoes(
     id                int not null,
      funcionario_id   int ,
     comissao_valor    real,
     comissao_situacao varchar(1) default 'A',
     data_criacao      timestamp,
     data_atualizacao  timestamp);
```

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

```
postgres=> alter table comissoes
           add constraint comissoes_pkey primary key(id);
```

Agora a FK que referencia a tabela `FUNCIONARIOS`:

```
postgres=> alter table comissoes
           add foreign key (funcionario_id) references funcionario
s(id);
```

Para verificarmos que tudo ocorreu bem, vamos listar os campos e as *constraints* com o comando `\d comissoes`, como mostra a figura a seguir.



Figura 3.8: \d vendas - listas dos campos da tabela e suas constraints

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

```
postgres=> alter table comissoes
           drop constraint comissoes_funcionario_id_fkey;
```

> **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, este recurso é utilizado para fazer validações e criar maneiras de se manter a integridades 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 esta situação? Podemos criar uma *constraint* para validar se o valor total da venda é positivo. Assim, não será necessário criar uma validação na aplicação e 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.

```
postgres=> alter table vendas 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 chegará se o campo está recebendo um valor nulo na inserção de um novo registro.

Sabendo disso, vamos criar uma *constraint* na tabela `FUNCIONARIOS` para não permitir que seja inserido nulo no campo `funcionario_nome` . Assim, o nosso banco vai verificar se tentamos inserir um valor nulo, assim como o `not null` faria.

```
postgres=> alter table funcionarios
            add check( funcionario_nome <> null);
```

3.6 CRIANDO SEQUÊNCIAS PARA AS NOSSAS TABELAS

Em todas nossas tabelas, temos uma chave primária. Em nosso padrão, adotei um campo chamado `id` como padrão para as nossas chaves primárias. Como este campo deve conter uma sequência numérica única, nada mais lógico do que termos um mecanismo para gerar este `id` único e automaticamente. Por isso temos as chamadas `sequences` , que são sequências numéricas autoincrementadas.

Criando uma `sequence` para cada tabela, o campo `id` vai se autoincrementar a cada inserção em cada tabela. Como comentei anteriormente, as chaves primárias, como boas práticas de programação, não é aconselhável utilizá-las para mostrar na tela.

Para cada tabela do projeto, vamos criar uma `sequence` , utilizando o comando `CREATE SEQUENCE` . Para o nome de nossas

`sequences`, usaremos o padrão `_NOMETABELA_NOMECOLUNA_SEQ_`. Este padrão é como eu gosto de utilizar. Você pode (e eu aconselho) criar o seu também. Com este padrão, o nome da `sequence` para a tabela `mesas` ficaria: `mesas_id_seq`. Então, vamos criar todas as sequências com os seguintes códigos:

```
postgres=> create sequence mesa_id_seq;
postgres=> create sequence vendas_id_seq;
postgres=> create sequence itens_vendas_id_seq;
postgres=> create sequence produtos_id_seq;
postgres=> create sequence funcionario_id_seq;
postgres=> create sequence comissoes_id_seq;
```

Depois das sequências criadas, devemos vincular cada `sequence` com suas respectivas tabelas, da seguinte maneira:

```
postgres=> alter table mesas
           alter column id set default nextval('mesa_id_seq');
postgres=> alter table vendas
           alter column id set default nextval('vendas_id_seq');
postgres=> alter table itens_vendas
           alter column id set default nextval('itens_vendas_id_se
q');
postgres=> alter table produtos
           alter column id set default nextval('produtos_id_seq');
postgres=> alter table funcionarios
           alter column id set default nextval('funcionario_id_seq
');
postgres=> alter table comissoes
           alter column id set default nextval('comissoes_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 este 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:

```
postgres=> drop sequence funcionario_id_seq cascade;
```

O resultado deverá ser o mostrado na figura a seguir.

```
postgres=#  drop sequence funcionario_id_seq cascade;
NOTICE:  drop cascades to default for table funcionarios column id
DROP SEQUENCE
```

Figura 3.9: Drop sequence ... cascade

Devemos utilizar o comando `cascade` no final do nosso comando para o banco também deletar o vínculo feito com a tabela `FUNCIONARIOS`. 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`:

```
postgres=> create sequence proximo_numero;
```

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

```
postgres=> drop sequence proximo_numero;
```

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 `COMISSOES`, o campo de `DATA_PAGAMENTO`, para informar a data 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 colun...` .

```
postgres=> alter table comissoes
             add column 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. Nesta situação, como não há ainda registros nesta nova coluna, temos duas escolhas: ou excluímos o campo e criamos novamente, ou modificamos o seu tipo.

Para excluir uma coluna, utilizamos o comando `alter table...drop column` .

```
postgres=> alter table comissoes
                  drop column data_pagamento;
postgres=> alter table comissoes
                   add column data_pagamento timestamp;
```

Se escolhermos apenas modificar o tipo da coluna, usamos o comando `alter table...alter column...type...` .

```
postgres=> alter table comissoes
           alter column data_pagamento type timestamp
           using data_pagamento_timestamp;
```

3.7 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, escolha nome de pessoas que você

conhece. Na tabela de produtos, utilize objetos com que você tenha familiaridade.

Inserindo registros

Para inserir registros, vamos utilizar o comando `INSERT INTO... VALUES...` . Usamos constantemente este 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. Esta 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.

```
postgres=> insert into mesas (mesa_codigo,
                              mesa_situacao,
                              data_criacao,
                              data_atualizacao)
                      values ('00001',
                              'A',
                              '01/01/2016',
                              '01/01/2016');

postgres=> insert into mesas (mesa_codigo,
                              mesa_situacao,
                              data_criacao,
                              data_atualizacao)
                     values ('00002',
                             'A',
                             '01/01/2016',
                             '01/01/2016');
```

Observe que, no comando, nós suprimos o campo `id` da tabela `MESAS` . Isso porque nós criamos uma `sequence` para inserir esse valor de todas as tabelas. Vamos continuar com a inserção dos registros.

```
postgres=>  insert into funcionarios(funcionario_codigo,
                                     funcionario_nome,
                                     funcionario_situacao,
```

```
                                    funcionario_comissao,
                                    funcionario_cargo,
                                    data_criacao)
                            values('0001',
                                   'VINICIUS CARVALHO',
                                   'A',
                                    5,
                                   'GERENTE',
                                   '01/01/2016');

postgres=> insert into funcionarios(funcionario_codigo,
                                   funcionario_nome,
                                   funcionario_situacao,
                                   funcionario_comissao,
                                   funcionario_cargo,
                                   data_criacao)
                            values('0002',
                                   'SOUZA',
                                   'A',
                                    2,
                                   'GARÇOM',
                                   '01/01/2016');

postgres=> insert into produtos (produto_codigo,
                                 produto_nome,
                                 produto_valor,
                                 produto_situacao,
                                 data_criacao,
                                 data_atualizacao)
                          values ('001',
                                  'REFRIGERANTE',
                                  10,
                                  'A',
                                  '01/01/2016',
                                  '01/01/2016');

postgres=> insert into produtos (produto_codigo,
                                 produto_nome,
                                 produto_valor,
                                 produto_situacao,
                                 data_criacao,
                                 data_atualizacao)
                           values ('002',
                                   'AGUA',
                                   3,
                                   'A',
                                   '01/01/2016',
                                   '01/01/2016');
```

```
postgres=> insert into produtos (produto_codigo,
                                 produto_nome,
                                 produto_valor,
                                 produto_situacao,
                                 data_criacao,
                                 data_atualizacao)
                          values ('003',
                                  'PASTEL',
                                  7,
                                  'A',
                                  '01/01/2016',
                                  '01/01/2016');

postgres=> insert into vendas (funcionario_id,
                               mesa_id,
                               venda_codigo,
                               venda_valor,
                               venda_total,
                               venda_desconto,
                               venda_situacao,
                               data_criacao,
                               data_atualizacao)
                        values (2,
                                1,
                                '0001',
                                '20',
                                '20',
                                '0',
                                'A',
                                '01/01/2016',
                                '01/01/2016');

postgres=> insert into vendas (funcionario_id,
                               mesa_id,
                               venda_codigo,
                               venda_valor,
                               venda_total,
                               venda_desconto,
                               venda_situacao,
                               data_criacao,
                               data_atualizacao)
                        values (2,
                                2,
                                '0002',
                                '21',
                                '21',
                                '0',
```

```
                                'A',
                                '01/01/2016',
                                '01/01/2016');


postgres=> insert into itens_vendas (produto_id,
                                     vendas_id,
                                     item_valor,
                                     item_quantidade,
                                     item_total,
                                     data_criacao,
                                     data_atualizacao)
                             values (1,
                                     1,
                                     10,
                                     2,
                                     20,
                                     '01/01/2016',
                                     '01/01/2016');

postgres=> insert into itens_vendas(produto_id,
                                    vendas_id,
                                    item_valor,
                                    item_quantidade,
                                    item_total,
                                    data_criacao,
                                    data_atualizacao)
                                values(1,
                                       2,
                                       7,
                                       3,
                                       21,
                                       '01/01/2016',
                                       '01/01/2016');
```

Não vamos inserir registro na tabela `COMISSOES` , pois nesta 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.

3.8 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 `MESAS`. Para isso, usaremos o `SELECT` e o `FROM`. Este é o responsável por dizer para o banco de dados qual é tabela que desejamos consultar. Nosso comando e o resultado ficarão da seguinte maneira:

```
postgres=> select * from mesas;
```

```
postgres=# select * from mesas;
 id | mesa_codigo | mesa_situacao |    data_criacao     |  data_atualizacao
----+-------------+---------------+---------------------+---------------------
  1 | 00001       | A             | 2016-01-01 00:00:00 | 2016-01-01 00:00:00
  2 | 00002       | A             | 2016-01-01 00:00:00 | 2016-01-01 00:00:00
(2 rows)
```

Figura 3.10: 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:

```
postgres=> select mesa_codigo, data_criacao from mesas;
```

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 que queremos buscar. Com o comando `where`, vamos informar qual o registro que desejamos buscar. Vamos então consultar a mesa que possui o `mesa_codigo` igual a dois.

```
postgres=> select * from mesas where mesa_codigo = '00002';
```

Observe que o código `00002` foi colocado entre aspas simples. Isto 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.

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`.

Vamos alterar o produto de `id = 2`, que é o produto `AGUA`, colocando um novo valor para ele. Agora ele passará a custar 4.

```
postgres=> update produtos set produto_valor = 4
                      where id = 2;
```

Observe que utilizamos também o comando `WHERE` para indicar qual produtos 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 `data_criacao` de todos os produtos.

```
postgres=> update produtos set data_criacao = '31/12/2016';
```

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

```
postgres=> select data_criacao from produtos;
```

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.

```
postgres=> delete from mesas where 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 `id = 2`. O `delete` sem `where` é muito famoso em pregar peças nos desenvolvedores desatentos.

3.9 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 utiliza 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. Nelas podemos escrever instruções para realizar operações como: consultar e retornar valores, realizar cálculos e retornar ou não valores, chamar outros procedimentos, e mais.

É comum no cotidiano dos desenvolvedores 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 `id`, eu crio uma `function` na qual passo o `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_situacao` . Para isso, vamos levar em consideração a seguinte tabela de possíveis valores para o campo `funcionario_situacao` .

Sigla	Descrição
A	Ativo
I	Inativo

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

No código, vamos passar como parâmetro o `id` do funcionário, e depois teremos uma consulta para buscar o funcionário e, 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.

```
postgres=# create or replace function
            retorna_nome_funcionario(func_id int)
            returns text as
            $$
            declare
            nome     text;
            situacao text;
            begin

              select funcionario_nome,
                     funcionario_situacao
                into nome, situacao
                from funcionarios
               where id = func_id;

               if situacao = 'A' then
                 return nome || ' Usuário Ativo';
               else
                 return nome || ' Usuário 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;
  situacao text;
```

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, este 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 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_situacao` do funcionário era igual a `A`, e escrevemos a instrução para que, se ele

fosse igual, concatenasse as palavras `Usuário Ativo` com o seu nome. Caso contrário (`else`), ele concatenaria as palavras `Usuário Inativo` .

Há a possibilidade de testarmos quantas condições que quisermos. Basta apenas 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_situacao` é igual a `A` , `I` , vazio ou diferente das três condições.

```
$$
begin

  if situacao = 'A' then

    'Usuário Ativo';

  elsif situacao = 'I' then

     'Usuário Inativo'

  elsif situacao is null then

    'Usuário Sem status'

  else

    'Usuário com status diferente de A e I'

  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 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 `id` do funcionário. Sabendo disso, vamos passar como parâmetro o `id 1` .

```
postgresql=> select retorna_nome_funcionario(1);
```

O resultado da nossa consulta será:

```
postgres=#  select retorna_nome_funcionario(1);
     retorna_nome_funcionario
-------------------------------------
 VINICIUS CARVALHO Usuário Ativo
(1 row)
```

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, quanto 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 utilizar mais à frente para criar um processo que calculará um valor de comissão de cada venda. Então, vamos lá,

mãos no teclado.

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

```
postgresql=> create or replace function
            rt_valor_comissao(func_id int)
            returns real as
            $$
            declare
              valor_comissao real;

            begin
              select funcionario_comissao
                into valor_comissao
                from funcionarios
               where id = func_id;
               return valor_comissao;
            end
            $$
            LANGUAGE plpgsql;
```

Vamos testar nossa `function`.

```
postgresql=> select rt_valor_comissao(1);
```

E como resultado, temos:

```
postgres=# select rt_valor_comissao(1);
 rt_valor_comissao
-------------------
                 5
(1 row)
```

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 funcionários das vendas que tiveram em algum período. Aproveite este exemplo para exercitar a inserção de valores nas tabelas e popule a tabela `vendas` e `itens_vendas`, para que você tenha vários registros para trabalhar, além dos que inserimos anteriormente. Agora, mãos no teclado e vamos a nossa `function`.

Para este 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_situacao` seja igual a `A`. Isso porque, após calcular a comissão da venda, vamos fazer um `udpate` nesse campo para `C`, 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 `comissoes`.

```
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  real := 0;
   porc_comissao   real := 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 int) IS
      select rt_valor_comissao(func_id);

 begin

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

    for reg in(
      select vendas.id id,
             funcionario_id,
             venda_total
        from vendas
       where data_criacao >= data_ini
         and data_criacao <= data_fim
         and venda_situacao = 'A')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 irá receber de comissão
-- daquela venda

      insert into comissoes(
                             funcionario_id,
```

```
                                comissao_valor,
                                comissao_situacao,
                                data_criacao,
                                data_atualizacao)
            values(reg.funcionario_id,
                   total_comissao,
                   'A',
                   now(),
                   now());

        -- update na situação da venda
        -- para que ela não seja mais comissionada

            update vendas set venda_situacao = 'C'
             where id = reg.id;

        -- devemos zerar as variáveis para reutilizá-las

             total_comissao := 0;
             porc_comissao  := 0;

        -- término do loop

            end loop;

         end
         $$ language plpgsql;
```

Para executar este processo, usaremos o seguinte comando:

```
postgresql=> select calc_comissao('01/01/2016 00:00:00','01/01/201
6 00:00:00');
```

Como resultado, teremos:

```
postgres=#  select calc_comissao('01/01/2016','01/01/2016');
 calc_comissao
---------------

(1 row)
```

Figura 4.3: Resultado da função calc_comissao

Agora vamos ver os registros na tabela `comissoes` . Vamos consultar as comissões do funcionário `id = 1` .

```
postgresql=> select comissao_valor,
                    data_criacao
               from comissoes;
```

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

```
postgres=# select comissao_valor,
postgres-#                    data_criacao
postgres-#                from comissoes;
 comissao_valor |        data_criacao
----------------+----------------------------
            0.4 | 2016-11-22 04:50:13.434059
           0.42 | 2016-11-22 04:50:13.434059
           2.55 | 2016-11-22 04:50:13.434059
              1 | 2016-11-22 04:50:13.434059
           2.25 | 2016-11-22 04:50:13.434059
(5 rows)
```

Figura 4.4: Comissões geradada do funcionário id = 1

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 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 é consumido um pouco de memória do servidor 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_comissoes` com o comando:

```
postgresql=> drop function calc_comissoes();
```

> **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. Isto por causa de linguagens de programação que fazem isso bem, como o 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 aos desenvolvedores 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 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 function! 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 no PostgreSQL, assim como em 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 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.

Estes 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 destes tipos de dados.

Uma preocupação levada em consideração nos mais variados SGBD, ao usarem este 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  vendas
where data_criacao >= 'data_ini'
  and data_criacao <= 'data_fim'
  and produto_situacao = 'A'
```

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 com eles.

Operador	Descrição
AND	Utilizamos quando queremos incluir duas ou mais condições em nossa operação. Os registros recuperados em uma declaração que une duas

	condições com este operador deverão suprir as duas ou mais condições
OR	Utilizamos quando queremos combinar duas ou mais condições em nossa operação. Os registros recuperados em uma declaração que une duas condições com este operador deverão suprir uma das duas condições
NOT	Utilizamos 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.

```
postgres=> insert into produtos (produto_codigo,
                                 produto_nome,
                                 produto_valor,
                                 produto_situacao,
                                 data_criacao,
                                 data_atualizacao)
                         values ('2832',
                                 'SUCO DE LIMÃO',
                                  15,
                                 'C',
                                 '02/02/2016',
                                 '02/02/2016');
```

Agora vamos consultar os produtos que possuem os produtos cancelados **E** os produtos que estão ativos.

