

O'REILLY®

Introdução à linguagem

**SQL**

---

ABORDAGEM PRÁTICA  
PARA INICIANTES



novatec

Thomas Nield

Introdução à linguagem

**SQL**

ABORDAGEM PRÁTICA PARA INICIANTES

**Thomas Nield**

**O'REILLY®**  
Novatec

Authorized Portuguese translation of the English edition of Getting Started with SQL, ISBN 9781491938614 © 2016 Thomas Nield. This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

Tradução em português autorizada da edição em inglês da obra Getting Started with SQL, ISBN 9781491938614 © 2016 Thomas Nield. Esta tradução é publicada e vendida com a permissão da O'Reilly Media, Inc., detentora de todos os direitos para publicação e venda desta obra.

© Novatec Editora Ltda. 2016.

Todos os direitos reservados e protegidos pela Lei 9.610 de 19/02/1998. É proibida a reprodução desta obra, mesmo parcial, por qualquer processo, sem prévia autorização, por escrito, do autor e da Editora.

Editor: Rubens Prates

Assistente editorial: Priscila A. Yoshimatsu

Tradução: Aldir José Coelho Corrêa da Silva

Revisão gramatical: Smirna Cavalheiro

Editoração eletrônica: Carolina Kuwabata

ISBN: 978-85-7522-746-6

Histórico de edições impressas:

Abril/2016 Primeira edição

Novatec Editora Ltda.

Rua Luís Antônio dos Santos 110

02460-000 – São Paulo, SP – Brasil

Tel.: +55 11 2959-6529

E-mail: [novatec@novatec.com.br](mailto:novatec@novatec.com.br)

Site: [www.novatec.com.br](http://www.novatec.com.br)

Twitter: [twitter.com/novateceditora](https://twitter.com/novateceditora)

Facebook: [facebook.com/novatec](https://facebook.com/novatec)

LinkedIn: [linkedin.com/in/novatec](https://linkedin.com/in/novatec)

# Sumário

## Introdução

## Prefácio

## Capítulo 1 ■ Por que aprender SQL?

O que é SQL e por que ele é vendável?

Para quem se destina o SQL?

## Capítulo 2 ■ Bancos de dados

O que é um banco de dados?

Examinando os bancos de dados relacionais

Por que tabelas separadas?

Selecionando uma solução de banco de dados

## Capítulo 3 ■ SQLite

O que é SQLite?

SQLiteStudio

Importando e procurando bancos de dados

## Capítulo 4 ■ SELECT

Recuperando dados com SQL

Expressões em instruções SELECT

Concatenação de texto

Resumo

## Capítulo 5 ■ WHERE

Filtrando registros

Usando WHERE com números

Instruções AND, OR e IN

Usando WHERE com texto

[Usando WHERE com booleanos](#)

[Manipulando NULL](#)

[Agrupando condições](#)

[Resumo](#)

## **Capítulo 6 ■ GROUP BY e ORDER BY**

[Agrupando registros](#)

[Ordenando registros](#)

[Funções de agregação](#)

[A instrução HAVING](#)

[Obtendo registros distintos](#)

[Resumo](#)

## **Capítulo 7 ■ Instruções CASE**

[A instrução CASE](#)

[Agrupando instruções](#)

[O truque da instrução CASE “Zero/Null”](#)

[Resumo](#)

## **Capítulo 8 ■ JOIN**

[Associando tabelas](#)

[INNER JOIN](#)

[LEFT JOIN](#)

[Outros tipos de operador JOIN](#)

[Associando várias tabelas](#)

[Agrupando JOINS](#)

[Resumo](#)

## **Capítulo 9 ■ Design de banco de dados**

[Planejando um banco de dados](#)

[A conferência SurgeTech](#)

[Participantes](#)

[Empresas](#)

[Apresentações](#)

[Salas](#)

[Comparecimento nas apresentações](#)

[Chaves primária e externa](#)  
[O esquema](#)  
[Criando um novo banco de dados](#)  
[CREATE TABLE](#)  
[Definindo as chaves externas](#)  
[Criando views](#)  
[Resumo](#)

## **Capítulo 10 ■ Gerenciando dados**

[INSERT](#)  
[Inserções múltiplas](#)  
[Testando as chaves externas](#)  
[DELETE](#)  
[TRUNCATE TABLE](#)  
[UPDATE](#)  
[DROP TABLE](#)  
[Resumo](#)

## **Capítulo 11 ■ Seguindo em frente**

### **Apêndice A ■ Operadores e funções**

[Apêndice A1 – consultas com expressões de literais](#)  
[Apêndice A2 – operadores matemáticos](#)  
[Apêndice A3 – operadores de comparação](#)  
[Apêndice A4 – operadores lógicos](#)  
[Apêndice A5 – operadores de texto](#)  
[Apêndice A6 – funções básicas comuns](#)  
[Apêndice A7 – funções de agregação](#)  
[Apêndice A8 – funções de data e hora](#)  
[Funções de data](#)  
[Funções de hora](#)  
[Funções de data/hora](#)

### **Apêndice B ■ Tópicos suplementares**

[Apêndice B1 – outros tópicos de interesse](#)  
[Apêndice B2 – melhorando o desempenho com índices](#)

Apêndice B3 – transações

Sobre o autor

Colofão

# Introdução

Nas três últimas décadas, os computadores conquistaram o mundo. Éramos analógicos vinte e cinco anos atrás. Comunicávamo-nos usando um telefone POTS analógico, sintonizávamos estações de rádio FM analógicas e íamos à biblioteca procurar informações nas estantes. Os prédios eram construídos com o uso de projetos desenhados à mão; artistas gráficos trabalhavam com caneta, pincel e tinta; os músicos dedilhavam cordas, tocavam instrumentos de sopro e gravavam em fita analógica; e os aviões eram controlados por cabos físicos que conectavam o manche às superfícies de controle.

Porém, agora tudo é computadorizado e digital. Consequentemente, todos os membros da sociedade precisam estar familiarizados com os computadores. Isso não significa ter o conhecimento profundo de um técnico, mas da mesma forma que os poetas precisam estudar um pouco de matemática e física, e os matemáticos precisam ler um pouco de poesia, atualmente todas as pessoas precisam saber algo sobre os computadores.

Acho que este livro ajuda a eliminar a lacuna existente entre o que sabem os técnicos e os leigos, fornecendo uma discussão acessível e fácil de ler sobre o SQL – uma tecnologia básica de banco de dados.

*Richard Hipp, criador do SQLite*

# Prefácio

Ninguém precisa aprender como um motor de carro funciona para poder dirigir. O que realmente importa em tecnologias como o SQL é que elas permitem que você se concentre no problema empresarial, sem se preocupar com a maneira como os detalhes técnicos são executados. Este livro fornece uma abordagem prática do uso do SQL e deixa de lado detalhes técnicos desnecessários não pertinentes às suas necessidades imediatas. Grande parte do conteúdo se baseia em exercícios práticos com bancos de dados reais que você pode baixar para ver como os conceitos são aplicados. Ao terminar o livro, você terá conhecimento prático para trabalhar com bancos de dados, assim como para usá-los na superação de seus desafios empresariais.

## Como usar o livro

Este livro foi planejado para ensinar os fundamentos do SQL e o trabalho com bancos de dados. Leitores que tiverem experiência no uso de planilhas do Excel vão achar o material acessível e ao mesmo tempo desafiador. Quem não conhece o Excel pode ter mais dificuldade. Pode ser útil familiarizar-se com os conceitos usados no Excel, como linhas, colunas, tabelas, expressões matemáticas (por exemplo, as fórmulas do Excel) e cálculos de agregação (como SOMA, MÉDIA, MÍNIMO, MÁXIMO, CONT.NÚM). Esses conceitos também serão ensinados aqui, mas alguma experiência prática em Excel ajudará a acelerar o aprendizado.

Um conhecimento básico em computação é necessário e os leitores devem saber como navegar em pastas e copiar/colar arquivos, além de baixar e salvar arquivos a partir da web.

Ao percorrer o material, use um computador para praticar os exemplos. Embora algumas pessoas consigam aprender apenas lendo, é melhor pôr

em prática o material para reforçar o aprendizado.

A habilidade chega pelo uso e prática contínuos. Em sua profissão, é provável que você use muito algumas funcionalidades do SQL e outras nem tanto. Não há problema. É mais importante dominar o que sua função demanda, e consultar este livro (ou o Google) como referência quando precisar de respostas sobre um tópico desconhecido.

No trabalho com tecnologia não é preciso saber tudo. Na verdade, as áreas tecnológicas são tão vastas que isso seria impossível. Logo, é útil ser mais focado e aprender apenas o suficiente para cumprir a tarefa atual. Caso contrário, você pode ficar sobrecarregado ou se distrair aprendendo tópicos irrelevantes. Espero que este livro lhe dê uma base de conhecimento e que depois você possa continuar o aprendizado com tópicos que lhe sejam pertinentes.

Sinta-se à vontade para entrar em contato comigo em [tmnield@outlook.com](mailto:tmnield@outlook.com) e responderei às perguntas da melhor maneira possível. Se tiver perguntas sobre como focar sua carreira de acordo com conjuntos de habilidades técnicas ou se tiver dúvidas sobre SQL, talvez eu possa ajudar. O ideal seria que, além de este material aumentar seu conjunto de habilidades e oportunidades de carreira, ele também despertasse novos interesses que o empolgasse como aconteceu comigo.

## Convenções usadas no livro

As convenções tipográficas a seguir são usadas neste livro:

### *Itálico*

Indica novos termos, URLs, emails e nomes e extensões de arquivo.

### **Largura constante**

Usada nas listagens de programas, assim como dentro de parágrafos para referenciar elementos do programa como nomes de variáveis ou funções, bancos de dados, tipos de dados, variáveis de ambiente, instruções e palavras-chave.

### **Largura constante em negrito**

Mostra comandos ou outros tipos de texto que devam ser digitados literalmente pelo usuário.

*Largura constante em itálico*

Mostra texto que deve ser substituído por valores fornecidos pelo usuário ou valores determinados pelo contexto.



Esse elemento significa uma nota geral.

## Usando exemplos de código (de acordo com a política da O'Reilly)

Há material complementar (exemplos de código, exercícios etc.) disponível para download em [https://github.com/thomasnield/oreilly\\_getting\\_started\\_with\\_sql](https://github.com/thomasnield/oreilly_getting_started_with_sql).

Este livro existe para ajudá-lo a executar seu trabalho. Os exemplos de código oferecidos no livro podem ser usados em seus programas e documentação. Não é preciso pedir permissão a menos que você esteja reproduzindo uma parte significativa do código. Por exemplo, escrever um programa que use vários trechos de código do livro não requer permissão. Vender ou distribuir um CD-ROM de exemplos de livros da O'Reilly requer permissão. Responder a uma pergunta citando este livro e seus exemplos de código não requer permissão. Incorporar um número substancial de exemplos de código do livro à documentação de seus produtos requer permissão.

Apreciamos, mas não exigimos, a atribuição de autoria. Geralmente a atribuição de autoria inclui o título, o autor, a editora e o ISBN. Por exemplo, “*Getting Started with SQL* de Thomas Nield (O'Reilly). Copyright 2016 Thomas Nield, 978-1-4919-3861-4”.

Se achar que o uso que está fazendo dos exemplos de código não se enquadra no uso justo ou na permissão discutida anteriormente, procure-nos em [permissions@oreilly.com](mailto:permissions@oreilly.com).

## Como entrar em contato conosco

Envie seus comentários e suas dúvidas sobre este livro à editora escrevendo para: [novatec@novatec.com.br](mailto:novatec@novatec.com.br).

Temos uma página web para este livro na qual incluímos erratas, exemplos e quaisquer outras informações adicionais.

- Página da edição em português

<http://www.novatec.com.br/catalogo/7522501-introducao-sql>

- Página da edição original em inglês

<http://bit.ly/getting-started-with-sql>

Para obter mais informações sobre os livros da Novatec, acesse nosso site em <http://www.novatec.com.br>.

## Agradecimentos

Tenho a sorte de contar com pessoas maravilhosas ao meu redor e reconheço sua importância em minha vida e em tudo que faço. Se não fosse por elas, provavelmente este livro não teria sido escrito.

Em primeiro lugar, gostaria de agradecer a meus pais. Eles fizeram de tudo para garantir meu futuro. Tenho certeza de que não teria as oportunidades que encontro hoje sem sua ajuda. Meu pai trabalhou muito para dar uma educação melhor para meus irmãos e para mim e minha mãe sempre me incentivou mesmo quando eu resistia. Ela me ensinou a nunca me acomodar e a sempre ultrapassar meus limites.

É difícil expressar quanto sou grato a meus chefes, gerentes e colegas do Gerenciamento de Receitas da Southwest Airlines. Justin Jones e Timothy Keeney têm um espírito guerreiro e um entusiasmo por inovações que poucos possuem. Eles representam realmente a liderança e a alma da Southwest Airlines, e o que é mais importante, são ótimas pessoas. Sempre serão meus amigos e tornam difícil imaginar uma vida sem a Southwest Airlines.

Robert Haun, Brice Taylor e Allison Russell se esforçam incansavelmente para nossa equipe ser líder em inovações e sempre perseguir novas ideias, e tenho a felicidade de trabalhar no ambiente que eles ajudaram a criar.

Também preciso agradecer a Matt Louis, por me trazer para o Gerenciamento de Receitas, e a Steven Barsalou, que me fez perceber que eu conhecia muito pouco sobre SQL. Steven foi a primeira pessoa em quem pensei quando precisei de um revisor para este livro e sou grato por ele ter participado do projeto.

Temos então a equipe de projeto com a qual trabalho diariamente: Brian Denholm, Paul Zigler, Bridget Green, Todd Randolph e Chris Solomon. As proezas que executamos como equipe sempre me surpreendem. Brian é o tipo de gerente de projeto que consegue mesclar com sucesso os jargões de tecnologia e negócios e não hesita em pôr mãos à obra e trabalhar diretamente com SQL e com a ocasional revisão de códigos. Quero oferecer um agradecimento especial a Chris Solomon, por me ajudar em tudo que faço hoje. Além de ter um talento especial para absorver grandes volumes de conhecimento técnico e mantê-los direcionados para o ponto de vista empresarial, ele também é uma pessoa da qual tenho o privilégio de ser amigo. Chris é sempre um participante essencial em qualquer projeto e fiquei empolgado quando ele aceitou revisar o livro.

Não posso me esquecer dos ótimos colegas que trabalhavam no Departamento de Conformidade de Operações Terrestres às Normas de Segurança da Southwest Airlines, entre eles Marc Stank, Reuben Miller, Mary Noel Hennes e todas as outras pessoas com as quais tive a sorte de trabalhar. Ingressei e interagi nesse departamento alguns anos atrás e algumas de minhas lembranças mais queridas vêm de lá. Foi nesse local que descobri minha paixão por tecnologia, e eles me deram muitas oportunidades para perseguir esse sonho, fosse lançando bancos de dados às pressas ou criando o protótipo de um aplicativo para iPad.

Quando anunciei que estava publicando este livro não esperava que Richard Hipp, fundador e criador do SQLite, me procurasse. Richard bondosamente se ofereceu para ser o revisor técnico e sua participação foi uma grande honra. A comunidade de tecnologia continua a me surpreender, e o fato de Richard Hipp ter participado do projeto mostra quanto ela é única e integrada.

Shannon Cutt foi o editor da O'Reilly designado para o livro. Este é meu primeiro livro e não sabia como seria a experiência de publicar. Shannon tornou publicar uma experiência tão agradável que estou ansioso para escrever novamente. Obrigado, Shannon, você foi fantástico!

Por fim, mas não menos importante, quero agradecer à Watermark Church e aos voluntários da Careers in Motion por serem o veículo que tornou este livro possível. Escrevi o “livro” inicialmente como um serviço público para ajudar profissionais desempregados na área de Dallas. Foi por seu encorajamento que decidi publicá-lo, e quero dedicar um agradecimento especial a Martha Garza, por sua insistência. Aprendi que coisas maravilhosas podem acontecer quando você doa seu tempo para ajudar os outros.

## CAPÍTULO 1

# Por que aprender SQL?

## O que é SQL e por que ele é vendável?

É uma declaração óbvia a de que o cenário empresarial está mudando rapidamente. Grande parte dessa tendência é viabilizada pela tecnologia e originada pelo boom dos dados empresariais. As empresas estão investindo grandes quantias para coletar e armazenar dados. No entanto, o que preocupa muitos líderes e gerentes empresariais atualmente é como tornar esses dados relevantes e usá-los. É aí que entra em cena o *SQL*, que é a abreviação de *Structured Query Language*. Ele fornece um meio de acessar e manipular dados de maneira significativa e fornece insights empresariais que antes não eram possíveis.

As empresas estão coletando dados a taxas exponenciais e há uma demanda igualmente crescente por pessoas que saibam como analisá-los e gerenciá-los. A Stack Overflow, a comunidade de programação mais ativa mundialmente, fez uma pesquisa abrangente com seus membros em 2015. A codificação da Apple era a tecnologia mais procurada e oferecia um salário médio que beirava os seis dígitos. Porém, o SQL apareceu em quinto lugar, com um salário que não ficava muito atrás. Nos últimos anos, de repente, os dados ganharam destaque – mesmo assim, poucas pessoas sabem como acessá-los significativamente, o que aumentou a demanda por especialistas em SQL.

## Para quem se destina o SQL?

Uma percepção equivocada que se tem sobre o SQL é a de que se trata de uma ferramenta de TI e, portanto, só é aplicável aos profissionais de tecnologia (e não de negócios). Da forma que o mundo se encontra hoje,

isso está longe de ser verdade. Executivos, gerentes, profissionais de TI e engenheiros podem se beneficiar do aprendizado de SQL para posicionar melhor suas carreiras. O SQL pode abrir muitas oportunidades de carreira porque permite que as pessoas conheçam melhor suas áreas de atuação por intermédio dos dados que as impulsionam. Pelo lado empresarial, o interesse em SQL pode levar a funções analíticas, gerenciais, estratégicas e baseadas em pesquisas ou em projetos. No que diz respeito à TI, pode levar a funções pertinentes ao design de bancos de dados, à administração de bancos de dados, à engenharia de sistemas, ao gerenciamento de projetos de TI e até mesmo ao desenvolvimento de softwares.

## CAPÍTULO 2

# Bancos de dados

### O que é um banco de dados?

Em uma definição mais ampla, um *banco de dados* é qualquer coisa que colete e organize dados. Uma planilha contendo reservas de clientes é um banco de dados, assim como um arquivo de texto simples contendo dados de horários de voos. Os próprios dados de texto simples podem ser armazenados em vários formatos, inclusive XML e CSV.

Profissionalmente, no entanto, quando alguém se refere a um “banco de dados” provavelmente está se referindo a um *sistema de gerenciamento de banco de dados relacional* (RDBMS, Relational Database Management System). Esse termo pode soar técnico e ameaçador, mas um RDBMS é simplesmente um tipo de banco de dados contendo uma ou mais tabelas que podem estar relacionadas entre si.

### Examinando os bancos de dados relacionais

Uma tabela é um conceito familiar. Ela tem colunas e linhas para armazenar dados, semelhante ao que ocorre em uma planilha. As tabelas podem se relacionar umas com as outras, como a tabela ORDER, que recorre a uma tabela CUSTOMER para obter informações de clientes.

Por exemplo, suponhamos que tivéssemos uma tabela ORDER com um campo chamado CUSTOMER\_ID (Figura 2.1).

	ORDER_ID	ORDER_DATE	SHIP_DATE	CUSTOMER_ID	PRODUCT_ID	ORDER_QTY	SHIPPED
1	3	2015-04-20	2015-04-23	3	5	300	false
2	4	2015-04-18	2015-04-22	5	4	375	false
3	1	2015-04-15	2015-04-18	1	1	450	false
4	5	2015-04-17	2015-04-20	3	2	500	false
5	2	2015-04-18	2015-04-21	3	2	600	false

*Figura 2.1 – Tabela ORDER com um campo CUSTOMER\_ID.*

É provável que haja outra tabela, talvez chamada `CUSTOMER` (Figure 2.2), contendo as informações de cliente de cada `CUSTOMER_ID`.

CUSTOMER ID	NAME	REGION	STREET ADDRESS	CITY	STATE	ZIP
1	LITE Industrial	Southwest	729 Ravine Way	Irving	TX	75014
2	Rex Tooling Inc	Southwest	6129 Collie Blvd	Dallas	TX	75201
3	Re-Barre Construction	Southwest	9043 Windy Dr	Irving	TX	75032
4	Prairie Construction	Southwest	264 Long Rd	Moore	OK	62104
5	Marsh Lane Metal Works	Southeast	9143 Marsh Ln	Avondale	LA	79782

*Figura 2.2 – Tabela CUSTOMER.*

Ao percorrer a tabela `ORDER`, podemos usar `CUSTOMER_ID` para procurar informações de clientes na tabela `CUSTOMER`. Essa é a ideia básica existente por trás de um “banco de dados relacional”, em que as tabelas podem ter campos que apontem para informações de outras tabelas. Talvez esse conceito soe familiar se você já usou PROCV no Excel para recuperar informações em uma planilha a partir de outra planilha de uma pasta de trabalho.

## Por que tabelas separadas?

Por que as tabelas são separadas e projetadas dessa forma? O motivo é a *normalização*, que é a separação dos diferentes tipos de dados em suas próprias tabelas em vez de serem inseridos na mesma tabela. Se tivéssemos todas as informações em uma única tabela, ela seria redundante, saturada e de difícil manutenção. Imagine se armazenássemos informações de clientes na tabela `ORDER`. A Figura 2.3 mostra como ela ficaria.

NAME	REGION	STREET ADDRESS	CITY	STATE	ZIP	ORDER ID	ORDER DATE	SHIP DATE	ORDER QTY	SHIPPED
1 LITE Industrial	Southwest	729 Ravine Way	Irving	TX	75014	1	2015-04-15	2015-04-18	450	false
2 Re-Barre Construction	Southwest	9043 Windy Dr	Irving	TX	75032	2	2015-04-18	2015-04-21	600	false
3 Re-Barre Construction	Southwest	9043 Windy Dr	Irving	TX	75032	3	2015-04-20	2015-04-23	300	false
4 Marsh Lane Metal Works	Southeast	9143 Marsh Ln	Avondale	LA	79782	4	2015-04-18	2015-04-22	375	false
5 Re-Barre Construction	Southwest	9043 Windy Dr	Irving	TX	75032	5	2015-04-17	2015-04-20	500	false

*Figura 2.3 – Tabela não normalizada.*

Observe que no caso da Re-Barre Construction, alguém precisou fornecer as informações de cliente (nome, região, endereço, cidade, estado e código postal) três vezes para todos os três pedidos. Isso é redundante, ocupa

espaço de armazenamento desnecessário e é de difícil manutenção. Imagine se o endereço de um cliente mudasse e você precisasse atualizar todos os pedidos. É por isso que é melhor separar CUSTOMERS e ORDERS em duas tabelas distintas. Se você precisar alterar o endereço de um cliente, só terá de alterar um único registro na tabela CUSTOMERS (Figura 2.4).

	CUSTOMER ID	NAME	REGION	STREET ADDRESS	CITY	STATE	ZIP
1		1 LITE Industrial	Southwest	729 Ravine Way	Irving	TX	75014
2		2 Rex Tooling Inc	Southwest	6129 Collie Blvd	Dallas	TX	75201
3		3 Re-Barre Construction	Southwest	10917 Long Way Rd	Irving	TX	75032
4		4 Prairie Construction	Southwest	264 Long Rd	Moore	OK	62104
5		5 Marsh Lane Metal Works	Southeast	9143 Marsh Ln	Avondale	LA	79782

Figura 2.4 – Tabela normalizada.

Examinaremos os relacionamentos entre tabelas novamente no Capítulo 8, e aprenderemos como usar o operador `JOIN` para mesclar tabelas em uma consulta de modo que as informações do cliente possam ser visualizadas junto com o pedido.

## Selecionando uma solução de banco de dados

Os bancos de dados relacionais e o SQL não são patenteados. No entanto, várias empresas e comunidades desenvolveram seu próprio software de banco de dados relacional, e todas elas usam tabelas e se beneficiam do SQL. Algumas soluções de banco de dados são leves e simples, armazenando dados em um único arquivo que pode ser acessado por um pequeno número de usuários. Outras são pesadas e executadas em um servidor, suportando milhares de usuários e aplicativos simultaneamente. Algumas soluções são gratuitas e open source, enquanto outras requerem licenças comerciais.

Para simplificar, dividiremos as soluções de banco de dados em duas categorias: *leves* e *centralizadas*. Essa não é necessariamente a categorização usada na indústria, mas ajudará a mostrar a diferença.

### Bancos de dados leves

Se você está procurando uma solução simples para um único usuário ou um pequeno número deles (por exemplo, seus colaboradores), um banco

de dados leve é um bom ponto de partida. Os bancos de dados leves têm pouco a nenhum overhead, o que significa que eles não têm servidores e são muito ágeis. Normalmente, os bancos de dados são armazenados em um arquivo que pode ser compartilhado com outras pessoas, embora comecem a travar quando várias pessoas fazem edições no arquivo simultaneamente. Se você se deparar com esse problema, pode ser melhor migrar para um banco de dados centralizado.

Os dois bancos de dados leves mais comuns são o SQLite e o Microsoft Access. Usaremos o SQLite neste livro. Ele é gratuito, leve e fácil de usar. É empregado na maioria dos dispositivos que usamos e pode ser encontrado em smartphones, satélites e sistemas de aviões e automóveis. Praticamente não tem limite de tamanho e é ideal para ambientes em que não é usado por mais de uma pessoa (ou no máximo por algumas pessoas). Entre outras aplicações, o SQLite é ideal para o aprendizado de SQL, já que é simples e fácil de instalar.

O Microsoft Access já é usado há algum tempo e é inferior ao SQL no que se refere à escalabilidade e desempenho. Porém, é bastante usado em ambientes empresariais e vale a pena conhecê-lo. Ele tem várias ferramentas gráficas para a criação de consultas sem uso de SQL, assim como criadores gráficos de formulários e recursos de macro. Muitas ocupações se beneficiam do uso e da manutenção de bancos de dados Microsoft Access, mas muitas também podem ser transferidas para plataformas de banco de dados melhores como a do MySQL.

## Bancos de dados centralizados

Se você espera que dezenas, centenas ou milhares de usuários e aplicativos usem um banco de dados simultaneamente, os bancos de dados leves não atenderão à demanda. Será preciso um banco de dados centralizado, que seja executado em um servidor e manipule o alto volume de tráfego eficientemente. Há uma ampla variedade de soluções de bancos de dados centralizados disponíveis para escolha, inclusive os seguintes:

- MySQL
- Microsoft SQL Server

- Oracle
- PostgreSQL
- Teradata
- IBM DB2
- MariaDB

Você pode instalar algumas dessas soluções em qualquer computador e transformá-lo em um *servidor*. Em seguida, é só conectar os computadores dos usuários (também conhecidos como *clientes*) ao servidor para que eles possam acessar os dados. O cliente enviará uma instrução SQL solicitando dados específicos e o servidor processará a solicitação e retornará a resposta. Essa é uma clássica *instalação cliente-servidor*. O cliente solicita algo e o servidor fornece.

Embora seja possível transformar qualquer MacBook ou PC barato em um servidor MySQL, volumes de tráfego maiores requerem computadores mais especializados (chamados de *computadores servidores*) otimizados para tarefas de servidor. Normalmente eles são mantidos por um departamento de TI cujos membros administram e controlam os bancos de dados formalmente considerados críticos para a empresa.



Não se engane com o uso do termo “SQL” no nome de plataformas de bancos de dados como MySQL, Microsoft SQL Server e SQLite. O SQL é a linguagem universal para o trabalho com dados em todas essas plataformas. Elas usam “SQL” em seus nomes somente por questões de marketing.

Quando entramos em um local de trabalho, quase sempre há um banco de dados centralizado contendo as informações desejadas e é necessário solicitar acesso a ele. Não vamos abordar os bancos de dados centralizados neste livro, mas a experiência oferecida pelas diferentes soluções de banco de dados é, em grande, parte a mesma. Em todas as soluções, usamos SQL para interagir com as tabelas de uma maneira bastante uniforme e até mesmo as ferramentas de edição de SQL são muito semelhantes. Cada solução pode ter nuances próprias no que diz respeito à sua implementação do SQL, como nas funcionalidades de data, mas tudo que se encontra neste livro deve ser universalmente aplicável.

Se você precisar criar uma solução de banco de dados centralizado, eu recomendaria o MySQL. Ele é open source, gratuito e fácil de instalar e configurar. É usado pelo Facebook, Google, eBay, Twitter e centenas de outras empresas do Vale do Silício.

Agora que temos um conhecimento conceitual dos bancos de dados, podemos começar a trabalhar com eles. Empregaremos o SQLite neste livro, mas já que ele usa SQL, o conhecimento adquirido pode ser aplicado a todas as plataformas de banco de dados.

## CAPÍTULO 3

# SQLite

### O que é SQLite?

Como discutido no capítulo anterior, há muitos lugares onde podemos colocar dados. Porém, muitas vezes, o que precisamos é de um local em que a inserção de dados seja rápida e fácil sem toda a complexidade de uma instalação cliente-servidor. Queremos armazenar dados em um arquivo simples e editá-los tão facilmente como um documento do Word. Essa é uma situação ideal para o uso do SQLite.

O SQLite é o banco de dados mais amplamente distribuído no mundo todo. Ele está embutido em iPhones, iPads, dispositivos Android, telefones Windows, termostatos, consoles automobilísticos, satélites e muitos outros dispositivos modernos que precisam armazenar e recuperar dados facilmente. É usado pesadamente no sistema operacional Windows 10 assim como na aeronave Airbus A350 XWB. Sobressai-se em situações em que simplicidade e baixo overhead são necessários. Também é ótimo na criação de protótipos de bancos de dados empresariais.

Porém, todas as tecnologias têm suas vantagens e desvantagens. Já que não há um servidor gerenciando o acesso, ele não é adequado em ambientes multiusuário em que várias pessoas podem editar simultaneamente o arquivo SQLite. Mesmo assim, para nosso treinamento, o SQLite é perfeito.

### SQLiteStudio

Há muitos editores de SQL que você pode usar para trabalhar com um banco de dados SQLite. Recomendo o uso do SQLiteStudio porque ele é intuitivo e torna fácil explorar e gerenciar um banco de dados. É esse

aplicativo que usaremos no livro. Você pode baixá-lo de <http://sqlitestudio.pl/?act=download>. Certifique-se de selecionar Windows, Mac ou Linux como seu sistema operacional. Em seguida, abra a pasta que baixou e copie-a em um local de sua escolha. Não é necessário instalar. Para iniciar o SQLiteStudio, clique duas vezes em *SQLiteStudio.exe* (Figura 3.1). Você também pode criar um atalho em sua área de trabalho para poder iniciar facilmente o aplicativo no futuro.