```
postgres=> select *
             from produtos
            where produto_situacao = 'A'
              and produto_situacao = 'C';
```

```
 id | produto_codigo | produto_nome | produto_valor | produto_situacao | data_criacao | data_atualizacao | produto_categoria | produ
to_estoque
----+----------------+--------------+---------------+------------------+--------------+------------------+-------------------+------
-----------
(0 rows)
```

Figura 5.1: Consulta com o operador AND

Como você pode ver, o resultado de nossa consulta não nos trouxe nada. 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.

```
postgres=> select *
             from produtos
            where produto_situacao = 'A'
               or produto_situacao = 'C';
```

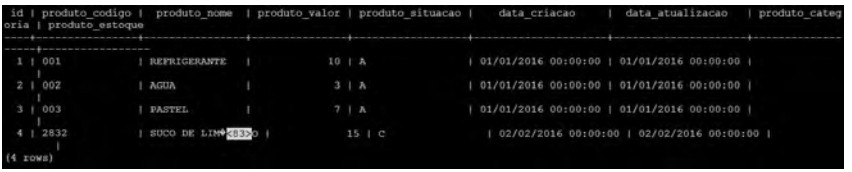

Figura 5.2: Consulta com o operador OR

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.

```
postgres=> select *
             from produtos
            where not produto_nome = 'SUCO DE LIMÃO';
```

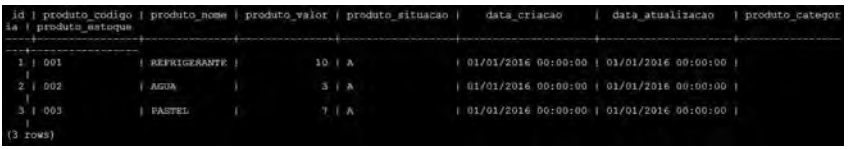

Figura 5.3: Consulta com o operador NOT

Na última consulta, negamos a condição declarada, que era `produto_nome = 'SUCO DE LIMÃO'`.

Além de utilizá-los separadamente, podemos criar consultas unindo-os. Mãos no teclado. Vamos criar uma consulta para buscar os produtos que estão ativos **OU** que estejam cancelados **E** possuam

data de criação igual a `02/02/2016` .

```
postgres=> select *
             from produtos
            where produto_situacao = 'A'
               or (produto_situacao = 'C'
                   and data_criacao = '02/02/2016');
```

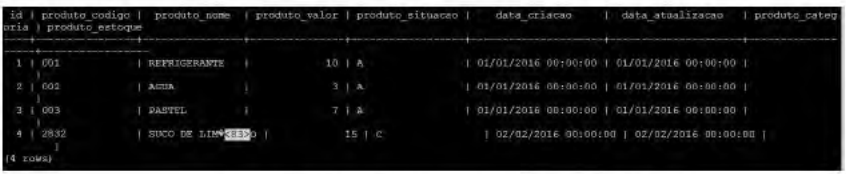

Figura 5.4: Consulta com os operadores AND e OR

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 id,
       funcionario_id,
       venda_total
  from vendas
 where data_criacao >= 'data_ini'
   and data_criacao <= 'data_fim'
   and venda_situacao = 'A';
```

Se quiséssemos excluir os valores das variáveis `data_ini` e `data_fim` utilizaríamos apenas os operadores `>` e `<`. Pois assim consultaríamos apenas as datas entre esses dois valores. Como por exemplo no código a seguir:

```
select id,
       funcionario_id,
       venda_total
  from vendas
 where data_criacao >= '01/01/2016'
   and data_criacao < '02/02/2016'
   and venda_situacao = 'A';
```

Nesta última consulta, as datas levadas em consideração seriam a `01/01/2016` e todas as menores que `02/02/2016`, assim excluindo a data `02/02/2016`, pois utilizamos o operador `<` e não o operador `<=` que iria incluir a data `02/02/2016` na consulta.

5.4 OPERADORES E FUNÇÕES MATEMÁTICAS

Operadores matemáticos

São fornecidos operadores matemáticos para muitos tipos de dado 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	2 + 3	5
-	subtração	2 - 3	-1
*	multiplicação	2 * 3	6
/	divisão (divisão inteira trunca o resultado)	4 / 2	2
%	módulo (resto)	5 % 4	1
^	exponenciação	2.0 ^ 3.0	8
!	fatorial	5 !	120
! !	fatorial (operador de prefixo)	!! 5	120
@	valor absoluto	@ -5.0	5

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.

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

```
postgres=> select 5!;
```

```
postgres=# select 5!;
 ?column?
----------
      120
(1 row)
```

Figura 5.5: 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 sabia 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	*select abs(-5);*	5
cbrt(variável)	Para calcular o valor da raiz cúbica de uma variável	*select cbrt(8);*	2
ceil(variável)	Para calcular o menor valor que seja maior ou igual à variável	*select ceil(14.2);*	15
ceiling(variável)	O mesmo que o ceil	*select ceil(-51.1);v	-31
degrees(variável)	Utilizado para converter um valor de radianos para graus	*select degrees(1)*	57.2957795130823
div(variável x, variável y)	Utilizado para fazer a divisão entre dois números	*select div(8,4);*	2
exp(variável)	Utilizado para fazer cálculo de exponenciação	*select exp(2);*	7.38905609893065
floor(variável)	Utilizado para encontrar o maior número inteiro não maior que a variável	*select floor(-12.9);*	-13
ln(variável)	Calcular e mostrar o valor do	*select ln(2.0)*	0.693147180559945

	logaritmo comum		
log(variável)	Utilizado para calcular logaritmo na base 10	select log(10)	2
log(variável b, variável k)	Logaritmo na base b	*select log(100.2);*	2.0008677215312267
mod(variável x, variável y)	Cálculo do resto da divisão de dois números	*select mod(11,2);*	1
pi()	Retorna o valor de pi	*select pi();*	3.14159265358979
power(variável x , variável y)	Faz o cálculo exponencial da variável x pela y	*select power(9.0, 3.0);*	729
radians(variável)	Faz o cálculo de conversão para graus radianos	*select rad(12);*	0.20943951023932
round(variável)	Faz o arredondamento da variável informada	*select round(25.2);*	25
round(variável x, variável y)	Realiza o arredondamento para o número especificado depois da vírgula	*select round(54.123,2);*	54.12
sqrt(variável)	Calcula a raiz quadrada de um número	*selecr sqrt(9);*	3
trunc(variável)	Utilizado para separar o número inteiro dos decimais	*select trunc(335.23);*	335
trunc(variável x, variável y)	Utilizado para separar uma quantidade específica de números decimais	*select trunc(335.123,2);*	335.12

Há também algumas funções trigonométricas. São elas:

Function	Descrição	Exemplo	Resultado

acos(variável X)	Utilizado para calcular o inverso do cosseno	*select acos(0);*	1.5707963267949
asin(variável X)	Utilizado para calcular o inverso do seno	*select asin(0.2);*	0.201357920790331
atan(variávelx)	Utilizado para calcular o inverso da tangente	*select atan(2);*	1.10714871779409
cos(variávelx)	Utilizado para calcular o valor do cosseno	*select cos(3);*	-0.989992496600445
cot(variávelx)	Utilizado para calcular o valor do cotangente	*select cot(3);*	-7.01525255143453
sin(variávelx)	Utilizado para calcular o valor do seno	*select sin(3);*	0.141120008059867
tan(variávelx)	Utilizado para calcular o valor da tangente	*select tan(3);*	-0.142546543074278

5.5 FUNÇÕES DE TEXTO

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

|| 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:

```
postgres=# select funcionario_codigo || funcionario_nome
               from funcionarios
              where id = 1;
```

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

```
select (funcionario_codigo ||' '|| funcionario_nome)
  from funcionarios
  where id = 1;
```

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 funcionarios
  where id = 1;
```

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.

```
postgres=# select char_length(funcionario_nome)
               from funcionarios
              where id = 2;
```

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, 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.

```
postgres=# select upper(funcionario_nome)
              from funcionarios;
```

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:

```
postgres=# 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`.

```
postgres=# 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.

```
postgres=# select lower(funcionario_nome)
             from funcionarios;
```

Substituindo string com overlay() e extraindo 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.

```
postgres=# select overlay(funcionario_nome placing '000000'
                          from 3 for 5)
              from funcionarios
             where id = 1;
```

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. Mãos no código.

```
postgres=# select substring(funcionario_nome from 3 for 5)
              from funcionarios
             where id = 1;
```

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 pesquisar o termo dentro de um texto. Esta função tem o objetivo de identificar em qual posição se encontra um caractere ou se inicia uma `string`.

Utilizando o nome do funcionário com `id = 1` cujo nome é `VINICIUS`, vamos localizar em qual posição se encontra a `string` `CIUS`.

```
postgres=# select position('CIUS' in funcionario_nome)
              from funcionarios
             where id = 1;
```

5.6 FUNÇÕES DATA/HORA

Se você já trabalha com programação ou se não trabalha, vai descobrir que trabalhar com data e hora em seu sistema dá um pouco de trabalho. Não importa a linguagem, de cada três

desenvolvedores, dois já tiveram problemas neste 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 esta informação, que é o `datestyle`.

```
postgres=# show datestyle;
```

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

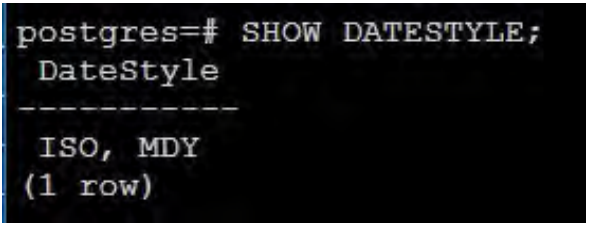

Figura 5.6: Formato de datas atual do banco de dados

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`.

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

E se você estiver logado no terminal do banco de dados, você pode executá-lo para aplicar essa alteração na sessão em que estiver logado.

```
postgres=# set datestyle to iso, dmy;
```

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

```
postgres=# SHOW DATESTYLE;
 DateStyle
-----------
 ISO, DMY
(1 row)
```

Figura 5.7: 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 sua idade exatamente. Mãos no teclado e vamos ao nosso código.

```
postgres=# select age(timestamp '04/11/1988');
```

```
postgres=# select age(timestamp '04/11/1988');
           age
-------------------------
 27 years 4 mons 21 days
(1 row)
```

Figura 5.8: Calculando a idade

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

```
postgresq=# select age(timestamp '07/05/2016',
                       timestamp '12/05/2007');
```

```
postgres=# select age(timestamp '07/05/2016',
postgres(#                      timestamp '12/05/2007');
          age
-----------------------
 8 years 11 mons 26 days
(1 row)
```

Figura 5.9: 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
clock_timestamp()	Data e hora atual	*select clock_timestamp();*
current_date	Data atual	*select current_date;*
current_time	Hora atual	*select current_time;*
current_timestamp	Data e hora atual	*select current_timestamp;*
localtime	Hora atual	*select localtime;*
localtimestamp	Data e hora atual	*select localtimestamp;*
now()	Data e hora atual	*select now();*
statement_timestamp()	Data e hora atual	*select statement_timestamp();*
timeofday()	Data e hora atual no formato de texto	*select timeofday();*

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
date_part('day', timestamp)	Extrai o dia da data/hora informada	*select date_part('day', timestamp '04/11/1988 20:38:40');*
date_part('month', timestamp)	Extrai o mês da data/hora informada	*select date_part('day', timestamp '04/11/1988 20:38:40');*
date_part('year', timestamp)	Extrai o ano da data/hora informada	*select date_part('day', timestamp '04/11/1988 20:38:40');*
date_part('hour', timestamp)	Extrai a hora da data/hora informada	*select date_part('day', timestamp '04/11/1988 20:38:40');*

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

Uma outra função muito interessante é o `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
century	Para extrair o século de uma determinada data	*select extract (century from timestamp '04/11/1988 12:21:13');*
day	Para extrair o dia de uma determinada data	*select extract (day from timestamp '04/11/1988 12:21:13');*
decade	Para a extrair a década de uma determinada data	*select extract (decade from timestamp '04/11/1988 12:21:13');*
doy	Para extrair o dia do ano de uma	*select extract (doy from timestamp*

	determinada data	*'04/11/1988 12:21:13');*
hour	Para extrair a hora de um determinado horário	*select extract (hour from timestamp '04/11/1988 12:21:13');*
year	Para extrair o ano de uma determinada data	*select extract (year from timestamp '04/11/1988 12:21:13');*
minute	Para extrair os minutos de um determinado horário	*select extract (minute from timestamp '04/11/1988 12:21:13');*
month	Para extrair o mês de uma determinada data	*select extract (month from timestamp '04/11/1988 12:21:13');*
second	Para extrair o valor dos segundos de um determinado horário	*select extract (second from timestamp '04/11/1988 12:21:13');*

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

```
postgres=# select extract(year from data_criacao)
             from funcionarios where id = 1;
```

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 de um programador. 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 `funcionarios` ? 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, o `count(*)`.

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

```
postgres=# select count(*)
             from funcionarios;
```

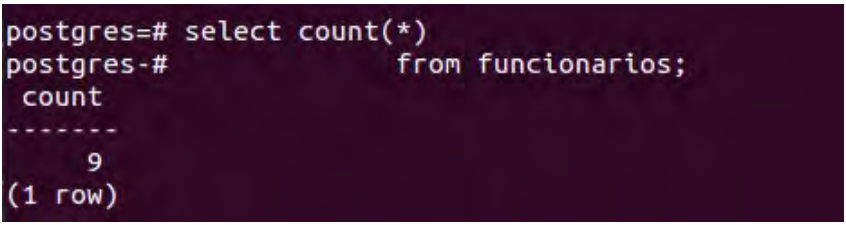

Figura 5.10: Contando os registro da tabela funcionarios

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(**id**) from funcionarios`.

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.

```
postgres=# select sum(venda_total)
             from vendas;
```

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

```
postgres=#  select sum(venda_total)
postgres-#              from vendas;
 sum
-----
  41
(1 row)
```

Figura 5.11: 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()`.

```
postgres=# select avg(produto_valor)
             from produtos;
```

```
postgres=# select avg(produto_valor) from produtos;
 avg
------
 8.75
(1 row)
```

Figura 5.12: 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!

```
postgres=# select max(produto_valor), min(produto_valor)
             from produtos;
```

```
postgres=# select max(produto_valor), min(produto_valor)
postgres-#              from produtos;
 max | min
-----+-----
  15 |   3
(1 row)
```

Figura 5.13: 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. Mãos no teclado e vamos à nossa consulta. Mas antes de criarmos, vamos inserir mais alguns registros nas tabelas `vendas` e `itens_vendas` para termos mais dados para testarmos.

```
postgres=> insert into vendas (id,
                               funcionario_id,
                               mesa_id,
                               venda_codigo,
                               venda_valor,
                               venda_total,
                               venda_desconto,
                               venda_situacao,
                               data_criacao,
                               data_atualizacao)
                       values (10000,
                               1,
                               1,
                               '10201',
                               '51',
                               '51',
                               '0',
                               'A',
                               '01/01/2016',
                               '01/01/2016');

postgres=> insert into itens_vendas (produto_id,
                                     vendas_id,
                                     item_valor,
                                     item_quantidade,
                                     item_total,
                                     data_criacao,
                                     data_atualizacao)
                             values (4,
                                     10000,
                                     15,
                                     2,
                                     30,
                                     '01/01/2016',
                                     '01/01/2016');

postgres=> insert into itens_vendas (produto_id,
                                     vendas_id,
                                     item_valor,
                                     item_quantidade,
```

```
                                item_total,
                                data_criacao,
                                data_atualizacao)
                         values (3,
                                10000,
                                7,
                                3,
                                21,
                                '01/01/2016',
                                '01/01/2016');

postgres=> insert into vendas (id,
                               funcionario_id,
                               mesa_id,
                               venda_codigo,
                               venda_valor,
                               venda_total,
                               venda_desconto,
                               venda_situacao,
                               data_criacao,
                               data_atualizacao)
                       values (10001,
                                1,
                                1,
                                '10201',
                                '20',
                                '20',
                                '0',
                                'A',
                                '01/01/2016',
                                '01/01/2016');

postgres=> insert into itens_vendas (produto_id,
                                     vendas_id,
                                     item_valor,
                                     item_quantidade,
                                     item_total,
                                     data_criacao,
                                     data_atualizacao)
                              values (1,
                                     10001,
                                     10,
                                     2,
                                     20,
                                     '01/01/2016',
                                     '01/01/2016');

postgres=> insert into vendas (id,
```

```
                                funcionario_id,
                                mesa_id,
                                venda_codigo,
                                venda_valor,
                                venda_total,
                                venda_desconto,
                                venda_situacao,
                                data_criacao,
                                data_atualizacao)
                        values (10002,
                                1,
                                1,
                                '10002',
                                '45',
                                '45',
                                '0',
                                'A',
                                '01/01/2016',
                                '01/01/2016');

postgres=> insert into itens_vendas (produto_id,
                                     vendas_id,
                                     item_valor,
                                     item_quantidade,
                                     item_total,
                                     data_criacao,
                                     data_atualizacao)
                             values (4,
                                     10002,
                                     15,
                                     3,
                                     45,
                                     '01/01/2016',
                                     '01/01/2016');
```

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.

```
postgres=# select produto_id , sum(iten_total)
             from itens_vendas
            group by produto_id;
```

Como resultado da nossa consulta, temos:

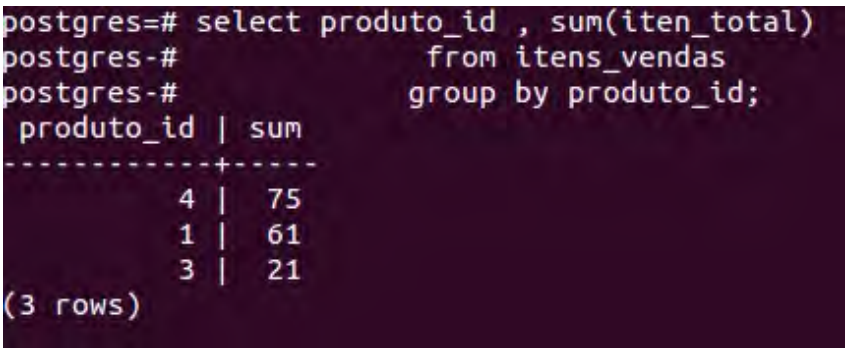

Figura 5.14: Valor vendido de cada produto

Não ficou muito legal a apresentação, pois só temos o `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. Mãos no teclado e vamos colocar em prática um recurso já aprendido.

```
postgres=# create or replace function
            retorna_nome_produto(prod_id int)
            returns text as
            $$
            declare
            nome     text;
            begin
              select produto_nome
                into nome
                from produtos
               where id = prod_id;
                 return nome;
            end
            $$
            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.

```
postgres=# select retorna_nome_produto(produto_id) , sum(iten_tota
l)
             from itens_vendas
            group by produto_id;
```

```
postgres=# select retorna_nome_produto(produto_id) , sum(iten_total)
postgres-#              from itens_vendas
postgres-#             group by produto_id;
 retorna_nome_produto | sum
----------------------+-----
 SUCO DE LIMÃO        |  75
 REFRIGERANTE         |  61
 PASTEL               |  21
(3 rows)
```

Figura 5.15: 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 melhor um pouco mais 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.

```
postgres=# select retorna_nome_produto(produto_id) PRODUTO,
                sum(item_total) VL_TOTAL_PRODUTO
             from itens_vendas
            group by produto_id
            order by vl_total_produto, produto;
```

```
postgres=# select retorna_nome_produto(produto_id) PRODUTO,
postgres-#                 sum(iten_total) VL_TOTAL_PRODUTO
postgres-#              from itens_vendas
postgres-#             group by produto_id
postgres-#             order by vl_total_produto, produto;
    produto    | vl_total_produto
---------------+------------------
 PASTEL        |               21
 REFRIGERANTE  |               61
 SUCO DE LIMÃO |               75
(3 rows)
```

Figura 5.16: Valor vendido de cada produto, uma function para retornar o nome do produto e ordenado e apelidado

Observe como conseguimos evoluir o nosso código. Para você

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:

```
postgres=# select retorna_nome_produto(produto_id) PRODUTO,
                  sum(item_total) VL_TOTAL_PRODUTO
             from itens_vendas
            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. Mãos no teclado e vamos criar a seguinte consulta:

```
postgres=#   select retorna_nome_produto(produto_id),
                    count(id) QTDE
               from itens_vendas
              group by produto_id;
```

Como resultado, teremos:

```
postgres=#  select retorna_nome_produto(produto_id),
postgres-#                    count(id) QTDE
postgres-#                from itens_vendas
postgres-#               group by produto_id;
 retorna_nome_produto | qtde
----------------------+------
 REFRIGERANTE         |    3
 SUCO DE LIMÃO        |    2
 PASTEL               |    1
(3 rows)
```

Figura 5.17: 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 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.

```
postgres=#   select retorna_nome_produto(produto_id) produto,
                    count(id) qtde
               from itens_vendas
              group by produto_id
             having count(produto_id) >= 2
              order by qtde;
```

E como resultado de nossa consulta, teremos:

```
postgres=# select retorna_nome_produto(produto_id) produto,
postgres-#                    count(id) qtde
postgres-#               from itens_vendas
postgres-#              group by produto_id
postgres-#             having count(produto_id) >= 2
postgres-#              order by qtde;
    produto     | qtde
----------------+------
 SUCO DE LIMÃO |    2
 REFRIGERANTE   |    3
(2 rows)
```

Figura 5.18: Having count como forma de filtrar as linhas apresentadas

Observe que, nesta 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 você for tendo 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
to_char(hora, formato)	Converte tipo hora para texto	*select to_char(current_timestamp, 'HH12:MI:SS');*	01:11:11
to_char(data, formato)	Converte tipo data para texto	*select to_char(current_date, 'DD/MM/YYYY');*	23/03/2016
to_char(data, formato)	Converte tipo data/hora para texto	*select to_char(current_timestamp, 'DD/MM/YYYY HH12:MI:SS');*	23/03/2016 01:11:11
to_char(número inteiro, número de caracteres que o primeiro número '999')	Converte número do tipo inteiro para texto	*select to_char(2322,'99999');*	2322

to_char(número double/real, '999D9')	Converte número do tipo double/real para texto	*select to_char(125.8::real, '999D9');*	125,8
to_date(texto, formato)	Converte texto para data	*select to_date('04 Nov 1988', 'DD Mon YYYY');*	1988-11-04
to_number(texto, formato)	Converte texto para dado numérico	*select to_number('52.3', '99G999D9S');*	523
to_timestamp(text, text)	Converte tipo data/hora com fuso horário para texto	*select to_timestamp('04 Nov 1988', 'DD Mon YYYY');*	1988-11-04

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`.

```
postgres=> insert into funcionarios(funcionario_codigo,
                                    funcionario_nome,
                                    funcionario_situacao,
                                    funcionario_comissao,
                                    funcionario_cargo,
                                    data_criacao)
                            values('0100',
                                    'VINICIUS SOUZA',
                                    'A',
                                     2,
                                    'GARÇOM',
                                    '01/03/2016');
```

```
postgres=> insert into funcionarios(funcionario_codigo,
                                    funcionario_nome,
                                    funcionario_situacao,
                                    funcionario_comissao,
                                    funcionario_cargo,
                                    data_criacao)
                            values('0101',
                                    'VINICIUS SOUZA MOLIN',
                                    'A',
                                     2,
                                    'GARÇOM',
                                    '01/03/2016');


postgres=> insert into funcionarios(funcionario_codigo,
                                    funcionario_nome,
                                    funcionario_situacao,
                                    funcionario_comissao,
                                    funcionario_cargo,
                                    data_criacao)
                            values('0102',
                                    'VINICIUS RANKEL C',
                                    'A',
                                     2,
                                    'GARÇOM',
                                    '01/03/2016');


postgres=> insert into funcionarios(funcionario_codigo,
                                    funcionario_nome,
                                    funcionario_situacao,
                                    funcionario_comissao,
                                    funcionario_cargo,
                                    data_criacao)
                            values('0103',
                                    'BATISTA SOUZA LUIZ',
                                    'A',
                                     2,
                                    'GARÇOM',
                                    '01/03/2016');


postgres=> insert into funcionarios(funcionario_codigo,
                                    funcionario_nome,
                                    funcionario_situacao,
                                    funcionario_comissao,
                                    funcionario_cargo,
```

```
                                    data_criacao)
                             values('0104',
                                    'ALBERTO SOUZA CARDOSO',
                                    'A',
                                     2,
                                    'GARÇOM',
                                    '01/03/2016');

postgres=> insert into funcionarios(funcionario_codigo,
                                    funcionario_nome,
                                    funcionario_situacao,
                                    funcionario_comissao,
                                    funcionario_cargo,
                                    data_criacao)
                             values('0105',
                                    'CARLOS GABRIEL ALMEIDA',
                                    'A',
                                     2,
                                    'GARÇOM',
                                    '01/03/2016');


postgres=> insert into funcionarios(funcionario_codigo,
                                    funcionario_nome,
                                    funcionario_situacao,
                                    funcionario_comissao,
                                    funcionario_cargo,
                                    data_criacao)
                             values('0106',
                                    'RENAN SIMOES SOUZA',
                                    'A',
                                     2,
                                    'GARÇOM',
                                    '01/03/2016');
```

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:

```
postgres=# select funcionario_nome from funcionarios;
    funcionario_nome
------------------------
 VINICIUS CARVALHO
 SOUZA
 VINICIUS SOUZA
 VINICIUS SOUZA MOLIN
 VINICIUS RANKEL C
 BATISTA SOUZA LUIZ
 ALBERTO SOUZA CARDOSO
 CARLOS GABRIEL ALMEIDA
 RENAN SIMOES SOUZA
(9 rows)
```

Figura 5.19: 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. Mãos no teclado e vamos ao nosso código.

```
postgresql=# select funcionario_nome
               from funcionarios
              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 dele e que é para ele buscar todos os resultados.

```
postgres=# select funcionario_nome
              from funcionarios
             where funcionario_nome like 'VINICIUS%';
   funcionario_nome
----------------------
 VINICIUS CARVALHO
 VINICIUS SOUZA
 VINICIUS SOUZA MOLIN
 VINICIUS RANKEL C
(4 rows)
```

Figura 5.20: 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.

```
postgresql=# select funcionario_nome
               from funcionarios
              where funcionario_nome like '%SOUZA%';
```

```
postgres=# select funcionario_nome
postgres-#               from funcionarios
postgres-#              where funcionario_nome like '%SOUZA%';
   funcionario_nome
----------------------
 SOUZA
 VINICIUS SOUZA
 VINICIUS SOUZA MOLIN
 BATISTA SOUZA LUIZ
 ALBERTO SOUZA CARDOSO
 RENAN SIMOES SOUZA
(6 rows)
```

Figura 5.21: 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:

```
postgresql=# select funcionario_nome
               from funcionarios
              where funcionario_nome = 'VINICIUS%';
```

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

```
postgresql=# select funcionario_nome
               from funcionarios
              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 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 por exemplo, 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: de 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 nesta nova tabela. Diferente dos tipos de funções que já

criamos, esta 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 de por meio 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 `logs_produtos`. 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. Esta é uma convenção que eu, particularmente, gosto de usar. Você pode criar a sua. :)

Vou criar uma coluna em nossa tabela chamada `alteracao` que vai armazenar o tipo de operação que foi feita na tabela, ou seja, gravará se foi uma inserção, uma alteração ou exclusão de registros. Isso é possível pois, durante a execução da `function` que retorna um tipo trigger, cria 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.

```
postgres=> create table logs_produtos(
             id                    int not null primary key,
             data_alteracao        timestamp,
             alteracao             varchar(10),
```

```
id_old                int,
produto_codigo_old    varchar(20),
produto_nome_old      varchar(60),
produto_valor_old     real,
produto_situacao_old  varchar(1) default 'A',
data_criacao_old      timestamp,
data_atualizacao_old  timestamp,
id_new                int,
produto_codigo_new    varchar(20),
produto_nome_new      varchar(60),
produto_valor_new     real,
produto_situacao_new  varchar(1) default 'A',
data_criacao_new      timestamp,
data_atualizacao_new  timestamp );
```

Vamos criar uma `sequence` para a nossa tabela de logs.

```
postgres=> create sequence logs_produto_id_seq;
```

Agora vamos vincular a `sequence` à coluna `id` da tabela `logs_produtos`.

```
postgres=> alter table logs_produtos
           alter column id set default
           nextval('logs_produto_id_seq');
```

Pronto! A tabela está pronta para receber os logs de alterações da tabela `produtos`. 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`, pois, 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 registro e ocorrerá um erro, dizendo que a variável não possui valor. Mãos no teclado e vamos ao código.

```
create or replace function gera_log_produtos()
returns trigger as

$$

Begin

  if TG_OP = 'INSERT' then

    insert into logs_produtos (
                alteracao
               ,data_alteracao
               ,id_new
               ,produto_codigo_new
               ,produto_nome_new
               ,produto_valor_new
               ,produto_situacao_new
               ,data_criacao_new
               ,data_atualizacao_new
              )

       values (
                 TG_OP
                ,now()
               ,new.id
               ,new.produto_codigo
               ,new.produto_nome
               ,new.produto_valor
               ,new.produto_situacao
               ,new.data_criacao
               ,new.data_atualizacao

                );
          return new;


  elsif TG_OP = 'UPDATE' then

      insert into logs_produtos (
                  alteracao
                 ,data_alteracao
                 ,id_old
                  ,produto_codigo_old
```

```
                ,produto_nome_old
                ,produto_valor_old
                ,produto_situacao_old
                ,data_criacao_old
                ,data_atualizacao_old
                ,id_new
                ,produto_codigo_new
                ,produto_nome_new
                ,produto_valor_new
                ,produto_situacao_new
                ,data_criacao_new
                ,data_atualizacao_new
               )

        values (
                  TG_OP
                 ,now()
                 ,old.id
                ,old.produto_codigo
                ,old.produto_nome
                ,old.produto_valor
                ,old.produto_situacao
                ,old.data_criacao
                ,old.data_atualizacao
                ,new.id
                ,new.produto_codigo
                ,new.produto_nome
                ,new.produto_valor
                ,new.produto_situacao
                ,new.data_criacao
                ,new.data_atualizacao

                 );
          return new;

elsif TG_OP = 'DELETE' then

  insert into logs_produtos (
                 alteracao
                ,data_alteracao
                ,id_old
                 ,produto_codigo_old
                ,produto_nome_old
                ,produto_valor_old
                ,produto_situacao_old
                ,data_criacao_old
                ,data_atualizacao_old
               )
```

```
            values (
                     TG_OP
                    ,now()
                    ,old.id
                   ,old.produto_codigo
                   ,old.produto_nome
                   ,old.produto_valor
                   ,old.produto_situacao
                   ,old.data_criacao
                   ,old.data_atualizacao

                    );
              return new;

    end if;

end;

$$ language 'plpgsql';
```

Com esta 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` .

```
postgres=>  create trigger tri_log_produtos
            after insert or update or delete on produtos
            for each row execute
            procedure gera_log_produtos();
```

Para sabermos que a trigger está criada corretamente na tabela que desejamos, podemos listar os objetos dessa tabela usando o comando `\d nome_tabela` .

```
postgres=> \d produtos;
```



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. Fiquem à vontade para criar quantos registros você quiser. Quanto mais, melhor!

```
postgres=> insert into produtos (produto_codigo,
                                 produto_nome,
                                 produto_valor,
                                 produto_situacao,
                                 data_criacao,
                                 data_atualizacao)
                         values ('1512',
                                 'LAZANHA',
                                 46,
                                 'A',
                                 '01/01/2016',
                                 '01/01/2016');

postgres=> insert into produtos (produto_codigo,
                                 produto_nome,
                                 produto_valor,
                                 produto_situacao,
                                 data_criacao,
                                 data_atualizacao)
                         values ('1613',
                                 'PANQUECA',
                                 38,
                                 'A',
                                 '01/01/2016',
                                 '01/01/2016');

postgres=> insert into produtos (produto_codigo,
                                 produto_nome,
```

```
                produto_valor,
                produto_situacao,
                data_criacao,
                data_atualizacao)
        values ('733',
                'CHURRASCO',
                72,
                'A',
                '01/01/2016',
                '01/01/2016');
```

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.

```
postgres=>
        select alteracao
              ,data_alteracao
              ,id_new
              ,produto_codigo_new
              ,produto_nome_new
              ,produto_valor_new
              ,produto_situacao_new
              ,data_criacao_new
              ,data_atualizacao_new
         from logs_produtos;
```

```
 alteracao |       data_alteracao       | id_new | produto_codigo_new | produto_nome_new | produto_valor_new
| produto_situacao_new |  data_criacao_new   | data_atualizacao_new
-----------+----------------------------+--------+--------------------+------------------+-------------------
+----------------------+---------------------+---------------------
 INSERT    | 2016-07-26 19:39:42.414343 |     12 | 1512               | LAZANHA          |                46
| A                    | 2016-01-01 00:00:00 | 2016-01-01 00:00:00
 INSERT    | 2016-07-26 19:46:09.962652 |     13 | 1613               | PANQUECA         |                38
| A                    | 2016-01-01 00:00:00 | 2016-01-01 00:00:00
 INSERT    | 2016-07-26 19:46:15.324571 |     14 | 733                | CHURRASCO        |                72
| A                    | 2016-01-01 00:00:00 | 2016-01-01 00:00:00
(3 rows)
```

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 `produtos`, e atualizar o preço do `'CHURRASCO'`.