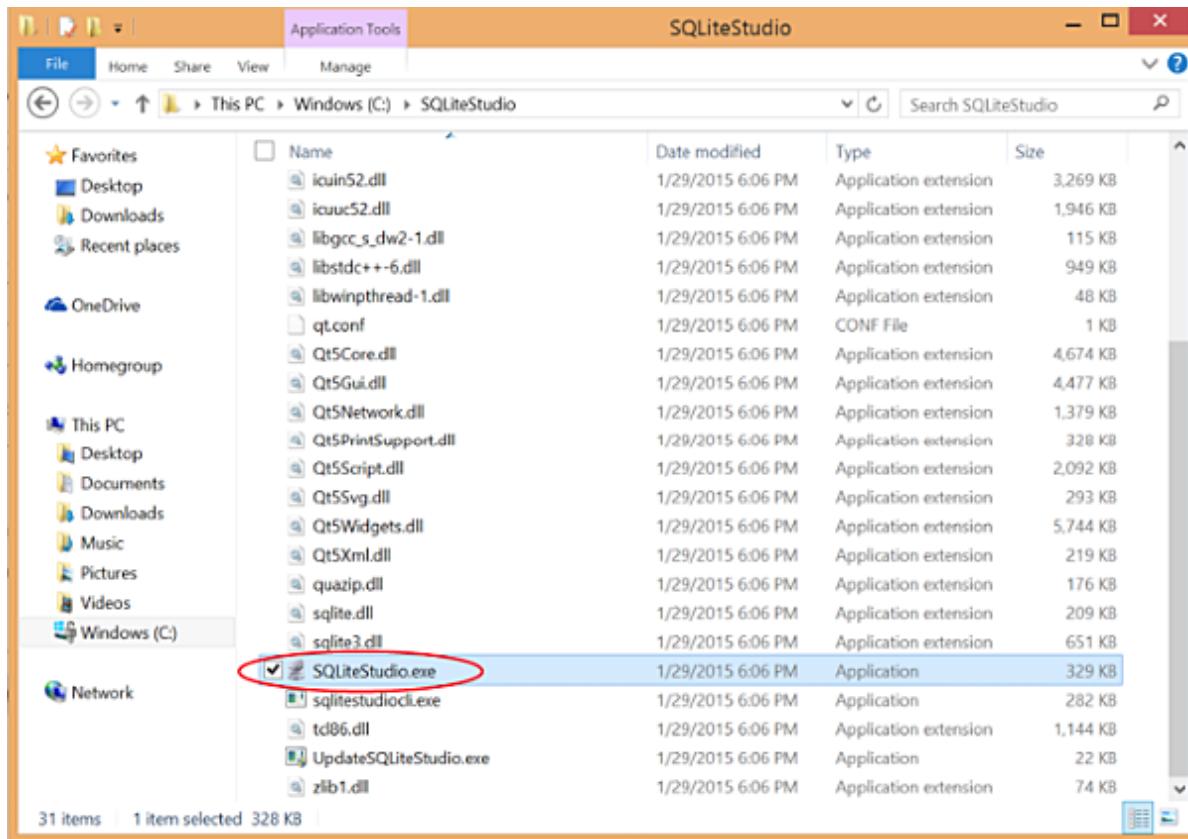


Figura 3.1 – Pasta do SQLiteStudio.

É bom ressaltar que o SQLiteStudio é um programa de terceiros não associado ao SQLite ou a seus desenvolvedores. O SQLite é um mecanismo de banco de dados construído por Richard Hipp e uma talentosa equipe de programadores. O SQLiteStudio apenas dá ao mecanismo uma interessante interface de usuário. Logo, se em algum momento você tiver problemas com o SQLiteStudio, deve entrar em contato com essa equipe e não com a do SQLite.

## Importando e procurando bancos de dados

Na primeira vez que você iniciar o SQLiteStudio, provavelmente verá um painel sem conteúdo (Figura 3.2). A janela esquerda é o navegador de bancos de dados e a área cinza à direita é a área de trabalho para SQL em que você escreverá SQL para os bancos de dados.

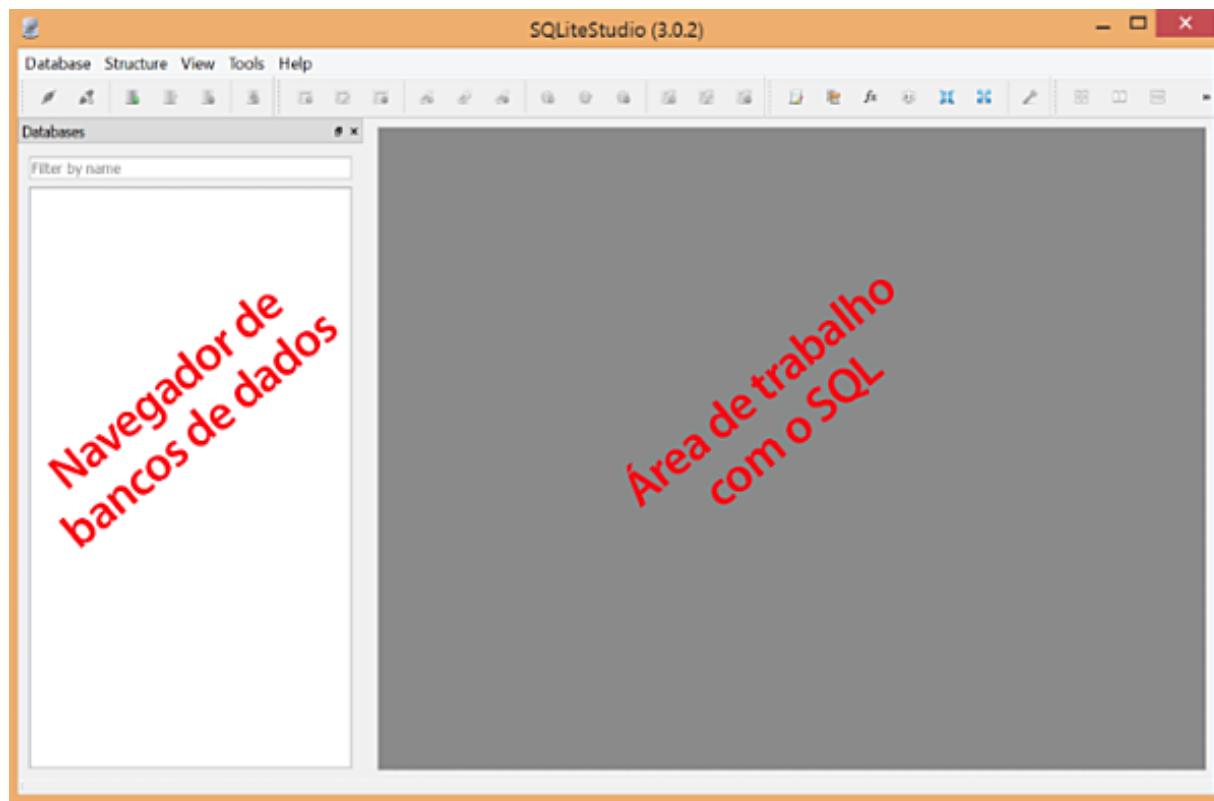


Figura 3.2 – Painel do SQLiteStudio.

Será preciso trazer bancos de dados para o SQLiteStudio. Alguns exemplos de bancos de dados SQLite usados neste livro são fornecidos em <http://bit.ly/1TLw1Gr>.

Baixe os bancos de dados clicando no botão Download ZIP e copie os conteúdos em uma pasta de sua escolha. Você deve dedicar essa pasta a todos os bancos de dados com os quais trabalhará no livro.

Após baixar os bancos de dados, acesse Database → Add a Database no menu superior (Figura 3.3).

*Figura 3.3 – Adicionando um banco de dados.*

Você verá uma caixa de diálogo solicitando um arquivo de banco de dados. Clique no ícone de pasta amarelo para selecionar um arquivo de banco de dados e importá-lo (Figura 3.4).

*Figura 3.4 – Abrindo um banco de dados.*

Procure a pasta que contém os bancos de dados salvos e clique duas vezes no arquivo de banco de dados *rexon\_metals.db* para carregá-lo no SQLiteStudio (Figura 3.5).

*Figura 3.5 – Procurando e abrindo arquivos de bancos de dados.*

Você verá que agora *rexon\_metals* foi adicionado ao navegador de bancos de dados (Figura 3.6). Clique duas vezes nele para ver seu conteúdo, que inclui três tabelas e duas views. Faça uma pausa para examinar esse banco de dados no navegador.

*Figura 3.6 – Navegando em um banco de dados.*

Repare que você pode clicar nas setas para obter informações mais detalhadas sobre diferentes objetos do banco de dados, como as tabelas (Figura 3.7). Por exemplo, clicar na seta da tabela *CUSTOMER* pode revelar informações, como as colunas que ela contém.

*Figura 3.7 – Expandido uma tabela para ver as colunas.*

Você deve estar se perguntando o que são “views”. Não é preciso se preocupar por enquanto. Basicamente, elas são consultas SQL pré-construídas que são usadas com tanta frequência que ficam convenientemente armazenadas no banco de dados.

Se você clicar duas vezes na tabela *CUSTOMER*, uma nova janela surgirá na área de trabalho contendo todos os tipos de informações sobre ela (Figura 3.8). Inicialmente, será aberta a guia *Structure*, que fornece informações detalhadas sobre cada coluna. No momento, o único detalhe com o qual

você precisa se preocupar é o tipo de dado das colunas.

Os campos `CUSTOMER_ID` e `ZIP` são armazenados como de tipo `INTEGER`, que é o tipo de dado de um número inteiro (não decimal). Isso significa que esses campos só devem conter valores `INTEGER` (inteiros). O resto das colunas é armazenado como `TEXT`. Outros tipos de dados que não são usados nessa tabela específica, como `DATETIME`, `BOOLEAN` (verdadeiro/falso) e `DECIMAL`, também poderiam ser empregados.

Por enquanto, se você conhece o conceito de tipos de dados, isso é tudo que precisa observar na guia Structure. Examinaremos o design de tabelas com detalhes quando criarmos nossas próprias tabelas posteriormente.

*Figura 3.8 – Cada coluna de uma tabela tem um tipo de dado, como inteiros ou texto.*

Clique na guia Data e verá os dados da tabela (Figura 3.9). Há apenas cinco registros (ou linhas) nessa tabela, mas o SQLite poderia conter milhões se precisasse. Você também pode editar convenientemente os valores da tabela (sem usar SQL) simplesmente clicando duas vezes em uma célula, editando-a, e então clicando na marca de seleção verde para salvá-la.

*Figura 3.9 – Tabela CUSTOMER.*

Reserve algum tempo para se familiarizar com o SQLiteStudio. Assim que estiver satisfeito com o que aprendeu, feche todas as janelas da área de trabalho. Em seguida, no menu superior, navegue para Tools → Open SQL Editor. Vimos que o SQLiteStudio fornece muitas maneiras de visualizar e manipular dados sem o uso de SQL, mas ele não chega perto da flexibilidade e do poder que o SQL oferece.

Agora que conhecemos nossas tabelas e sabemos com o que estamos trabalhando, será intuitivo escrever instruções SQL. É difícil consultar bancos de dados sem antes conhecer as tabelas que eles contêm.

# CAPÍTULO 4

## SELECT

No trabalho com bancos de dados e SQL, a tarefa mais comum é solicitar dados de uma ou mais tabelas e exibi-los. A instrução `SELECT` faz isso. No entanto, `SELECT` pode fazer muito mais que simplesmente recuperar e exibir dados. Como aprenderemos em capítulos posteriores, podemos transformar esses dados de maneira significativa e construir resumos poderosos a partir de milhões de registros.

Porém, primeiro aprenderemos a selecionar colunas de uma única tabela e a compor expressões com elas.

### Recuperando dados com SQL

Se ainda não o fez, clique em Tools → Open SQL Editor no menu superior e verifique se o banco de dados `rexon_metals` está aberto, como mencionado no capítulo anterior. Seu espaço de trabalho no SQLiteStudio deve ficar parecido com o da Figura 4.1. Observe que agora o espaço para uso do SQL está dividido em dois painéis: o painel SQL Editor e o painel Query Results.

O painel SQL Editor é onde você escreverá suas instruções SQL e o painel Query Results exibirá os resultados das instruções.

Escreveremos nossa primeira instrução SQL. A operação SQL mais comum é a instrução `SELECT`, que extrai dados de uma tabela e exibe os resultados. Clique no painel SQL Editor e escreva a instrução a seguir:

```
SELECT * FROM CUSTOMER;
```

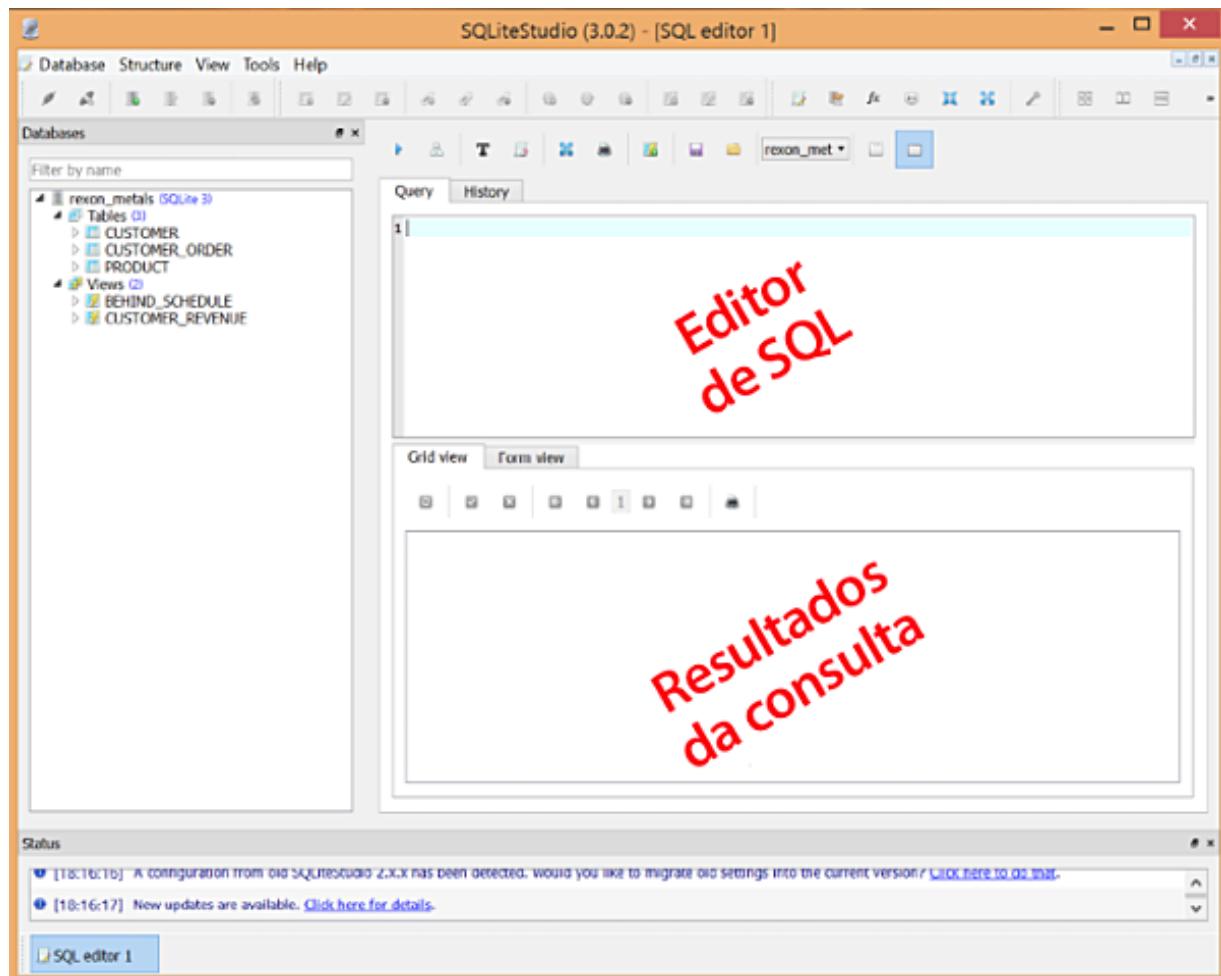


Figura 4.1 – Espaço de trabalho para uso do SQL.

Clique no botão de triângulo azul ou pressione F9 para executar a instrução SQL.

Você acabou de executar sua primeira consulta e os resultados devem ser exibidos no painel inferior (Figura 4.2).

Vejamos detalhadamente o que ocorreu. Uma instrução SELECT permite selecionar que colunas serão acessadas em uma tabela. Logo, a primeira parte da instrução SQL mostrada aqui poderia ser entendida como “Selecione todas as colunas”, em que o asterisco (\*) é um placeholder que especifica todas as colunas:

```
SELECT * FROM CUSTOMER;
```

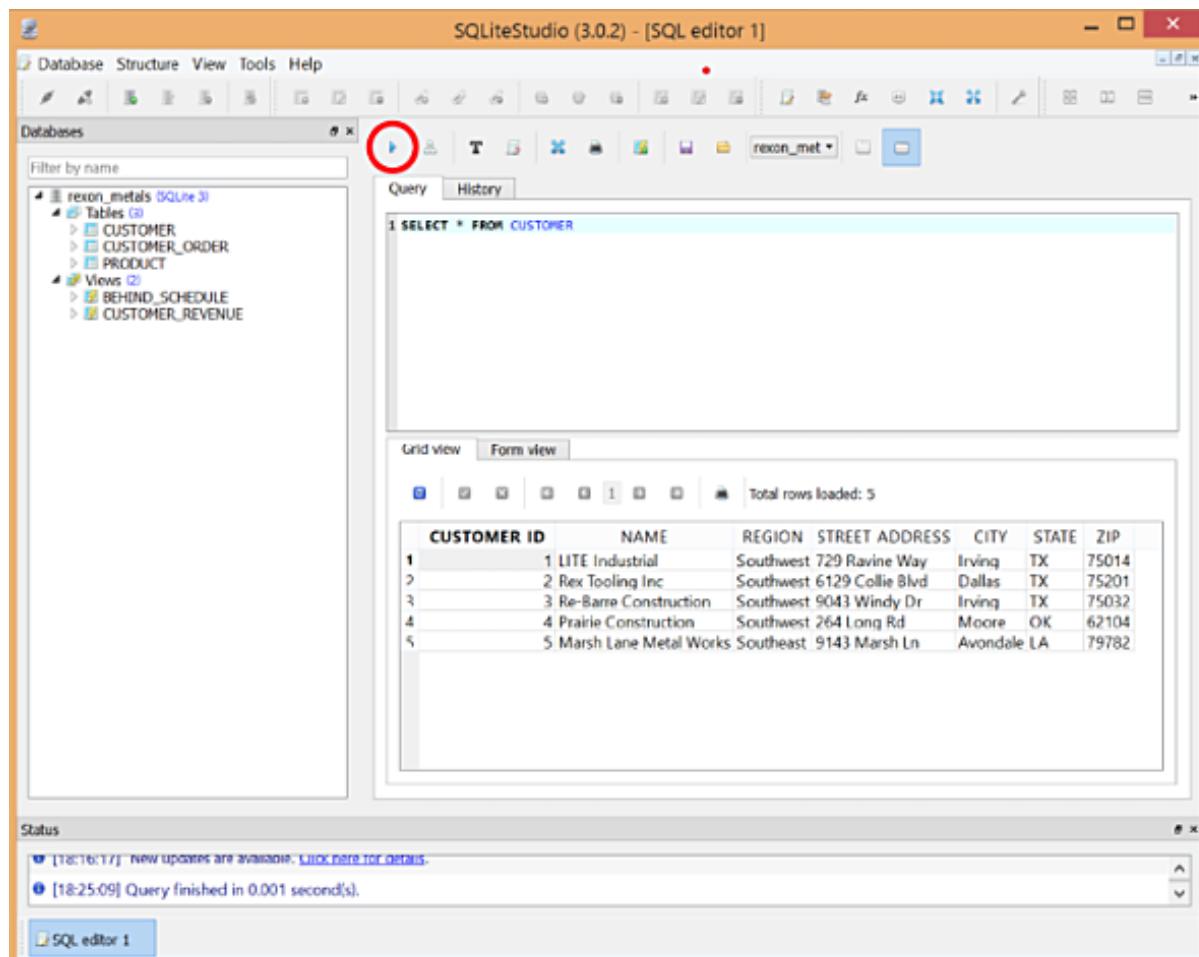


Figura 4.2 – Executando uma consulta SELECT.

E as colunas que estão sendo acessadas são da tabela CUSTOMER:

```
SELECT * FROM CUSTOMER;
```

Quando essa instrução SELECT for executada, ela retornará todas as colunas da tabela CUSTOMER e as exibirá (Figura 4.3).

The screenshot shows the results grid from Figure 4.3. The table has the same structure as in Figure 4.2, displaying five rows of customer data. The first column is labeled 'CUSTOMER ID' and the last column is 'ZIP'.

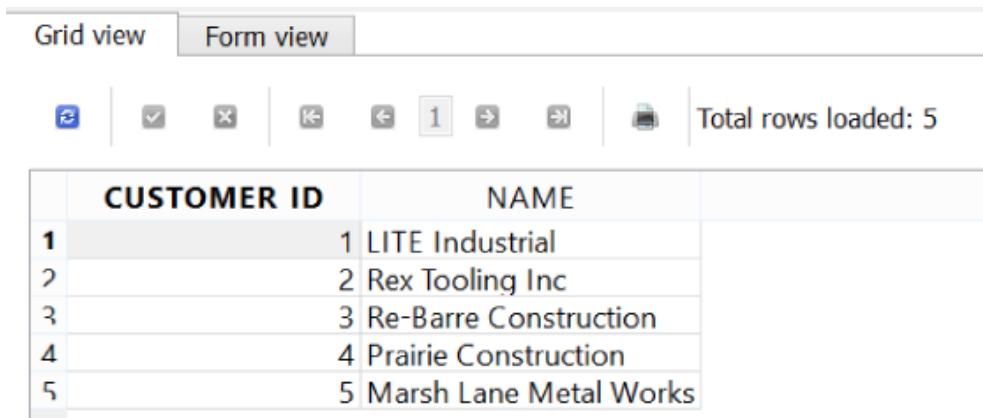
CUSTOMER ID	NAME	REGION	STREET ADDRESS	CITY	STATE	ZIP
1	LITE Industrial	Southwest	729 Ravine Way	Irving	TX	75014
2	Rex Tooling Inc	Southwest	6129 Collie Blvd	Dallas	TX	75201
3	Re-Barre Construction	Southwest	9043 Windy Dr	Irving	TX	75032
4	Prairie Construction	Southwest	264 Long Rd	Moore	OK	62104
5	Marsh Lane Metal Works	Southeast	9143 Marsh Ln	Avondale	LA	79782

Figura 4.3 – Selecionando todos os registros da tabela CUSTOMER.

Não precisamos acessar todas as colunas em uma instrução SELECT. Também podemos selecionar apenas as colunas em que estivermos interessados. A consulta a seguir só acessará as colunas CUSTOMER\_ID e NAME:

```
SELECT CUSTOMER_ID, NAME FROM CUSTOMER;
```

E a saída só exibirá essas duas colunas (Figura 4.4).



CUSTOMER ID	NAME
1	LITE Industrial
2	Rex Tooling Inc
3	Re-Barre Construction
4	Prairie Construction
5	Marsh Lane Metal Works

Figura 4.4 – Selecionando apenas duas colunas de uma tabela.



Opcionalmente, uma única instrução SQL pode terminar com ponto e vírgula (;), como mostrado nos exemplos anteriores. No entanto, o símbolo de ponto e vírgula é necessário na execução de várias instruções SQL de uma só vez, o que é útil para gravar dados, como abordado no Capítulo 10.

Selecionar colunas pode não parecer interessante no momento, mas nos permite direcionar nossa atenção para o que queremos. A redução do escopo a apenas certas colunas também ajudará nas tarefas de agregação realizadas por GROUP BY, como veremos no Capítulo 6.

## Expressões em instruções SELECT

A instrução SELECT pode fazer muito mais que apenas selecionar colunas. Você também pode efetuar cálculos em uma ou mais colunas e incluí-los no resultado de sua consulta.

Trabalharemos com outra tabela, chamada PRODUCT. Primeiro, execute uma operação “selecionar tudo” para ver os dados (Figura 4.5):

```
SELECT * FROM PRODUCT;
```

Grid view   Form view

	PRODUCT ID	DESCRIPTION	PRICE
1	1	Copper	7.51
2	2	Aluminum	2.58
3	3	Silver	15
4	4	Steel	12.31
5	5	Bronze	4
6	6	Duralumin	7.6
7	7	Solder	14.16
8	8	Stellite	13.31
9	9	Brass	4.75

Figura 4.5 – Tabela PRODUCT.

Suponhamos que quiséssemos gerar uma coluna calculada chamada TAXED\_PRICE em que o preço fosse 7% mais alto que em PRICE. Poderíamos usar uma consulta SELECT para fazer o cálculo dinamicamente para nós (Figura 4.6):

```

SELECT
  PRODUCT_ID,
  DESCRIPTION,
  PRICE,
  PRICE * 1.07 AS TAXED_PRICE
FROM PRODUCT;
  
```

Grid view   Form view

	PRODUCT ID	DESCRIPTION	PRICE	TAXED PRICE
1	1	Copper	7.51	8.0357
2	2	Aluminum	2.58	2.7606
3	3	Silver	15	16.05
4	4	Steel	12.31	13.1717
5	5	Bronze	4	4.28
6	6	Duralumin	7.6	8.132
7	7	Solder	14.16	15.1512
8	8	Stellite	13.31	14.2417
9	9	Brass	4.75	5.0825

*Figura 4.6 – Usando expressões para calcular uma coluna TAXED\_PRICE.*



Essa instrução `SELECT` demonstra que podemos distribuir o SQL em várias linhas para torná-lo mais legível. O software ignora espaços em branco irrelevantes e linhas separadas, logo, podemos usá-los para facilitar a leitura.

Observe como a coluna `TAXED_PRICE` foi calculada dinamicamente na consulta `SELECT`. Essa coluna não é armazenada na tabela; em vez disso, é calculada e exibida sempre que executamos a consulta. Esse é um recurso poderoso do SQL, que nos permite armazenar os dados em um nível mais simples e usar consultas para fazer cálculos com eles.

Examinemos a coluna `TAXED_PRICE` para detalhar como foi criada. Primeiro, vemos que `PRICE` é multiplicado por 1,07 para o cálculo da quantia acrescida da taxa. Geramos o valor de `TAXED_PRICE` para cada registro:

```
SELECT  
PRODUCT_ID,  
DESCRIPTION,  
PRICE,  
PRICE * 1.07 AS TAXED_PRICE  
FROM PRODUCT
```

Observe também que demos um nome ao valor calculado usando uma instrução `AS` (esse nome é conhecido como *alias*):

```
SELECT  
PRODUCT_ID,  
DESCRIPTION,  
PRICE,  
PRICE * 1.07 AS TAXED_PRICE  
FROM PRODUCT
```

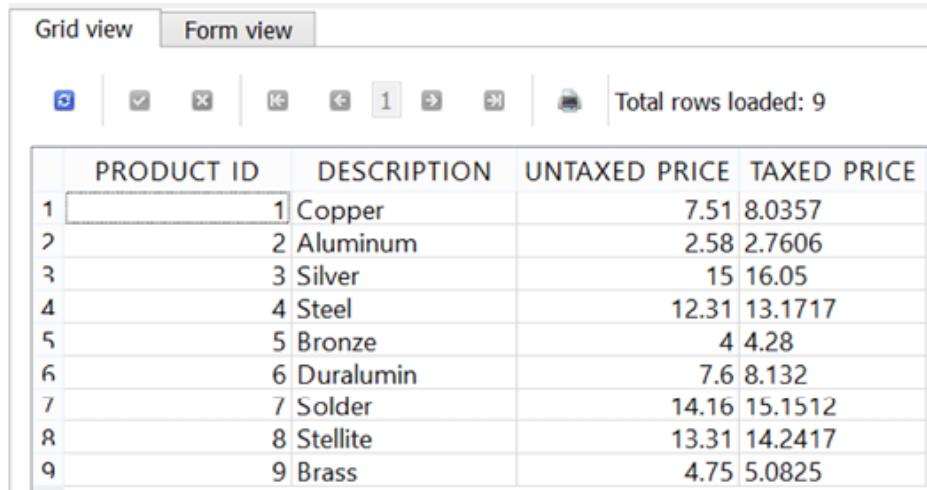
Podemos usar aliases para dar nomes a expressões. Também podemos usá-los para aplicar um novo nome a uma coluna existente dentro da consulta. Por exemplo, poderíamos aplicar o alias `UNTAXED_PRICE` à coluna `PRICE` (Figura 4.7). Isso não muda o nome da coluna na tabela, mas dá a ela um novo nome dentro do escopo da instrução `SELECT`:

```
SELECT  
PRODUCT_ID,  
DESCRIPTION,  
PRICE AS UNTAXED_PRICE,
```

```

PRICE * 1.07 AS TAXED_PRICE
FROM PRODUCT

```



The screenshot shows a SQLite database interface with a grid view of products. The columns are PRODUCT\_ID, DESCRIPTION, UNTAXED PRICE, and TAXED PRICE. The TAXED PRICE column contains values like 8.0357, 2.7606, etc., which are rounded to two decimal places.

	PRODUCT ID	DESCRIPTION	UNTAXED PRICE	TAXED PRICE
1	1	Copper	7.51	8.0357
2	2	Aluminum	2.58	2.7606
3	3	Silver	15	16.05
4	4	Steel	12.31	13.1717
5	5	Bronze	4	4.28
6	6	Duralumin	7.6	8.132
7	7	Solder	14.16	15.1512
8	8	Stellite	13.31	14.2417
9	9	Brass	4.75	5.0825

Figura 4.7 – Fornecendo o alias UNTAXED\_PRICE para a coluna PRICE.



Ao dar nomes para qualquer coisa em SQL (seja um alias, um nome de coluna, um nome de tabela ou o nome de qualquer outra entidade), use sempre um underscore (\_) como placeholder para espaços. Caso contrário, causará erros.

Se quiséssemos distribuir os resultados dessa instrução SQL como um relatório em nosso local de trabalho, seria aconselhável fixar um arredondamento em TAXED\_PRICE. Pode não ser interessante haver mais de duas casas decimais. Todas as plataformas de banco de dados têm *funções* internas que ajudam nesses tipos de operações e o SQLite fornece uma função `round()` que aceita dois argumentos em parênteses separados por uma vírgula: o valor a ser arredondado e o número de casas decimais do arredondamento. Para arredondar TAXED\_PRICE para duas casas decimais, podemos passar a expressão de multiplicação `PRICE * 1.07` como o primeiro argumento e 2 como o segundo:

```

SELECT
PRODUCT_ID,
DESCRIPTION,
PRICE,
round(PRICE * 1.07, 2) AS TAXED_PRICE
FROM PRODUCT;

```

Execute a instrução e verá que ela está arredondando TAXED\_PRICE, que fica com uma aparência muito melhor com duas casas decimais (Figura 4.8).