```
postgres=>
      update produtos 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.

```
postgres=> select *
              from logs_produtos;
```

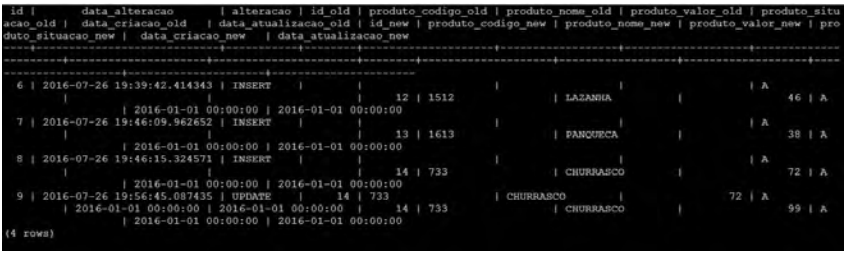

Figura 6.3: Logs de inserção (insert) e alteração (update)

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

```
postgres=> delete from produtos where produto_nome = 'PANQUECA';
```

Executando a mesma consulta:

```
postgres=> select *
              from logs_produtos;
```



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 nesta 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:

```
postgres=> alter table produtos
           disable trigger tri_log_produtos;
```

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

```
postgres=> insert into produtos (produto_codigo,
                                 produto_nome,
                                 produto_valor,
                                 produto_situacao,
                                 data_criacao,
                                 data_atualizacao)
                         values ('912',
                                 'SORVETE',
                                  6,
                                 'A',
```

```
                                '01/01/2016',
                                '01/01/2016');
```

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

```
postgres=> select *
             from logs_produtos;
```

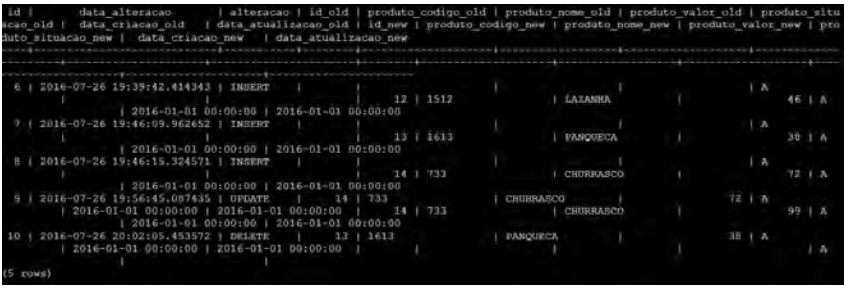

Figura 6.5: O banco não registrou a inserção, pois a trigger está desabilitada

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

Agora vamos habilitar a trigger novamente:

```
postgres=> alter table produtos
           enable trigger tri_log_produtos;
```

Vamos alterar o preço do `'SORVETE'` .

```
postgres=> update produtos set produto_valor = 10
            where produto_nome = 'SORVETE';
```

Consultando novamente:

```
postgres=> select *
             from logs_produtos;
```

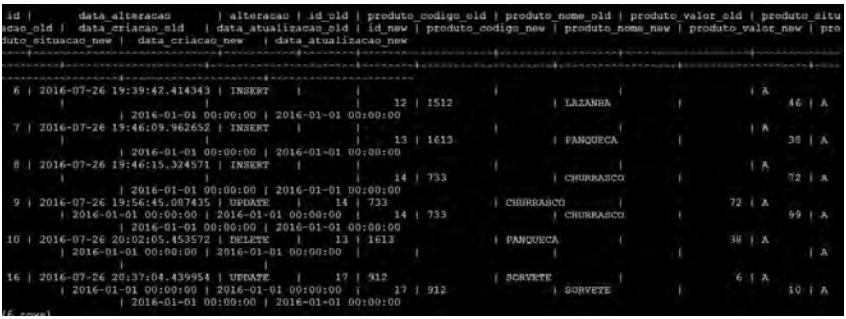

Figura 6.6: Trigger voltando a gravar as alterações

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

```
postgres=> drop trigger tri_log_produtos on produtos;
```

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

```
postgres=> \d produtos
```



Figura 6.7: Ausencia da trigger

Para excluir uma trigger, tenha certeza de que ela não será mais necessária, uma vez que a exclusão pode prejudicar processos, tanto simples, como a gravação de logs, ou até mesmo cálculos que são executados por esses gatilhos. Por exemplo, imagine que você tem uma `trigger` que grava os logs de todas 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 segurança, já que é uma das maneiras de saber quem alterou algo no sistema, e até mesmo uma maneira para restaurar registros deletados indevidamente. Você agora pode criar uma tabela de log para cada tabela que criamos em nosso projeto. Uma maneira para 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 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. Neste caso, 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 isto está fora de cogitação! :)

Uma maneira para relacionar duas tabelas é por meio de uma

subconsulta, na qual buscaremos os funcionários que possuem vínculo com uma ou mais vendas.

```
postgres=# select funcionario_nome
             from funcionarios
            where id in (select funcionario_id
                           from vendas);
```



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.

```
postgres=# select funcionario_nome
             from funcionarios
            where id in (select funcionario_id
                           from vendas
                          where date_part('year', data_criacao) =
'2016');
```

```
 funcionario_nome
-------------------
 SOUZA
 VINICIUS CARVALHO
(2 rows)
```

Figura 7.2: Consulta com subconsulta e parâmetros

Observe que trouxe 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.

```
postgres=# select distinct funcionario_nome
             from funcionarios
            where id in (select funcionario_id
                           from vendas)
            order by funcionario_nome;
```

```
postgres=# select distinct funcionario_nome
postgres-#              from funcionarios
postgres-#             where id in (select funcionario_id
postgres(#                                 from vendas)
postgres-#             order by funcionario_nome;
 funcionario_nome
-------------------
 SOUZA
 VINICIUS CARVALHO
(2 rows)
```

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.

```
postgres=# select distinct funcionario_nome
             from funcionarios, vendas
            where funcionarios.id = vendas.funcionario_id
            order by funcionario_nome;
```

```
postgres=# select distinct funcionario_nome
postgres-#              from funcionarios, vendas
postgres-#             where funcionarios.id = vendas.funcionario_id
postgres-#             order by funcionario_nome;
 funcionario_nome
------------------
 SOUZA
 VINICIUS CARVALHO
(2 rows)
```

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:

```
postgres=# select distinct funcionario_nome
             from funcionarios, vendas
            order by funcionario_nome;
```

```
postgres=# select distinct funcionario_nome
postgres-#              from funcionarios, vendas
postgres-#             order by funcionario_nome;
    funcionario_nome
-----------------------
 ALBERTO SOUZA CARDOSO
 BATISTA SOUZA LUIZ
 CARLOS GABRIEL ALMEIDA
 dadasf
 RENAN SIMOES SOUZA
 SOUZA
 TESTE FUNCIO
 VINICIUS CARVALHO
 VINICIUS FUNCIONARIO
 VINICIUS RANKEL C
 VINICIUS SOUZA
 VINICIUS SOUZA MOLIN
(12 rows)
```

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.

Esta é a forma mais popular para se escrever uma `join`, e o jeito mais simples para ler as consultas. Eu, particularmente, prefiro escrevê-las desta 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, este 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`.

```
postgres=# select distinct funcionario_nome
             from funcionarios
            inner join vendas
               on (funcionarios.id = vendas.funcionario_id)
            order by funcionario_nome;
```

```
postgres=# select distinct funcionario_nome
postgres-#              from funcionarios
postgres-#             inner join vendas
postgres-#                on (funcionarios.id = vendas.funcionario_id)
postgres-#             order by funcionario_nome;
 funcionario_nome
------------------
 SOUZA
 VINICIUS CARVALHO
(2 rows)
```

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`, que teremos o mesmo resultado.

```
postgres=# select distinct funcionario_nome
             from funcionarios
             join vendas
               on (funcionarios.id = vendas.funcionario_id)
            order by funcionario_nome;
```

```
postgres=#  select distinct funcionario_nome
postgres-#              from funcionarios
postgres-#              join vendas
postgres-#                on (funcionarios.id = vendas.funcionario_id)
postgres-#             order by funcionario_nome;
 funcionario_nome
------------------
 SOUZA
 VINICIUS CARVALHO
(2 rows)
```

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 (quanto 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, que vai ficar mais claro para visualizarmos este cenário.

```
postgres=# select funcionario_nome, v.id
             from funcionarios f
             left join vendas v
               on f.id = v.funcionario_id
            order by funcionario_nome desc;
```

```
postgres=# select funcionario_nome, v.id
postgres-#              from funcionarios f
postgres-#              left join vendas v
postgres-#                on f.id = v.funcionario_id
postgres-#             order by funcionario_nome desc;
    funcionario_nome     |  id
-------------------------+-------
 VINICIUS SOUZA MOLIN    |
 VINICIUS SOUZA          |
 VINICIUS RANKEL C       |
 VINICIUS FUNCIONARIO    |
 VINICIUS FUNCIONARIO    |
 VINICIUS CARVALHO       | 10002
 VINICIUS CARVALHO       | 10000
 VINICIUS CARVALHO       | 10001
 TESTE FUNCIO            |
 SOUZA                   |     1
 SOUZA                   |     2
 RENAN SIMOES SOUZA      |
 dadasf                  |
 CARLOS GABRIEL ALMEIDA  |
 BATISTA SOUZA LUIZ      |
 ALBERTO SOUZA CARDOSO   |
(16 rows)
```

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.id`, que é o identificador da venda. E para os funcionários que não possuem nenhum registro na tabela `vendas`, 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.

```
postgres=# select v.id, v.venda_total, funcionario_nome
             from vendas v
```

```
right join funcionarios f
   on v.funcionario_id = f.id
order by v.venda_total;
```

```
postgres=# select v.id, v.venda_total, funcionario_nome
postgres-#              from vendas v
postgres-#             right join funcionarios f
postgres-#                on v.funcionario_id = f.id
postgres-#             order by v.venda_total;
  id   | venda_total |    funcionario_nome
-------+-------------+------------------------
     1 |          20 | SOUZA
 10001 |          20 | VINICIUS CARVALHO
     2 |          21 | SOUZA
 10002 |          45 | VINICIUS CARVALHO
 10000 |          51 | VINICIUS CARVALHO
       |             | ALBERTO SOUZA CARDOSO
       |             | CARLOS GABRIEL ALMEIDA
       |             | RENAN SIMOES SOUZA
       |             | TESTE FUNCIO
       |             | VINICIUS FUNCIONARIO
       |             | VINICIUS FUNCIONARIO
       |             | dadasf
       |             | VINICIUS SOUZA
       |             | VINICIUS SOUZA MOLIN
       |             | VINICIUS RANKEL C
       |             | BATISTA SOUZA LUIZ
(16 rows)
```

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? Pois pense como o 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.

```
postgres# create or replace view vendas_do_dia as
                select distinct produto_nome
                     , sum(vendas.venda_total)
                  from produtos, itens_vendas, vendas
                 where produtos.id = itens_vendas.produto_id
                   and vendas.id   = itens_vendas.vendas_id
                   and vendas.data_criacao = '01/01/2016'
                 group by produto_nome;
```

```
postgres=# create or replace view vendas_do_dia as
postgres-#                 select distinct produto_nome
postgres-#                      , sum(vendas.venda_total)
postgres-#                   from produtos, itens_vendas, vendas
postgres-#                  where produtos.id = itens_vendas.produto_id
postgres-#                    and vendas.id   = itens_vendas.vendas_id
postgres-#                    and vendas.data_criacao = '01/01/2016'
postgres-#                  group by produto_nome;
CREATE VIEW
```

Figura 7.10: Criando views para agilizar o dia a dia

Agora que temos esta visão criada em nosso banco de dados,

podemos consultá-la quando desejarmos, da seguinte maneira:

```
postgres# select * from vendas_do_dia;
```

```
postgres=# select * from vendas_do_dia;
  produto_nome   | sum
-----------------+-----
 SUCO DE LIMÃO |  96
 PASTEL          |  51
 COXINHA         | 102
(3 rows)
```

Figura 7.11: Resultado da consulta da view

Podemos ainda fazer consultas em nossa view com outras cláusulas, como por exemplo, buscar se um determinado produto foi vendido no dia inserido na visão.

```
postgres=# select *
             from vendas_do_dia
            where produto_nome = 'PASTEL';
```

```
postgres=# select *
postgres-#              from vendas_do_dia
postgres-#             where produto_nome = 'PASTEL';
 produto_nome | sum
--------------+-----
 PASTEL       |  51
(1 row)
```

Figura 7.12: Adicionando cláusula em consulta com views

Podemos também criar uma visão que traria diversos campos e você poderia passar outros parâmetros.

```
postgresq# create or replace view produtos_vendas as
               select produtos.id PRODUTO_ID
                    , produtos.produto_nome PRODUTO_NOME
                    , vendas.id VENDA_ID
                    , itens_vendas.id ITEM_ID
                    , itens_vendas.item_valor ITEM_VALOR
                    , vendas.data_criacao DATA_CRIACAO
                 from produtos, vendas, itens_vendas
```

```
 where vendas.id   = itens_vendas.vendas_id
   and produtos.id = itens_vendas.produto_id
 order by data_criacao desc;
```

Agora podemos consultar qualquer uma das colunas que estão contidas na `view` e também realizar comparações com elas.

```
postgres=# select *
             from produtos_vendas;
```

```
postgres=# select *
             from produtos_vendas;
 produto_id |  produto_nome  | venda_id | item_id | item_valor |    data_criacao
------------+----------------+----------+---------+------------+---------------------
          4 | SUCO DE LIMÃO |    10002 |       6 |         15 | 01/01/2016 00:00:00
          4 | SUCO DE LIMÃO |    10000 |       3 |         15 | 01/01/2016 00:00:00
          1 | COXINHA       |        2 |       8 |          7 | 01/01/2016 00:00:00
          1 | COXINHA       |        1 |       7 |         10 | 01/01/2016 00:00:00
          1 | COXINHA       |    10001 |       5 |         10 | 01/01/2016 00:00:00
          1 | COXINHA       |        2 |       2 |          7 | 01/01/2016 00:00:00
          1 | COXINHA       |        1 |       1 |         10 | 01/01/2016 00:00:00
          3 | PASTEL        |    10000 |       4 |          7 | 01/01/2016 00:00:00
(8 rows)
```

Figura 7.13: Uma view com diversos campos

```
postgres=# select produto_nome
             from produtos_vendas
            where data_criacao = '01/01/2016';
```

```
postgres=# select produto_nome
postgres-#              from produtos_vendas
postgres-#             where data_criacao = '01/01/2016';
  produto_nome
----------------
 SUCO DE LIMÃO
 SUCO DE LIMÃO
 COXINHA
 COXINHA
 COXINHA
 COXINHA
 COXINHA
 PASTEL
(8 rows)
```

Figura 7.14: 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.

```
postgres=# create or replace view produtos_estoque as
               select *
                 from produtos;
```

Agora podemos consultar qualquer campo da tabela, só que agora pela `view`.

```
postgres=# select produto_nome
             from produtos_estoque;
```

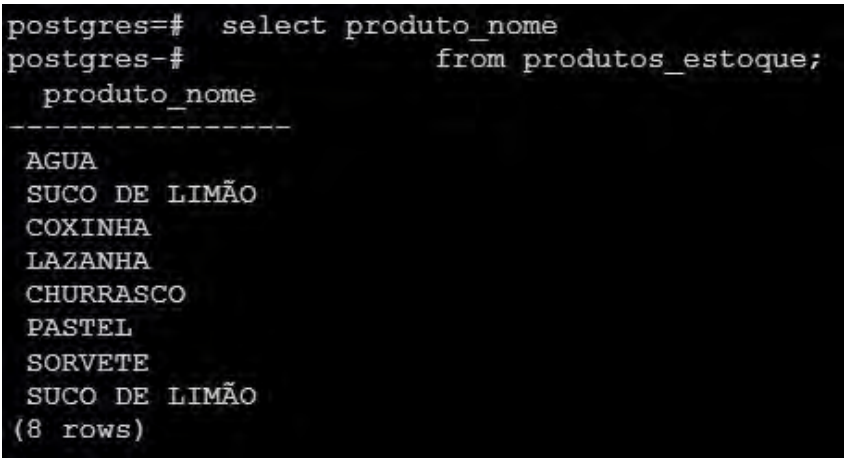

Figura 7.15: View de uma tabela inteira

Mas 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 fornecemos acesso a toda 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 um bom programador. 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 todo programador deve conhecer um pouco de infraestrutura. Além de administração, aprenderemos um pouco sobre performance de nosso banco e de consultas.

Capítulo 8

ADMINISTRAÇÃO DO BANCO 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 DE BANCO DE DADOS VS. DESENVOLVEDOR

A pessoa responsável pela administração do banco de dados é conhecida como DBA (*Data Base Administrator*, ou no bom e velho português, Administrador de Banco de Dados). Em muitas empresas, vocês podem se deparar com o próprio desenvolvedor fazendo o papel de administrador do banco de dados. Isto é muito comum, principalmente em pequenas empresas que não têm um capital disponível para a contratação de um profissional capacitado para esta função.

O DBA é responsável por administrar aspectos de infraestrutura do banco, como performance, arquitetura, criação, importação e backup. Mas como nem sempre as empresas possuem esse papel, essas e muitas outras funções ficam a cargo do desenvolvedor.

Muitas pessoas divergem sobre até onde o desenvolvedor deve

influenciar no banco de dados. Alguns dizem que ele 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 administrador do banco. Atualmente, como existem vários *frameworks* que criam automaticamente as tabelas e os código do banco de dados, acaba que, para o administrator, fica a tarefa de administrar aspectos da performance, segurança e a estrutura do banco.

Se você não é e não pretende ser um desenvolvedor, 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 o ocorre no dia a dia de um administrador do banco de dados cobra, além de saber comandos para a configuração e otimização do servidor. E se você é um desenvolvedor e não tem disponível um profissional para executar essas tarefas, é importante que você tenha algum conhecimento.