	PRODUCT ID	DESCRIPTION	UNNTAXED PRICE	TAXED PRICE
1	1	Copper	7.51	8.04
2	2	Aluminum	2.58	2.76
3	3	Silver	15	16.05
4	4	Steel	12.31	13.17
5	5	Bronze	4	4.28
6	6	Duralumin	7.6	8.13
7	7	Solder	14.16	15.15
8	8	Stellite	13.31	14.24
9	9	Brass	4.75	5.08

Figura 4.8 – Usando a função `round()` para limitar as casas decimais de `TAXED_PRICE`.

Aqui está um pequeno resumo dos operadores matemáticos que você pode usar em SQL (veremos como são empregados no decorrer do livro):

Operador	Descrição	Exemplo
+	Soma dois números	STOCK + NEW_SHIPMENT
-	Subtrai dois números	STOCK - DEFECTS
*	Multiplica dois números	PRICE * 1.07
/	Divide dois números	STOCK / PALLET_SIZE
%	Divide dois números, mas retorna o resto	STOCK % PALLET_SIZE

## Concatenação de texto

As expressões não precisam ser usadas somente com números. Você também pode usar expressões com texto e outros tipos de dados. Um operador útil quando usado com texto é o de *concatenação*, que mescla dois ou mais dados. O operador de concatenação é especificado por um pipe duplo (`||`); os valores de dados a serem concatenados devem ser colocados nos dois lados do pipe duplo.

Por exemplo, você pode concatenar os campos `CITY` e `STATE` da tabela

CUSTOMER e inserir uma vírgula e um espaço entre eles para criar um valor LOCATION (Figura 4.9):

```
SELECT NAME,  
CITY || ', ' || STATE AS LOCATION  
FROM CUSTOMER;
```

	NAME	LOCATION
1	LITE Industrial	Irving, TX
2	Rex Tooling Inc	Dallas, TX
3	Re-Barre Construction	Irving, TX
4	Prairie Construction	Moore, OK
5	Marsh Lane Metal Works	Avondale, LA

Figura 4.9 – Concatenando CITY e STATE.

Você pode até mesmo concatenar vários campos em um único valor SHIP\_ADDRESS (Figura 4.10):

```
SELECT NAME,  
STREET_ADDRESS || ' ' || CITY || ', ' || STATE || ' ' || ZIP AS SHIP_ADDRESS  
FROM CUSTOMER;
```

NAME	SHIP_ADDRESS
LITE Industrial	729 Ravine Way Irving, TX 75014
Rex Tooling Inc	6129 Collie Blvd Dallas, TX 75201
Re-Barre Construction	9043 Windy Dr Irving, TX 75032
Prairie Construction	264 Long Rd Moore, OK 62104
Marsh Lane Metal Works	9143 Marsh Ln Avondale, LA 79782

Figura 4.10 – Concatenando vários campos para criar SHIP\_ADDRESS.

A concatenação deve funcionar com qualquer tipo de dado (números, datas, etc.), que é tratado como texto após a mesclagem. O campo ZIP mostrado aqui é um número, mas foi implicitamente convertido em texto durante a concatenação.

Outras operações de texto serão abordadas no próximo capítulo, mas a concatenação é definitivamente uma das mais importantes.



Muitas plataformas de banco de dados usam pipes duplos (||) na concatenação, exceto o MySQL e algumas outras que requerem o uso de uma função CONCAT().

## Resumo

Neste capítulo vimos como usar a instrução `SELECT`, a operação SQL mais comum. Ela recupera e transforma os dados de uma tabela sem afetar a tabela propriamente dita. Também aprendemos a selecionar colunas e escrever expressões. Dentro de expressões, podemos usar operadores e funções para executar tarefas como as de arredondamento, cálculo matemático e concatenação.

No próximo capítulo conheceremos a instrução `WHERE`, que permite filtrar registros de acordo com critérios especificados.

# capítulo 5

## WHERE

Nos próximos capítulos adicionaremos mais funcionalidades à instrução SELECT. No trabalho com dados, uma tarefa muito comum é a filtragem de registros de acordo com critérios, o que pode ser feito com uma instrução WHERE.

Conheceremos mais funções e aprenderemos como usá-las aplicando-as à cláusula WHERE, mas também podemos usá-las nas instruções SELECT, como discutido no capítulo anterior. Geralmente as expressões e funções podem ser usadas em qualquer parte de uma instrução SQL.

### Filtrando registros

Abriremos outro banco de dados, chamado weather\_stations. Você deve adicioná-lo ao seu navegador de bancos de dados (consulte o Capítulo 3 se esqueceu como se faz). Clique duas vezes no banco de dados e verá que há uma única tabela chamada STATION\_DATA. Ela contém amostragens de dados relacionados ao clima coletadas em várias estações meteorológicas.

Execute SELECT em todas as colunas para ver os dados que elas contêm:

```
SELECT * FROM station_data;
```

Há um grande volume de dados aqui: cerca de 28.000 registros (Figura 5.1). Não coletaremos muitas informações interessantes rolando pelos registros um a um. Precisamos conhecer mais alguns recursos SQL para transformar os dados em algo significativo. Começaremos examinando a instrução WHERE, que pode ser usada na filtragem de recursos de acordo com um critério.

Figura 5.1 – Banco de dados weather\_stations.

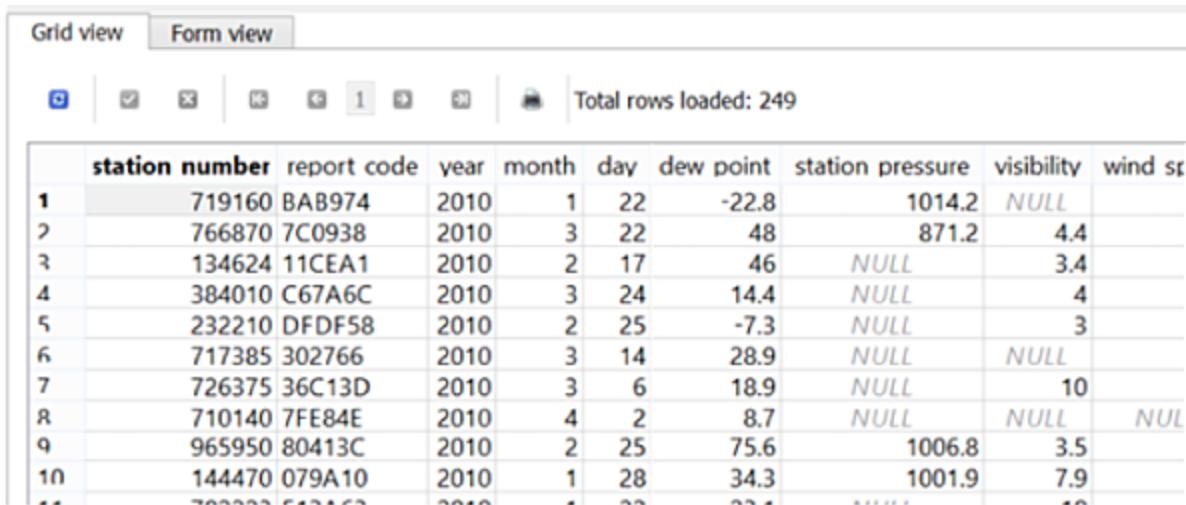
Os nomes e as colunas das tabelas podem ser definidos em letras maiúsculas ou minúsculas. Comandos SQL como SELECT, FROM e WHERE também podem estar em maiúsculas ou minúsculas.

## Usando WHERE com números

Suponhamos que só estivéssemos interessados nos registros de station\_data do ano 2010. É muito fácil usar WHERE com um critério simples como esse. Com essa consulta, você só obterá registros em que o campo year for igual a 2010 (Figura 5.2):

```
SELECT * FROM station_data
```

```
WHERE year = 2010;
```



The screenshot shows a database grid interface with two tabs at the top: "Grid view" (selected) and "Form view". Below the tabs is a toolbar with various icons. To the right of the toolbar, it says "Total rows loaded: 249". The main area displays a table with 11 columns: station number, report code, year, month, day, dew point, station pressure, visibility, wind, and sr. The "year" column contains the value 2010 for all rows. The table has 11 rows, indexed from 1 to 10. Row 11 is indicated by three dots. The data includes various station identifiers like 719160, 766870, etc., and weather values like -22.8, 48, 1014.2, etc.

station number	report code	year	month	day	dew point	station pressure	visibility	wind	sr
1	719160 BAB974	2010	1	22	-22.8	1014.2	NULL		
2	766870 7C0938	2010	3	22	48	871.2	4.4		
3	134624 11CEA1	2010	2	17	46	NULL		3.4	
4	384010 C67A6C	2010	3	24	14.4	NULL		4	
5	232210 DFDF58	2010	2	25	-7.3	NULL		3	
6	717385 302766	2010	3	14	28.9	NULL	NULL		
7	726375 36C13D	2010	3	6	18.9	NULL		10	
8	710140 7FE84E	2010	4	2	8.7	NULL	NULL		NUL
9	965950 80413C	2010	2	25	75.6	1006.8	3.5		
10	144470 079A10	2010	1	28	34.3	1001.9	7.9		
..	.....	....	..	..	..	.....	.....	..	..

Figura 5.2 – Registros do ano 2010.

Inversamente, você pode usar != ou <> para obter tudo, exceto 2010. Por exemplo:

```
SELECT * FROM station_data
```

```
WHERE year != 2010
```

Ou:

```
SELECT * FROM station_data
```

```
WHERE year <> 2010
```

Essas duas sintaxes fazem a mesma coisa. Atualmente o SQLite e a maioria das plataformas suportam ambas. No entanto, o Microsoft Access e o IBM DB2 só dão suporte a `<>`.

Também podemos qualificar intervalos inclusivos usando uma instrução `BETWEEN`, como mostrado aqui (“inclusivo” significa que 2005 e 2010 estão incluídos no intervalo):

```
SELECT * FROM station_data
```

```
WHERE year BETWEEN 2005 and 2010
```

## Instruções AND, OR e IN

Uma instrução `BETWEEN` pode ser expressa alternativamente com o uso das expressões maior ou igual a e menor ou igual a e uma instrução `AND`. É um pouco mais verboso, mas demonstra que podemos usar duas condições com a instrução `AND`. No caso em questão, o ano deve ser maior ou igual a 2005 e menor ou igual a 2010:

```
SELECT * FROM station_data
```

```
WHERE year >= 2005 AND year <= 2010
```

Se quiséssemos todos os dados entre 2005 e 2010 exclusivamente – isto é, não incluindo esses dois anos – bastaria remover os caracteres `=`. Só 2006, 2007, 2008 e 2009 se qualificariam:

```
SELECT * FROM station_data
```

```
WHERE year > 2005 AND year < 2010
```

Também temos a opção de usar OR. Em uma instrução OR, pelo menos um dos critérios deve ser atendido para o registro se qualificar. Se só quiséssemos registros dos meses 3, 6, 9 ou 12, poderíamos usar OR para obtê-los:

```
SELECT * FROM station_data
```

```
WHERE MONTH = 3
```

```
OR MONTH = 6
```

```
OR MONTH = 9
```

```
OR MONTH = 12
```

Ficou um pouco verboso. Uma maneira mais eficiente de fazer isso é usando uma instrução IN que forneça uma lista válida de valores:

```
SELECT * FROM station_data
```

```
WHERE MONTH IN (3,6,9,12)
```

Se quiséssemos todos os dados exceto os relativos aos meses 3, 6, 9 e 12, poderíamos usar NOT IN:

```
SELECT * FROM station_data
```

```
WHERE MONTH NOT IN (3,6,9,12)
```

Você também pode usar outras expressões matemáticas em suas instruções WHERE. Anteriormente, tentamos fazer a filtragem para obter dados dos meses 3, 6, 9 e 12. Se quiséssemos só meses divisíveis por 3 (os meses “do fim de um trimestre”), poderíamos usar o operador de módulo (%). Esse operador é semelhante ao operador de divisão (/), mas retorna o resto em vez do quociente. Um resto igual a 0 significa que não há resto, logo, você pode se beneficiar dessa lógica procurando um resto igual a 0 com o módulo 3.

Em português claro e simples, isso significa “me dê todos os meses em que a divisão por 3 forneça 0 de resto”:

```
SELECT * FROM station_data
```

```
WHERE MONTH % 3 = 0
```



O Oracle não dá suporte ao operador de módulo. Em vez disso, ele usa a função MOD().

## Usando WHERE com texto

Abordamos vários exemplos de como qualificar campos numéricos em instruções WHERE. As regras para a qualificação de campos de texto seguem a mesma estrutura, embora haja diferenças sutis. Você pode usar instruções =, AND, OR e IN com texto. No entanto, ao usar texto, é preciso inserir os literais (ou os valores textuais que você especificar) em aspas simples. Por exemplo, se você quisesse filtrar um report\_code (código de relatório) específico, poderia executar essa consulta:

```
SELECT * FROM station_data
```

```
WHERE report_code = '513A63'
```

Observe que, já que o campo report\_code é de texto (e não numérico), precisamos inserir '513A63' em aspas simples para qualificá-lo. Se não o fizermos, o software SQL ficará confuso e pensará que 513A63 é uma coluna em vez de um valor de texto. Isso causará erro e a consulta falhará.

A regra das aspas simples se aplica a todas as operações de texto, inclusive a essa operação IN:

```
SELECT * FROM station_data
```

```
WHERE report_code IN ('513A63','1F8A7B','EF616A')
```

Há outras operações e funções de texto úteis que você pode usar em instruções WHERE e SELECT. Por exemplo, a função length() conta o número de caracteres de um valor específico. Logo, se fôssemos responsáveis pelo controle de qualidade e precisássemos assegurar que todos os report\_codes tivessem seis caracteres, precisaríamos ter certeza de que nenhum registro fosse retornado pela execução dessa consulta:

```
SELECT * FROM station_data  
WHERE length(report_code) != 6
```

Outra operação comum é o uso de curingas em uma expressão LIKE, onde % significa qualquer número de caracteres e \_ um único caractere. Os outros caracteres são interpretados literalmente. Então, se você quisesse encontrar todos os códigos de relatório que começassem com a letra “A”, executaria essa instrução para encontrar a letra “A” seguida por quaisquer que fossem os outros caracteres:

```
SELECT * FROM station_data  
WHERE report_code LIKE 'A%'
```

Se quisesse encontrar todos os códigos de relatório que tivessem um “B” como o primeiro caractere e um “C” como o segundo, precisaria especificar um underscore (\_) para a segunda posição e qualquer número de caracteres após o “C”:

```
SELECT * FROM station_data  
WHERE report_code LIKE 'B_C%'
```

Não se confunda pelo uso do símbolo % para duas finalidades diferentes. Anteriormente o usamos para executar uma operação de módulo, mas em uma instrução LIKE ele é um curinga em um padrão de texto. Como ocorre com outros símbolos e caracteres em SQL, o contexto em que são usados define sua funcionalidade.

Há muitas outras funções de texto úteis, como INSTR, SUBSTR e REPLACE. Para simplificar, não abordaremos mais funções de texto aqui, mas você pode consultar o “Apêndice A6 – Funções básicas comuns”, e o “Apêndice A8 – Funções de data e hora”, para saber mais sobre essas funções.

Funções de texto como LIKE, SUBSTR e INSTR podem se tornar tediosas e verbosas ao qualificar padrões de texto complexos. Recomendo que você tente usar expressões regulares se perceber isso. Elas não são material para iniciantes, mas serão úteis se você precisar desse suporte intermediário.

## Usando WHERE com booleanos

Booleanos são valores de tipo verdadeiro/falso. No universo dos bancos de dados, normalmente falso é expresso como 0 e verdadeiro como 1.

Algumas plataformas de banco de dados (como a do MySQL) permitem o uso implícito das palavras true e false na qualificação, como mostrado aqui:

```
SELECT * FROM station_data
```

```
WHERE tornado = true AND hail = true;
```

O SQLite, no entanto, não permite isso. Ele demanda o uso explícito de 1 para verdadeiro e 0 para falso. Se você quisesse todos os registros em que aparecessem tornado e granizo, executaria essa instrução:

```
SELECT * FROM station_data
```

```
WHERE tornado = 1 AND hail = 1;
```

Se quisesse apenas valores verdadeiros, não precisaria nem mesmo usar a expressão = 1. Como os campos são booleanos (no fim das contas, todas as condições WHERE se resumem a uma expressão booleana), inherentemente eles já se qualificam. Logo, você pode obter os mesmos resultados executando a consulta a seguir:

```
SELECT * FROM station_data  
WHERE tornado AND hail;
```

Porém, a qualificação de condições falsas precisa ser explícita. Para obter todos os registros sem tornado e com granizo, execute essa consulta:

```
SELECT * FROM station_data  
WHERE tornado = 0 AND hail = 1;
```

Você também pode usar a palavra-chave NOT para qualificar tornado como falso:

```
SELECT * FROM station_data  
WHERE NOT tornado AND hail;
```

## Manipulando NULL

Você deve ter notado que algumas colunas, como station\_pressure e snow\_depth, têm valores null (Figura 5.3). O valor nulo é aquele que não apresenta valor. É a ausência completa de qualquer conteúdo. Trata-se de um estado de vacuidade. Em linguagem coloquial, seria um valor “em branco”.

Figura 5.3 – A tabela station\_data tem valores NULL.

Os valores nulos não podem ser determinados com `=`. Você precisa usar as instruções IS NULL ou IS NOT NULL para identificar valores nulos. Portanto, para obter todos os registros sem definição de profundidade da neve (snow\_depth), poderíamos executar essa consulta:

```
SELECT * FROM station_data  
WHERE snow_depth IS NULL;
```

Com frequência, valores nulos são indesejáveis. A coluna station\_number deve ser projetada para nunca permitir nulos ou poderíamos ter dados órfãos não pertencentes a nenhuma estação. No entanto, pode fazer sentido existirem valores nulos em snow\_depth ou precipitation, não porque fez um dia de sol (nesse caso, é melhor registrar os valores como 0), mas porque algumas estações podem não ter os instrumentos necessários para obter essas medidas. Configurar esses valores com 0 (que indica que dados foram registrados) pode levar a resultados incorretos, logo, eles devem ser deixados como nulos.

Isso mostra que os nulos podem ser ambíguos e que pode ser difícil determinar seu significado para a empresa. É importante que colunas anuláveis (colunas que podem ter valores nulos) documentem o que um valor nulo significa do ponto de vista empresarial. Caso contrário, os nulos devem ser banidos dessas colunas nas tabelas.

Não confunda os valores nulos com texto vazio, que é indicado por duas aspas simples sem nada entre elas (""). Isso também se aplica a texto de espaço em branco (' '). Esses casos são tratados como valores e nunca são considerados nulos. Um valor nulo definitivamente também não é o mesmo que 0, já que 0 é um valor e nulo é a ausência de valor.

Os valores nulos podem ser muito difíceis de manipular na composição de instruções WHERE. Se você quisesse consultar todos os registros em que precipitation fosse menor que 0,5, poderia escrever essa instrução:

```
SELECT * FROM station_data
```

```
WHERE precipitation <= 0.5;
```

Porém, considerou os valores nulos? E se quisesse que nulos fossem incluídos nessa consulta? Já que nulo não é 0 ou algum outro número, ele não se qualificará. Raramente os nulos se qualificam como alguma coisa e quase sempre são deixados de fora na filtragem de uma instrução WHERE a menos que sejam manipulados explicitamente. Logo, é preciso usar OR para incluir nulos:

```
SELECT * FROM station_data  
WHERE precipitation IS NULL OR precipitation <= 0.5;
```

Uma maneira mais elegante de manipular valores nulos é usando a função coalesce(), que converte um valor possivelmente nulo em um valor-padrão especificado se ele for realmente nulo. Caso contrário, ela deixa o valor no estado em que se encontra. O primeiro argumento é o valor possivelmente nulo e o segundo é o valor a ser usado se ele for realmente nulo. Dessa forma, se quiséssemos que todos os nulos fossem tratados como 0 dentro de nossa condição, por intermédio de coalesce() poderíamos fazer o campo precipitation converter nulo em 0:

```
SELECT * FROM station_data  
WHERE coalesce(precipitation, 0) <= 0.5;
```

Como qualquer função, coalesce() também pode ser usada na instrução SELECT e não apenas em WHERE. Ela será útil se você quiser melhorar a aparência de um relatório sem exibir valores nulos e exibindo algum placeholder – por exemplo, 0, “N/A” ou “None” – que seja mais significativo para as pessoas:

```
SELECT report_code, coalesce(precipitation, 0) as rainfall  
FROM station_data;
```

## Agrupando condições

Quando você começar a encadear instruções AND e OR, agrupe-as cuidadosamente. É preciso se certificar de organizar cada conjunto de condições existente entre cada instrução OR de uma maneira que condições relacionadas fiquem agrupadas. Digamos que você estivesse procurando condições de chuva com neve ou apenas de neve. Para a primeira condição ocorrer, é preciso haver chuva e uma temperatura menor ou igual a 32 graus Fahrenheit. Você pode procurar essa condição de chuva com neve ou de profundidade da neve maior que 0, como mostrado aqui:

```
SELECT * FROM station_data  
WHERE rain = 1 AND temperature <= 32  
OR snow_depth > 0;
```

No entanto, há um problema aqui. Embora tecnicamente essa operação funcione, há alguma ambiguidade e tivemos sorte de o SQLite interpretá-la de maneira correta. Isso ocorre porque a resposta à pergunta “Que condições pertencem a AND e quais pertencem a OR?” não está clara. O interpretador de SQL poderia se confundir e interpretar de maneira incorreta que estamos procurando chuva E (AND) outra condição em que a temperatura esteja abaixo de 32 OU (OR) a profundidade da neve seja maior que 0. A semântica não está clara, e em uma consulta SQL mais complicada, poderia confundir não só as pessoas como também a máquina.

É por isso que é melhor agrupar as condições explicitamente em parênteses:

```
SELECT * FROM station_data  
WHERE (rain = 1 AND temperature <= 32)  
OR snow_depth > 0
```

Nesse caso, agrupamos a expressão que representa chuva com neve dentro de parênteses para que ela seja avaliada unitariamente e temperature não seja considerada pertencente ao operador OR e erroneamente associada a snow\_depth. O agrupamento com parênteses em instruções WHERE torna a semântica mais clara e a execução mais segura. Isso lembra a ordem das operações (PEMDAS) que aprendemos nas aulas de matemática na época escolar. Qualquer coisa que estiver em parênteses é calculada antes. Quando você começar a escrever condições WHERE complicadas, essa prática será ainda mais crítica.

## Resumo

Neste capítulo aprendemos a filtrar os registros de uma instrução SELECT eficientemente usando uma cláusula WHERE. Também conhecemos novas funções e operadores de expressões que podem ser usados em quase qualquer parte da instrução SQL. Para concluir, examinamos como encadear de maneira cuidadosa e segura várias condições na mesma instrução WHERE.

Espero que o SQL já esteja mostrando sua utilidade. Você pode filtrar dados de maneira rápida e fácil em condições muito explícitas, em um nível difícil de atingir com ferramentas corriqueiras como o Excel.

Apesar de termos abordado bastante material, isso é só o começo. O próximo capítulo examinará a agregação de dados, que adiciona ainda mais valor ao repertório do SQL. Uma coisa é limitar os registros e fazer a filtragem de acordo com critérios específicos. Outra é converter milhões de registros em apenas alguns que resumam os dados.

Abordamos algumas funções neste capítulo, mas há muitas outras que você pode pesquisar e usar conforme necessário. Veremos mais algumas no decorrer do livro. O “Apêndice A6 – funções básicas comuns”, e o “Apêndice A8 – funções de data e hora”, examinam várias delas, e você pode ver uma lista completa de funções do SQLite em [https://www.sqlite.org/lang\\_corefunc.html](https://www.sqlite.org/lang_corefunc.html).

# CAPÍTULO 6

## GROUP BY e ORDER BY

A *agregação* de dados (também chamada de *totalização*, *resumo* ou *agrupamento de dados*) é a criação de algum tipo de total a partir de vários registros. Soma, mínimo, máximo, contagem e média são operações de agregação comuns. Em SQL, você pode agrupar esses totais em qualquer coluna especificada, o que permite controlar facilmente o escopo das agregações.

### Agrupando registros

Primeiro, executaremos a operação de agregação mais simples: contar o número de registros de uma tabela. Abra o editor de SQL e obtenha a contagem de registros de `station_data`:

```
SELECT COUNT(*) AS record_count FROM station_data;
```

`COUNT(*)` significa contar os registros. Também podemos usar essa função junto com outras operações SQL, como `WHERE`. Para contar o número de registros em que um tornado estava presente, digite o seguinte:

```
SELECT COUNT(*) AS record_count FROM station_data  
WHERE tornado = 1;
```

Identificamos três mil registros com tornados presentes. Porém, e se quiséssemos separar a contagem por ano (Figura 6.1)? Também podemos fazer isso com esta consulta:

```
SELECT year, COUNT(*) AS record_count FROM station_data  
WHERE tornado = 1  
GROUP BY year;
```

	year	record_count
51	1990	66
52	1991	75
53	1992	52
54	1993	52
55	1994	92
56	1995	45
57	1996	31
58	1997	29
59	1998	41
60	1999	48
61	2000	54
62	2001	21

Figura 6.1 – Obtendo a contagem de tornados por ano.

Repentinamente, esses dados se tornaram mais significativos. Agora estamos vendo a diferença nas contagens de tornados por ano. Detalharemos essa consulta para saber como isso ocorreu.

Primeiro selecionamos o ano (`year`), depois selecionamos a partir de que registros seria feita a contagem (`COUNT(*)`), e filtramos para obter apenas registros nos quais a ocorrência de um `tornado` fosse verdadeira:

```
SELECT year, COUNT(*) AS record_count FROM station_data
WHERE tornado = 1
GROUP BY year;
```

No entanto, também especificamos que estamos *agrupando* por `ano`. É isso que nos permite contar o número de registros *anuais*. A última linha, realçada em negrito, executa esse agrupamento:

```
SELECT year, COUNT(*) AS record_count FROM station_data
WHERE tornado = 1
GROUP BY year;
```

Podemos fatiar esses dados considerando mais de um campo. Se quiséssemos uma contagem por `ano` e `mês`, também poderíamos fazer o agrupamento considerando o campo `month` (Figura 6.2):

```
SELECT year, month, COUNT(*) AS record_count FROM station_data
WHERE tornado = 1
GROUP BY year, month
```

The screenshot shows a database interface with a toolbar at the top. The 'Grid view' tab is selected. Below the toolbar, there are several icons for navigating through the data. To the right of these icons, it says 'Total rows loaded: 831'. The main area is a grid table with three columns: 'year', 'month', and 'record count'. The data is as follows:

year	month	record count
1	1930	6 3
2	1930	10 2
3	1932	3 3
4	1933	3 3
5	1933	7 3
6	1935	7 2
7	1936	8 7
8	1936	9 3
9	1936	10 5
10	1936	11 3
11	1937	2 3
12	1937	3 3
13	1937	7 6

Figura 6.2 – Contagem de tornados por ano e mês.

Alternativamente, podemos usar *posições ordinais* em vez de especificar as colunas em `GROUP BY`. As posições ordinais correspondem à posição numérica de cada item na instrução `SELECT`. Logo, em vez de escrever `GROUP BY year, month`, poderíamos usar `GROUP BY 1, 2` (o que será particularmente útil se nossa instrução `SELECT` tiver expressões ou nomes de colunas longos e não quisermos reescrevê-los em `GROUP BY`):

```
SELECT year, month, COUNT(*) AS record_count FROM station_data
WHERE tornado = 1
GROUP BY 1, 2
```

É bom ressaltar que nem todas as plataformas dão suporte às posições ordinais. Com o Oracle e o SQL Server, por exemplo, você precisará reescrever a expressão ou o nome da coluna inteiro em `GROUP BY`.

## Ordenando registros

Observe que a coluna `month` não está na ordem natural esperada. Essa é uma boa oportunidade para examinarmos o operador `ORDER BY`, que pode ser inserido no fim de uma instrução SQL (após as instruções `WHERE` e `GROUP BY`). Se você quisesse classificar por ano, e depois por mês, só teria que adicionar esse comando:

```

SELECT year, month, COUNT(*) AS record_count FROM station_data
WHERE tornado = 1
GROUP BY year, month
ORDER BY year, month

```

No entanto, é provável que esteja mais interessado em dados recentes e que eles apareçam primeiro. Por padrão, a classificação é executada com o operador `ASC`, que ordena os dados em ordem crescente. Se quiser classificar em ordem decrescente, aplique o operador `DESC` à ordem de `year` para fazer os registros mais recentes aparecerem no início dos resultados:

```

SELECT year, month, COUNT(*) AS record_count FROM station_data
WHERE tornado = 1
GROUP BY year, month
ORDER BY year DESC, month

```

## Funções de agregação

Já usamos a função `COUNT(*)` para contar registros. Porém, há outras funções de agregação, entre elas `SUM()`, `MIN()`, `MAX()` e `AVG()`. Podemos usar funções de agregação em uma coluna específica para executar algum tipo de cálculo nela.

Porém, primeiro examinaremos outra maneira de usar `COUNT()`. A função `COUNT()` pode ser usada para uma finalidade diferente da de simplesmente contar registros. Se você especificar uma coluna em vez de um asterisco, ela contará quantos valores não nulos existem nessa coluna. Por exemplo, poderíamos fazer a contagem de registros de `snow_depth`, que retornaria o número de valores não nulos (Figura 6.3):

```

SELECT COUNT(snow_depth) as recorded_snow_depth_count
FROM STATION_DATA

```

Grid view		Form view	
			Total rows loaded: 1
<b>recorded_snow_depth_count</b>			
1	1552		

Figura 6.3 – Contagem de registros não nulos para a profundidade da neve.

Pode ser útil contar os valores não nulos de uma coluna, logo, lembre-se de que `COUNT()` também cumpre essa tarefa quando aplicada a colunas.



Funções de agregação como `COUNT()`, `SUM()`, `AVG()`, `MIN()` e `MAX()` nunca incluem valores nulos em seus cálculos. Só valores não nulos são considerados.

Passemos então para outras tarefas de agregação. Se você quisesse calcular a temperatura média de cada mês a partir do ano 2000, poderia fazer uma filtragem aceitando apenas o ano 2000 e anos posteriores, agrupar por mês e calcular a média da temperatura (Figura 6.4):

```
SELECT month, AVG(temperature) as avg_temp
FROM station_data
WHERE year >= 2000
GROUP BY month
```

month	avg temp
1	41.5558544303798
2	38.980631276901
3	48.9750626566416
4	52.7998516320474
5	58.4553191489361
6	64.344536423841
7	69.74756302521
8	68.0215384615385
9	62.632428115016
10	56.3823008849557
11	47.5029556650247
12	41.1165938864629

Figura 6.4 – Temperatura média por mês desde o ano 2000.