Agora se você é desenvolvedor e tem a disponibilidade de um administrador 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 auxiliá-lo na utilização e administração do cotidiano, temos alguns comandos que são de extrema importância. Se você dominá-los, não terá necessidade da utilização de uma ferramenta. Dominando 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
`sudo -i -u postgres psql -l`	Para listar os bancos de dados
`sudo -i -u postgres psql -U nomeusuario nomebanco`	Para conectar ao console psql no banco de dados
`sudo -i -u postgres psql banco -E`	Para mostrar internamente como cada consulta é realizada
`sudo -i -u postgres psql –version`	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 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
`\?`	Para lhe ajudar 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 e criar também um novo usuário. Então, mãos no teclado.

```
postgres=> create user nomedousuario superuser;
```

Alteramos a senha do novo usuário:

```
postgres=> 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 `novobanco` :

```
postgres=> 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` .

```
postgres=# \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.

```
newbase=# \l;
```

Se você não criou nenhum outro banco, você obterá o resultado a seguir:

```
postgres=# \l
                                  List of databases
   Name    |     Owner     | Encoding | Collate | Ctype |   Access privileges
-----------+---------------+----------+---------+-------+-----------------------
 newbase   | nomedousuario | LATIN1   | en_US   | en_US |
 postgres  | postgres      | LATIN1   | en_US   | en_US |
 template0 | postgres      | LATIN1   | en_US   | en_US | =c/postgres          +
           |               |          |         |       | postgres=CTc/postgres
 template1 | postgres      | LATIN1   | en_US   | en_US | =c/postgres          +
           |               |          |         |       | postgres=CTc/postgres
(4 rows)
```

Figura 8.1: Bancos de dados criados

Observe que a lista dos bancos criados e dos usuários que os criaram. Até agora estávamos usando o banco de dados `postgres` e o usuário `postgres` . E 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 deste do início de nosso projeto, vamos exportar o banco de dados `postgres` . Saia do terminal do Postgre e digite o comando a seguir, que exportará todos os objetos e registros do banco `postgres` .

```
$> pg_dump --host localhost --port 5432 --username postgres --form
at tar --file nomearquivo.backup postgres
```

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

A importação é ainda mais simples. Vamos digitar o comando passando o nome do banco cuja importação queremos fazer e o usuário. O comando ficará da seguinte maneira:

```
$> pg_restore --host localhost --port 5432 --username nomedousuari
o --dbname newbase nomearquivo.backup
```

Para verificarmos se foram importados todos os objetos, vamos utilizar o comando `d` .

```
newbase=# \d;
```

```
                List of relations
 Schema |         Name          |   Type   |  Owner
--------+-----------------------+----------+----------
 public | comissoes             | table    | postgres
 public | comissoes_id_seq      | sequence | postgres
 public | funcionario_id_seq    | sequence | postgres
 public | funcionarios          | table    | postgres
 public | itens_vendas          | table    | postgres
 public | itens_vendas_id_seq   | sequence | postgres
 public | logs_produto_id_seq   | sequence | postgres
 public | logs_produtos         | table    | postgres
 public | mesa_id_seq           | sequence | postgres
 public | mesas                 | table    | postgres
 public | produtos              | table    | postgres
 public | produtos_estoque      | view     | postgres
 public | produtos_id_seq       | sequence | postgres
 public | produtos_vendas       | view     | postgres
 public | vendas                | table    | postgres
 public | vendas_do_dia         | view     | postgres
 public | vendas_id_seq         | sequence | postgres
(17 rows)
```

Figura 8.2: Objetos criados com a importação

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	GARÇOM	01/01/2016	01/01/2016
3	154	NOVO FUNCIONARIO	A	75	GARÇOM	01/01/2016	01/01/2016

Figura 8.3: 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.

```
postgres=#

COPY funcionarios
(
   funcionario_codigo,
   funcionario_nome,
   funcionario_situacao,
   funcionario_comissao,
   funcionario_cargo,
```

```
   data_criacao,
   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: 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:

```
postgres=# select * from funcionarios where funcionario_cargo = 'G
ARÇOM';
```

Como não temos um índice nesta 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 mais provavelmente vão 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. Por se usado em coluna, 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 `funcionarios`.

```
postgres=# create index idx_cargo on funcionarios(funcionario_carg
o);
```

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:

```
posgtres=# drop index idx_cargo;
```

Tipos de índices

Temos alguns tipos de índices no PostgreSQL, sendo que 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 deste 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` :

```
postgres=# create index idx_cargo on
            funcionarios(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 estas razões, desencoraja-se a utilização dos índices hash.

Para criarmos o índice do tipo hash, utiliza-se o comando:

```
postgres=# create index idx_codigo on
            funcionarios 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 este tipo de índice, usamos o comando:

```
postgres=# create index concurrentyle idx_nome on
              funcionarios 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 este tipo de índice, usamos o comando:

```
postgres=# create index idx_funcionario_id_codigo on
              funcionarios(id, funcionario_codigo);
```

Este tipo de índice seria útil para consultas do tipo:

```
postgres=# select *
             from funcionarios
            where id > 10
              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:

```
postgres=# create unique index idx_unique_codigo on
                funcionarios(funcionario_codigo);
```

Para sabermos se este índice está funcionando corretamente, vamos tentar inserir um novo registro na tabela `funcionarios` e, no `insert` do registro, utilizar um código de funcionário que já exista na tabela. Primeiro, vamos fazer uma consulta para poder pegar um `funcionario_codigo` de um funcionário existente no banco.

```
postgres=# select funcionario_codigo
             from funcionarios;
```

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.

```
postgres=# insert into funcionarios(funcionario_codigo, funcionari
o_nome)
            values('0001', 'DANIEL VINICIUS SOUZA');
```

Ao clicarmos `enter` , será exibido o erro:

```
ERROR:  duplicate key value violates unique constraint "idx_unique_codigo"
DETAIL:  Key (funcionario_codigo)=(0001) already exists.
```

Figura 8.4: Erro de unique key

Utilize em coluna que ainda não possua itens duplicados, ou antes de duplicar itens, pois se a coluna já possuir itens em duplicidade, este 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. Este 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 `analyse` , 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:

```
postgres=# analyze verbose;
```

Será exido o resultado da análise das estatísticas de todas as tabelas. Agora, pediremos para que seja analisada a tabela de funcionários.

```
postgres=# analyze verbose funcionarios;
```

Neste caso, apenas será atualizado as estatísticas da tabela de funcionários. Podemos ir além e pedir para executar a análise apenas da coluna `funcionario_cargo` .

```
postgres=# analyze verbose funcionario(funcionario_cargo);
```

Dentre 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 de que as estatísticas podem mudar a cada execução do `analyze` , assim alterando 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 for criado. Essa ação fará com que o índice volte a performar melhor em seu SGBD. Para fazer essa reindexação, vamos usar o comando:

```
postgres=# reindex table funcionarios;
```

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 a perfeição? *#FicaADica.*

Outro ponto que vimos neste capítulo foi a importação via CSV. É 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 utiliza um banco de dados, você já poderá ajudá-lo.

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 tipo 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 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.

Os tipos de campos `array` e `JSON`, você verá que terá uma boa aplicação em nosso projeto e poderá lhe ser útil 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. Mãos no teclado para criamos o código para criação do nosso campo.

```
postgresql=# alter table produtos
               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.

```
postgresql=#

insert into produtos (produto_codigo,
                      produto_nome,
                      produto_valor,
                      produto_situacao,
                      data_criacao,
                      data_atualizacao,
                      produto_categoria)
              values ('03251',
                      'ESFIRRA',
                       5,
                      'A',
                      '01/01/2016',
                      '01/01/2016',
                      '{"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 deste novo registro em nossa tabela, podemos fazer uma consulta para verificar como ficam os dados no banco de dados. Mãos no teclado e vamos fazer uma consulta usando somente o campo que criamos agora.

```
postgresql=# select produto_categoria
               from produtos
              where produto_nome like 'ESFIRRA';
```

Como resultado, teremos:

```
postgres=# select produto_categoria from produtos where produto_nome like 'ESFIRRA';
       produto_categoria
-------------------------------
 {CARNE,SALGADO,ASSADO,QUEIJO}
(1 row)
```

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:

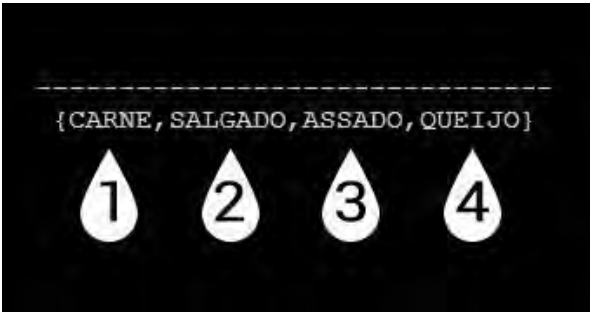

Figura 9.3: Elementos do nosso array

Observando esta 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 o somente segundo item da lista?

Baseando-se nesta figura dos itens, podemos deduzir que cada item encontra-se 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 dois, indicando para o banco de dados que desejamos que ele nos retorne o item que está na posição 2 da lista. Mãos no teclado e vamos para o nosso código.

```
postgresql=# select produto_categoria[2]
               from produtos
              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:

```
postgres=# select produto_categoria[2] from produtos where produto_nome like 'ESFIRRA';
 produto_categoria
-------------------
 SALGADO
(1 row)
```

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.

```
postgresql=# select produto_categoria[2:4]
               from produtos
              where produto_nome like 'ESFIRRA';
```

Observe em nosso código que agora usamos dois pontos para separar a posição inicial da final. Como resultado, teremos:

```
postgres=# select produto_categoria[2:4] from produtos where produto_nome like 'ESFIRRA';
    produto_categoria
-------------------------
 {SALGADO,ASSADO,QUEIJO}
(1 row)
```

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 neste 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 se relacionar. Se você quiser se aprofundar em banco de dados NoSQL, aconselho conhecer os livros sobre o assunto da Casa do Código sobre NoSQL "NoSQL Como armazenar os dados de uma aplicação moderna" e sobre MongoDB "MongoDB Construa novas aplicações com novas tecnologias".

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 este 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 este tipo de campo. Basta apenas informar `produto_estoque json`. Então, vamos ao código.

```
postgresql=# alter table produtos
               add column produto_estoque json;
```

Cenário simples

Agora vamos conhecer a estrutura que devemos inserir os registros nesta 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 objetos filhos `tem_estoque` para indicar se o produto tem no estoque, o objeto filho `quantidade` para indicar quantos produtos possui no estoque e o objeto filho `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.

```
postgresql=#
 insert into produtos(produto_codigo,
                      produto_nome,
                      produto_valor,
                      produto_situacao,
                      data_criacao,
                      data_atualizacao,
                      produto_categoria,
                      produto_estoque)
             values('6234',
                    'COCA-COLA',
                    6,
                    'A',
                    '01/01/2016',
                    '01/01/2016',
                    '{"REFRIGERANTE",
                      "LATA",
                      "BEBIDA" ,
                      "COLA"}',
                    '{ "info_estoque":
```

```
                    { "tem_estoque": "SIM",
                      "quantidade": 17,
                      "ultima_compra": "01/01/16" }
                }'
              );
```

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.

```
postgresql=# select produto_estoque
               from produtos
              where produto_nome like 'COCA-COLA';
```

E como resultado, teremos:

```
postgres=# select produto_estoque from produtos where produto_nome like 'COCA-COLA';
 { "info_estoque":                                  +
                    { "tem_estoque": "SIM",         +
                      "quantidade": 17,             +
                      "ultima_compra": "01/01/16" } +
                  }
```

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:

```
postgresql=#
  select produto_estoque->'info_estoque'->>'quantidade'
         as quantidade
    from produtos
   where produto_nome like 'COCA-COLA';
```

E como resultado, teremos o valor inserido no objeto filho `quantidade`.

Operadores -> e ->>

Observe que, nesta última consulta, foi usado o operador `->>` e o banco de dados retornou o valor em formato de texto do objeto. E 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, em seguida, com o operador `->>` .

```
postgresql=#
  select produto_estoque->'info_estoque'->'ultima_compra'
         as ultima_compra
    from produtos
   where produto_nome like 'COCA-COLA';
```

Como retorno, teremos `"01/01/16"` . 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 operador `->>` .

```
postgresql=#
  select produto_estoque->'info_estoque'->>'ultima_compra'
         as ultima_compra
    from produtos
   where produto_nome like 'COCA-COLA';
```

Como retorno, teremos `01/01/16` . Agora tivemos como retorno apenas o texto do objeto, sem as aspas. Conhecendo estes operadores, podemos passar para um exemplo um pouco mais complexo.

Cenário complexo

Nesta 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 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.

```
postgresql=#
 insert into produtos(produto_codigo,
                      produto_nome,
                      produto_valor,
                      produto_situacao,
                      data_criacao,
                      data_atualizacao,
                      produto_categoria,
                      produto_estoque)
             values('77978',
                    'GATORADE',
                    6,
                    'A',
                    '01/01/2016',
                    '01/01/2016',
                    '{"ISOTONICO",
                      "GARRAFA",
                      "BEBIDA" }',
                    '{ "info_estoque":
                       { "tem_estoque": "SIM",
                         "quantidade": 17,
                         "ultima_compra": "01/01/16" },
                        "ultima_venda": "02/01/2016"
                     }'
                    );
```

Em seguida, após inserir este 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 `02/01/2016` .

```
postgresql=#
   select produto_estoque
     from produtos
    where produto_estoque->>'ultima_venda' = '02/01/2016';
```

Como resultado, teremos o conteúdo do campo `produto_estoque` .

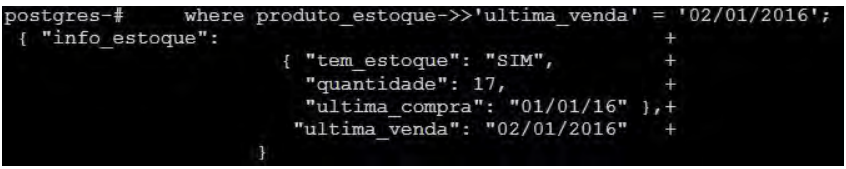

Figura 9.7: Consulta com JSON com mais de um objeto pai

Observe como é muito simples trabalhar com JSON no PostgreSQL, mesmo tendo um ou mais objetos pais. Este 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 campo deste tipo em seu banco de dados, ficará mais fácil você receber e armazenar esses 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 isto 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 que você 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.

Para o próximo capítulo, teremos algumas questões para você treinar conceitos aprendidos neste livro. São questões sobre banco de dados extraídas de concursos públicos. Boa sorte! *Keep Programming!* :)

Capítulo 10

EXERCÍCIOS DE CONCURSO

"A sua maior frustração pode se transformar em sua maior ideia" — Troy Osinoff

10.1 CONCURSOS PELO BRASIL

Cada dia mais vem crescendo o número de concursos na área de desenvolvimento de software e Tecnologia de Informação no Brasil. E a tendência desse número é aumentar.

95% dos concursos possuem perguntas sobre banco de dados, e muitas vezes sobre um SGBD específico. Assim, fiz um compilado de exercícios de várias provas de concursos aplicadas ao redor do Brasil, sendo que as respostas para todas as perguntas estão no meio dos capítulos que compõem este livro.

Algumas perguntas que extraí de provas de concursos adaptei para o cenário do nosso projeto para que fique claro o conceito e a pergunta que está sendo feita. *Bora estudar?*

10.2 EXERCÍCIOS

[0000 0001] O comando a seguir permite adicionar à tabela `itens_vendas` uma chave estrangeira com o

nome `fk_produtos` do campo `id` que pertence à tabela `produtos`.

```
alter table itens_vendas
  alter column fk_produtos
     references produtos(id);
```

- Certo
- Errado

[0000 0010] A instrução SQL em PostgreSQL a seguir está mal formulada. Isto aconteceu porque:

```
create view vista as select 'Hello World';
```

a) A criação de uma visualização requer a utilização da cláusula `WHERE` para a restrição dos dados.

b) Não é possível criar uma `view` sem a identificação do tipo de dado e sem a determinação da quantidade de registros selecionados.

c) O comando `create view` deve utilizar a cláusula `FROM` para o nome da tabela.

d) A criação de uma visualização (`view`) requer a definição de um gatilho (`trigger`) correspondente ao nome da coluna.

e) Por padrão, o tipo de dado será considerado indefinido (`unknown`) e a coluna vai utilizar o nome padrão `?column?` .

[0000 0011] Considere o trecho do comando em SQL a seguir.

```
create table produtos (
    id integer not null,
    nome_produto varchar(40) not null,
    primary key (id)
    );
```

- Certo
- Errado

[0000 0100] No PostgreSQL, diversos gatilhos podem ser associados a uma mesma condição. Entretanto, se o primeiro gatilho retornar `null`, os demais não serão executados.

- Certo
- Errado

[0000 0101] Considere o trecho em PostgreSQL a seguir.