Como sempre, você pode usar funções nos valores agregados e executar tarefas como a de arredondamento para lhes dar uma aparência melhor (Figura 6.5):

```
SELECT month, round(AVG(temperature),2) as avg_temp
FROM station_data
WHERE year >= 2000
GROUP BY month
```

month	avg temp
1	41.56
2	38.98
3	48.98
4	52.8
5	58.46
6	64.34
7	69.75
8	68.02
9	62.63
10	56.38
11	47.5
12	41.12

Figura 6.5 – Arredondando a temperatura média por mês.

`SUM()` é outra operação comum de agregação. Para calcular a soma dos valores de profundidade da neve por ano desde 2000, execute esta consulta:

```
SELECT year, SUM(snow_depth) as total_snow
FROM station_data
WHERE year >= 2000
GROUP BY year
```

Não há limite para quantas operações de agregação você pode usar em uma única consulta. Na consulta a seguir estamos calculando `total_snow` e `total_precipitation` para cada ano desde o ano 2000 na mesma consulta, além de obter `max_precipitation`:

```
SELECT year,
SUM(snow_depth) as total_snow,
SUM(precipitation) as total_precipitation,
MAX(precipitation) as max_precipitation
FROM station_data
WHERE year >= 2000
GROUP BY year
```

Talvez você ainda não tenha notado, mas podemos calcular alguns totais

muito específicos usando `WHERE`. Se quiséssemos calcular a precipitação total por ano somente quando um tornado estivesse presente, teríamos de fazer uma filtragem para encontrar `tornado` quando seu valor fosse verdadeiro. Esta consulta só incluirá nos totais precipitações envolvendo tornados:

```
SELECT year,  
       SUM(precipitation) as tornado_precipitation  
    FROM station_data  
   WHERE tornado = 1  
GROUP BY year
```

## A instrução HAVING

Suponhamos que você quisesse filtrar registros de acordo com um valor agregado. Inicialmente, poderia ficar tentado a usar uma instrução `WHERE`, mas isso não funcionará porque `WHERE` filtra registros e não agregações. Por exemplo, se você tentar usar `WHERE` para encontrar resultados em que `total_precipitation` seja maior ou igual a 30, a filtragem falhará:

```
SELECT year,  
       SUM(precipitation) as total_precipitation  
    FROM station_data  
   WHERE total_precipitation > 30  
GROUP BY year
```

Por que não funcionou? Você não pode filtrar campos agregados usando `WHERE`. É preciso usar a palavra-chave `HAVING` para fazer isso. A agregação funciona com o software processando registro a registro e encontrando os que ele deseja manter de acordo com a condição de `WHERE`. Depois, ele agrupa os registros em `GROUP BY` e executa as funções de agregação solicitadas, como `SUM()`. Se quiséssemos filtrar pelo valor de `SUM()`, seria preciso que a filtragem ocorresse após o cálculo. É aí que entra em cena `HAVING`:

```
SELECT year,  
       SUM(precipitation) as total_precipitation  
    FROM station_data  
GROUP BY year  
HAVING total_precipitation > 30
```

`HAVING` é o equivalente na agregação a `WHERE`. A palavra-chave `WHERE` filtra registros individuais, mas `HAVING` filtra agregações.

Devo lembrar que algumas plataformas, inclusive a do Oracle, não dão suporte a aliases na instrução `HAVING` (como ocorre com `GROUP BY`). Isso significa que você deve especificar a função de agregação novamente na instrução `HAVING`. Se estivéssemos executando a consulta anterior em um banco de dados Oracle, precisaríamos escrevê-la dessa forma:

```
SELECT year,  
       SUM(precipitation) as total_precipitation  
  FROM station_data  
 GROUP BY year  
 HAVING SUM(precipitation) > 30
```

## Obtendo registros distintos

É comum tentar obter um conjunto de resultados distintos a partir de uma consulta. Sabemos que há 28.000 registros em nossa tabela `station_data`. E se quiséssemos obter uma lista distinta contendo os valores de `station_number`? Se executarmos a consulta a seguir, teremos duplicatas:

```
SELECT station_number FROM station_data
```

Se quisermos uma lista distinta contendo os números das estações sem duplicatas, podemos usar a palavra-chave `DISTINCT`:

```
SELECT DISTINCT station_number FROM station_data
```

Você também pode obter resultados distintos para mais de uma coluna. Se precisar dos conjuntos exclusivos de `station_number` e `year`, basta incluir essas duas colunas na instrução `SELECT`:

```
SELECT DISTINCT station_number, year FROM station_data
```

## Resumo

Neste capítulo aprendemos como agregar e classificar dados usando `GROUP BY` e `ORDER BY`. Também empregamos as funções de agregação `SUM()`, `MAX()`, `MIN()`, `AVG()` e `COUNT()` para reduzir milhares de registros a apenas alguns registros totalizados de maneira significativa. Já que não podemos usar

`WHERE` para filtrar campos agregados, usamos a palavra-chave `HAVING` para fazer isso. Também utilizamos o operador `DISTINCT` para obter resultados distintos em nossas consultas e eliminar duplicatas.

Espero que a esta altura você já tenha percebido a flexibilidade que o SQL oferece para o desenvolvimento rápido de relatórios significativos baseados em milhares ou milhões de registros. Antes de prosseguir, eu recomendaria que você fizesse testes com tudo que aprendeu até agora e empregasse `SELECT`, `WHERE` e `GROUP BY` em suas consultas. Faça a si próprio perguntas úteis como “A temperatura vem aumentando em todo mês de janeiro nos últimos 20 anos?” ou “Quantas vezes tivemos e não tivemos granizo durante um tornado?”. Tente criar consultas SQL com os dados climáticos para responder a essas perguntas.

Familiarize-se com o que aprendeu até aqui, mas não se preocupe em memorizar todas as funcionalidades do SQL. Isso virá com o tempo à medida que você usar e testar repetidamente a linguagem. Mais conhecimentos serão adquiridos nos próximos capítulos e não há problemas em recorrer ao Google ou a este guia se você esquecer como compor as instruções.



É possível obter a lista completa de funções de agregação do SQLite no “Apêndice A7 – funções de agregação” ou em [https://www.sqlite.org/lang\\_aggfunc.html](https://www.sqlite.org/lang_aggfunc.html).

# CAPÍTULO 7

## Instruções CASE

Estamos quase prontos para aprender o recurso que realmente define o SQL, o operador `JOIN`. Porém, antes de fazê-lo, reservaremos um pequeno capítulo para abordar um operador muito útil chamado `CASE`. Esse comando nos permite substituir o valor de uma coluna por outro valor, de acordo com uma ou mais condições.

### A instrução CASE

Uma instrução `CASE` permite mapear uma ou mais condições para um valor correspondente a cada condição. Ela deve ser iniciada com a palavra `CASE` e finalizada com `END`. Entre essas palavras-chave devemos especificar cada condição com a sintaxe `WHEN [condição] THEN [valor]`, em que a `[condição]` e o `[valor]` correspondente nós é que fornecemos. Após especificar os pares condição-valor, podemos estabelecer um valor geral a ser usado como padrão se nenhuma das condições for atendida, que deve ser definido em `ELSE`. Por exemplo, poderíamos classificar a velocidade do vento (`wind_speed`) em categorias relacionadas ao seu rigor (`wind_severity`) (Figura 7.1), em que velocidades com valor maior que 40 seriam altas ('`HIGH`'), de 30 a 40 moderadas ('`MODERATE`') e velocidades menores seriam consideradas baixas ('`LOW`'):

```
SELECT report_code, year, month, day, wind_speed,  
CASE  
    WHEN wind_speed >= 40 THEN 'HIGH'  
    WHEN wind_speed >= 30 AND wind_speed < 40 THEN 'MODERATE'  
    ELSE 'LOW'  
END as wind_severity  
FROM station_data
```

Grid view   Form view

Total rows loaded: 28000

	report_code	year	month	day	wind_speed	wind_severity
22	6757E5	1992	10	9	43.9	HIGH
23	E681EA	1984	2	10	44.9	HIGH
24	F7D723	2003	5	23	45.3	HIGH
25	B6A78C	1973	7	17	42.3	HIGH
26	1C8A2A	1998	11	21	50	HIGH
27	34DDA7	2002	12	21	0.2	LOW
28	39537B	1998	10	1	6.7	LOW
29	C3C6D5	2001	5	18	4.3	LOW
30	145150	2007	10	14	2.5	LOW
31	EF616A	1967	7	29	1.2	LOW

Figura 7.1 – Categorizando o rigor do vento como alto, moderado e baixo.

Na verdade, podemos omitir a condição `AND wind_speed < 40`. Entenda o motivo: a máquina lê uma instrução `CASE` de cima para baixo, e a primeira condição verdadeira que ela encontra é a que é usada (ela não avalia as condições subsequentes). Logo, se tivermos um registro com velocidade do vento igual a 43, poderemos ter certeza de que ele será avaliado na categoria 'HIGH'. Embora ele seja maior que 30, não será considerado 'MODERATE' porque não chegará a esse ponto. Diante disso, podemos criar uma consulta um pouco mais eficiente:

```
SELECT report_code, year, month, day, wind_speed,
CASE
    WHEN wind_speed >= 40 THEN 'HIGH'
    WHEN wind_speed >= 30 THEN 'MODERATE'
    ELSE 'LOW'
END as wind_severity
FROM station_data
```

## Agrupando instruções

Quando você criar instruções `CASE` e agrupá-las, poderá provocar algumas transformações muito poderosas. Converter valores de acordo com uma ou mais condições antes de agregá-los nos abre ainda mais possibilidades de fatiar dados de maneiras interessantes. Estendendo-nos um pouco

mais em nosso exemplo anterior, podemos executar o agrupamento baseando-nos em `year` e `wind_severity` e obter uma contagem de registros de cada um desses itens como mostrado aqui (observe também que usamos posições ordinais para não precisarmos reescrever a expressão `case` de `wind_severity` em `GROUP BY`):

```
SELECT year,  
CASE  
    WHEN wind_speed >= 40 THEN 'HIGH'  
    WHEN wind_speed >= 30 THEN 'MODERATE'  
    ELSE 'LOW'  
END as wind_severity,  
COUNT(*) as record_count  
FROM station_data  
GROUP BY 1, 2
```

## O truque da instrução CASE “Zero/Null”

Podemos fazer alguns truques interessantes com a instrução `CASE`. Um padrão simples, porém útil, é o truque da instrução `CASE` “zero/null”. Ele permite aplicar diferentes “filtros” a valores agregados distintos na mesma consulta `SELECT`. Você nunca conseguiria fazer isso na instrução `WHERE` porque ela aplica um filtro a todos os itens. No entanto, é possível usar `CASE` para criar uma condição de filtro diferente para cada valor agregado.

Digamos que você quisesse agregar a precipitação em duas somas, `tornado_precipitation` e `non_tornado_precipitation`, e agrupar por ano e mês. A lógica depende principalmente de dois campos: `precipitation` e `tornado`, mas como exatamente podemos codificar isso?

Se você pensar bem, verá que não pode fazê-lo com uma instrução `WHERE`, a menos que execute duas consultas separadas (uma para `tornado` retornando verdadeiro e a outra para falso):

### *Precipitação com tornado*

```
SELECT year, month,  
SUM(precipitation) as tornado_precipitation  
FROM station_data
```

```

WHERE tornado = 1
GROUP BY year, month

```

### *Precipitação sem tornado*

```

SELECT year, month,
SUM(precipitation) as non_tornado_precipitation
FROM station_data
WHERE tornado = 0
GROUP BY year, month

```

Entretanto, é possível fazê-lo em uma única consulta com o uso de uma instrução CASE. Você pode mover as condições de ocorrência de tornado da instrução WHERE para uma instrução CASE e retornar o valor 0 se a condição produzir falso como resultado. Em seguida, some as instruções (Figura 7.2):

```

SELECT year, month,
SUM(CASE WHEN tornado = 1 THEN precipitation ELSE 0 END) as
tornado_precipitation,
SUM(CASE WHEN tornado = 0 THEN precipitation ELSE 0 END) as
non_tornado_precipitation
FROM station_data
GROUP BY year, month

```

	year	month	tornado_precipitation	non_tornado_precipitation
811	2008	8 0		3.45
812	2008	9 0		1.77
813	2008	10 0		8.88
814	2008	11 0		2.14
815	2008	12 0.52		8.19
816	2009	1 0		4.02
817	2009	2 0		0.84
818	2009	3 0		14.41
819	2009	4 0		4.08
820	2009	5 0		8.4
821	2009	6 0.41		3.32

*Figura 7.2 – Obtendo a precipitação com e sem tornados por ano e mês.*

A instrução CASE pode realizar um grande volume de trabalho,

principalmente em tarefas de agregação complexas. Beneficiando-nos de uma condição que retorne o valor 0 quando não for atendida, ignoramos esse valor e o excluímos da soma (porque a soma de 0 não altera nada).

Você também poderia fazer isso com a operação `MIN` ou `MAX` e usar nulo em vez de 0 para assegurar que valores de determinadas condições nunca fossem considerados. É possível calcular a precipitação máxima quando tornados estavam ou não presentes (Figura 7.3) conforme descrito a seguir:

```
SELECT year,  
       MAX(CASE WHEN tornado = 0 THEN precipitation ELSE NULL END)  
             as max_non_tornado_precipitation,  
       MAX(CASE WHEN tornado = 1 THEN precipitation ELSE NULL END)  
             as max_tornado_precipitation  
FROM station_data  
GROUP BY year
```

The screenshot shows a database grid interface with two tabs: 'Grid view' (selected) and 'Form view'. At the top, there are buttons for search, refresh, and navigation, followed by the text 'Total rows loaded: 79'. The grid has three columns: 'year', 'max\_non\_tornado\_precipitation', and 'max\_tornado\_precipitation'. The data is as follows:

year	max_non_tornado_precipitation	max_tornado_precipitation
59	1990 2.48	0.59
60	1991 2.36	1.93
61	1992 1.5	1.51
62	1993 1.18	2.13
63	1994 1.26	1.16
64	1995 0.91	0.35
65	1996 3.31	0.68
66	1997 1.18	0.08
67	1998 1.22	0.2
68	1999 2.64	0.25
69	2000 0.87	0.24

Figura 7.3 – Precipitações máximas com e sem tornados por ano.

Da mesma forma que na instrução `WHERE`, você pode usar expressões booleanas em uma instrução `CASE`, o que inclui funções e as instruções `AND`, `OR` e `NOT`. A consulta a seguir calculará as temperaturas médias por mês quando chuva/granizo estavam ou não presentes após o ano 2000:

```
SELECT month,  
       AVG(CASE WHEN rain OR hail THEN temperature ELSE null END)
```

```
AS avg_precipitation_temp,  
AVG(CASE WHEN NOT (rain OR hail) THEN temperature ELSE null END)  
AS avg_non_precipitation_temp  
FROM STATION_DATA  
WHERE year > 2000  
GROUP BY month
```

O truque da instrução `CASE` zero/null é uma ótima aplicação da instrução `CASE`. Ele oferece muitas possibilidades para a execução de várias agregações com diferentes critérios e, portanto, é útil conhecê-lo.

## Resumo

Dedicamos um capítulo ao aprendizado das instruções `CASE` porque elas fornecem muita flexibilidade. Você pode substituir os valores de uma coluna por outro conjunto de valores de acordo com as condições que fornecer. Quando agregamos instruções `CASE`, criamos mais oportunidades de fatiar dados de maneiras interessantes e embutir mais informações na mesma consulta.

Espero que a essa altura você já tenha uma base de conhecimento sólida e esteja pronto para aprender a parte que define o SQL: o operador `JOIN`. Faça uma pausa. Tome algumas xícaras de café. Só após conhecer `JOIN` é que você poderá se apresentar como desenvolvedor SQL.

# CAPÍTULO 8

## JOIN

### Associando tabelas

A associação é a funcionalidade definidora do SQL que o distingue de outras tecnologias de dados. É preciso que você tenha certeza de que entendeu o material que examinamos até agora, portanto dedique algum tempo para fazer exercícios e revisões antes de prosseguir.

Voltemos ao início do livro, quando discutimos os bancos de dados relacionais. Lembra-se de como os bancos de dados “normalizados” com frequência tinham tabelas com campos que apontavam para outras tabelas? Por exemplo, considere a tabela `CUSTOMER_ORDER`, que tem um campo `CUSTOMER_ID` (Figura 8.1).

ORDER_ID	ORDER_DATE	SHIP_DATE	CUSTOMER_ID	PRODUCT_ID	ORDER_QTY	SHIPPED
1	3 2015-04-20	2015-04-23	3	5	300	false
2	4 2015-04-18	2015-04-22	5	4	375	false
3	1 2015-04-15	2015-04-18	1	1	450	false
4	5 2015-04-17	2015-04-20	3	2	500	false
5	2 2015-04-18	2015-04-21	3	2	600	false

Figura 8.1 – A tabela `CUSTOMER_ORDER` tem um campo `CUSTOMER_ID`.

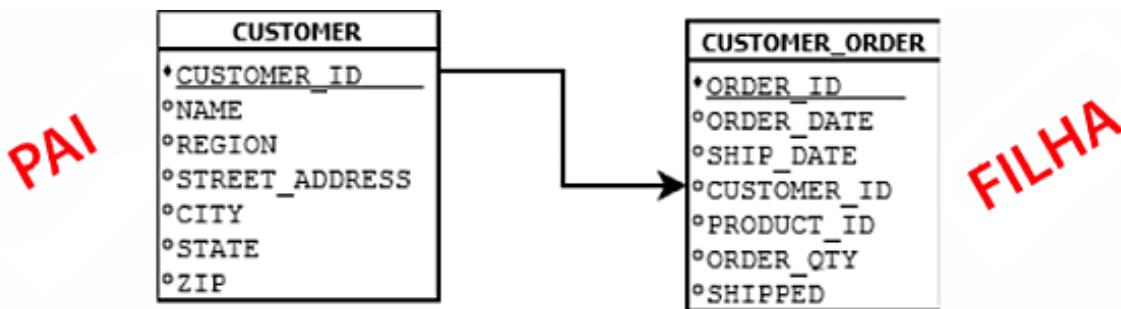
O campo `CUSTOMER_ID` fornece uma *chave* para buscas na tabela `CUSTOMER`. Diante disso, não é nenhuma surpresa a tabela `CUSTOMER` também ter um campo `CUSTOMER_ID` (Figura 8.2).

CUSTOMER_ID	NAME	REGION	STREET ADDRESS	CITY	STATE	ZIP
1	LITE Industrial	Southwest	729 Ravine Way	Irving	TX	75014
2	Rex Tooling Inc	Southwest	6129 Collie Blvd	Dallas	TX	75201
3	Re-Barre Construction	Southwest	9043 Windy Dr	Irving	TX	75032
4	Prairie Construction	Southwest	264 Long Rd	Moore	OK	62104
5	Marsh Lane Metal Works	Southeast	9143 Marsh Ln	Avondale	LA	79782

*Figura 8.2 – A tabela CUSTOMER tem um campo de chave CUSTOMER\_ID que pode ser usado na obtenção de informações de clientes.*

Podemos recuperar as informações de cliente de um pedido nessa tabela, de maneira semelhante ao que ocorre com PROCV no Excel.

Esse é um exemplo de *relacionamento* entre a tabela CUSTOMER\_ORDER e a tabela CUSTOMER. Podemos dizer que CUSTOMER é *pai* de CUSTOMER\_ORDER. Já que CUSTOMER\_ORDER depende das informações de CUSTOMER, ela é *filha* de CUSTOMER. Inversamente, CUSTOMER não pode ser filha de CUSTOMER\_ORDER porque não depende dela para obter informações. O diagrama da Figura 8.3 demonstra esse relacionamento; a seta mostra que CUSTOMER fornece informações para CUSTOMER\_ORDER via CUSTOMER\_ID.



*Figura 8.3 – CUSTOMER é pai de CUSTOMER\_ORDER, porque CUSTOMER\_ORDER depende dela para obter informações de clientes.*

O outro aspecto que devemos considerar em um relacionamento é quantos registros da tabela-filha podem estar associados a um único registro da tabela-pai. Examine as tabelas CUSTOMER e CUSTOMER\_ORDER e verá que esse é um *relacionamento um-para-muitos*, em que um único registro de cliente pode estar associado a vários pedidos. Consultemos a Figura 8.4 para ver um exemplo específico: o cliente “Re-Barre Construction”, que tem a ID de cliente 3, está associado a três pedidos.

## CUSTOMER

CUSTOMER_ID	NAME	REGION	STREET ADDRESS	CITY	STATE	ZIP
1	LITE Industrial	Southwest	729 Ravine Way	Irving	TX	75014
2	Rex Tooling Inc	Southwest	6129 Collie Blvd	Dallas	TX	75201
3	Re-Barre Construction	Southwest	9043 Windy Dr	Irving	TX	75032
4	Prairie Construction	Southwest	264 Long Rd	Moore	OK	62104
5	Marsh Lane Metal Works	Southeast	9143 Marsh Ln	Avondale	LA	79782

## CUSTOMER\_ORDER

ORDER_ID	ORDER_DATE	SHIP_DATE	CUSTOMER_ID	PRODUCT_ID	ORDER_QTY	SHIPPED
1	2015-04-20	2015-04-23	3	5	300	false
2	2015-04-18	2015-04-22	5	4	375	false
3	2015-04-15	2015-04-18	1	1	450	false
4	2015-04-17	2015-04-20	3	2	500	false
5	2015-04-18	2015-04-21	3	2	600	false

Figura 8.4 – Relacionamento um-para-muitos entre CUSTOMER e CUSTOMER\_ORDER.

O tipo um-para-muitos é o relacionamento mais comum porque acomoda a maioria das necessidades empresariais, como quando um único cliente possui vários pedidos. Os dados empresariais de um banco de dados bem projetado costumam se enquadrar em um padrão um-para-muitos. Os *relacionamentos um-para-um* e *muitos-para-muitos* (às vezes chamados de produto cartesiano) são menos comuns. Você pode estudá-los depois, mas a fim de nos atermos ao escopo do livro, não nos preocuparemos com eles.

## INNER JOIN

Agora que conhecemos os relacionamentos entre as tabelas, poderíamos achar útil unir duas delas; dessa forma, veríamos informações de CUSTOMER e CUSTOMER\_ORDER lado a lado. Se não o fizermos, teremos de executar manualmente várias buscas com CUSTOMER\_ID, o que pode ser tedioso. É possível evitar isso com os operadores JOIN, e o primeiro que conhceremos é INNER JOIN.

O operador INNER JOIN nos permite mesclar duas tabelas. Porém, se você pretende mesclar tabelas, precisamos definir um atributo comum entre elas para que seus registros se alinhem. Precisamos definir um ou mais campos que elas tenham em comum e fazer a associação a partir deles. Se

quisermos consultar a tabela `CUSTOMER_ORDER` e associá-la a `CUSTOMER` para acessar informações de clientes, é preciso definir `CUSTOMER_ID` como atributo comum.

Abra o banco de dados `rexon_metals` e, em seguida, abra uma nova janela do editor de SQL. Executaremos nossa primeira operação `INNER JOIN`:

```
SELECT ORDER_ID,  
CUSTOMER.CUSTOMER_ID,  
ORDER_DATE,  
SHIP_DATE,  
NAME,  
STREET_ADDRESS,  
CITY,  
STATE,  
ZIP,  
PRODUCT_ID,  
ORDER_QTY  
  
FROM CUSTOMER INNER JOIN CUSTOMER_ORDER  
ON CUSTOMER.CUSTOMER_ID = CUSTOMER_ORDER.CUSTOMER_ID
```

A primeira coisa a notar é que conseguimos consultar campos tanto da tabela `CUSTOMER` quanto de `CUSTOMER_ORDER`. É quase como se as tivéssemos pegado e mesclado temporariamente em uma única tabela, da qual obtivemos a consulta. Na verdade, foi exatamente o que fizemos!

Vejamos em detalhes como isso ocorreu. Primeiro, selecionamos os campos que queríamos nas tabelas `CUSTOMER` e `CUSTOMER_ORDER`:

```
SELECT CUSTOMER.CUSTOMER_ID,  
NAME,  
STREET_ADDRESS,  
CITY,  
STATE,  
ZIP,  
ORDER_DATE,  
SHIP_DATE,  
ORDER_ID,  
PRODUCT_ID,  
ORDER_QTY  
  
FROM CUSTOMER INNER JOIN CUSTOMER_ORDER  
ON CUSTOMER.CUSTOMER_ID = CUSTOMER_ORDER.CUSTOMER_ID
```

Nesse caso, queremos exibir os endereços de clientes referentes a cada pedido. Observe também que como `CUSTOMER_ID` aparece nas duas tabelas, precisamos selecionar explicitamente uma das ocorrências (não importando qual). No exemplo, selecionamos `CUSTOMER_ID` de `CUSTOMER` usando uma sintaxe explícita, `CUSTOMER.CUSTOMER_ID`.

Para concluir, temos a parte importante que mescla temporariamente as duas tabelas em uma. Foi na instrução `FROM` que executamos `INNER JOIN`. Especificamos que estamos extrairindo informações de `CUSTOMER` e mesclando-as a `CUSTOMER_ORDER`, e que o atributo comum se encontra nos campos `CUSTOMER_ID` (que precisam ser iguais para se alinhar):

```
SELECT CUSTOMER.CUSTOMER_ID,  
       NAME,  
       STREET_ADDRESS,  
       CITY,  
       STATE,  
       ZIP,  
       ORDER_DATE,  
       SHIP_DATE,  
       ORDER_ID,  
       PRODUCT_ID,  
       ORDER_QTY  
FROM CUSTOMER INNER JOIN CUSTOMER_ORDER  
ON CUSTOMER.CUSTOMER_ID = CUSTOMER_ORDER.CUSTOMER_ID
```

Se você estivesse trabalhando no Excel, poderia considerar o que fizemos como uma operação PROCV mais robusta, em que, em vez de procurar `CUSTOMER_ID` e obter um único valor em outra tabela, estamos obtendo o registro coincidente inteiro. Isso nos permite selecionar qualquer número de campos na outra tabela.

Agora examine os resultados (Figura 8.5). Graças ao operador `INNER JOIN`, essa consulta fornece uma visualização que inclui os detalhes do cliente em cada pedido.

Grid view Form view

Total rows loaded: 5

ORDER ID	CUSTOMER ID	ORDER DATE	SHIP DATE	NAME	STREET ADDRESS	CITY	STATE	ZIP	PRODUCT ID	ORDER QTY
1	1	2015-05-15	2015-05-18	LITE Industrial	729 Ravine Way	Irving	TX	75014	1	450
2	2	2015-05-18	2015-05-21	Re-Barre Construction	9043 Windy Dr	Irving	TX	75032	2	600
3	3	2015-05-20	2015-05-23	Re-Barre Construction	9043 Windy Dr	Irving	TX	75032	5	300
4	4	2015-05-18	2015-05-22	Marsh Lane Metal Works	9143 Marsh Ln	Avondale	LA	79782	4	375
5	5	2015-05-17	2015-05-20	Re-Barre Construction	9043 Windy Dr	Irving	TX	75032	2	500

Figura 8.5 – CUSTOMER mesclada a CUSTOMER\_ORDER.

As associações fornecem realmente o melhor de dois mundos. Armazenamos dados eficientemente por intermédio da normalização, mas podemos usar associações para mesclar tabelas a partir de campos comuns para criar visualizações mais descriptivas dos dados.

Há um comportamento de `INNER JOIN` com o qual devemos tomar cuidado. Faça uma pausa para examinar os resultados da consulta anterior. Podemos ver que temos três pedidos da “Re-Barre Construction”, assim como um pedido da “LITE Industrial” e outro da “Marsh Lane Metal Works”. No entanto, não estamos nos esquecendo de alguém?

Se você examinar a tabela `CUSTOMER`, verá que há cinco clientes. Nossa consulta `INNER JOIN` só capturou três. “Rex Tooling Inc” e “Prairie Construction” não aparecem nos resultados da consulta. O que ocorreu? Não há pedidos para Rex Tooling Inc e Prairie Construction e, portanto, `INNER JOIN` as excluiu da consulta. Ela só exibirá registros que existam nas duas tabelas (Figura 8.6).

Com o uso de `INNER JOIN`, qualquer registro que não tenha um valor comum nas duas tabelas será excluído. Se quisermos incluir todos os registros da tabela `CUSTOMER`, podemos fazê-lo com `LEFT JOIN`.

## CUSTOMER

CUSTOMER ID	NAME	REGION	STREET ADDRESS	CITY	STATE	ZIP
1	LITE Industrial	Southwest	729 Ravine Way	Irving	TX	75014
2	Rex Tooling Inc	Southwest	6129 Collie Blvd	Dallas	TX	75201
3	Re-Barre Construction	Southwest	9043 Windy Dr	Irving	TX	75032
4	Prairie Construction	Southwest	264 Long Rd	Moore	OK	62104
5	Marsh Lane Metal Works	Southeast	9143 Marsh Ln	Avondale	LA	79782

## CUSTOMER\_ORDER

ORDER_ID	ORDER_DATE	SHIP_DATE	CUSTOMER_ID	PRODUCT_ID	ORDER_QTY	SHIPPED
1	2015-05-15	2015-05-18	1	1	450	false
2	2015-05-18	2015-05-21	3	2	600	false
3	2015-05-20	2015-05-23	3	5	300	false
4	2015-05-18	2015-05-22	5	4	375	false
5	2015-05-17	2015-05-20	3	2	500	false

## INNER JOINED

ORDER_ID	CUSTOMER_ID	ORDER_DATE	SHIP_DATE	NAME	STREET_ADDRESS	CITY	STATE	ZIP	PRODUCT_ID	ORDER_QTY
1	1	2015-05-15	2015-05-18	LITE Industrial	729 Ravine Way	Irving	TX	75014	1	450
2	3	2015-05-18	2015-05-21	Re-Barre Construction	9043 Windy Dr	Irving	TX	75032	2	600
3	3	2015-05-20	2015-05-23	Re-Barre Construction	9043 Windy Dr	Irving	TX	75032	5	300
4	5	2015-05-18	2015-05-22	Marsh Lane Metal Works	9143 Marsh Ln	Avondale	LA	79782	4	375
5	3	2015-05-17	2015-05-20	Re-Barre Construction	9043 Windy Dr	Irving	TX	75032	2	500

Figura 8.6 – Visualização da mesclagem de CUSTOMER e CUSTOMER\_ORDER (observe que os dois clientes foram omitidos porque não têm pedidos associados).

## LEFT JOIN

Os dois clientes citados, Rex Tooling Inc e Prairie Construction, foram excluídos da operação `INNER JOIN` baseada em `CUSTOMER_ID` porque não tinham pedidos associados. Porém, digamos que quiséssemos incluí-los mesmo assim. Não é raro alguém querer associar tabelas e ver, por exemplo, todos os clientes, mesmo se eles não tiverem pedidos.