```
postgresql=> insert into products (product_no, name, price)
             values (1,'Cheese', 9.99)
                    (2,'Bread',1.99)
                    (3, 'Milk', 2.99)
```

Considerando a existência prévia da tabela `products`, contendo as colunas `product_no`, `name` e `price`, e desconsiderando os tipos de dados, esse trecho resultará:

a) Na adição de 3 novas colunas na tabela `products`.

b) Na adição de 3 novas linhas na tabela `products`.

c) Em erro, pois não é possível múltiplas inserções em um único comando SQL.

d) Em erro, pois, para se realizar múltiplas inserções, é necessária a utilização da cláusula `select`.

e) Em erro, pois múltiplas inserções são possíveis somente com a utilização de colchetes para a limitação dos registros.

[0000 0110] O comando em SQL capaz de serializar dados de uma tabela para um arquivo em disco, ou

efetuar a operação contrária, transferindo dados de um arquivo em disco para uma tabela de um banco de dados é o:

a) `COPY`

b) `TRANSFER`

c) `SERIALIZE`

d) `FILE TRANSFER`

e) `EXPORT`

[0000 0111] Suponha que exista determinada tabela alunos, com os campos `id_aluno` , `nome_aluno` , `telefone` e `idade` . Nesse caso, o comando a seguir é apropriado para listar todos os alunos que tenham idade superior a 34 anos e obter o resultado de forma ordenada por aluno.

```
select * from alunos where idade > 34 group by
nome_aluno having count(*) > 34.
```

- Certo
- Errado

[0000 1000] Suponha que tenha sido identificado que uma tabela, cujo nome é `funcionarios` , não apresentava nenhum índice criado que estivesse associado ao campo `funcionario_cargo` . Nessa situação, o comando seguinte permite a criação desse índice com o nome `idx_cargo` .

```
CREATE INDEX IN funcionarios ON idx_cargo
(funcionario_cargo).
```

- Certo

- Errado

[0000 1001] Em SQL, uma visão é uma relação que não está no modelo lógico do banco de dados, mas que é visível ao usuário como uma relação virtual. Marque a alternativa que possui o comando utilizado para a criação desta visão.

a) `CREATE VIEW [NOME DA VISAO] AS [EXPRESSAO DA CONSULTA]`

b) `CREATE VIEW [NOME DA VISAO] FROM [EXPRESSAO DA CONSULTA]`

c) `SELECT VIEW [NOME DA VISAO] AS [EXPRESSAO DA CONSULTA]`

d) `SELECT VIEW [NOME DA VISAO] FROM [EXPRESSAO DA CONSULTA]`

e) `UPDATE VIEW [NOME DA VISAO] FROM [EXPRESSAO DA CONSULTA]`

[0000 1010] Em bancos de dados PostgreSQL, o comando `declare` é usado para:

a) Criar uma classe de operadores que define como um determinado tipo de dado pode ser usado em um índice.

b) Criar cursores que podem ser utilizados para retornar, de cada vez, um pequeno número de linhas em uma consulta.

c) Criar uma tabela, inicialmente vazia, no banco de dados corrente.

d) Registrar um novo tipo de dado para uso no banco de dados corrente.

e) Registrar uma nova linguagem procedural a ser utilizada em consultas ao banco de dados.

[0000 1011] Qual o tipo de dados que é retornado quando a função `extract` é executada em um campo de data e hora?

a) `string`

b) `int[]`

c) `int`

d) `double`

e) `varchar`

[0000 1100] O comando `extract` na linguagem SQL é usado para extrair dados de uma tabela.

- Certo
- Errado

[0000 1101] Na linguagem de consulta estruturada (SQL), é correto utilizar o comando `truncate table`, com a finalidade de excluir todos os dados de uma tabela.

- Certo
- Errado

[0000 1110] Marque a alternativa que possui o comando SQL usado para que sejam selecionadas as informações (nome do correntista e o número de conta corrente) dos correntistas do Banco do Brasil.

Tabela: **Bancos**

Código	Nome
001	Banco do Brasil
033	Santander
237	Bradesco
341	Itaú

Tabela: **Pessoas**

CPF	Nome
86277635697	Jóse da Silva
88208811874	Manoel da Silva
66516764743	Maria dos Santos

Tabela: **Conta_Corrente**

Banco	Pessoa	Número
033	86277635697	98876788
237	86277635697	96645727
341	66516764743	9102947
001	88208811874	8120938

a) `SELECT Nome, Numero FROM Pessoas, Conta_Corrente WHERE Pessoa = CPF AND Banco IN (SELECT Codigo AS Banco FROM Bancos WHERE Nome='Banco do Brasil')`

b) `SELECT Nome, Numero FROM Pessoas, Conta_Corrente, Bancos WHERE Pessoa=CPF AND Banco IN (SELECT Codigo AS Banco FROM Bancos WHERE Nome='Banco do Brasil')`

c) `SELECT Nome, Numero FROM Pessoas, Conta_Corrente WHERE Pessoa=CPF AND Banco='Banco do Brasil'`

d) `SELECT Nome, Numero FROM Pessoas, Conta_Corrente, Bancos WHERE Pessoa=CPF AND Nome='Banco do Brasil'`

e) `SELECT Nome, Numero FROM Pessoas, Conta_Corrente WHERE Nome='Banco do Brasil'`

[0000 1111] Em PostgreSQL, a função que converte a primeira letra da string informada em letra maiúscula, alterando todas as letras subsequentes dessa string para minúsculas, chama-se:

a) `chgstr`

b) `altertext`

c) `initcap`

d) `upper`

e) `toupper`

[0001 0000] Em PostgreSQL, qual o comando correto para a criação de um banco de dados com o nome `escola`.

a) `create base escola;`

b) `create database escola;`

c) `create new database escola;`

d) `create escola as database;`

e) `create escola;`

[0001 0001] Quando queremos iniciar uma nova transação no banco de dados PostgreSQL, qual o comando que podemos usar?

a) `alter`

b) `rollback`

c) `transfer`

d) `begin`

e) `create`

[0001 0010] Qual o nome do comando usado para recuperar os dados deletados de uma transação?

a) `ROLLBACK`

b) `END`

c) `TRANSFER`

d) `EFFECTIVE`

e) `SELECT`

[0001 0011] Dois países diferentes podem possuir o formato de datas diferente um do outro. O formato de datas do PostgreSQL pode ser alterado através da alteração de um parâmetro do banco de dados. Qual é esse parâmetro?

a) `dateinit`

b) `styledate`

c) `datastyle`

d) `confinit`

e) `datestyle`

[0001 0100] Julgue o item a seguir.

O PostgreSQL, diferente de outros bancos de dados, como o MySQL, não permite a criação de `triggers` , sendo esta a principal diferença entre os dois gerenciadores de banco de dados.

- Certo
- Errado

[0001 0101] Considerando que um SGBD é um pacote de software para a implementação e manutenção de bancos de dados computacionais, julgue o item a seguir.

PostgreSQL e MySQL são exemplos de SGBD que executam em ambiente Linux e Windows.

- Certo
- Errado

[0001 0110] Armazenar dados em planilhas é algo comum e praticado por pequenas empresas que não possuem um sistema informatizado. As planilhas são, em sua grande maioria, o primeiro banco de dados das empresas. Se houvesse uma maneira de importar essas planilhas diretamente para o PostgreSQL, iria facilitar muito a vida dos programadores e das empresas. No entanto, este é um suporte que o PostgreSQL não oferece.

- Certo
- Errado

[0001 0111] É possível criar diversas tabelas em banco de dados e, ao criar tabelas, o objetivo passa a ser

consultar os registros das tabelas criadas. Com base nos conhecimentos sobre consulta, julgue o comando a seguir para fazer a consulta do campo `nome` na tabela `funcionarios`.

```
select nome
    to funcionarios
 where id = 123;
```

- Certo
- Errado

[0001 1000] Considerando que um SGBD é um pacote de software para a implementação e manutenção de bancos de dados computacionais, julgue o item a seguir.

Considerando-se bases de dados muito grandes, o MySQL é mais rápido que o PostgreSQL; entretanto, o PostgreSQL oferece uma série de recursos extras que o tornam especializado em operações complexas.

- Certo
- Errado

[0001 1001] No PostgreSQL, para a atribuição de privilégios para criar uma tabela com restrição de chave estrangeira, é necessário possuir, também na tabela com a chave referenciada, o privilégio:

a) `REFERENCES`

b) `RULE`

c) `TRIGGER`

d) `GRANT OPTION`

e) `PUBLIC`

[0001 1010] A consulta a seguir está certa ou errada?

```
SELECT *
  FROM ESTADOS
 WHERE PAIS_ID IN (SELECT *
                     FROM PAISES
                    WHERE NOME_PAIS LIKE 'BRASIL');
```

- Certo
- Errado

[0001 1011] Julgue o item a seguir, em relação às características do PostgreSQL.

No PostgreSQL, o arquivo `pg_hba.conf` é o responsável pelo controle da autenticação de usuário.

- Certo
- Errado

[0001 1100] 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 de por meio 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.

- Certo
- Errado

[0001 1101] A criação da `view` a seguir está certa ou errada?

```
create view colecao as (select * from livros where id in (select l
ivro_id from livraria));
```

- Certo
- Errado

[0001 1110] O PostgreSQL é um sistema de gerenciamento de banco de dados objeto relacional muito usado, no entanto, ele seria mais utilizado se fosse gratuito. Por ser uma tecnologia paga, tem dificuldade de se popularizar.

- Certo
- Errado

[0001 1111] O SGBD PostgreSQL possui vários operadores que combinam o resultado de duas consultas em um único resultado e são denominados de operadores de conjuntos. No intuito de usar estes operadores, são seguidas as seguintes regras.

1. As colunas correspondentes nos comandos `SELECT` devem ser do mesmo tipo de dados e o comando `SELECT` deve ter o mesmo número de colunas.

2. O comando `SELECT` deve ter o mesmo número de colunas e o nome da coluna do primeiro `SELECT` deve ser usado como cabeçalho.

3. O resultado do operador não possui qualquer linha duplicada, a menos que a cláusula `ALL` seja usada e o nome da coluna do primeiro `SELECT` usado como cabeçalho.

Assinale:

a) Se somente a regra 1 estiver correta.

b) Se somente a regra 2 estiver correta.

c) Se somente a regra 3 estiver correta.

d) Se somente as regras 1 e 2 estiverem corretas.

e) Se todas as regras estiverem corretas.

[0010 0000] O operador `WHERE` de um comando `SELECT` da SQL do SGBD PostgreSQL tem por finalidade:

a) Indicar a tabela que se deseja consultar.

b) Retornar as tuplas da segunda consulta que não estão na primeira.

c) Criar uma condição para a consulta.

d) Gerar uma exception em um comando SQL contido em uma consulta.

e) Confirmar a consulta no SGBD.

[0010 0001] Com SQL no PostgreSQL, é possível retornar dados de duas ou mais colunas através de `JOIN` entre tabelas.

- Certo
- Errado

[0010 0010] O PostgreSQL:

1. Permite a criação de consultas usando simultaneamente várias bases de dados.
2. Permite a geração de consultas pré-programadas através de stored procedures.
3. Permite o armazenamento de dados binários através do

campo tipo `BYTEA`.

Assinale a alternativa correta:

a) Somente as afirmativas 1 e 2 são verdadeiras.

b) Somente as afirmativas 2 e 3 são verdadeiras.

c) Somente a afirmativa 1 é verdadeira.

d) Somente a afirmativa 2 é verdadeira.

e) Somente a afirmativa 3 é verdadeira.

[0010 0011] Um DBA criou uma tabela em um Banco de Dados usando o seguinte comando:

```
CREATE TABLE mec (
        cidade varchar(80),
        temp_baixa int,
        temp_alta int,
        nivel_precip int,
        data timestamp );
```

Qual alternativa a seguir corresponde ao comando de inserção de dados na tabela `mec`?

a) `INSERT IN mec(cidade, temp_baixa, temp_alta, nivel_precip, data) VALUES ('Brasilia', 20, 34, 0.2, '13/09/2015');`

b) `INSERT ON mec(cidade, temp_baixa, temp_alta, nivel_precip, data) VALUES ('Brasilia', 20, 34, 0.2, '13/09/2015');`

c) `INSERT FROM mec(cidade, temp_baixa, temp_alta, nivel_precip, data) VALUES ('Brasilia', 20, 34, 0.2, '13/09/2015');`

d) `INSERT OVER mec(cidade, temp_baixa, temp_alta,`

```
nivel_precip, data) VALUES ('Brasilia', 20, 34, 0.2,
'13/09/2015');
```

e) `INSERT INTO mec(cidade, temp_baixa, temp_alta, nivel_precip, data) VALUES ('Brasilia', 20, 34, 0.2, '13/09/2015');`

[0010 0100] Um DBA criou uma tabela em um banco de dados utilizando o comando:

```
CREATE TABLE OBJETOS(
    NOME varchar(20),
    TIPO varchar(30));
```

Se o DBA tentar criar uma view com o mesmo nome, qual mensagem o banco vai retornar?

a) *ERROR: syntax error at or "create"*

b) *ERROR: relation "objetos" already exists*

c) *ERROR: column "objetos" of relation "objetos" already exists*

d) Não retornará erro e sim apenas uma advertência sobre os nomes duplicados.

e) Não retornará erro e deixará fazer a criação da `view`.

[0010 0101] O DBA está com dúvida de qual comando usar para fazer update em uma coluna `AUTOR` da tabela `livros`. Ajude-o informando o comando correto.

a) `UPDATE LIVROS AUTOR = 'VINICIUS CARVALHO';`

b) `UPDATE LIVROS INTO AUTOR = 'VINICIUS CARVALHO';`

c) `UPDATE LIVROS OVER 'AUTOR' = 'VINICIUS CARVALHO':`

d) `UPDATE LIVROS SET AUTOR = 'VINICIUS CARVALHO';`

e) `UPDATE LIVROS SET AUTOR 'VINICIUS CARVALHO';`

[0010 0110] O sistema de gerenciamento de banco de dados (SGBD) PostgreSQL é um modelo em código aberto que tem como base o modelo de desenvolvimento bazar.

- Certo
- Errado

[0010 0111] Com relação a cópias de segurança (backups), é correto afirmar que o `pg_dump` , ao ser executado, faz a importação do arquivo indicado.

- Certo
- Errado

[0010 1000] A função `count()` é usada para calcular a média entre dois números.

- Certo
- Errado

[0010 1001] A função `group by()` é utilizada para fazer a ordenação de uma consulta realizada.

- Certo
- Errado

[0010 1010] A função `having count()` é usada para agrupar os registros iguais do resultado de uma consulta.

- Certo
- Errado

[0010 1011] Qual o comando responsável por deletar uma tabela?

a) `CREATE TABLE...`

b) `ALTER TABLE...`

c) `UPDATE COLUMN...`

d) `DELETE TABLE...`

e) `DROP TABLE...`

[0010 1100] Quais tipos de dados a seguir não existem no PostgreSQL?

1. JSON
2. VARCHAR2
3. XML
4. NULL

Escolha a opção correta:

a) 1 e 3.

b) 1 e 4.

c) 2 e 4.

d) 3, 2 e 4.

e) 1 e 2.

[0010 1101] Quais tipos de dados a seguir são válidos para o PostgreSQL?

1. INT
2. PERSONALIZADO
3. NUMBER
4. VARCHAR2
5. TIMESTAMP

Escolha a opção correta:

a) 1 e 3.

b) 1, 2 e 5.

c) 2 e 4.

d) 1, 2 e 3.

e) 1, 2 e 4.

[0010 1110] Na criação de uma tabela, está sendo retornando um erro de sintaxe. Qual comando a seguir está correto?

a) `CREATE TABLE mytable(f1 number, f2 float, f3 varchar2(20));`

b) `CREATE TABLE mytable(f1 int, f2 float, f3 number);`

c) `CREATE TABLE mytable(f1 int, f2 float, f3 varchar2(10));`

d) `CREATE TABLE mytable(f1 int, f2 float, f3 text);`

[0010 1111] Em bancos de dados, um termo representa uma expressão booleana associada a um BD e que precisa ser avaliada como TRUE , por todo o tempo. Vamos supor um banco de dados de fornecedores e peças.

1. O valor do status de cada fornecedor está no intervalo de 500 a 900, inclusive.
2. Se houver peças, uma delas tem de ser amarela.
3. Dois fornecedores diferentes não têm o mesmo número de fornecedor.
4. Cada fornecedor com negócios no Brasil tem status 700.
5. Cada remessa envolve um fornecedor existente.
6. Nenhum fornecedor com status menor que 700 fornece peça alguma com uma quantidade maior que 350.

O exemplo descrito caracteriza o termo denominado de restrição de:

a) Atividade

b) Integridade

c) Confiabilidade

d) Disponibilidade

[0011 0000] Armazenar dados do tipo JSON, além de ser muito esperada, foi uma necessidade muito bem recebida pela comunidade de desenvolvedores que utilizam PostgreSQL, que passaram a armazenar JSON no banco de dados. Com base em seu conhecimento

sobre inserção de JSON em tabelas do banco de dados, selecione a inserção correta de um campo do tipo JSON.

a)

```
insert into produtos(produto_codigo,
                     produto_nome,
                     produto_valor,
                     produto_situacao,
                     data_criacao,
                     data_atualizacao,
                     produto_categoria,
                     produto_estoque)
            values('6234',
                   'COCA-COLA',
                   6,
                   'A',
                   '01/01/2016',
                   '01/01/2016',
                   '{"REFRIGERANTE",
                     "LATA",
                     "BEBIDA" ,
                     "COLA"}',
                   '{ "info_estoque":
                      { "tem_estoque": "SIM",
                        "quantidade": 17,
                        "ultima_compra": "01/01/16" }
                    }'
                   );
```

b)