Se você entendeu o funcionamento de `INNER JOIN`, a associação externa à esquerda não é muito diferente. Porém, há uma diferença sutil. Modifique a consulta anterior substituindo `INNER JOIN` por `LEFT JOIN`, as palavras-chave de uma associação externa à esquerda. Como mostrado na Figura 8.7, a tabela especificada no lado “esquerdo” do operador `LEFT JOIN` (CUSTOMER) terá

todos os seus registros incluídos, mesmo se não tiverem registros-filho na tabela da “direita” (CUSTOMER\_ORDER).

```

SELECT CUSTOMER.CUSTOMER_ID,
NAME,
STREET_ADDRESS,
CITY,
STATE,
ZIP,
ORDER_DATE,
SHIP_DATE,
ORDER_ID,
PRODUCT_ID,
ORDER_QTY
FROM CUSTOMER LEFT JOIN CUSTOMER_ORDER
ON CUSTOMER.CUSTOMER_ID = CUSTOMER_ORDER.CUSTOMER_ID
    
```

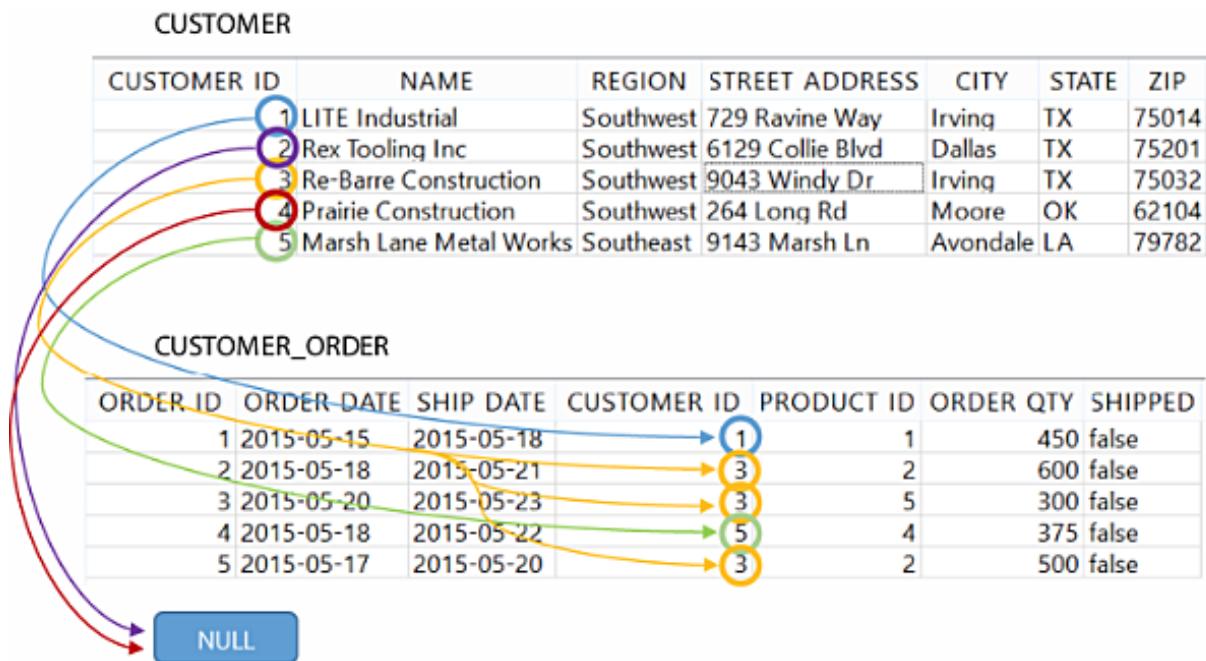
Tabela da “esquerda”      Tabela da “direita”

*Figura 8.7 – LEFT JOIN incluirá todos os registros da tabela da “esquerda”, mesmo se eles não tiverem um registro associado na tabela da “direita” (que então será nulo).*

Ao executar essa operação, teremos resultados semelhantes aos obtidos na consulta INNER JOIN anterior, mas com dois registros adicionais para os clientes que não têm pedidos (Figura 8.8). No caso desses dois clientes, observe que todos os campos que vêm de CUSTOMER\_ORDER são nulos porque não há pedidos a serem incluídos na associação. Em vez de omiti-los, como fez INNER JOIN, LEFT JOIN apenas os tornou nulos (Figura 8.9).

CUSTOMER ID	NAME	STREET ADDRESS	CITY	STATE	ZIP	ORDER DATE	SHIP DATE	ORDER ID	PRODUCT ID	ORDER QTY
1	LITE Industrial	729 Ravine Way	Irving	TX	75014	2015-05-15	2015-05-18	1	1	450
2	Rex Tooling Inc	6129 Collie Blvd	Dallas	TX	75201	NULL	NULL	NULL	2	500
3	Re-Barre Construction	9043 Windy Dr	Irving	TX	75032	2015-05-17	2015-05-20	5	2	600
3	Re-Barre Construction	9043 Windy Dr	Irving	TX	75032	2015-05-18	2015-05-21	2	5	300
3	Re-Barre Construction	9043 Windy Dr	Irving	TX	75032	2015-05-20	2015-05-23	3	NULL	NULL
4	Prairie Construction	264 Long Rd	Moore	OK	62104	NULL	NULL	NULL	4	375
5	Marsh Lane Metal Works	9143 Marsh Ln	Avondale	LA	79782	2015-05-18	2015-05-22	4	4	NULL

*Figura 8.8 – CUSTOMER em uma associação à esquerda com CUSTOMER\_ORDER (os campos nulos de CUSTOMER\_ORDER indicam que não foram encontrados pedidos desses dois clientes).*



#### LEFT OUTER JOINED

CUSTOMER_ID	NAME	STREET ADDRESS	CITY	STATE	ZIP	ORDER_DATE	SHIP_DATE	ORDER_ID	PRODUCT_ID	ORDER_QTY
1	LITE Industrial	729 Ravine Way	Irving	TX	75014	2015-05-15	2015-05-18	1	1	450
2	Rex Tooling Inc	6129 Collie Blvd	Dallas	TX	75201	NULL	NULL	NULL	NULL	NULL
3	Re-Barre Construction	9043 Windy Dr	Irving	TX	75032	2015-05-17	2015-05-20	5	2	500
3	Re-Barre Construction	9043 Windy Dr	Irving	TX	75032	2015-05-18	2015-05-21	2	2	600
3	Re-Barre Construction	9043 Windy Dr	Irving	TX	75032	2015-05-20	2015-05-23	3	5	300
4	Prairie Construction	264 Long Rd	Moore	OK	62104	NULL	NULL	NULL	NULL	NULL
5	Marsh Lane Metal Works	9143 Marsh Ln	Avondale	LA	79782	2015-05-18	2015-05-22	4	4	375

*Figura 8.9 – CUSTOMER em uma associação à esquerda com CUSTOMER\_ORDER (observe que “Rex Tooling” e “Prairie Construction” foram associadas a NULL, já que não há pedidos aos quais serem associadas).*

Também é comum o uso de LEFT JOIN na busca de registros filhos “órfãos” (registros sem pai) ou de um pai que não tenha filhos (por exemplo, pedidos que não tenham clientes ou clientes que não tenham pedidos). Você pode usar uma instrução WHERE para procurar valores nulos que sejam resultado de LEFT JOIN. Modificando nosso exemplo anterior, podemos encontrar clientes que não tenham pedidos filtrando campos da tabela da direita que sejam nulos:

```

SELECT
CUSTOMER.CUSTOMER_ID,
NAME AS CUSTOMER_NAME
FROM CUSTOMER LEFT JOIN CUSTOMER_ORDER

```

```
ON CUSTOMER.CUSTOMER_ID = CUSTOMER_ORDER.CUSTOMER_ID  
WHERE ORDER_ID IS NULL
```

Como era de se esperar, você só verá Rex Tooling Inc e Prairie Construction listadas, já que elas não têm pedidos.

## Outros tipos de operador JOIN

Há um operador `RIGHT JOIN`, que executa uma associação externa à direita que é quase idêntica à associação externa à esquerda. Ele inverte a direção da associação e inclui todos os registros da tabela da direita. No entanto, `RIGHT JOIN` é raramente usado e deve ser evitado. Você deve obedecer à convenção preferindo as associações externas à esquerda com `LEFT JOIN` e posicionar a tabela “que contém todos os registros” no lado esquerdo do operador de associação.

Também há um operador de associação externa completa chamado `OUTER JOIN` que inclui todos os registros das duas tabelas. Ele executa uma `LEFT JOIN` e uma `RIGHT JOIN` simultaneamente e pode ter registros nulos nas duas tabelas. Pode ser útil na busca de registros órfãos nas duas direções simultaneamente em uma única consulta, mas também é pouco usado.



Os operadores `RIGHT JOIN` e `OUTER JOIN` não têm suporte no SQLite devido à sua natureza altamente segmentada. Porém, a maioria das soluções de banco de dados faz uso deles.

## Associando várias tabelas

Os bancos de dados relacionais podem ser muito complexos em termos de relacionamentos entre tabelas. Uma tabela específica pode ser filha de mais de uma tabela-pai, e uma tabela pode ser pai de uma tabela e filha de outra. Bem, como isso tudo funciona?

Examinamos o relacionamento entre `CUSTOMER` e `CUSTOMER_ORDER`. No entanto, há outra tabela que podemos incluir que tornará nossos pedidos mais significativos: a tabela `PRODUCT`. Lembre-se de que a tabela `CUSTOMER_ORDER` tem uma coluna `PRODUCT_ID`, que corresponde a um produto da tabela `PRODUCT`.

É possível fornecer informações não só de `CUSTOMER` para a tabela `CUSTOMER_ORDER`, mas também de `PRODUCT` usando `PRODUCT_ID` (Figura 8.10).

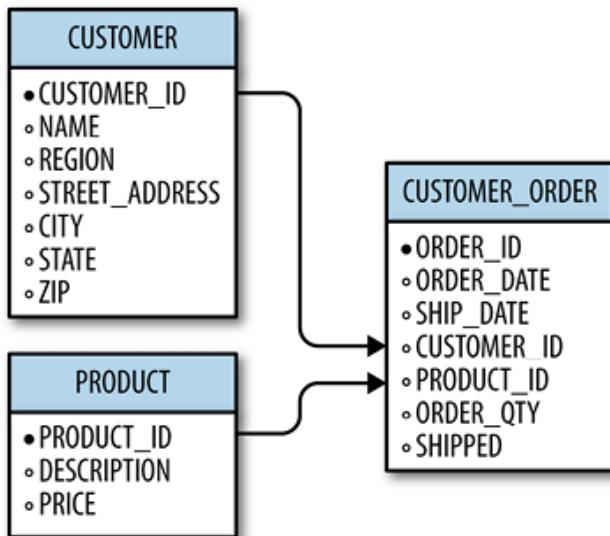


Figura 8.10 – Associando várias tabelas.

Podemos usar esses dois relacionamentos para executar uma consulta que exiba pedidos com informações de clientes e de produtos simultaneamente. Só precisamos definir as duas associações entre `CUSTOMER_ORDER` e `CUSTOMER` e entre `CUSTOMER_ORDER` e `PRODUCT` (Figura 8.11). Se achar confuso, compare a consulta a seguir com o diagrama da Figura 8.10 e verá que as associações foram construídas estritamente a partir desses relacionamentos:

```
SELECT
  ORDER_ID,
  CUSTOMER.CUSTOMER_ID,
  NAME AS CUSTOMER_NAME,
  STREET_ADDRESS,
  CITY,
  STATE,
  ZIP,
  ORDER_DATE,
  PRODUCT_ID,
  DESCRIPTION,
  ORDER_QTY
FROM CUSTOMER
```

```

INNER JOIN CUSTOMER_ORDER
ON CUSTOMER_ORDER.CUSTOMER_ID = CUSTOMER.CUSTOMER_ID
INNER JOIN PRODUCT
ON CUSTOMER_ORDER.PRODUCT_ID = PRODUCT.PRODUCT_ID

```

ORDER ID	CUSTOMER ID	CUSTOMER NAME	STREET ADDRESS	CITY	STATE	ZIP	ORDER DATE	PRODUCT ID	DESCRIPTION	ORDER QTY
1	1	1 LITE Industrial	729 Ravine Way	Irving	TX	75014	2015-05-15	1	Copper	450
2	2	3 Re-Barre Construction	9043 Windy Dr	Irving	TX	75032	2015-05-18	2	Aluminum	600
3	3	3 Re-Barre Construction	9043 Windy Dr	Irving	TX	75032	2015-05-20	5	Bronze	300
4	4	5 Marsh Lane Metal Works	9143 Marsh Ln	Avondale	LA	79782	2015-05-18	4	Steel	375
5	5	3 Re-Barre Construction	9043 Windy Dr	Irving	TX	75032	2015-05-17	2	Aluminum	500

Figura 8.11 – Associando campos de ORDER, CUSTOMER e PRODUCT.

Esses pedidos são muito mais descritivos agora que nos beneficiamos de CUSTOMER\_ID e PRODUCT\_ID para trazer informações de clientes e de produtos. Na verdade, já que mesclamos as tabelas, podemos usar campos das três para criar expressões. Se quisermos calcular a receita obtida com cada pedido, podemos multiplicar ORDER\_QTY e PRICE, mesmo que esses campos existam em tabelas separadas:

```

SELECT
ORDER_ID,
CUSTOMER.CUSTOMER_ID,
NAME AS CUSTOMER_NAME,
STREET_ADDRESS,
CITY,
STATE,
ZIP,
ORDER_DATE,
PRODUCT_ID,
DESCRIPTION,
ORDER_QTY,
ORDER_QTY * PRICE as REVENUE
FROM CUSTOMER
INNER JOIN CUSTOMER_ORDER
ON CUSTOMER.CUSTOMER_ID = CUSTOMER_ORDER.CUSTOMER_ID
INNER JOIN PRODUCT
ON CUSTOMER_ORDER.PRODUCT_ID = PRODUCT.PRODUCT_ID

```

Agora temos a receita fornecida por cada pedido, ainda que as colunas necessárias venham de duas tabelas separadas.

## Agrupando JOINs

Daremos continuidade usando o mesmo exemplo. Temos os pedidos com suas receitas, graças à associação que construímos. E se quiséssemos calcular a receita total por cliente? Ainda temos de usar todas as três tabelas e mesclá-las usando nosso esquema de associação atual, porque precisamos da receita que acabamos de calcular. E também precisamos usar GROUP BY.

Isso é legítimo e perfeitamente exequível. Como queremos fazer a agregação por cliente, o agrupamento precisa se basear em CUSTOMER\_ID e CUSTOMER\_NAME. Em seguida, precisamos somar a expressão ORDER\_QTY \* PRICE para obter a receita total (Figura 8.12). Para deixar claro o escopo de GROUP BY, omitimos todos os outros campos:

```
SELECT
  CUSTOMER.CUSTOMER_ID,
  NAME AS CUSTOMER_NAME,
  sum(ORDER_QTY * PRICE) as TOTAL_REVENUE
FROM CUSTOMER_ORDER
INNER JOIN CUSTOMER
ON CUSTOMER.CUSTOMER_ID = CUSTOMER_ORDER.CUSTOMER_ID
INNER JOIN PRODUCT
ON CUSTOMER_ORDER.PRODUCT_ID = PRODUCT.PRODUCT_ID
GROUP BY 1,2
```

CUSTOMER ID	CUSTOMER NAME	TOTAL REVENUE
1	LITE Industrial	3379.5
3	Re-Barre Construction	4038.0
5	Marsh Lane Metal Works	4616.25

Figura 8.12 – Calculando TOTAL\_REVENUE pela associação e agregação de três tabelas.

Se quisermos ver todos os clientes, inclusive os que não têm pedidos, é só usar LEFT JOIN em vez de INNER JOIN nas operações de associação (Figura 8.13):

```
SELECT
  CUSTOMER.CUSTOMER_ID,
```

```

NAME AS CUSTOMER_NAME,
sum(ORDER_QTY * PRICE) as TOTAL_REVENUE
FROM CUSTOMER
LEFT JOIN CUSTOMER_ORDER
ON CUSTOMER.CUSTOMER_ID = CUSTOMER_ORDER.CUSTOMER_ID
LEFT JOIN PRODUCT
ON CUSTOMER_ORDER.PRODUCT_ID = PRODUCT.PRODUCT_ID
GROUP BY 1,2

```

CUSTOMER ID	CUSTOMER NAME	TOTAL REVENUE
1	LITE Industrial	3379.5
2	Rex Tooling Inc	NULL
3	Re-Barre Construction	4038.0
4	Prairie Construction	NULL
5	Marsh Lane Metal Works	4616.25

Figura 8.13 – Usando LEFT JOIN para incluir todos os clientes e sua receita total.



Precisamos associar à esquerda os dois pares de tabelas porque combinar LEFT JOIN e INNER JOIN faria INNER JOIN ter precedência, o que resultaria na exclusão dos dois clientes sem pedidos. Isso ocorre porque valores nulos não podem servir de base para uma associação interna e são sempre excluídos. LEFT JOIN tolera valores nulos.

Agora a Rex Tooling Inc e a Prairie Construction estão presentes, mesmo não tendo pedidos. Também poderíamos fazer os valores adotarem como padrão 0 em vez de nulo se não houver vendas. Isso pode ser feito com a função `coalesce()` que conhecemos no Capítulo 5 para transformar zeros em valores nulos (Figura 8.14):

```

SELECT
CUSTOMER.CUSTOMER_ID,
NAME AS CUSTOMER_NAME,
coalesce(sum(ORDER_QTY * PRICE), 0) as TOTAL_REVENUE
FROM CUSTOMER
LEFT JOIN CUSTOMER_ORDER
ON CUSTOMER.CUSTOMER_ID = CUSTOMER_ORDER.CUSTOMER_ID
LEFT JOIN PRODUCT
ON CUSTOMER_ORDER.PRODUCT_ID = PRODUCT.PRODUCT_ID

```

```
GROUP BY 1,2
```

CUSTOMER_ID	CUSTOMER_NAME	TOTAL_REVENUE
1	LITE Industrial	3379.5
2	Rex Tooling Inc	0
3	Re-Barre Construction	4038.0
4	Prairie Construction	0
5	Marsh Lane Metal Works	4616.25

Figura 8.14 – Transformando em 0 os valores nulos das receitas totais.

## Resumo

As associações são o tópico mais desafiador do SQL, mas também o mais compensador. Elas nos permitem pegar dados distribuídos em várias tabelas e reuni-los como algo mais significativo e descritivo. Podemos pegar duas ou mais tabelas e associá-las em uma tabela maior com mais contexto. No próximo capítulo aprenderemos mais sobre as associações e como elas são naturalmente definidas por relacionamentos entre as tabelas.

## CAPÍTULO 9

# Design de banco de dados

## Planejando um banco de dados

Até agora, neste livro, só aprendemos como consumir dados com a instrução `SELECT`. Fizemos operações de análise que leem dados e os transformam de maneiras interessantes, mas nada disso muda fisicamente os dados nas tabelas. Uma instrução `SELECT` é uma operação somente de leitura. Em algumas situações, no entanto, podemos querer criar novas tabelas, assim como inserir, atualizar e excluir registros.

Quando você criar suas próprias tabelas para dar suporte à empresa, isso não pode ser feito de qualquer maneira. É preciso planejar com cuidado porque certamente um design de banco de dados inapropriado causará arrependimentos mais à frente. Há perguntas críticas que devem orientar o design:

### *Perguntas relativas ao design*

- Quais são os requisitos do negócio?
- Que tabelas serão necessárias para atender a esses requisitos?
- Que colunas cada tabela conterá?
- Como as tabelas serão normalizadas?
- Quais serão seus relacionamentos pai/filho?

Pode ser uma boa ideia rascunhar um diagrama que exiba as tabelas e como elas estão relacionadas. No entanto, o design não é o único fator a considerar. O fornecimento de dados também deve fazer parte do processo de planejamento. Se os dados não forem editáveis e atuais, o design terá falhado. Esse fator é com frequência negligenciado e pode facilmente fazer um projeto de banco de dados falhar.

### *Perguntas relacionadas aos dados*

- Quantos dados serão fornecidos nessas tabelas?
- Quem ou o que fornecerá os dados para as tabelas?
- De onde virão os dados?
- Precisamos de processos que preencham automaticamente as tabelas?

A recepção de dados precisa ocorrer a partir de algum local. Dependendo da natureza dos dados, eles podem ser criados dentro de sua empresa ou recebidos de terceiros. Se você precisar armazenar um alto volume de dados de atualização regular, talvez uma pessoa não consiga executar essa tarefa manualmente. Será preciso que um processo escrito em Java, Python ou outra linguagem de codificação a execute.

Embora a segurança e a administração não façam parte do escopo deste livro, geralmente bancos de dados centralizados lidam com essas áreas. A administração de privilégios e da segurança é por si só uma tarefa de tempo integral normalmente executada por administradores de bancos de dados (DBAs, database administrators). Em bancos de dados centralizados, fatores relacionados à segurança devem ser considerados.

### *Perguntas relacionadas à segurança*

- Quem deve ter acesso a esse banco de dados?
- Quem deve ter acesso a que tabelas? Acesso somente de leitura? Acesso de gravação?
- Esse banco de dados é crítico para as operações empresariais?
- Que planos de backup temos para o caso de desastre/falha?
- As alterações feitas nas tabelas devem ser registradas?
- Se o banco de dados for usado por sites ou aplicativos web, isso é seguro?

A segurança é um assunto difícil de resolver. Uma segurança excessiva cria burocracia e atrapalha a agilidade, mas a segurança ineficiente é um convite ao caos. Como ocorre com qualquer assunto complexo, é preciso que haja um ponto de equilíbrio entre os dois extremos. Entretanto, a segurança deve ter alta prioridade quando o banco de dados for usado

por um site. A conexão de qualquer coisa com a web a torna mais vulnerável a vazamentos e ataques maliciosos.



Um dos ataques mais maliciosos é a *injeção de SQL*. Se o desenvolvedor web não implementar medidas de segurança em um site, será possível digitar em um navegador web uma instrução **SELECT** cuidadosamente elaborada e fazer os resultados da consulta serem retornados diretamente para você! 130 milhões de números de cartão de crédito foram roubados dessa forma em 2009.

O SQLite tem poucos recursos de segurança ou administrativos, já que esses recursos seriam um exagero em um banco de dados leve. Se seus bancos de dados SQLite precisarem de proteção, proteja os arquivos da mesma forma como faria com qualquer outro arquivo. Oculte-os, copie-os em um backup ou distribua cópias para seus colaboradores para que eles não precisem acessar a “cópia-mestra”.

Lembrando de todas essas considerações, projetaremos nosso primeiro banco de dados.

## A conferência SurgeTech

Você é membro da equipe da conferência SurgeTech, uma reunião de startups de tecnologia procurando publicidade e investidores. O organizador lhe incumbiu da tarefa de criar um banco de dados para gerenciar os participantes, as empresas, as apresentações, as salas e o comparecimento nas apresentações. Como esse banco de dados deve ser projetado?

Primeiro, examine as diferentes entidades e pense como elas serão estruturadas em tabelas. Essa tarefa pode dar a impressão de envolver um grande número de perguntas operacionais, mas qualquer problema complexo pode ser dividido em componentes mais simples.

## Participantes

Os participantes são convidados registrados (inclusive alguns VIPs) que estão conhecendo as startups de tecnologia. Teremos de registrar a ID, o nome, o número de telefone, o email e o status VIP de cada participante.

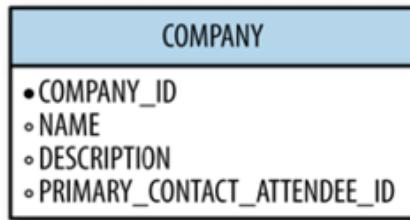
Pegando todas essas informações, poderíamos projetar a tabela **ATTENDEE**

com essas colunas:



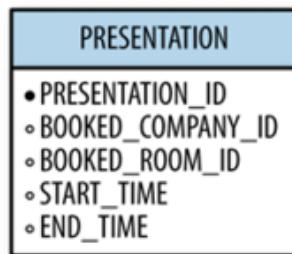
## Empresas

As startups também precisam ser registradas. A ID, o nome, a descrição e o contato principal (que deve ser listado como participante) de cada empresa devem ser registrados:



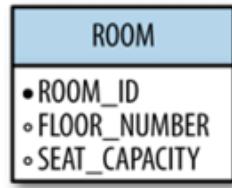
## Apresentações

As empresas agendarão a apresentação para um intervalo de tempo específico (com uma hora de início e uma hora de término). Cada período de apresentação também deve ser reservado para a empresa condutora e reservar um número de sala:



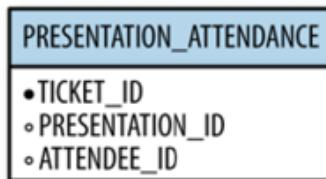
## Salas

Haverá salas disponíveis para as apresentações, e cada sala terá um número identificador, um número de andar e a capacidade de pessoas sentadas:



## Comparecimento nas apresentações

Se os participantes estiverem interessados em comparecer à apresentação de uma empresa, poderão adquirir um tíquete (que terá uma identificação) e ter o acesso permitido. Isso ajudará a rastrear quem compareceu a quais apresentações. Para implementar esse recurso, a tabela PRESENTATION\_ATTENDANCE registrará as IDs dos tíquetes e unirá as apresentações aos participantes por intermédio de suas IDs para exibir quem esteve onde:



## Chaves primária e externa

Você deve sempre tentar ter uma chave primária em suas tabelas. A *chave primária* é um campo especial (ou uma combinação de campos) que fornece uma identidade exclusiva para cada registro. Com frequência ela define um relacionamento e costuma formar a base de associações. A tabela ATTENDEE tem um campo ATTENDEE\_ID como sua chave primária, COMPANY tem COMPANY\_ID, e assim por diante. Embora você não precise designar um campo como chave primária para usar como base de associações, isso permite que o software do banco de dados execute consultas com muito mais eficiência. Além disso, age como uma restrição para assegurar a

integridade dos dados. Não são permitidas duplicatas da chave primária, o que significa que você não pode ter dois registros em `ATTENDEE` com `ATTENDEE_ID` igual a 2. O banco de dados não deixará que isso aconteça e lançará um erro.



Para limitar nosso escopo neste livro, não criaremos uma chave primária com mais de um campo. Porém, lembre-se de que múltiplos campos podem agir como chave primária, e não podemos ter combinações duplicadas desses campos. Por exemplo, se você especificar sua chave primária com base nos campos `REPORT_ID` e `APPROVER_ID`, não poderá ter dois registros com essa combinação de campos.

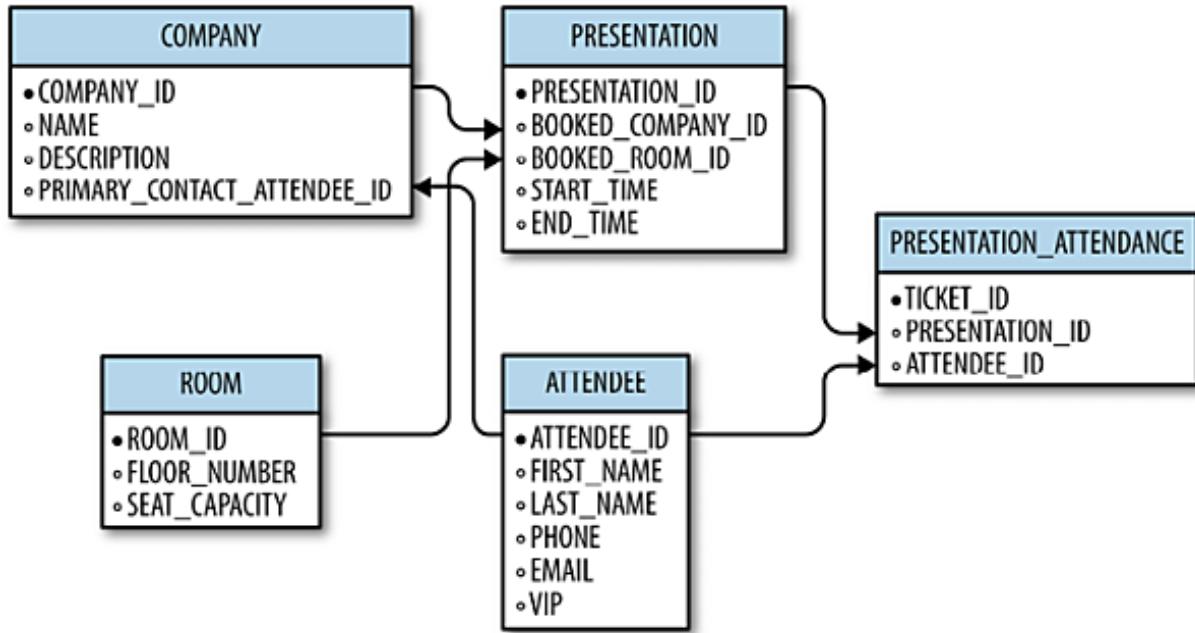
Não confunda a chave primária com uma *chave externa*. A chave primária existe na tabela-pai e a chave externa fica na tabela-filha. A chave externa de uma tabela-filha aponta para a chave primária de sua tabela-pai. Por exemplo, o campo `ATTENDEE_ID` da tabela `ATTENDEE` é uma chave primária, mas o campo `ATTENDEE_ID` da tabela `PRESENTATION_ATTENDANCE` é uma chave externa. As duas estão associadas por um relacionamento um-para-muitos. Ao contrário da chave primária, a chave externa não exige exclusividade, por isso é representada pelo termo “muitos” do relacionamento “um-para-muitos”.

A chave primária e a chave externa não precisam compartilhar o mesmo nome de campo. O campo `BOOKED_COMPANY_ID` da tabela `PRESNTATION` é uma chave externa que aponta para o campo `COMPANY_ID` de sua tabela-pai `COMPANY`. O nome do campo pode ser diferente na chave externa para descrever melhor o seu uso. Nesse caso, `BOOKED_COMPANY_ID` (`ID_de_Empresa_com_Reserva`) é mais descritivo que apenas `COMPANY_ID` (`ID_de_Empresa`). A semântica é subjetiva, mas será legítima se o jargão empresarial ficar claro.

## O esquema

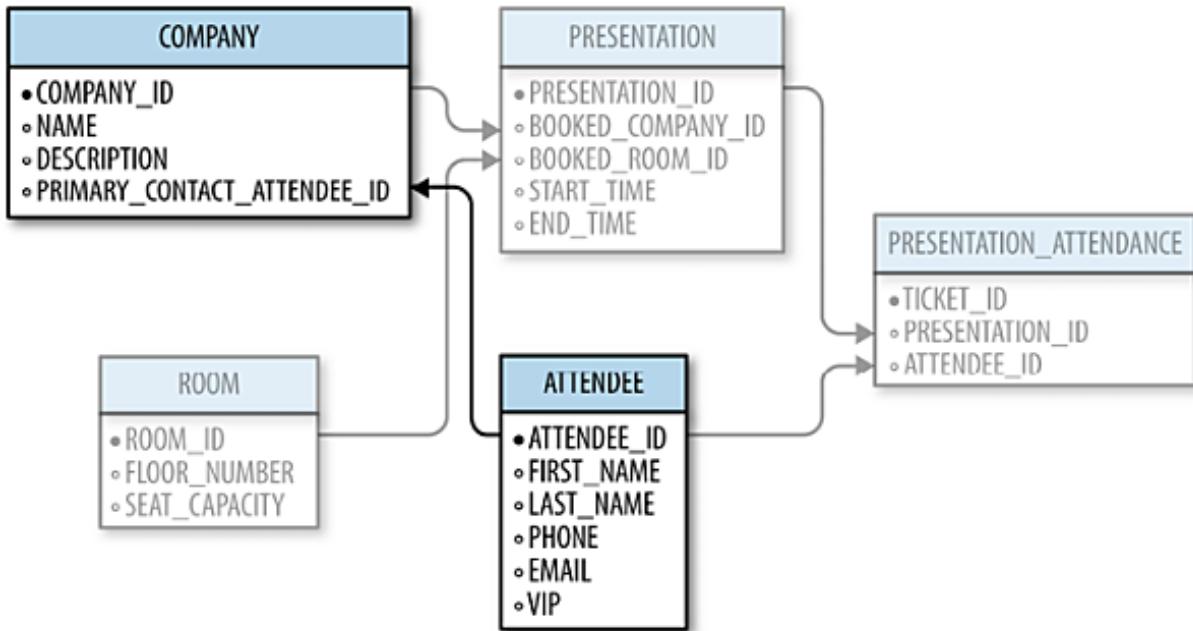
Aplicando nosso conhecimento de chaves primárias e chaves externas, podemos estabelecer os relacionamentos entre essas cinco tabelas e desenhar um esquema de banco de dados como mostrado na Figura 9.1. Um *esquema de banco de dados* é um diagrama que exibe as tabelas, suas colunas e seus relacionamentos. Todas as chaves primárias e chaves externas são conectadas por setas. O lado sem ponta da seta sai de uma

chave primária e o outro lado aponta para uma chave externa. As setas demonstram como cada tabela-pai fornece dados para uma tabela-filha.



*Figura 9.1 – Esquema de banco de dados da conferência SurgeTech, com todas as tabelas e relacionamentos.*

Pode assustar olhar para todas essas tabelas e relacionamentos de uma só vez. No entanto, qualquer estrutura complexa pode ser dividida em partes mais simples. Provavelmente você nunca escreverá uma consulta `SELECT` que use todas as tabelas, e só fará a seleção a partir de duas (talvez três) tabelas. Logo, o segredo para seguir um esquema é se concentrar apenas em duas ou três tabelas de cada vez e em seus relacionamentos. Ao analisar o design que esboçou, você pode verificar se as tabelas foram normalizadas eficientemente e se as chaves primárias/externas estão sendo usadas de maneira eficaz (Figura 9.2).



*Figura 9.2 – Examinando apenas duas tabelas e seus relacionamentos (aqui podemos ver facilmente que PRIMARY\_CONTACT\_ATTENDEE\_ID fornece informações de nome e contato a partir da tabela ATTENDEE).*

Se você conseguir visualizar facilmente diferentes associações e consultas SELECT que usariam os dados, o esquema de banco de dados deve estar correto.

## Criando um novo banco de dados

Agora que temos um design bem planejado, é hora de criar o banco de dados. Usaremos ferramentas do SQLiteStudio para criar as tabelas e os componentes. Porém, enquanto fazemos isso, o SQLiteStudio nos mostrará o código SQL que ele usa para criar e modificar nossas tabelas.

Primeiro, navegue para Database → Add a Database (Figura 9.3).

Clique no botão de “sinal de adição” verde que foi marcado na Figura 9.4 para criar um novo banco de dados.

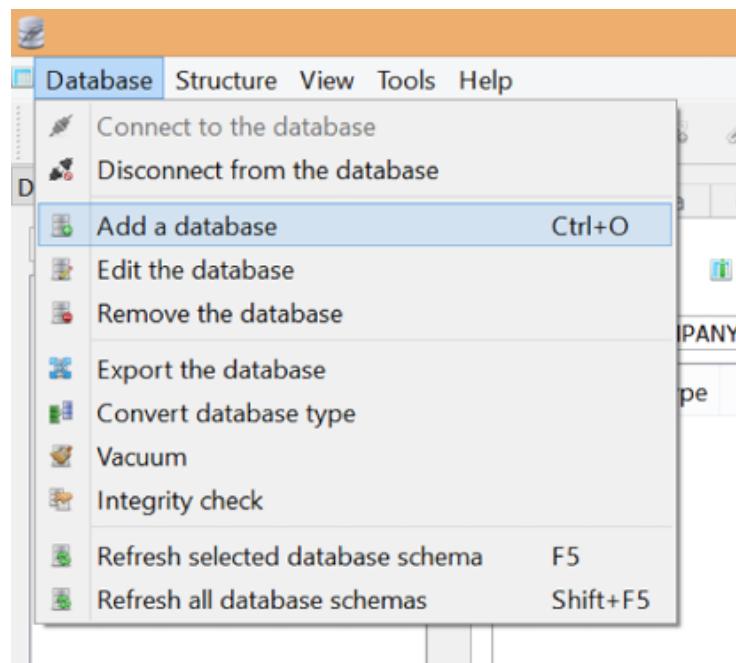


Figura 9.3 – Adicionando um banco de dados.

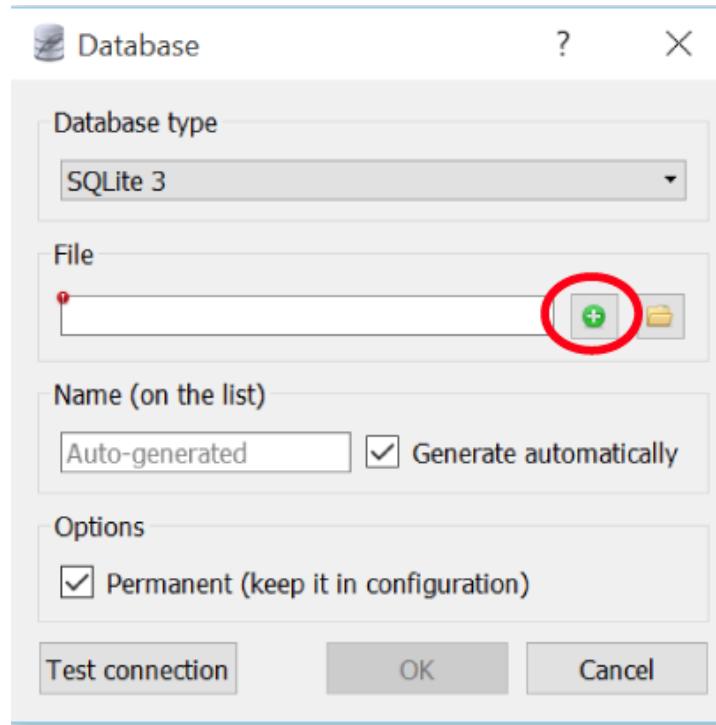


Figura 9.4 – Criando um banco de dados.

Navegue até a pasta em que deseja salvar o banco de dados. No campo “File name” forneça um nome para o arquivo. Geralmente é boa prática terminar o nome com a extensão de arquivo .db. Nesse caso, poderíamos

chamar o arquivo de *surgetech\_conference.db* (Figura 9.5).

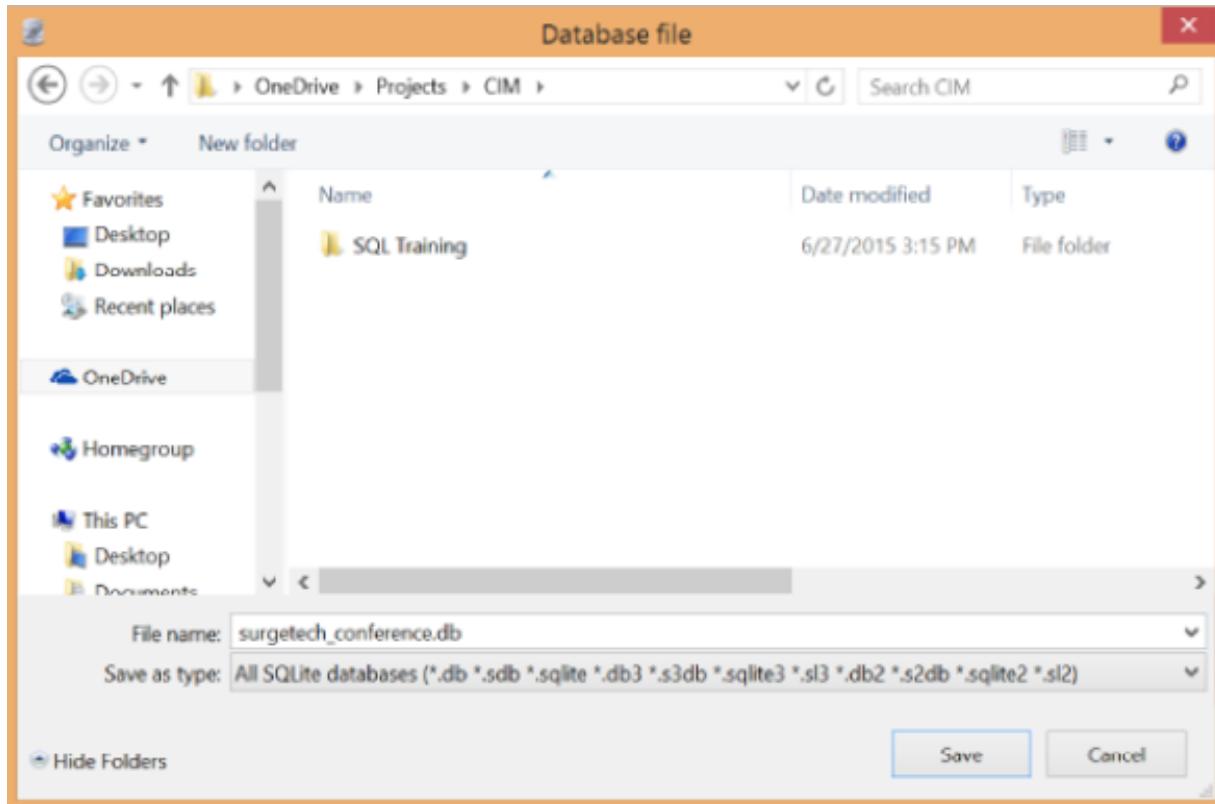


Figura 9.5 – Selecionando um local para criar um banco de dados.

Clique em Save e em OK. Agora você deve ver o novo banco de dados em seu navegador (Figura 9.6).

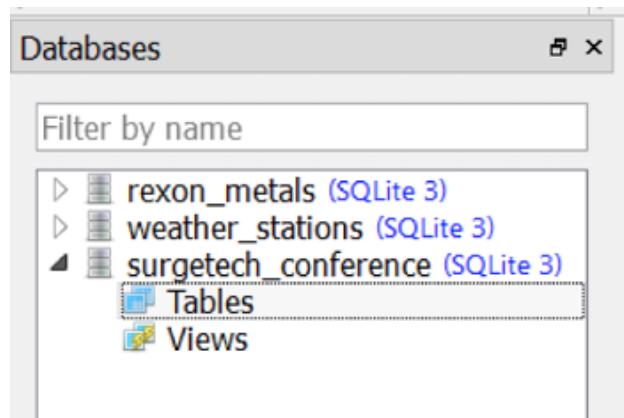


Figura 9.6 – Nosso novo banco de dados *surgetech\_conference*.

Esse banco de dados está vazio, logo, adicionaremos algumas tabelas.

## CREATE TABLE

Para criar uma tabela em SQL devemos usar a instrução `CREATE TABLE`. No entanto, defendo o uso de ferramentas que facilitem as tarefas. Usaremos as ferramentas gráficas do SQLiteStudio para criar a tabela e ao terminarmos ele irá gerar e exibir a instrução `CREATE TABLE` que construiu para nós.

Clique com o botão direito do mouse no item Tables a partir do navegador e clique em Create a table, como mostrado na Figura 9.7.

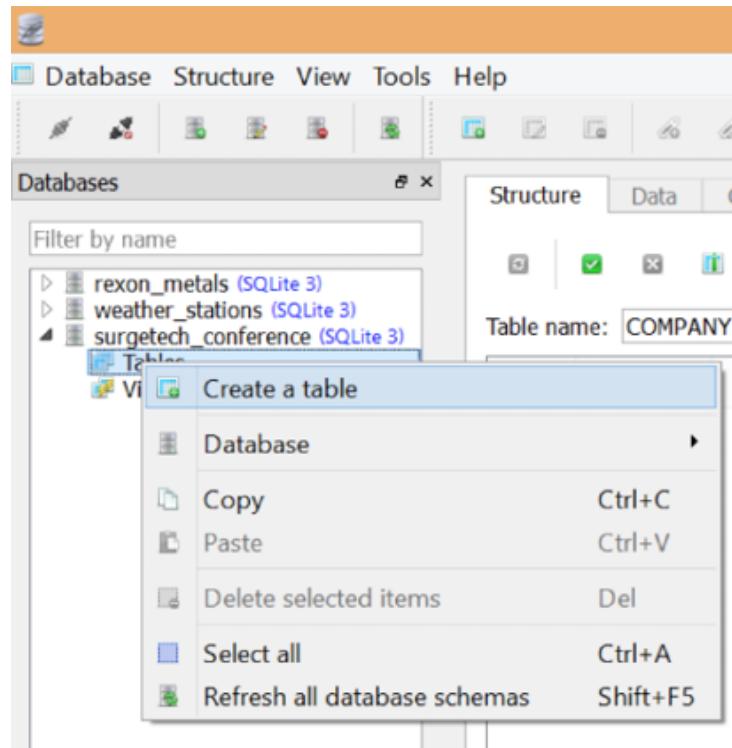


Figura 9.7 – Criando uma tabela.

Você será levado à guia Structure. Nela podemos adicionar, modificar e remover colunas em nossa tabela (Figura 9.8).

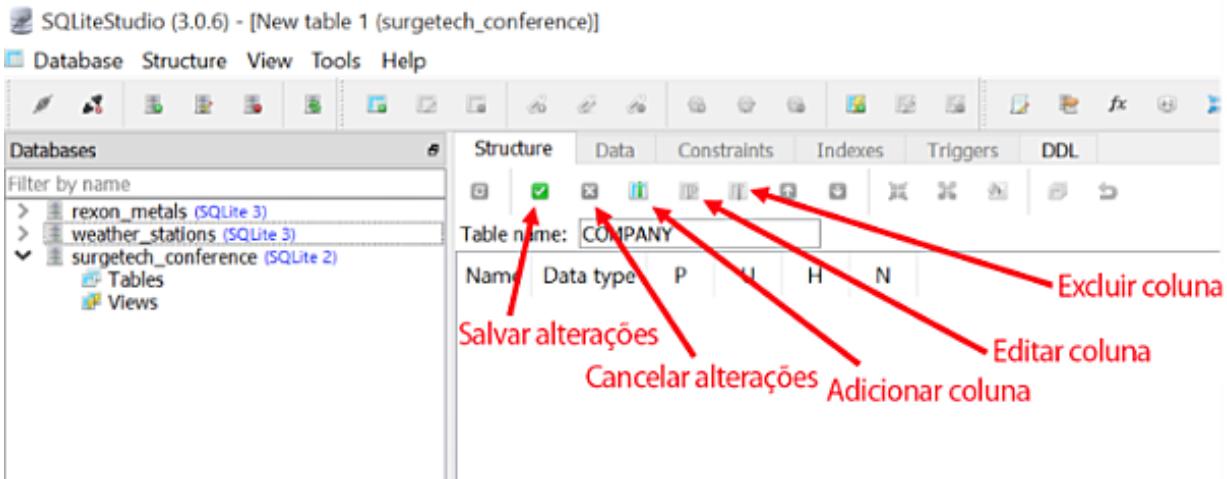
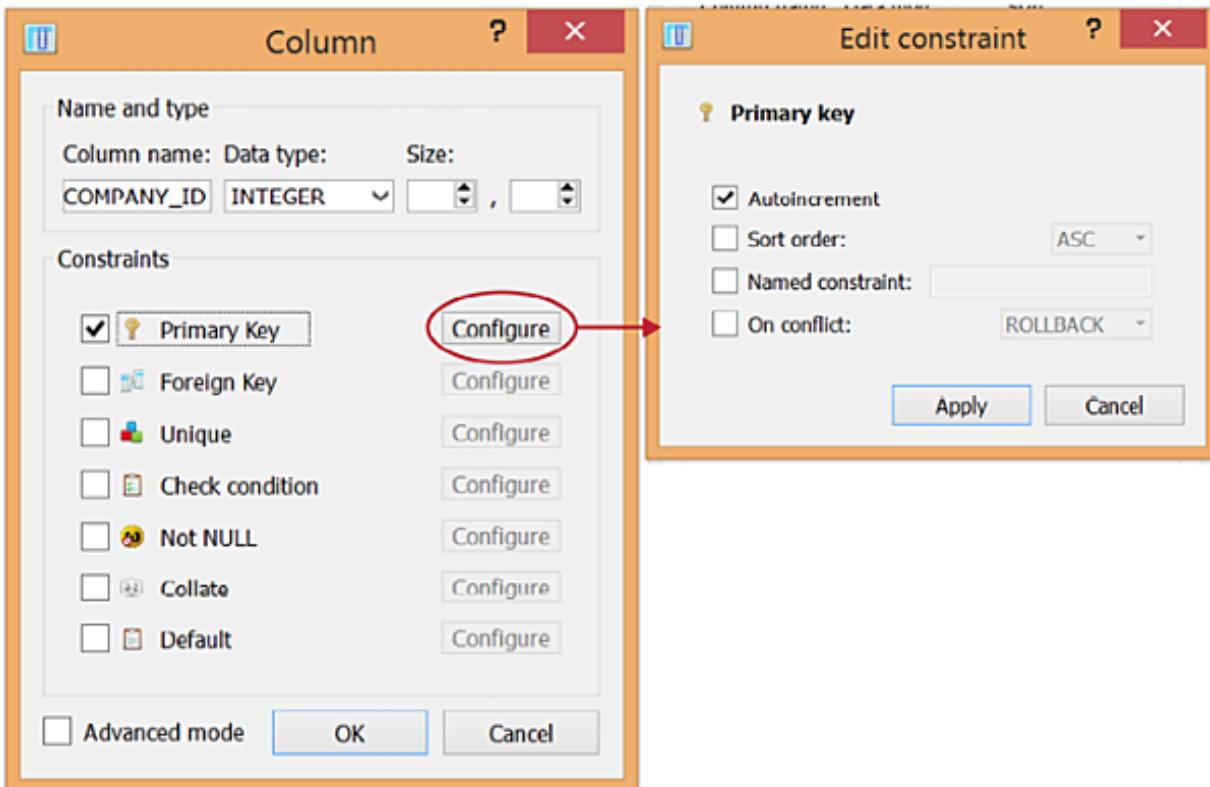


Figura 9.8 – A guia Strucuture, que podemos usar para adicionar, modificar e remover colunas em uma tabela.

Podemos definir várias restrições para assegurar que os dados inseridos nas colunas estejam de acordo com as regras especificadas. Também podemos fornecer um nome para essa tabela no campo “Table name”. Digite **COMPANY** como o nome da tabela. Observe que há um botão para salvar suas edições e outro para adicionar uma nova coluna.

Clique no botão Add Column e verá uma caixa de diálogo para a definição de uma nova coluna e seus atributos. Nomeie a coluna como **COMPANY\_ID** e estabeleça seu tipo de dado como “INTEGER”, como mostrado na Figura 9.9.



*Figura 9.9 – Definindo uma nova coluna chamada COMPANY\_ID que contém inteiros; ela também foi configurada para ser a chave primária e fornecerá automaticamente um valor via “Autoincrement” para cada registro inserido.*

Esse é o campo `COMPANY_ID` e precisamos defini-lo como chave primária da tabela `COMPANY`. Normalmente, a maneira mais fácil de atribuir valores de chave é fazendo a atribuição sequencialmente para cada novo registro. O primeiro registro terá 1 como valor de `COMPANY_ID`, o segundo terá 2, o terceiro 3, e assim por diante. Quando inserirmos registros no próximo capítulo, vai ser difícil fazer isso manualmente. Porém, podemos configurar o SQLite para atribuir uma ID automaticamente para cada registro que inserirmos. Apenas marque Primary Key, clique em Configure, selecione Autoincrement e clique em Apply (Figura 9.9).

Para concluir, clique em OK na janela Column e verá nossa primeira coluna definida (Figura 9.10).

The screenshot shows the MySQL Workbench interface with the 'Structure' tab selected. The table name is set to 'COMPANY'. The table definition is as follows:

Name	Data type	P	F	U	H	N	C
1 COMPANY_ID	INTEGER						NULL

Figura 9.10 – Nossa primeira coluna está definida; observe o símbolo de chave indicando que essa coluna é a chave primária.

Definimos nossa primeira coluna, e já que ela é a coluna de chave primária, demandou mais trabalho. O resto das colunas será mais fácil de definir.

Clique no botão Add Column novamente para criar outra coluna (Figura 9.11).

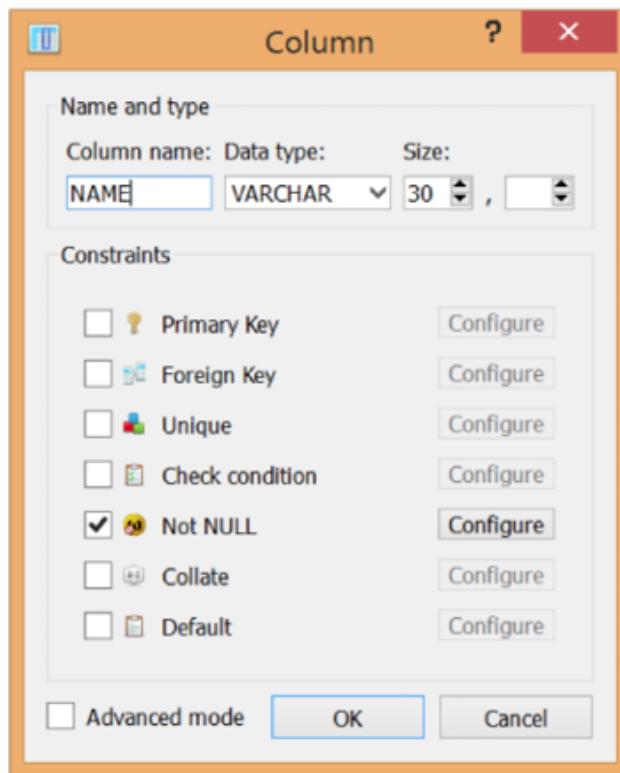


Figura 9.11 – Criando uma coluna “NAME” de tipo VARCHAR, com 30 como o número máximo de caracteres e com a restrição “Not NULL”.

Chame-a de NAME e defina-a como de tipo VARCHAR, que é para texto

que pode ter vários tamanhos. Especifique 30 como o número máximo de caracteres. Já que não vamos querer que esse campo tenha um valor nulo, marque a restrição “Not NULL”. Se algum registro for adicionado ou modificado com NAME configurado com nulo, o banco de dados rejeitará as edições.

Clique em OK e crie mais duas colunas, DESCRIPTION e PRIMARY\_CONTACT\_ATTENDEE\_ID, com as configurações mostradas na Figura 9.12. Lembre-se de que PRIMARY\_CONTACT\_ATTENDEE\_ID deve ser uma chave externa, mas ainda não definimos isso. Retornaremos para definir essa configuração após criarmos seu pai, a tabela ATTENDEE.

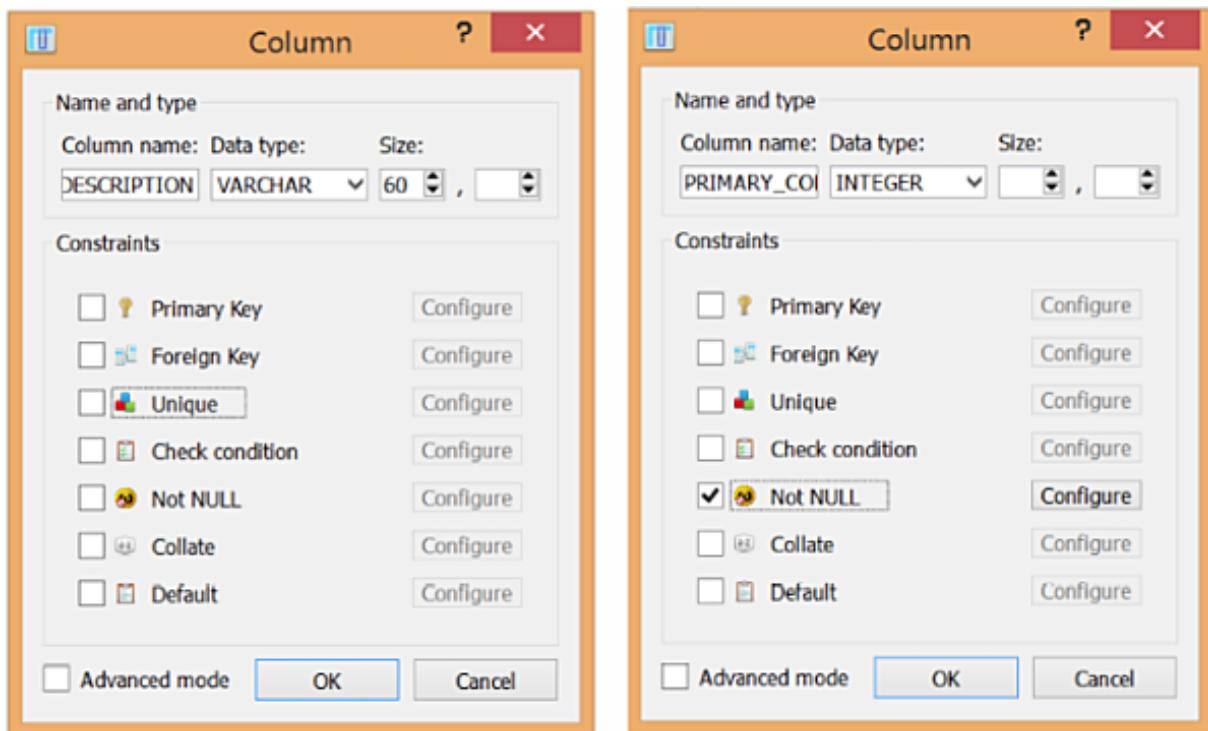
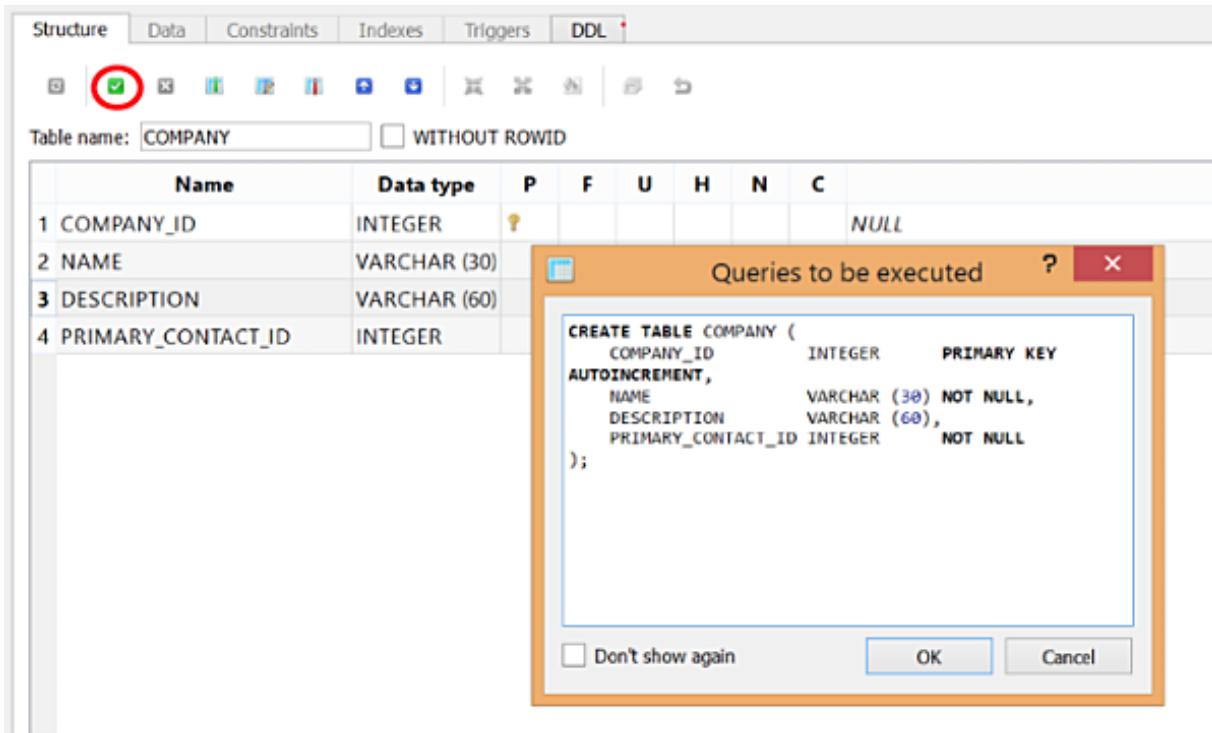


Figura 9.12 – Criando as duas colunas restantes.

Por fim, clique no botão Save Table. Deve ser exibida uma instrução CREATE TABLE construída pelo SQLiteStudio, que ele executará quando você a aprovar (Figura 9.13).



*Figura 9.13 – Clique no botão Save Table na barra de ferramentas superior e o SQLiteStudio apresentará a instrução CREATE TABLE que ele executará de acordo com as definições fornecidas.*

Não é ótimo? O SQLiteStudio escreveu a instrução SQL para você de acordo com as definições de tabela fornecidas. Antes de clicar em OK, examinaremos rapidamente a instrução CREATE TABLE para ver como ela funciona:

```

CREATE TABLE COMPANY (
    COMPANY_ID INTEGER PRIMARY KEY AUTOINCREMENT,
    NAME VARCHAR(30) NOT NULL,
    DESCRIPTION VARCHAR(60),
    PRIMARY_CONTACT_ID INTEGER NOT NULL
);

```

Se você examinar a consulta SQL, verá que a instrução CREATE TABLE declara uma nova tabela chamada COMPANY. Tudo que se encontra em parênteses depois disso define as colunas da tabela. Cada coluna é definida por um nome, seguido pelo tipo e por qualquer restrição ou regra existente como PRIMARY KEY, AUTOINCREMENT ou NOT NULL.

Você poderia copiar essa instrução e executá-la no editor de SQL, mas

apenas clique em OK e ele a executará. Em seguida, sua nova tabela deve aparecer no navegador (Figura 9.14).