```
insert into produtos(produto_codigo,
                     produto_nome,
                     produto_valor,
                     produto_situacao,
                     data_criacao,
                     data_atualizacao,
                     produto_categoria,
                     produto_estoque)
            values('6234',
                   'COCA-COLA',
                   6,
                   'A',
                   '01/01/2016',
                   '01/01/2016',
```

```
                    {"REFRIGERANTE",
                      "LATA",
                      "BEBIDA" ,
                      "COLA"},
                    '{ "info_estoque":
                       { "tem_estoque": "SIM",
                         "quantidade": 17,
                         "ultima_compra": "01/01/16" }
                     }'
                    );
```

c)

```
insert into produtos(produto_codigo,
                     produto_nome,
                     produto_valor,
                     produto_situacao,
                     data_criacao,
                     data_atualizacao,
                     produto_categoria,
                     produto_estoque)
            values('6234',
                   'COCA-COLA',
                   6,
                   'A',
                   '01/01/2016',
                   '01/01/2016',
                   {"REFRIGERANTE",
                     "LATA",
                     "BEBIDA" ,
                     "COLA"},
                   { "info_estoque":
                      { "tem_estoque": "SIM",
                        "quantidade": 17,
                        "ultima_compra": "01/01/16" }
                    }
                   );
```

d)

```
insert into produtos(produto_codigo,
                     produto_nome,
                     produto_valor,
                     produto_situacao,
                     data_criacao,
                     data_atualizacao,
                     produto_categoria,
                     produto_estoque)
```

```
            values('6234',
                   'COCA-COLA',
                   6,
                   'A',
                   '01/01/2016',
                   '01/01/2016',
                   {"REFRIGERANTE",
                     "LATA",
                     "BEBIDA" ,
                     "COLA"}',
                   '{ "info_estoque":
                      { "tem_estoque": "SIM",
                        "quantidade": 17,
                        "ultima_compra": "01/01/16" }
                    }
                   );
```

e)