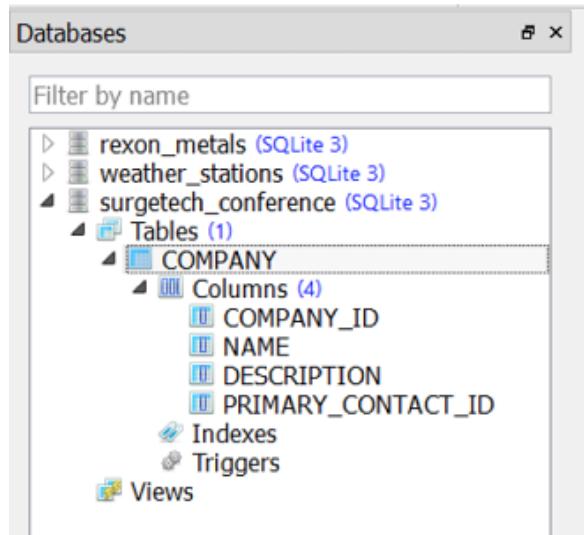


Figura 9.14 – Tabela COMPANY no navegador.



A restrição `AUTOINCREMENT` do SQLite não é necessária. Foi usada aqui para praticarmos o seu uso porque ela é necessária em outras plataformas, inclusive no MySQL. No SQLite, transformar uma coluna de tipo `INTEGER` em uma chave primária a fará manipular automaticamente sua própria atribuição de IDs. Na verdade, no SQLite é mais eficiente não usar `AUTOINCREMENT` e deixar a chave primária se encarregar do autoincremento implicitamente.

Crie as quatro tabelas restantes da mesma forma. A seguir temos as instruções `CREATE TABLE` (você pode optar por construir as tabelas usando a guia Structure ou pode executar as instruções `CREATE TABLE` textualmente no editor de SQL):

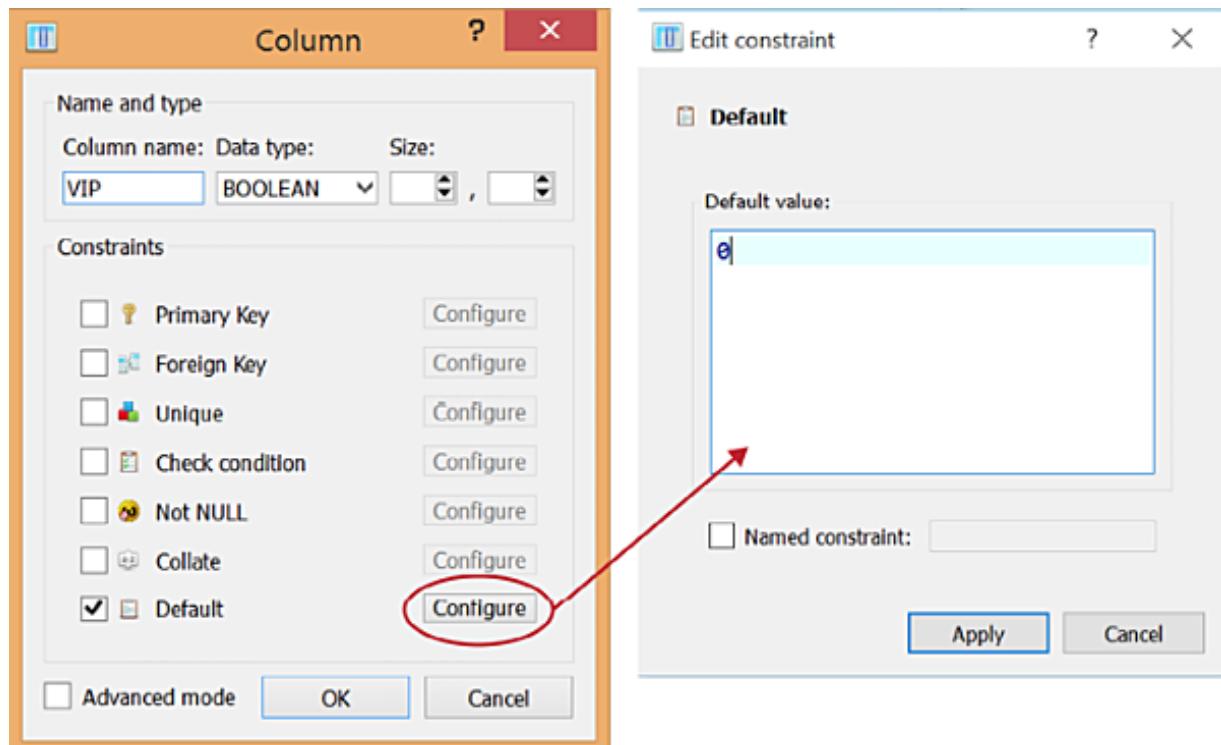
```
CREATE TABLE ROOM (
    ROOM_ID INTEGER PRIMARY KEY AUTOINCREMENT,
    FLOOR_NUMBER INTEGER NOT NULL,
    SEAT_CAPACITY INTEGER NOT NULL
);
CREATE TABLE PRESENTATION (
    PRESENTATION_ID INTEGER PRIMARY KEY AUTOINCREMENT,
    BOOKED_COMPANY_ID INTEGER NOT NULL,
    BOOKED_ROOM_ID INTEGER NOT NULL,
    START_TIME TIME,
    END_TIME TIME
);
```

```

CREATE TABLE ATTENDEE (
    ATTENDEE_ID INTEGER PRIMARY KEY AUTOINCREMENT,
    FIRST_NAME VARCHAR (30) NOT NULL,
    LAST_NAME VARCHAR (30) NOT NULL,
    PHONE INTEGER,
    EMAIL VARCHAR (30),
    VIP BOOLEAN DEFAULT (0)
);
CREATE TABLE PRESENTATION_ATTENDANCE (
    TICKET_ID INTEGER PRIMARY KEY AUTOINCREMENT,
    PRESENTATION_ID INTEGER,
    ATTENDEE_ID INTEGER
);

```

Observe que a tabela ATTENDEE tem um campo VIP que é um valor booleano (verdadeiro/falso). Por padrão, se um registro não especificar um valor para uma coluna, o valor será nulo. Pode ser uma boa ideia usar falso (0) como padrão para esse campo específico, se não for fornecido um valor. O fragmento SQL anterior reflete essa abordagem, mas você também pode defini-la no construtor de colunas como mostrado na Figura 9.15.



*Figura 9.15 – Definindo um valor-padrão para uma coluna.*

Todas as cinco tabelas foram criadas e tiveram suas restrições definidas, exceto as chaves externas (Figura 9.16).

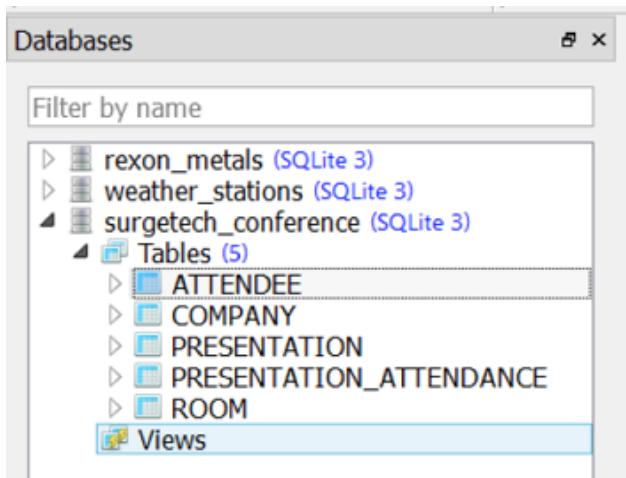


Figura 9.16 – Todas as tabelas foram construídas.



A maioria das soluções de banco de dados impõe valores a uma coluna somente pelo tipo de dado especificado. O SQLite não. No SQLite, você pode inserir um valor de tipo TEXT em uma coluna de tipo INTEGER. Outras soluções de banco de dados desaprovam isso. Embora pareça contraditório, os criadores do SQLite agiram dessa forma por razões técnicas que não fazem parte do escopo deste livro.

## Definindo as chaves externas

Há uma última tarefa que tornará nossas tabelas irretocáveis. Definimos as chaves primárias, mas não as chaves externas. Lembre-se de que a chave externa de uma tabela-filha está associada à chave primária de uma tabela-pai. Logicamente, não devemos nunca ter um valor de chave externa que não tenha um valor de chave primária correspondente.

Por exemplo, não pode haver um registro em PRESENTATION com um valor para BOOKED\_COMPANY\_ID que não exista na coluna COMPANY\_ID da tabela COMPANY. Se houver um valor igual a 5 para BOOKED\_COMPANY\_ID, é preciso que também haja em COMPANY um registro COMPANY\_ID igual a 5. Caso contrário, haverá um registro órfão. Podemos impor isso definindo restrições de chave externa.

Abra a tabela PRESENTATION e clique duas vezes na coluna BOOKED\_COMPANY\_ID para modificá-la (Figura 9.17). Marque Foreign Key e clique em Configure. Defina a tabela externa como CUSTOMER e a coluna externa como CUSTOMER\_ID.

Isso limitará BOOKED\_COMPANY\_ID somente aos valores da coluna CUSTOMER\_ID da tabela CUSTOMER. Clique em Apply e em OK.

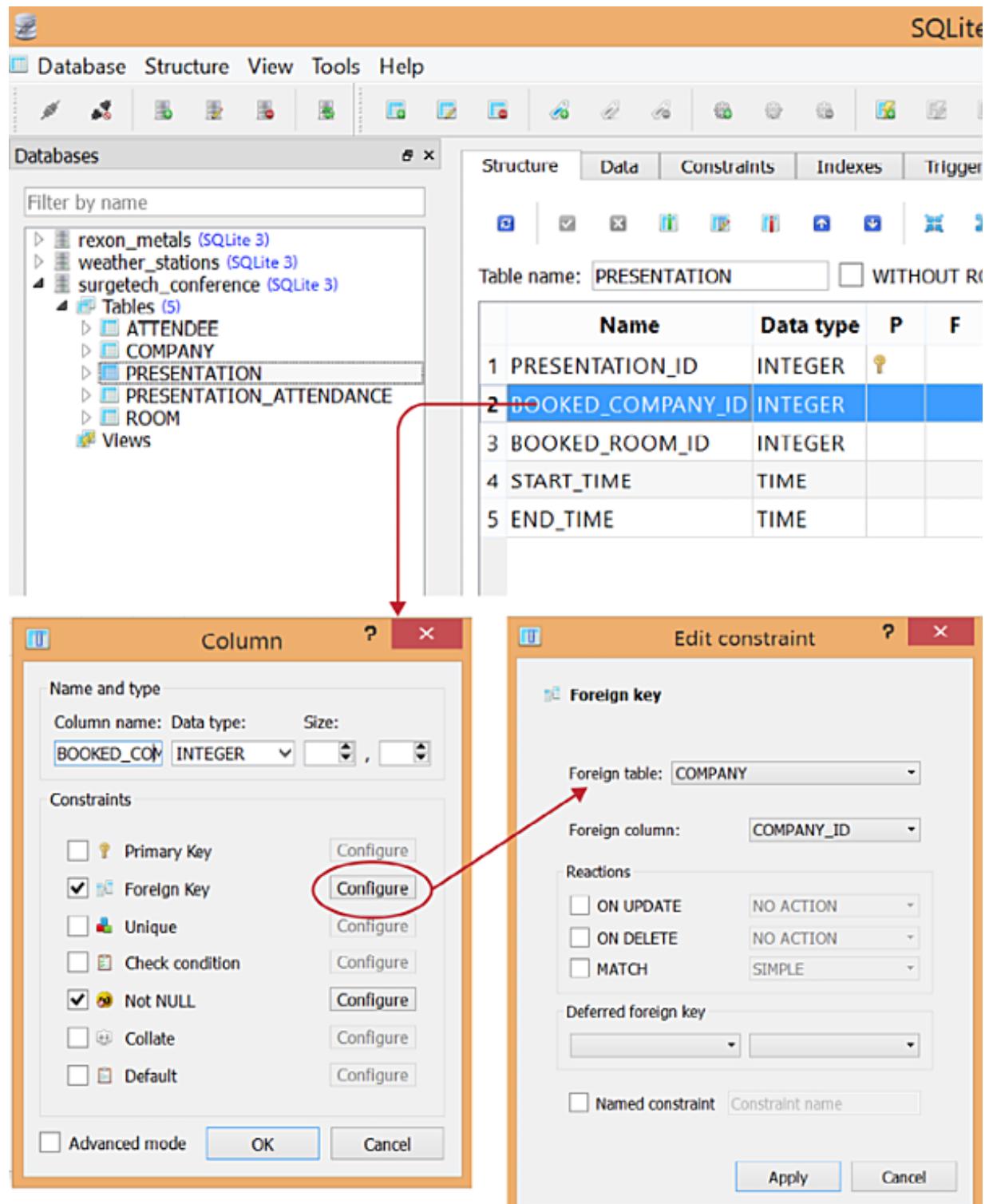


Figura 9.17 – Tornando BOOKED\_COMPANY\_ID a chave externa de

*COMPANY\_ID na tabela COMPANY.*

Clique no botão Commit Changes da guia Structure e uma série de instruções SQL será gerada para implementar a chave externa. Você pode examinar o código SQL se estiver curioso, mas ele só confirmará o trabalho que o SQLiteStudio realizou automaticamente. Em seguida, clique em OK para confirmar a alteração.

O uso de chaves externas dá integridade aos dados e impede que dados incorretos corrompam os relacionamentos. Devemos definir restrições de chave externa para todos os relacionamentos desse banco de dados a fim de que não haja nenhum registro órfão.

Agora você pode criar chaves externas para todos os relacionamentos pai-filho a seguir repetindo o mesmo procedimento:

Crie a chave externa de [Tabela].[Campo]	Chave primária no pai [Tabela].[Campo]
PRESENTATION.BOOKED_ROOM_ID	ROOM.ROOM_ID
PRESENTATION_ATTENDANCE.PRESENTATION_ID	PRESENTATION.PRESENTATION_ID
PRESENTATION_ATTENDANCE.ATTENDEE_ID	ATTENDEE.ATTENDEE_ID
COMPANY.PRIMARY_CONTACT_ATTENDEE_ID	ATTENDEE.ATTENDEE_ID

Asseguramos que cada registro-filho tenha um registro-pai e que nenhum órfão seja permitido no banco de dados.



Se em algum momento você usar o SQLite fora do SQLiteStudio, lembre-se de que a imposição da restrição de chave externa pode ter de ser ativada antes. O SQLiteStudio a ativa por padrão, mas outros ambientes do SQLite talvez não o façam.

## Criando views

Não é raro consultas SELECT muito usadas serem armazenadas em um banco de dados. Quando salvamos uma consulta em um banco de dados, ela se chama *view*. Uma view se comporta de forma semelhante a uma tabela. É possível executar instruções SELECT com uma view, que também pode ser associada a outras views e tabelas. Porém, os dados serão totalmente derivados da consulta SELECT que você especificou, logo, em

muitos casos não é possível (e nem faria sentido) modificá-los.

Suponhamos que executássemos uma consulta `SELECT` com muita frequência para obter uma visualização mais descritiva da tabela `PRESENTATION`, que trouxesse as informações da empresa cuja reserva foi feita e da sala reservada:

```
SELECT
COMPANY.NAME as BOOKED_COMPANY,
ROOM.ROOM_ID as ROOM_NUMBER,
ROOM.FLOOR_NUMBER as FLOOR,
ROOM.SEAT_CAPACITY as SEATS,
START_TIME,
END_TIME
FROM PRESENTATION
INNER JOIN COMPANY
ON PRESENTATION.BOOKED_COMPANY_ID = COMPANY.COMPANY_ID
INNER JOIN ROOM
ON PRESENTATION.BOOKED_ROOM_ID = ROOM.ROOM_ID
```

Agora suponhamos que quiséssemos armazenar essa consulta no banco de dados para que ela pudesse ser chamada facilmente. Podemos fazê-lo clicando com o botão direito do mouse no item Views do navegador e clicando em Create a view (Figura 9.18).

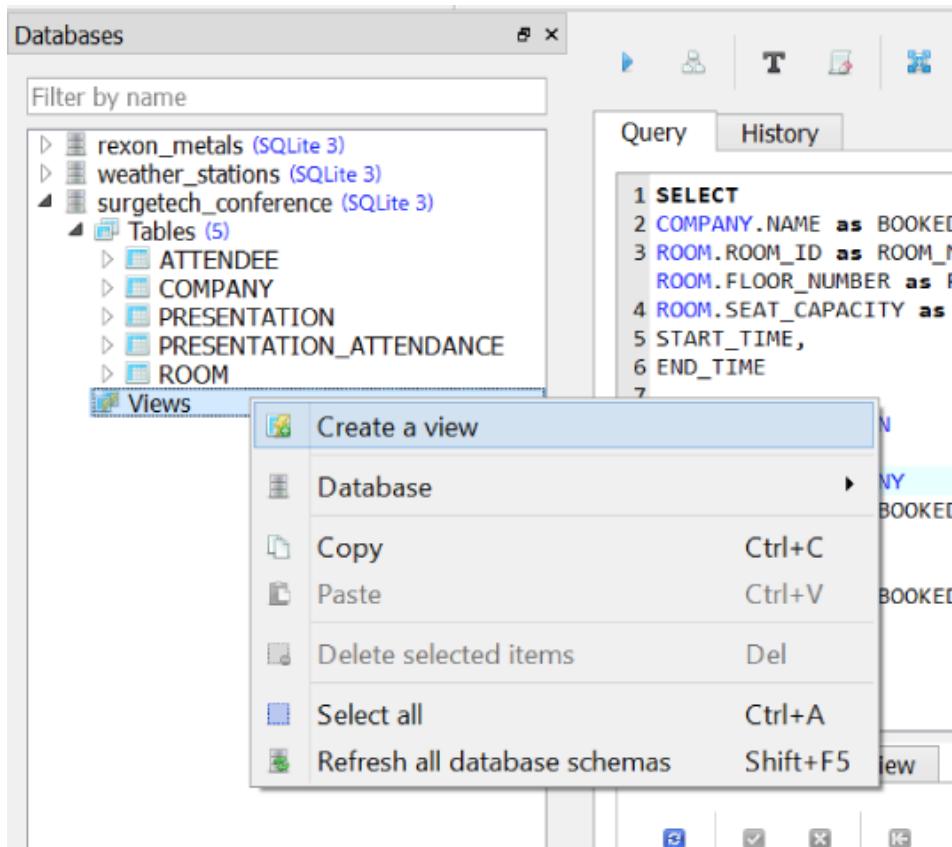


Figura 9.18 – Criando uma view.

Você será levado a uma janela de design de views (Figura 9.19). Navegue para a guia Query. Aqui você colará sua instrução SELECT. No campo “View name”, chame essa view de **PRESENTATION\_VW** (onde “VW” é a abreviação de “VIEW”) e clique na marca de seleção verde para salvá-la. Antes de executar a consulta SQL para criar a view, o SQLiteStudio a exibirá para revisão. Como mostrado a seguir, a sintaxe SQL que cria uma view é muito simples. Ela se apresenta na forma **CREATE VIEW [nome\_da\_view] AS [uma consulta SELECT]**.

Quando você clicar em OK, deve ver a view em seu navegador sob “Views” (Figura 9.20). Clique duas vezes nela e verá na guia Query a consulta que foi usada e na guia Data os resultados da consulta.

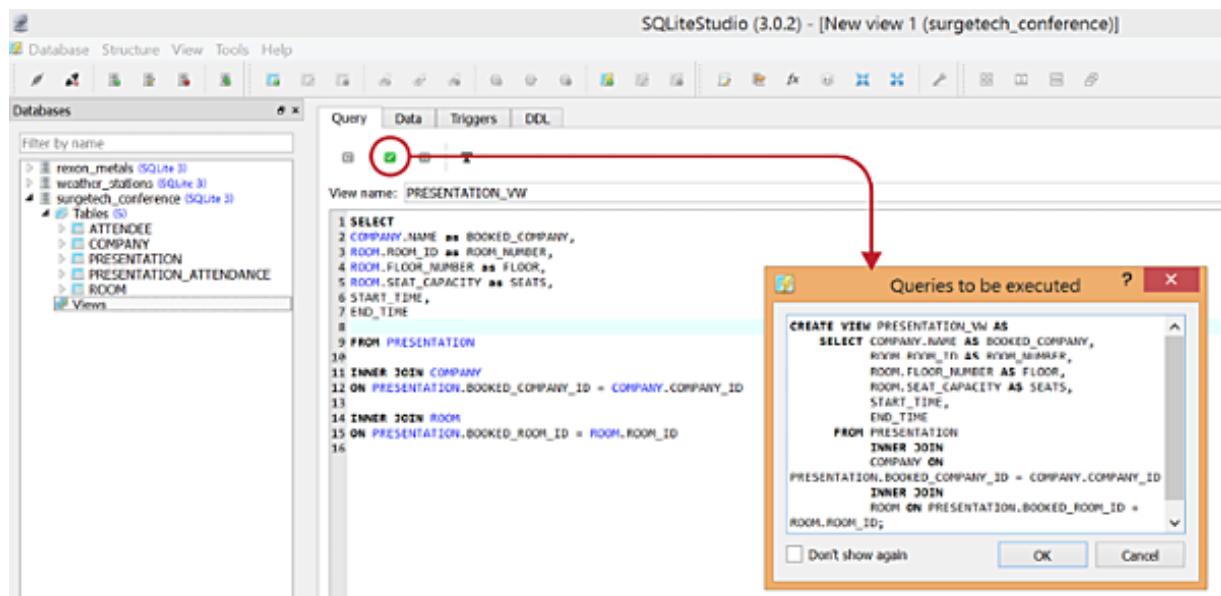


Figura 9.19 – Criando uma view a partir de uma consulta SELECT.

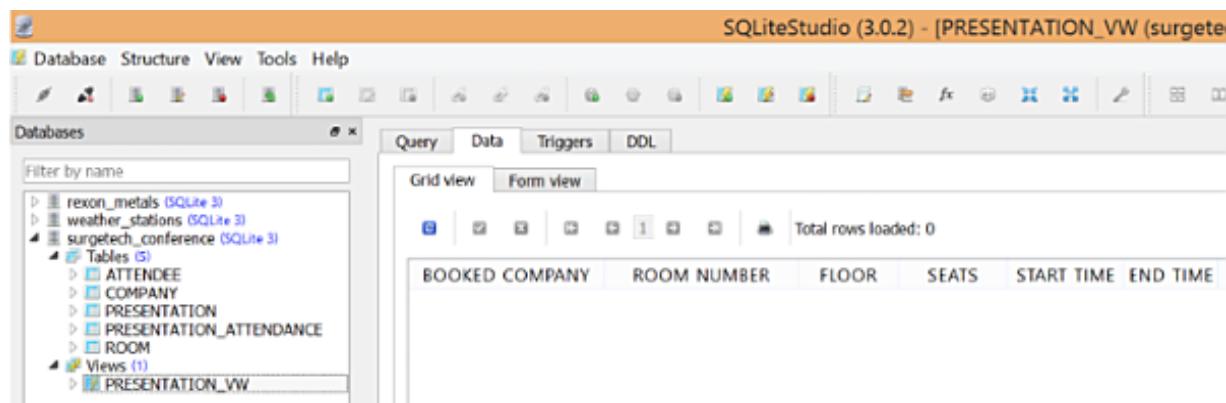


Figura 9.20 – Embora ainda não haja dados, a consulta SELECT foi salva como uma view chamada PRESENTATION\_VW.

A guia Data estará em branco, até as tabelas consultadas serem preenchidas com dados.

É bom ressaltar também que podemos consultar uma view como se fosse uma tabela (e aplicar filtros, executar operações de associação e fazer qualquer coisa que faríamos em uma instrução SELECT com uma tabela):

```

SELECT * FROM PRESENTATION_VW
WHERE SEAT_CAPACITY >= 30

```

## Resumo

Neste capítulo, nos aprofundamos na criação de nossos próprios bancos de dados e aprendemos como projetá-los eficientemente. Estudamos os relacionamentos entre as tabelas, que ajudam a definir claramente suas associações. Também examinamos algumas das diversas restrições de colunas (que incluem `PRIMARY KEY`, `FOREIGN KEY`, `NOT NULL`, `AUTOINCREMENT` e `DEFAULT`) para manter a consistência dos dados e assegurar que sigam as regras que definimos.

No próximo capítulo forneceremos e modificaremos os dados desse banco de dados. Testemunharemos nosso design em ação e veremos que valeu a pena o tempo investido em seu planejamento. Um bom design com restrições bem definidas gera um banco de dados resiliente.



Um tópico que este capítulo não abordou é o relativo aos índices. Os índices são úteis em tabelas com um grande número de registros, mas geram problemas de desempenho para as instruções `SELECT`. O “Apêndice B2 – melhorando o desempenho com indices”, discute os índices e quando eles devem ou não ser usados.

## CAPÍTULO 10

# Gerenciando dados

No capítulo anterior, aprendemos não só como criar um banco de dados, mas como criá-lo bem. Se planejarmos adequadamente o design das tabelas, as restrições de colunas e os relacionamentos, eles serão muito úteis quando começarmos a inserir dados nas tabelas. Com um design robusto para as tabelas e relacionamentos bem planejados entre elas, podemos criar associações de maneira fácil e eficiente. Quando um dado precisar ser alterado (por exemplo, o endereço de um cliente), ele só será modificado em um local em vez de em vários. Se um dado incorreto chegar, restrições de bloqueio suficientes devem impedi-lo de entrar no banco de dados.

Neste capítulo aprenderemos como inserir, excluir e atualizar registros. Felizmente, é muito mais fácil escrever esses tipos de operações que escrever instruções `SELECT`.

As operações de gravação em SQL não precisam ser executadas por uma pessoa. Com frequência, softwares (escritos em Java, Python ou outras linguagens de codificação) geram e executam instruções SQL de leitura e gravação da mesma forma que uma pessoa o faria, mas de maneira bem mais eficiente. Embora não façam parte do escopo do livro, as linguagens de codificação serão mencionadas no próximo capítulo, caso lhe interessem.

## **INSERT**

Em um banco de dados relacional só existem dados quando o banco de dados recebe registros. A instrução `INSERT` faz exatamente isso: insere um registro no banco de dados. Você pode selecionar que campos do registro

serão preenchidos e o resto será nulo ou usará um valor-padrão.

Primeiro, inseriremos um registro de participante no banco de dados da SurgeTech que criamos no último capítulo. Uma operação `INSERT` para você mesmo se incluir no banco de dados poderia ser semelhante à mostrada a seguir. Execute essa instrução com seu nome:

```
INSERT INTO ATTENDEE (FIRST_NAME, LAST_NAME)
VALUES ('Thomas','Nield')
```

Examinaremos a instrução por partes:

```
INSERT INTO ATTENDEE (FIRST_NAME, LAST_NAME)
VALUES ('Thomas','Nield')
```

Para começar, declaramos que estamos inserindo um registro na tabela `ATTENDEE`, e os campos que selecionamos para preencher são `FIRST_NAME` e `LAST_NAME`:

```
INSERT INTO ATTENDEE (FIRST_NAME, LAST_NAME)
VALUES ('Thomas','Nield')
```

Em seguida, especificamos os valores de cada um desses campos. É bom ressaltar que especificamos os valores na mesma ordem em que declaramos os campos: 'Thomas' corresponde a `FIRST_NAME` e 'Nield' a `LAST_NAME`.

Execute `SELECT * FROM ATTENDEE` para ver se a instrução `INSERT` foi bem-sucedida. Ótimo, o registro já existe (Figura 10.1).

The screenshot shows a database grid interface. At the top, there are two tabs: "Grid view" (selected) and "Form view". Below the tabs is a toolbar with various icons for database operations like insert, update, delete, and search. To the right of the toolbar, it says "Total rows loaded: 1". The main area is a grid table with the following columns: ATTENDEE ID, FIRST NAME, LAST NAME, PHONE, EMAIL, and VIP. There is one row of data: ATTENDEE ID is 1, FIRST NAME is Thomas, LAST NAME is Nield, PHONE is NULL, EMAIL is NULL, and VIP is 0.

ATTENDEE ID	FIRST NAME	LAST NAME	PHONE	EMAIL	VIP
1	Thomas	Nield	NULL	NULL	0

Figura 10.1 – Registro recém-inserido na tabela ATTENDEE.

Há várias coisas a serem observadas aqui. Não preenchemos todas as colunas em `INSERT`, mas, de acordo com as regras que criamos no capítulo anterior, algumas das colunas receberam um valor-padrão.

O campo `ATTENDEE_ID` se autoatribuiu o valor 1 devido às nossas regras

`PRIMARY KEY` e `AUTOINCREMENT`. Se inseríssemos outro registro, ele receberia automaticamente uma `ATTENDEE_ID` igual a 2, depois 3 seria atribuído, e assim por diante. Em uma instrução `INSERT`, você deve evitar preencher o campo `ATTENDEE_ID` por sua própria conta e deixar que ele atribua sua própria ID.



Como já vimos, a restrição `AUTOINCREMENT` do SQLite não é necessária. Porém, ela é requerida pelo MySQL e outras plataformas e é por isso que estamos praticando como usá-la. No SQLite, basta transformar uma coluna de tipo `INTEGER` em uma chave primária para que seja acionada a atribuição automática de IDs a novos registros.

`PHONE` e `EMAIL` não foram especificados em `INSERT`, logo, foram deixados nulos. Se uma dessas colunas tivesse uma restrição `NOT NULL` sem uma política de valor-padrão, `INSERT` teria falhado. Porém, em nosso design, permitimos que esses dois campos fossem nulos para o caso de os participantes preferirem manter-se em sigilo.

O status `VIP` também não foi especificado em `INSERT`, mas demos a esse campo um valor-padrão igual a falso (0). Portanto, em vez de torná-lo nulo, o SQLite recorreu ao uso do valor-padrão que especificamos.

Você já deve ter percebido que o design está funcionando eficientemente. Devido às políticas que definimos, as colunas recorreram aos valores-padrão quando não receberam outro valor.

## Inserções múltiplas

Se você tiver muitos registros para inserir, não é preciso inseri-los um de cada vez. É possível especificar vários registros em um único comando `INSERT`. Apenas repita a cláusula que vem após `VALUES` e separe cada entrada com uma vírgula:

```
INSERT INTO ATTENDEE (FIRST_NAME, LAST_NAME, PHONE, EMAIL, VIP)
VALUES
('Jon', 'Skeeter', 4802185842, 'john.skeeter@rex.net', 1),
('Sam', 'Scala', 2156783401, 'sam.scala@gmail.com', 0),
('Brittany', 'Fisher', 5932857296, 'brittany.fisher@outlook.com', 0)
```

Executar várias inserções dessa maneira é muito mais eficiente, principalmente se você tiver milhares de registros. Se um processo escrito

em Java ou Python estiver preenchendo uma tabela, ele deve usar essa sintaxe para inserir grandes quantidades de registros em vez de executar uma única instrução `INSERT` de cada vez. Caso contrário, a execução do processo pode ser muito lenta.