```
insert into produtos(produto_codigo,
                     produto_nome,
                     produto_valor,
                     produto_situacao,
                     data_criacao,
                     data_atualizacao,
                     produto_categoria,
                     produto_estoque)
            values('6234',
                   'COCA-COLA',
                   6,
                   'A',
                   '01/01/2016',
                   '01/01/2016',
                '{"REFRIGERANTE",
                     "LATA",
                     "BEBIDA" ,
                     "COLA"}',
                   '{ "info_estoque":
                      { "tem_estoque": "SIM",
                        "quantidade": 17,
                        "ultima_compra"  "01/01/16" }
                    }
                   );
```

[0011 0001] PGSQL é uma linguagem procedural carregável desenvolvida para o sistema de banco de dados PostgreSQL. Como a maioria dos produtos de

banco de dados relacional, o PostgreSQL suporta funções de agregação. Uma função de agregação computa um único resultado para várias linhas de entrada. Por exemplo, para calcular a média, deve ser usada a seguinte função de agregação:

a) `query`

b) `media`

c) `med`

d) `avg`

e) `like`

[0011 0010] Paulo utiliza o `pg_dump` do PostgreSQL para fazer cópia de segurança de um banco de dados. Normalmente, ele faz cópias de segurança no formato `tar` e usa o `pg_restore` para reconstruir o banco de dados, quando necessário. O `pg_restore` pode selecionar o que será restaurado, ou mesmo reordenar os itens antes de restaurá-los, além de permitir salvar e restaurar objetos grandes. Certo dia, Paulo fez uma cópia de segurança do banco de dados chamado `trt13` para o arquivo `tribunal.tar` , incluindo os objetos grandes. Paulo utilizou uma instrução que permitiu a seleção manual e reordenação de itens arquivados durante a restauração. Porém, a ordem relativa de itens de dados das tabelas não pôde ser alterada durante o processo de restauração.

Paulo usou, em linha de comando, a instrução:

a) `pg_dump -Ec -h trt13 > tribunal.tar`

b) `pg_dump -Ft -b trt13 > tribunal.tar`

c) `pg_dump -tar -a trt13 > tribunal.tar`

d) `pg_dump -tar -c trt13 > tribunal.tar`

e) `pg_dump -Fp -b trt13 > tribunal.tar`

Capítulo 11

GABARITO

Pergunta	Reposta	Pergunta	Reposta
[0000 0001]	Errado	**[0001 1010]**	Errado
[0000 0010]	E	**[0001 1011]**	Certo
[0000 0011]	Certo	**[0001 1100]**	Certo
[0000 0100]	Certo	**[0001 1101]**	Certo
[0000 0101]	B	**[0001 1110]**	Errado
[0000 0110]	A	**[0001 1111]**	E
[0000 0111]	Errado	**[0010 0000]**	C
[0000 1000]	Errado	**[0010 0001]**	Certo
[0000 1001]	A	**[0010 0010]**	E
[0000 1010]	B	**[0010 0011]**	E
[0000 1011]	D	**[0010 0100]**	C
[0000 1100]	Errado	**[0010 0101]**	D
[0000 1101]	Certo	**[0010 0110]**	Certo
[0000 1110]	A	**[0010 0111]**	Errado
[0000 1111]	C	**[0010 1000]**	Errado
[0001 0000]	B	**[0010 1001]**	Errado
[0001 0001]	D	**[0010 1010]**	Errado
[0001 0010]	A	**[0010 1011]**	E
[0001 0011]	E	**[0010 1100]**	C
[0001 0100]	Errado	**[0010 1101]**	B

[0001 0101]	Certo	**[0010 1110]**	D
[0001 0110]	Errado	**[0010 1111]**	B
[0001 0111]	Errado	**[0011 0000]**	A
[0001 1000]	Certo	**[0011 0001]**	D
[0001 1001]	A	**[0011 0010]**	B

11.1 PARA PENSAR!

E aí? Como foi nos exercícios? Espero que tenha ido bem, para se sair bem nos próximos concursos de que for participar. A ideia foi dar uma noção de como são os exercícios das provas de concursos, e também para você colocar um pouco em prática a teoria aprendida no decorrer deste livro.

Se a sua intenção é fazer algum concurso público, existem vários sites com perguntas e gabaritos de provas já aplicadas, direcionadas para a área de tecnologia da informação. Eu retirei algumas questões do site `https://www.qconcursos.com` Bons estudos!

Sempre que existe aquela fase bônus no jogo é muito divertido. Que tal agora uma fase bônus no próximo capítulo para relaxarmos?! Aproveite! *#Bônus*

Capítulo 12

APÊNDICE A — PROFISSÃO DBA

"Falhar é uma opção aqui. Se as coisas não estão dando errado, você não está inovando o suficiente". — Elon Musk

Como dito, DBA é a abreviatura para *Data Base Administrator*, ou no bom e velho português, Administrador de Banco de Dados. Se você é da área, já deve ter ouvido falar. E se você está chegando agora, vai ouvir muito sobre DBA e pode até vir a precisar de um.

Cada vez mais tem sido um profissional mais requisitado no mercado de trabalho. Estamos vivendo na era da informação e, em meio dessa grande quantidade de dados, os DBAs estão se destacando no mercado de trabalho. Ainda são poucos os profissionais que se especializam em administração de banco de dados e áreas correlacionadas.

O que faz um DBA? Qual seria o perfil de um DBA? Como se tornar um?

Dia a dia de um DBA

O DBA é o responsável desde o hardware do servidor do banco de dados até a performance do banco de dados. Mas de muitas atividades, como principais tarefas de um DBA, temos:

- Analisar, definir e instalar o hardware do servidor, ou

então pela contratação do serviço nas nuvens que fará o papel do servidor.
- Instalar o software do banco de dados e fazer a sua manutenção, desde cuidar das atualizações a possíveis problemas.
- Lembra do projeto que fizemos durante o livro? Também é uma função do DBA, que é a criação das tabelas e outros objetos do banco de dados.
- Garantir a estabilidade e disponibilidade do banco de dados.
- Fazer backup do banco de dados, cuidar da integridade dos backups e, de preferência, que o este seja diário.
- Fazer o monitoramento diário e manter a performance do banco de dados.

Estas são apenas algumas atividades que um DBA realiza em seu cotidiano. Se você quer conhecer mais a fundo sobre a rotina de um administrador de banco de dados, nada melhor do que procurar um profissional que trabalhe como DBA.

Costumo dizer que os profissionais de T.I. estão disponíveis nas redes sociais. Busque por DBAs no LinkedIn ou em grupos do Facebook, e tire suas dúvidas. Tenho certeza de que você encontrará alguém para tirar suas dúvidas.

Perfil do DBA

Muitas vezes, você pode até gostar de fazer uma determinada atividade. No entanto, pode descobrir que não possui o perfil para trabalhar com aquela atividade em seu dia a dia. Toda profissão tem seus desafios, mas cada uma exige um perfil com características específicas.

As três características que acho essenciais para o profissional

que vai trabalhar administrando banco de dados são:

- **Alta capacidade de concentração:** pois o se o banco de dados parar, todos cobrarão o DBA. E neste momento será necessário possuir uma alta capacidade de concentração para desenvolver uma solução para o problema que estiver enfrentando.
- **Gostar de estudar:** acredito que, se você está na área da computação e não gosta de estudar, talvez está não seja sua área. A tecnologia de hoje provavelmente amanhã já estará ultrapassada. Não pare de estudar e se aperfeiçoar nunca.
- **Responder rapidamente sob pressão:** o banco de dados caiu! Você é o DBA responsável por ele. Você deve tomar uma atitude o mais rápido possível para minimizar os prejuízos. Está é, na minha opinião, a característica que determinará se você vai gostar de trabalhar como DBA.

Ninguém nasce dominando essas três características. Você pode desenvolvê-las a cada dia. Apenas você deve estar predisposto a desenvolvê-las.

Como se tornar um DBA

Estude, estude e, depois, pratique e pratique. Comece dominando os comandos `SQL` , que em sua maioria, são comuns aos bancos de dados relacionais. Pesquise os bancos de dados existentes e as características de cada um. Algum SGBD pode lhe atrair mais do que outros.

Se você escolher focar em um banco de dados, busque por certificações. São exames que alguma instituição é autorizada a aplicar e atestará que você domina a tecnologia. Busque por livros,

artigos, sites especializados, entre tantas outras fontes que você pode procurar na internet.

Depois que tiver dominando um banco de dados e todas as atividades inerentes ao papel de DBA, você estará apto a trabalhar como Administrador de Banco de Dados.

12.1 COMANDOS BÁSICOS E ÚTEIS

É sempre muito útil você ter na mão alguns comuns que são usados no dia a dia. Aqui estão alguns poucos, mas que são muito úteis e que, com certeza, serão utilizados frequentemente.

Comando	Descrição
`\?`	Obtém a lista completa de comando psql, incluindo os que não estão listados aqui
`\h`	Obtém ajudar sobre comandos SQL
`\q`	Sair do terminar psql
`\d`	Lista as tabelas, views e sequences da base de dados
`\du`	Listas os perfis disponíveis
`\dp`	Lista os privilégios de acesso
`\dt`	Lista as tabelas
`\l`	Lista todas as bases de dados
`\c`	Conecta a um banco de dados diferente. Deve-se colocar o nome do banco depois do comando
`\password`	Para alterar a senha do usuário conectado
`\conninfo`	Obtém informação sobre o banco de dados conectado e sobre a conexão

12.2 TRABALHANDO COM PGADMIN

Até agora, trabalhamos em nosso projeto somente usando o

console do banco de dados e linha de comando para realizar todas as tarefas. Fizemos tudo na "unha"! Mas existe uma outra forma de manipular os dados, por uma ferramenta visual. Ela possibilita fazer tudo o que fizemos até agora, só que de uma maneira mais simples.

Umas das ferramentas mais utilizadas para trabalhar com o PostgreSQL é 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/, e lá escolher a versão para o seu sistema operacional.

Após baixar e instalar, ao abrir pela primeira vez, o pgAdmin vai identificar os servidores do PostgreSQL que estão executando em sua máquina e solicitar a senha que você criou para o seu servidor. Desde o começo do nosso projeto, a minha senha é `senha`. E se você não alterou a sua, então também será. Na figura a seguir mostra onde você vai inserir a sua senha.

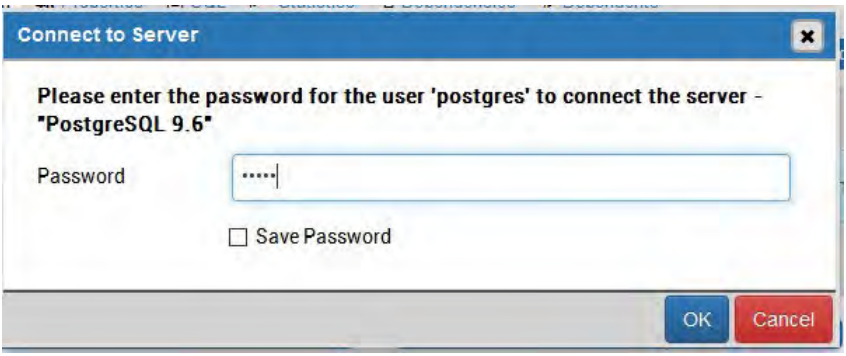

Figura 12.1: Informando a senha para se conectar ao servidor

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

objetos do nosso banco de dados.

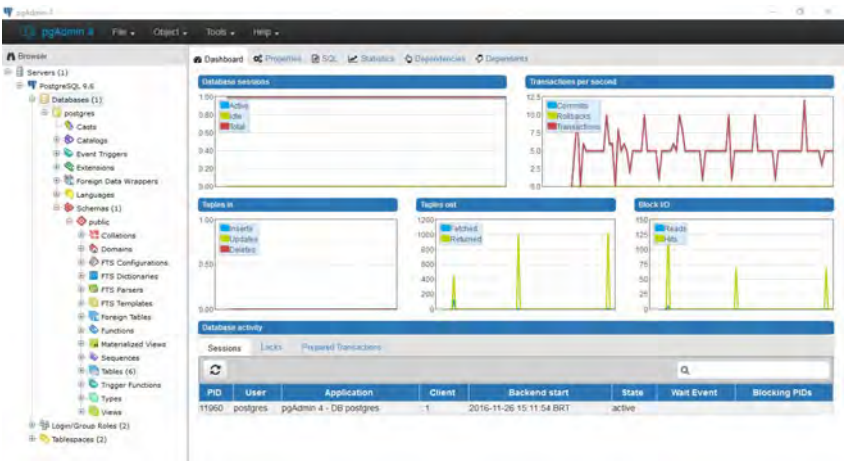

Figura 12.2: Tela inicial do pgAdmin

Executamos todos os nossos códigos até agora no console do PostgreSQL, e agora com pgAdmin os comandos podem ser executados na ferramenta através da Query Tool. Ela pode ser acessada por meio do `Menu > Tools > Query Tool` , conforme mostra a figura a seguir.

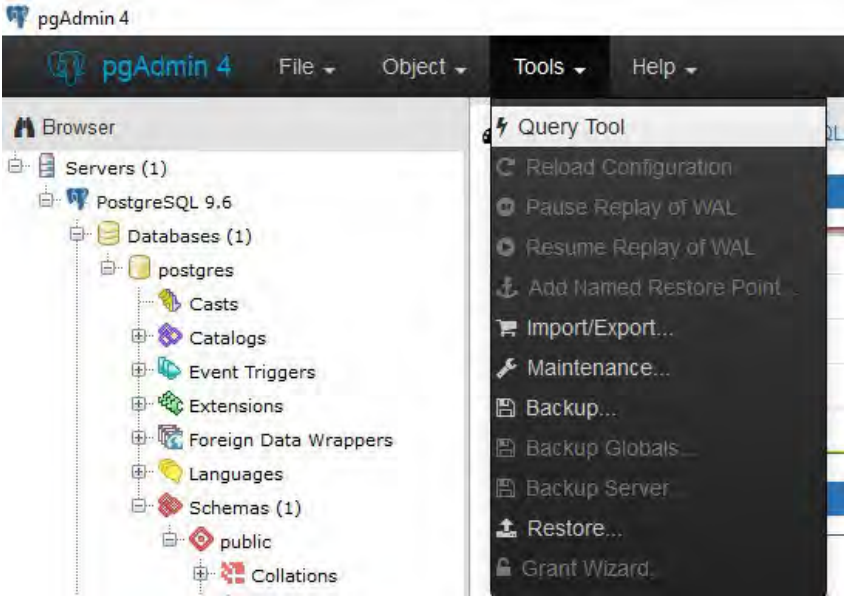

Figura 12.3: Área para executar os comandos

Ao acessar a Query Tool, abrirá um espaço no qual podemos escrever os comandos e mandar executar, conforme mostra a figura seguinte. Nela escrevi o comando o `select`, que vai retornar os registros da tabela `funcionarios`.

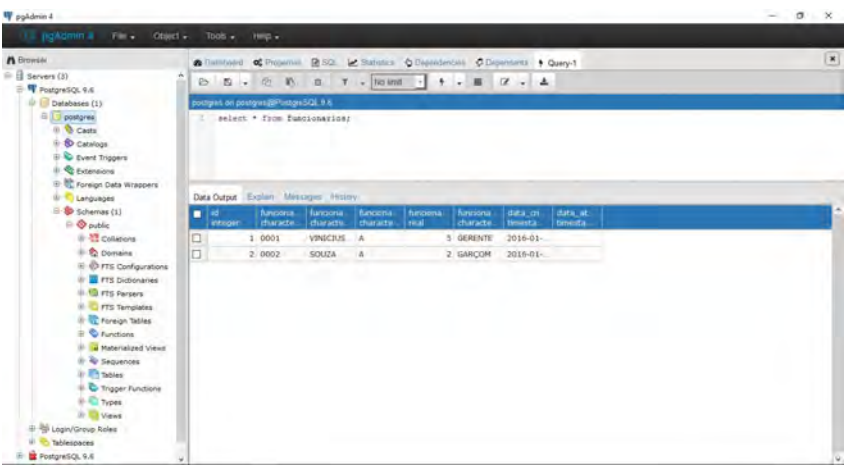

Figura 12.4: Fazer consulta na ferramenta

Observe que o resultado da consulta foi exibido na parte abaixo do espaço onde os comandos serão inseridos. Tudo que fizemos via console podemos fazer aqui na ferramenta. Então, é possível criarmos um banco de dados e uma tabela por ela.

Para criar um banco de dados, selecione do lado esquerdo `Databases` e, em seguida, vamos até a opção do menu `Object > Create > Database`.

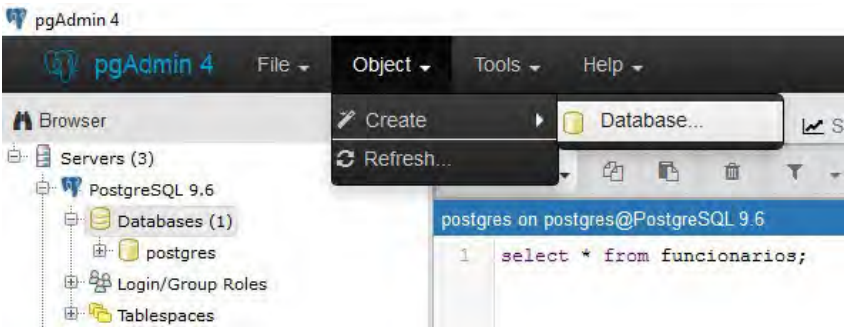

Figura 12.5: Criando um novo banco de dados pela ferramenta

Isso abrirá uma tela de configurações do banco de dados. Vamos inserir o nome do novo banco de dados `banco2` no campo `Database`, e clicar no botão `Save`.

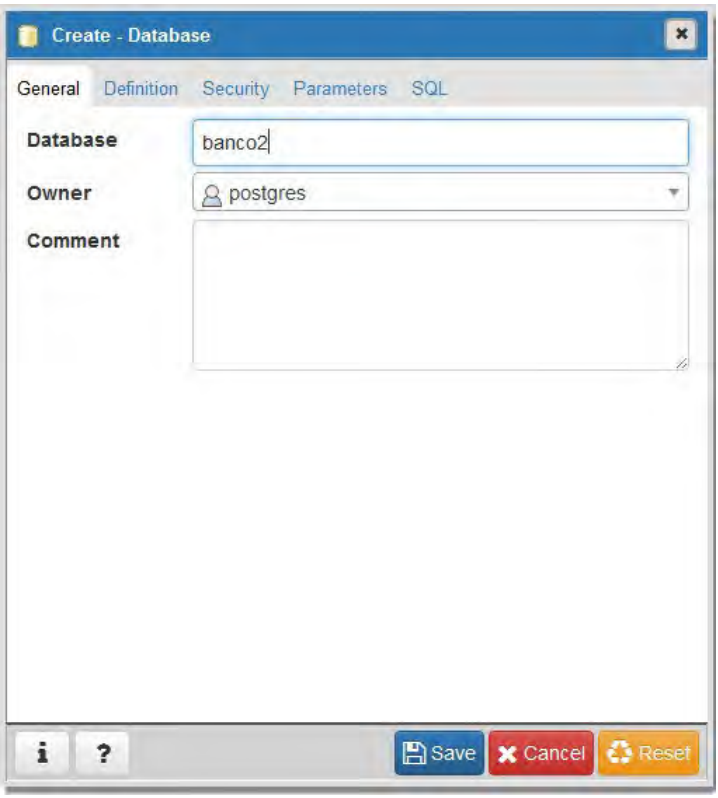

Figura 12.6: Inserindo o nome do novo banco

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

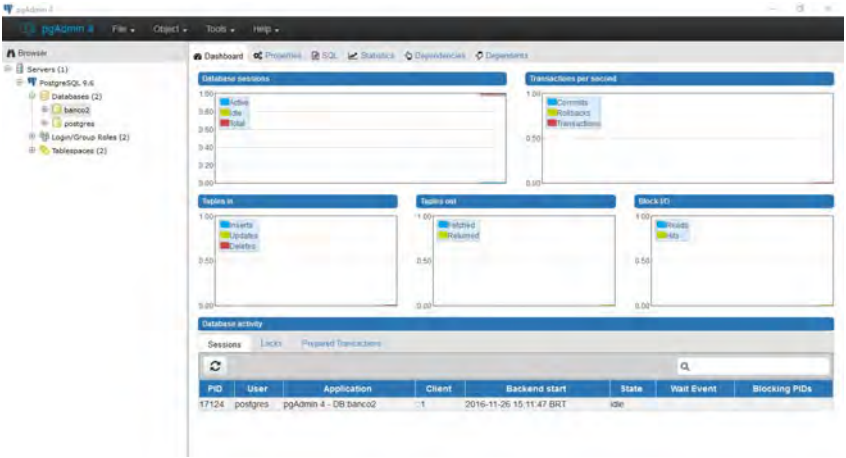

Figura 12.7: Criando um novo banco de dados pela ferramenta

Agora, utilizando o novo banco de dados criado, criaremos uma tabela através da ferramenta. Podemos escrever o código no espaço através da Query Tool, mas isso já sabemos fazer, por isso vamos criar a nova tabela pela ferramenta.

Na lista do lado esquerdo, navegue até `Tables` e, com o botão direito do mouse, clique em `Create > Table` . Uma tela aparecerá para nos auxiliar nessa criação.

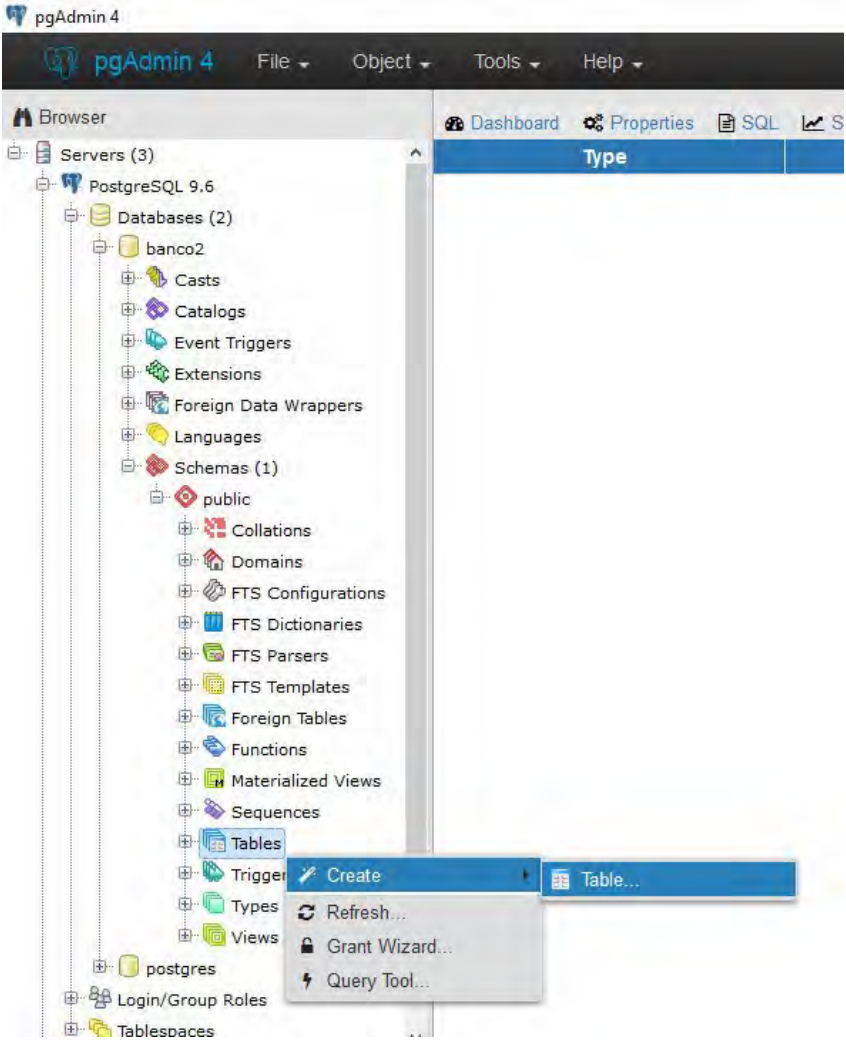

Figura 12.8: Criando tabela pela ferramenta

Primeiro, vamos informar o nome da tabela na aba `General`. A nossa nova tabela vai se chamar `tabela_produto`.

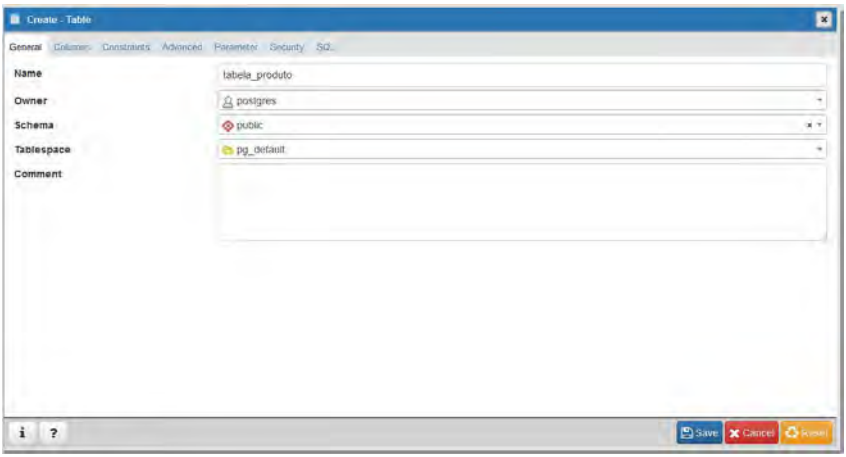

Figura 12.9: Nomeando a nova tabela

Na sequência na aba `Columns`, vamos inserir as suas colunas. Em vez de escrever qual a configuração de cada coluna, da maneira que fazemos quando escrevemos o comando, agora selecionaremos o tipo de cada coluna, o tamanho (se for necessário), se poderá ser nula e se o campo é uma chave primaria. Então, faremos a adição dos campos pelo botão com o sinal `+`, no canto superior direito.

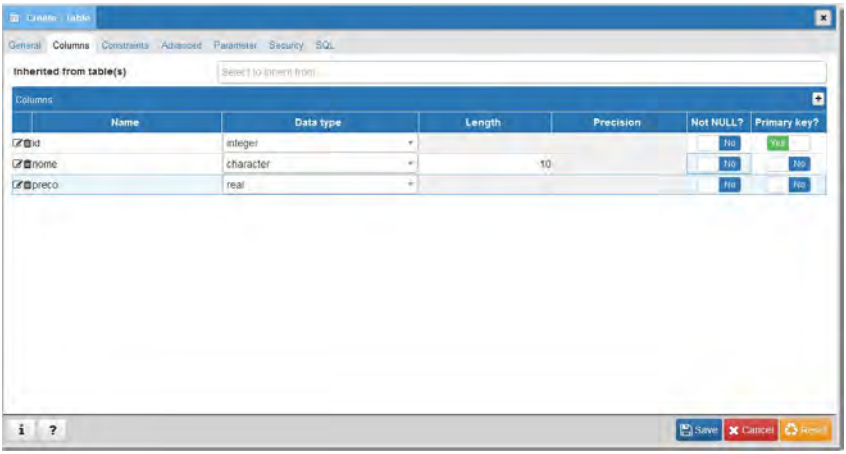

Figura 12.10: Inserindo colunas na nova tabela

Inserindo as colunas, passamos para a aba `Constraints`. Como a coluna `id` será uma chave primária, ela já vai aparecer na listagem. Precisamos apenas nomeá-la, conforme a próxima figura.

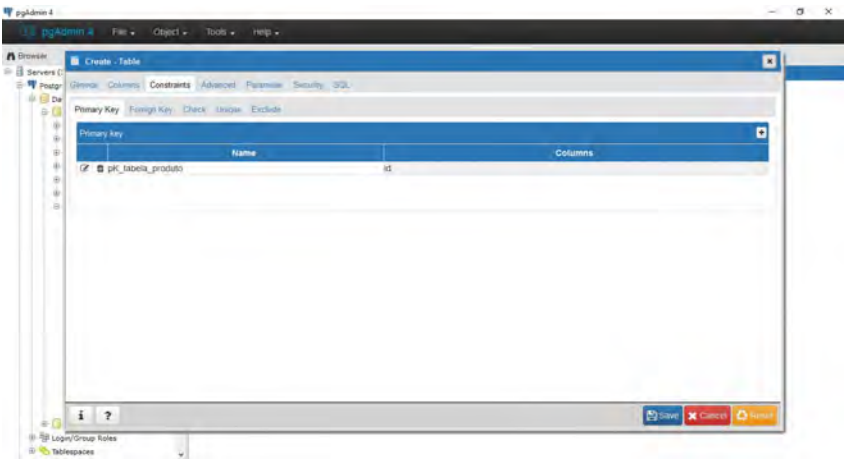

Figura 12.11: Constraint na nova tabela

Após fazer as configurações das tabelas, vamos até a aba `SQL`. Lá visualizaremos o código para criação da tabela que será aplicado no banco de dados. Na sequência, basta clicar no botão `Save`, que a tabela será criada.

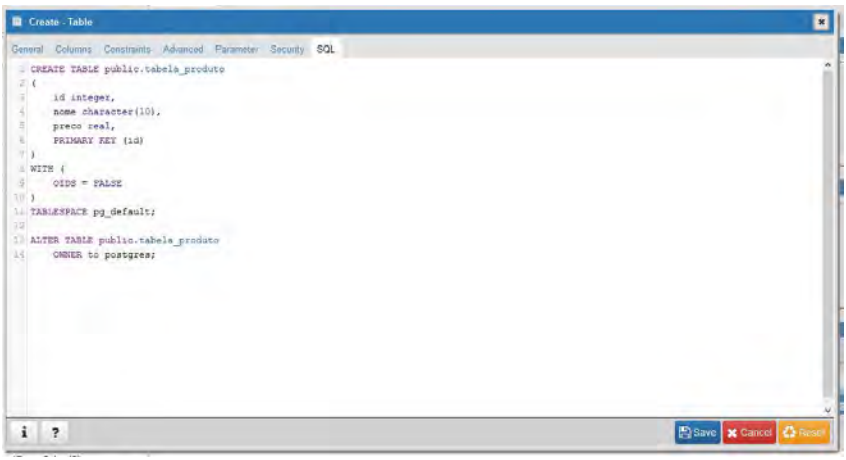

Figura 12.12: SQL gerado pela ferramenta

Muito simples, não é mesmo!? Esta ferramenta facilita bastante a nossa vida. Depois que você já conhece os comandos, não precisa ficar escrevendo eles na mão. Dependendo do cenário que você estiver trabalhando, até necessite utilizar o console do SGBD. Mas em outras ocasiões, poderá usar uma ferramenta visual como o pgAdmin.

Para melhor aprendizado, foi interessante ter trabalhado até agora somente com o console do PostgreSQL, pois é uma forma de entendermos e fixarmos bem todos os comandos aprendidos. Agora que os conhecemos bem, podemos migrar para o pgAdmin, já que ele nos dará mais produtividade durante o dia a dia.

Há quem prefira trabalhar apenas pelo console. Fica a seu critério. Dependendo do cenário em que você estiver trabalhando, poderá trabalhar tanto pelo console quanto pela ferramenta visual.

12.3 PARA PENSAR E AGRADECER!

E aí, ficou com vontade de se tornar um DBA? É uma excelente e promissora carreira. Você tem a possibilidade de se aperfeiçoar em diversos bancos, ou se aprofundar em apenas um. Se quer conhecer outros gerenciadores de banco de dados, procure os outros livros da Casa do Código. Tem sobre Oracle "SQL: Uma abordagem para bancos de dados Oracle" e "PL/SQL: Domine a linguagem do banco de dados Oracle", MySQL (o meu primeiro livro) "MySQL: Comece com o principal banco de dados open source do mercado" e NoSQL "NoSQL: Como armazenar os dados de uma aplicação moderna".

Outra dica é procurar certificações focadas em determinados banco de dados. Elas são muito bem conceituadas no mercado de trabalho. Vale muito a pena.

Muito obrigado por chegar até aqui. Espero que tenha gostado

do conteúdo do livro e que 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. Isso que vai lhe diferenciar no mercado de trabalho. Ótimos estudos!

"Keep Coding" — Vinícius Carvalho