Também é possível inserir registros usando os resultados de uma consulta `SELECT`. Isso será útil se você precisar migrar dados de uma tabela para outra. Apenas certifique-se de que os campos de `SELECT` estejam alinhados com os de `INSERT`, que estejam na mesma ordem e que tenham os mesmos tipos de dados:

```
INSERT INTO ATTENDEE (FIRST_NAME, LAST_NAME, PHONE, EMAIL)
SELECT FIRST_NAME, LAST_NAME, PHONE, EMAIL
FROM SOME_OTHER_TABLE
```

## Testando as chaves externas

Aproveitaremos a oportunidade para ver outra política de nosso design em ação: as chaves externas.

Atualmente, devemos ter apenas quatro participantes com as IDs 1, 2, 3 e 4. Porém, teste essa funcionalidade inserindo um registro de empresa com o valor de `PRIMARY_CONTACT_ID` igual a 5:

```
INSERT INTO COMPANY (NAME, DESCRIPTION, PRIMARY_CONTACT_ID)
VALUES ('RexApp Solutions', 'A mobile app delivery service', 5)
```

Essa consulta deve falhar e uma mensagem de erro será exibida na parte inferior da janela (Figura 10.2).



Figura 10.2 – A restrição de chave externa funcionou e lançou um erro.

Isso é bom porque significa que a restrição de chave externa funcionou: ela não aceitou um registro órfão. `INSERT` precisa ter um valor de `PRIMARY_CONTACT_ID` que exista. Logo, se o alterarmos para 3 (Sam Scala), `INSERT` deve funcionar corretamente:

```
INSERT INTO COMPANY (NAME, DESCRIPTION, PRIMARY_CONTACT_ID)
```

```
VALUES ('RexApp Solutions', 'A mobile app delivery service', 3)
```

## DELETE

A instrução `DELETE` é perigosamente simples. Ela exclui todos os registros de uma tabela:

```
DELETE FROM ATTENDEE
```

No entanto, você pode excluir registros condicionalmente com uma instrução `WHERE`. Se quiséssemos remover todos os registros que não têm informações de contato, poderíamos filtrar os registros em que `PHONE` e `EMAIL` sejam nulos:

```
DELETE FROM ATTENDEE  
WHERE PHONE IS NULL  
AND EMAIL IS NULL
```

Já que é perigosamente fácil cometer erros com uma instrução `DELETE`, é boa prática substituí-la por `SELECT *`. A execução dessa consulta permite visualizar que registros serão excluídos:

```
SELECT * FROM ATTENDEE  
WHERE PHONE IS NULL  
AND EMAIL IS NULL
```

## TRUNCATE TABLE

Na seção anterior examinamos um meio de excluir todos os registros de uma tabela:

```
DELETE FROM ATTENDEE
```

Embora não seja usada no SQLite, em algumas plataformas de banco de dados (como no MySQL) a maneira preferida de excluir todos os registros de uma tabela é com `TRUNCATE TABLE`:

```
TRUNCATE TABLE ATTENDEE
```

O uso desse comando permite que o mecanismo de banco de dados redefina os autoincrementos de chaves primárias assim como de outros comportamentos de restrição. Também permite que sejam feitas algumas otimizações em segundo plano para a redefinição da tabela.

Embora o SQLite não dê suporte a `TRUNCATE TABLE`, ele permite algumas otimizações semelhantes na execução de `DELETE` sem `WHERE`.

## UPDATE

Para concluir, chegamos ao comando `UPDATE`. `UPDATE` modifica registros existentes. Se quiséssemos atualizar os valores de `EMAIL` de todos os registros para que ficassem em maiúsculas, poderíamos fazê-lo com essa instrução usando a função `UPPER()`:

```
UPDATE ATTENDEE SET EMAIL = UPPER(EMAIL)
```

Também podemos atualizar vários campos ao mesmo tempo. Basta separar cada expressão posterior à palavra-chave `SET` com uma vírgula. Para atualizar tanto o campo `FIRST_NAME` quanto o campo `LAST_NAME` para maiúsculas, execute este comando:

```
UPDATE ATTENDEE SET FIRST_NAME = UPPER(FIRST_NAME),  
LAST_NAME = UPPER(LAST_NAME)
```

Como ocorre com `DELETE`, podemos usar `WHERE` para aplicar condicionalmente atualizações a registros. Execute a consulta a seguir para configurar o campo `VIP` com verdadeiro onde `ATTENDEE_ID` for 3 ou 4:

```
UPDATE ATTENDEE SET VIP = 1  
WHERE ATTENDEE_ID IN (3,4)
```

## DROP TABLE

Pode ocorrer uma situação em que você queira remover uma tabela inteira do banco de dados. Apenas digite `DROP TABLE` seguido do nome da tabela que deseja excluir (essa também é uma instrução perigosa porque exclui a tabela permanentemente, logo, tome cuidado e tenha certeza do que está fazendo):

```
DROP TABLE MY_UNWANTED_TABLE
```

## Resumo

A essa altura você já tem as ferramentas necessárias para criar seu próprio banco de dados e gerenciar seus dados. Talvez tenha dúvidas sobre como

fazer tudo isso eficientemente ou sobre como fornecer meios práticos para os usuários inserirem e atualizarem dados, porque provavelmente você não vai poder ensinar SQL a todos eles, que vão querer uma interface gráfica de usuário. Ou talvez queira saber como introduzir grandes quantidades de dados em um banco de dados automaticamente ou como sincronizá-los com outra fonte de dados. Esses tópicos serão abordados resumidamente no último capítulo.

## CAPÍTULO 11

# Seguindo em frente

Se você alcançou este capítulo e seguiu todo o material até aqui, parabéns! Aprendeu um conjunto de habilidades adaptável e vendável que pode ser aplicado às áreas de negócios, tecnologia da informação e engenharia. Com a prática e o uso constantes, ficará à vontade para se posicionar profissionalmente no que diz respeito ao SQL e ao design de bancos de dados.

Quando se trata de entender e usar tecnologia, as pessoas têm níveis de ambição diferentes. Você pode ter lido este livro e considerar o material não essencial para seus objetivos profissionais, considerando-o apenas complementar. Pode ser esse o caso se você for um gerente interdepartamental generalista e só quiser conhecer TI um pouco melhor. O livro deve ter fornecido o insight necessário, o que é perfeitamente aceitável. Só você pode definir seus objetivos profissionais e priorizar o que aumenta seu valor!

No entanto, talvez o SQL tenha lhe empolgado e deixado dúvidas sobre como pode se aprofundar mais. Talvez queira ter um conhecimento maior de SQL e inteligência empresarial. Ou, talvez, queira criar softwares e interfaces gráficas de usuário que interajam com um banco de dados. Mesmo após ler este material você pode se sentir como se ainda faltasse algo e ter respostas não respondidas sobre como criar soluções empresariais completas. Caso se identifique, recomendando que continue pesquisando e perseguindo o conhecimento necessário para atingir seus objetivos.

Se você deseja aprender funcionalidades mais detalhadas fornecidas pelo SQL, há vários livros e recursos disponíveis. É possível expandir seu conhecimento em SQL com funcionalidades mais avançadas como

subconsultas e açãoadores. Se quiser obter um conhecimento mais profundo do SQLite, examine *Using SQLite* de Jay A. Kreibich (O'Reilly). Você também pode tentar conhecer outra plataforma de banco de dados, como o MySQL, com mais detalhes. Porém, o que quer que faça, não hesite em usar a internet como recurso. Ela tem um número infinito de tutoriais, guias, instruções passo a passo e documentação para ajudá-lo a expandir o conhecimento.

Embora o SQL por si só já aumente as oportunidades, você pode ser mais adaptável conhecendo uma ou duas tecnologias relevantes adicionais. Se estiver ansioso para aumentar seu conjunto de habilidades, considere aprender e integrar outro tópico. Isso abrirá várias oportunidades profissionais. Integrar algumas habilidades tecnológicas pode torná-lo ainda mais interessante que se especializar em um único conjunto de habilidades.

Python é uma ótima linguagem de programação, acessível para iniciantes e mesmo assim suficientemente poderosa para ser usada tanto por programadores profissionais quanto por hackers e seus testes de segurança. Você pode usar Python para processar dados em bancos de dados ou criar aplicativos e processos que deem suporte a um negócio. É uma linguagem de programação adaptável e se ajusta bem ao desenvolvimento de bancos de dados. Um ótimo recurso de introdução à linguagem Python é o popular livro de Al Sweigart, *Automate the Boring Stuff with Python* (No Starch Press).

A linguagem R pode ser usada na programação de inteligência empresarial. Trata-se de uma linguagem de programação estatística usada na execução de um profundo aprendizado de máquina e análise de dados. Percebi que ela é preferida por pessoas da área de negócios/ciências/matemática porque se presta muito bem a essas áreas. A linguagem possui funcionalidades para a análise de qualquer coisa, de sequências de DNA a tendências do mercado empresarial. Ela se sai muito bem na aplicação de modelos estatísticos clássicos como a regressão linear. Não usei muitos recursos que tratam da linguagem R, mas ouvi dizer que o Coursera (<https://www.coursera.org/>) oferece ótimos cursos

online sobre ela.

A adaptabilidade da linguagem Python está alcançando a da linguagem R, já que agora ela possui bibliotecas de data mining. As duas tecnologias são gratuitas e open source. Quando uma instrução `SELECT` não fornecer funcionalidade suficiente para uma dúvida que você tenha sobre seus dados, Python e R são ferramentas poderosas que poderão coletar essas informações.

Se você está interessado no processo integral de desenvolvimento de softwares e não apenas na criação casual de scripts, linguagens como Java, Swift (Apple) ou C# (Microsoft) podem ser boas opções. Com essas linguagens, é possível criar aplicativos móveis e soluções de software de negócios de nível comercial. Pode ser demorado dominar a programação plena e ela é desafiadora. No entanto, se você se destacar, as funções e as oportunidades serão intermináveis. Se quiser conhecer Java, o livro *Java: A Beginner's Guide*, de Herbert Schildt (McGraw Hill Professional) é um bom ponto de partida.

Esses definitivamente não são os únicos caminhos profissionais que você pode tomar. As necessidades tecnológicas são tão direcionadas atualmente que você pode acabar desempenhando uma função que nunca foi desempenhada antes. Apenas certifique-se de dar maior ênfase ao aprendizado de materiais que sejam pertinentes às suas necessidades imediatas. Além de livros, também há ótimos sites como o Pluralsight (<https://www.pluralsight.com/>) e o W3Schools (<http://www.w3schools.com/>) para a obtenção de conhecimentos básicos sobre qualquer tópico. E não subestime o Google quando tiver uma dúvida específica. Se estiver com dúvidas ou com problemas, provavelmente outra pessoa já passou por isso e a resposta foi postada online.

Se não conseguir encontrar uma resposta para a sua pergunta, há um ótimo site de perguntas e respostas chamado Stack Overflow (<http://stackoverflow.com/>) com profissionais e entusiastas de todas as áreas tecnológicas. Você deve postar uma pergunta bem definida e embasada e obterá respostas dos outros membros gratuitamente. Os colaboradores mais produtivos geralmente trabalham para o Google,

Oracle, Microsoft e outras grandes empresas de tecnologia. Alguns deles escreveram livros ou são celebridades do Vale do Silício que pertencem à comunidade tecnológica. Essas pessoas fornecem aconselhamento especializado apenas porque amam o que fazem.

Por fim, fique à vontade para me enviar emails se tiver alguma pergunta, preocupação ou comentário. Se quiser opinar sobre o livro ou tiver perguntas mais genéricas, farei o possível para ajudar. Meu email é *tmnield@outlook.com* – espero notícias.

Boas consultas!

## APÊNDICE A

# Operadores e funções

Este apêndice tem por objetivo ser uma fonte de consulta conveniente das operações e funções comuns usadas no SQLite. Embora o livro não examine todas as funcionalidades do SQLite, as que podem ter demanda imediata estão listadas aqui. O objetivo é demonstrar os conceitos do SQL aplicáveis à maioria das plataformas, em vez de ensinar as nuances da linguagem em detalhes.

Uma abordagem abrangente dos recursos do SQLite pode ser encontrada em <https://www.sqlite.org/docs.html>.

## Apêndice A1 – consultas com expressões de literais

Você pode testar os operadores e as funções facilmente sem consultar nenhuma tabela. Simplesmente use SELECT com uma expressão de literais como na consulta a seguir, que encontrará o valor 12:

```
SELECT 5 + 7
```

Outras funções e literais, inclusive strings de texto, também podem ser testados dessa maneira. A consulta a seguir verificará se a palavra 'TONY' existe na string 'TONY STARK' e deve retornar 1:

```
SELECT INSTR('TONY STARK', 'TONY')
```

Trata-se de uma ótima maneira de testar operadores e funções sem usar tabelas. Este apêndice mostrará muitos exemplos com essa abordagem e você pode usá-la em seus próprios testes.

## Apêndice A2 – operadores matemáticos

O SQLite tem um pequeno conjunto de operadores matemáticos básicos. Geralmente tarefas mais avançadas são executadas com funções, mas aqui estão os cinco operadores matemáticos básicos.

*Suponha  $x = 7$  e  $y = 3$*

Operador	Descrição	Exemplo	Resultado
+	Soma dois números	$x + y$	10
-	Subtrai dois números	$x - y$	4
*	Multiplica dois números	$x * y$	21
/	Divide dois números	$x / y$	2
%	Divide dois números, mas retorna o resto	$x \% y$	1

## Apêndice A3 – operadores de comparação

Os operadores de comparação geram um valor verdadeiro (1) ou falso (0) como resultado de uma avaliação comparativa.

*Suponha  $x = 5$  e  $y = 10$*

Operador	Descrição	Exemplo	Resultado
$= e ==$	Verifica se dois valores são iguais	$x = y$	0 (falso)
$!= e <>$	Verifica se dois valores são diferentes	$x != y$	1 (verdadeiro)
$>$	Verifica se o valor da esquerda é maior que o valor da direita	$x > y$	0 (falso)
$<$	Verifica se o valor da esquerda é menor que o valor da direita	$x < y$	1 (verdadeiro)
$\geq$	Verifica se o valor da esquerda é maior ou igual ao valor da direita	$x \geq y$	0 (falso)
$\leq$	Verifica se o valor da esquerda é menor ou igual ao valor da direita	$x \leq y$	1 (verdadeiro)

## Apêndice A4 – operadores lógicos

Os operadores lógicos nos permitem combinar expressões booleanas e executar outras operações condicionais.

*Suponha  $x = \text{verdadeiro} (1)$  e  $y = \text{falso} (0)$*

*Suponha  $a = 4$  e  $b = 10$*

Operador	Descrição	Exemplo	Resultado
AND	Verifica se todas as expressões booleanas são verdadeiras	$x \text{ AND } y$	0 (falso)
OR	Verifica se alguma expressão booleana é verdadeira	$x \text{ OR } y$	1 (verdadeiro)
BETWEEN	Verifica se um valor se encaixa inclusivamente dentro de um intervalo	$a \text{ BETWEEN } 1 \text{ and } b$	1 (verdadeiro)
IN	Verifica se um valor existe em uma lista de valores	$a \text{ IN } (1,5,6,7)$	0 (falso)
NOT	Nega e inverte o valor em uma expressão booleana	$a \text{ NOT IN } (1,5,6,7)$	1 (verdadeiro)
IS NULL	Verifica se um valor é nulo	$a \text{ IS NULL}$	0 (falso)
IS NOT NULL	Verifica se um valor não é nulo	$a \text{ IS NOT NULL}$	1 (verdadeiro)

## Apêndice A5 – operadores de texto

Há um conjunto limitado de operadores de texto, já que a maioria das tarefas processadoras de texto é executada com funções. No entanto, existem alguns. Lembre-se também de que as expressões regulares não fazem parte do escopo deste livro, mas é útil conhecê-las caso você venha a trabalhar intensamente com padrões de texto.

*Suponha  $\text{city} = \text{'Dallas'}$  e  $\text{state} = \text{'TX'}$*

Operador	Descrição	Exemplo	Resultado

Operador	Descrição	Exemplo	Resultado
	Concatena um ou mais valores convertendo-os em texto	city    ','    state	Dallas, TX
LIKE	Permite que os curingas _ e % representem padrões de texto	city LIKE 'D_l%'	1 (verdadeiro)
REGEXP	Procura um padrão de texto usando uma expressão regular	state REGEXP '[A-Z]{2}'	1 (verdadeiro)



Nota especial para programadores: REGEXP não é implementado diretamente no SQLite, logo, se usar o SQLite você pode ter de compilar ou implementar esse operador para seu aplicativo ou programa.

## Apêndice A6 – funções básicas comuns

O SQLite tem muitas funções básicas internas. Embora essa não seja uma lista abrangente, ela mostra as funções mais usadas. Uma lista completa das funções e sua documentação podem ser encontradas em [http://www.sqlite.org/lang\\_corefunc.html](http://www.sqlite.org/lang_corefunc.html).

Suponha  $x = -5$ ,  $y = 2$  e  $z = \text{NULL}$

Operador	Descrição	Exemplo	Resultado
abs()	Calcula o valor absoluto de um número	abs(x)	5
coalesce()	Converte um possível valor nulo em um valor-padrão se ele for realmente nulo	coalesce(z, y)	2
instr()	Verifica se uma string de texto contém outra string de texto; se contiver, retorna o índice da posição em que ela foi encontrada. Caso contrário, retorna 0	instr('HTX','TX')	2
length()	Fornece o número de caracteres de uma string	length('Dallas')	6
trim()	Remove espaços desnecessários nos dois lados de uma string	trim(' TX ')	TX
ltrim()	Remove espaços desnecessários no lado esquerdo de uma string	ltrim(' TX')	TX

Operador	Descrição	Exemplo	Resultado
rtrim()	Remove espaços desnecessários no lado direito de uma string	rtrim('LA ')	LA
random()	Retorna um número pseudoaleatório contido no intervalo que vai de -9223372036854775808 a +9223372036854775807	random()	7328249
round()	Arredonda um decimal para um número especificado de casas decimais	round(182.245, 2)	182.24
replace()	Substitui uma substring de texto contida em uma string por outra string	replace('Tom Nield','Tom', 'Thomas')	Thomas Nield
substr()	Extrai um intervalo de caracteres de uma string de acordo com suas posições numéricas	substr('DOG',2,3)	OG
lower()	Converte todas as letras de uma string para minúsculas	lower('DoG')	dog
upper()	Converte todas as letras de uma string para maiúsculas	upper('DoG')	DOG

## Apêndice A7 – funções de agregação

O SQLite tem um conjunto de funções de agregação que você pode usar com uma instrução GROUP BY para obter um escopo de agregação.

*X = coluna especificada para a agregação*

Função	Descrição
avg(X)	Calcula a média de todos os valores da coluna (omite valores nulos)
count(X)	Conta o número de valores não nulos da coluna
count(*)	Conta o número de registros
max(X)	Encontra o valor máximo da coluna (omite valores nulos)
min(X)	Encontra o valor mínimo da coluna (omite valores nulos)

Função	Descrição
sum(X)	Calcula a soma dos valores da coluna (omite valores nulos)
group_concat(X)	Concatena todos os valores não nulos da coluna. Você também pode fornecer um segundo argumento que especifique um separador, como a vírgula

## Apêndice A8 – funções de data e hora

A funcionalidade de data e hora em SQL varia muito entre as plataformas de banco de dados. Consequentemente, o livro não abordará o tópico fora deste apêndice. É preciso que você aprenda a sintaxe de data/hora de sua plataforma de banco de dados específica. Algumas plataformas, como a do MySQL, tornam o trabalho com valores de data/hora muito intuitivo, enquanto outras não são tão simples.

No que diz respeito ao SQLite, as funções de data/hora não podem ser abordadas de maneira abrangente aqui porque nos desviariamos do escopo do livro. No entanto, as tarefas de data e hora mais comuns serão abordadas nesta seção. A documentação completa da manipulação de data e hora no SQLite pode ser encontrada no site da plataforma ([http://www.sqlite.org/lang\\_datefunc.html](http://www.sqlite.org/lang_datefunc.html)).

## Funções de data

No trabalho com datas, é mais fácil armazená-las no formato de string AAAA-MM-DD já que a maioria das plataformas de banco de dados o entende (tecnicamente ele se chama formato ISO8601). Primeiro temos um ano de quatro dígitos, seguido por um mês de dois dígitos e um dia também de dois dígitos, e esses componentes são separados por um travessão (por exemplo, 2015-06-17). Se você formatar suas strings de data dessa forma, não precisará fazer conversões explícitas.

Quando uma consulta é executada, qualquer string com o formato 'AAAA-MM-DD' é interpretada como uma data. Isso significa que você pode executar tarefas cronológicas como comparar uma data com outra:

```
SELECT '2015-05-14' > '2015-01-12'
```

Se você não usar o formato ISO8601, o SQLite as comparará como strings de texto e verificará se o primeiro texto vem alfabeticamente antes do segundo. É claro que isso não é desejável já que você quer que as datas sejam avaliadas, comparadas e tratadas como datas.

Convenientemente, é possível obter a data de hoje passando a string 'now' para a função DATE():

```
SELECT DATE('now')
```

O SQLite também permite que você passe qualquer número de argumentos modificadores para a função DATE(). Por exemplo, é possível obter a data de ontem passando '-1 day' como argumento:

```
SELECT DATE('now','-1 day')
```

Também podemos passar uma string de data para a função DATE() e adicionar qualquer número de modificadores para transformar a data. Esse exemplo adicionará três meses a 2015-12-07 e subtrairá um dia:

```
SELECT DATE('2015-12-07','+3 month','-1 day')
```

Há várias transformações de data avançadas que você pode executar. Acesse o link de funções de data do SQLite fornecido no começo desta seção para obter uma visão geral dessas funcionalidades.

## Funções de hora

A hora também tem um formato típico, que é HH:MM:SS (também um padrão ISO8601). HH é um formato de hora militar de dois dígitos, MM é um formato de minutos com dois dígitos e SS representa um formato de segundos com dois dígitos. O separador é o símbolo de dois pontos. Se você formatar as horas dessa forma, as strings sempre serão interpretadas como valores de hora. A string a seguir representa o valor de hora de 4:31 PM e 15 segundos:

```
SELECT '16:31:15'
```

Você pode omitir o valor dos segundos se não estiver interessado nele. O SQLite o interpretará como 00:

```
SELECT '16:31'
```

Como ocorre com as datas, podemos executar várias operações com as horas, como comparar um valor de hora com outro:

```
SELECT '16:31' < '08:31'
```

A string 'now' também funciona com a função TIME() para a obtenção da hora atual:

```
SELECT TIME('now')
```

Como também é feito com as datas, você pode usar a função TIME() para executar transformações, como adicionar ou subtrair horas, minutos e segundos:

```
SELECT TIME('16:31','+1 minute')
```

## Funções de data/hora

Você pode ter uma data que também tenha um valor de hora. Como era de se esperar, o formato de string-padrão é composto pelos formatos de data e hora concatenados, separados por um espaço: ‘AAAA-MM-DD HH:MM:SS’. O SQLite reconhecerá uma string com esse formato como sendo um valor de data/hora:

```
SELECT '2015-12-13 16:04:11'
```

Todas as regras das funções DATE() e TIME() podem ser aplicadas à função DATETIME(). Você pode usar 'now', transformações e qualquer outra operação cronológica que aprendemos. Por exemplo, é possível subtrair um dia de um valor de data/hora e adicionar três horas:

```
SELECT DATETIME('2015-12-13 16:04:11','-1 day','+3 hour')
```

Há várias outras funções e recursos para o trabalho com datas e horas no SQLite, inclusive para a conversão de datas para formatos alternativos e para compensações referentes aos fusos horários. Também há o suporte à hora do Unix e ao sistema Juliano de numeração de dias. Como mencionado, acesse [http://www.sqlite.org/lang\\_datefunc.html](http://www.sqlite.org/lang_datefunc.html) para obter uma lista abrangente dessas funcionalidades.

## APÊNDICE B

# Tópicos suplementares

Há algumas tarefas relacionadas a bancos de dados e ao SQL que, quando este texto foi escrito, não se encaixavam no objetivo do livro. A intenção era que você examinasse os aspectos básicos do SQL sem ficar abarrotado de informações sobre cada funcionalidade disponível. Existem tópicos que, sem dúvida, não se encaixam na missão principal do livro, contudo também não merecem ser omitidos. Eles serão mencionados aqui e o ajudarão a aumentar sua proficiência.

## Apêndice B1 – outros tópicos de interesse

Este é um livro para iniciantes em SQL. Logo, o escopo e o enfoque foram restritos a tópicos básicos. Todavia, se você terminou de ler e deseja expandir seu conhecimento em bancos de dados e SQL, aqui estão algumas sugestões de tópicos que podem ser examinados e pesquisados:

Tópico	Descrição
UNION e UNION ALL	Acrescentam os resultados de duas ou mais consultas a um único conjunto de resultados
Subconsultas	Consultam outras consultas como se fossem tabelas
Índices	Melhoram o desempenho de SELECT em uma tabela com grandes volumes de dados (abordado brevemente no “Apêndice B2 – Melhorando o Desempenho com Índices”)
Transações	Executam várias instruções UPDATE/DELETE/INSERT como um único lote à prova de falhas (abordado brevemente no “Apêndice B3 – Transações”)
Acionadores	Reagem a instruções UPDATE/DELETE/INSERT e executam tarefas como a de logging e a de validação avançada de dados

Tópico	Descrição
Expressões regulares	Usam uma sintaxe universal para procurar facilmente padrões de texto avançados – semelhantes aos curingas das expressões LIKE, porém mais poderosas
Administração de bancos de dados	Adapta os bancos de dados de produção aos amplos ambientes corporativos

Você deve se deparar com vários outros tópicos, principalmente ao examinar as peculiaridades das diferentes plataformas de bancos de dados. No entanto, os sugeridos aqui contêm pistas que expandirão seu conhecimento em bancos de dados para além do que vimos no livro.

## Apêndice B2 – melhorando o desempenho com índices

À medida que seu banco de dados crescer, o desempenho pode ficar lento em consultas `SELECT`. A máquina precisará processar cada registro para encontrar os que atendem a condição `WHERE` e é claro que a existência de mais registros retardará esse processo.

Uma maneira comum de melhorar o desempenho significativamente é usando *índices*, mecanismos que permitem a execução de buscas com mais rapidez de modo muito semelhante a como funciona o índice de um livro. O índice acelera o desempenho de `SELECT`, mas retarda as instruções `INSERT`, `UPDATE` e `DELETE`. Ele também aumenta o arquivo do banco de dados. Esses são fatores que você precisa avaliar ao decidir se deve usá-los. Você não deve cogitar criar índices ao projetar um banco de dados. Faça-o posteriormente, se achar que está tendo problemas de desempenho.

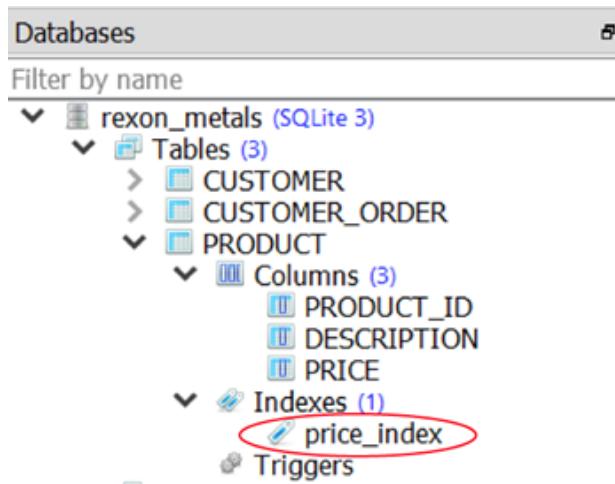
Os índices devem ser especificados em uma ou mais colunas e isso deve ocorrer nas colunas usadas com mais frequência para qualificar consultas. Por exemplo, se você consulta com frequência a tabela `PRODUCT` e usa uma instrução `WHERE` baseada na coluna `price`, pode aplicar um índice a essa coluna como mostrado aqui:

```
CREATE INDEX price_index ON PRODUCT(price);
```

O índice recebeu o nome `price_index`, foi aplicado à tabela `PRODUCT` e, entre parênteses, o especificamos para a coluna `price`. O SQLite manterá um

mapa de que registros têm quais valores de `price`. O desempenho melhorará significativamente com o uso de `price` como qualificador. Porém, é claro que, quando modificarmos registros, o índice precisará ser atualizado, logo, esse overhead retardará as operações `INSERT`, `UPDATE` e `DELETE`.

É possível ver no navegador de bancos de dados do SQLiteStudio que a tabela contém todos os objetos de índice criados (Figura B-1).



*Figura B.1 – O índice `price_index` foi adicionado aos índices da tabela `PRODUCT`.*

Você também pode criar um índice exclusivo (`UNIQUE INDEX`) para uma coluna que nunca tenha valores duplicados, e o SQLite fará otimizações especiais para esse caso:

```
CREATE UNIQUE INDEX name_index ON CUSTOMER(name);
```

É possível usar índices compostos se dois ou mais campos forem normalmente usados juntos em uma qualificação, mas esse recurso não faz parte do escopo do livro.

Para remover um índice, basta executar uma instrução `DROP INDEX` com o nome do índice:

```
DROP INDEX price_index;
```

Apenas como lembrete, os índices só devem ser usados em tabelas muito grandes que reconhecidamente tenham problemas de desempenho com instruções `SELECT`. Você deve evitar usar índices em tabelas pequenas

porque, na verdade, o overhead piorará o desempenho (o que significa que esse exemplo foi apenas uma demonstração e não é algo que deva ser feito com a tabela `PRODUCT`). Também devemos evitar usar índices em tabelas que tenham atualizações volumosas e frequentes.

## Apêndice B3 – transações

Em algumas situações, você pode querer executar várias instruções `INSERT`, `UPDATE` ou `DELETE` como um lote, contanto que todas sejam concluídas com sucesso, e se uma falhar, que todas falhem. Isso é conhecido como *atomicidade*, ou seja, todas as ações devem ser bem-sucedidas ou nenhuma será executada.

Um bom exemplo em que esse tipo de comportamento é necessário é em transações financeiras, como em transferências entre contas bancárias ou em serviços de pagamento como o do PayPal. Quando tiramos dinheiro de uma conta e o colocamos em outra, precisamos nos certificar de que as duas operações sejam bem-sucedidas.

Observe essas duas instruções `INSERT` que retiram 187.56 dólares de uma conta e os depositam em outra:

```
INSERT INTO ACCOUNT_ACTIVITY (ACCOUNT_ID,AMOUNT) VALUES (1563,-187.56);
INSERT INTO ACCOUNT_ACTIVITY (ACCOUNT_ID,AMOUNT) VALUES (3067,187.56);
```

O que aconteceria se a primeira instrução `INSERT` fosse bem-sucedida, mas a segunda falhasse? Bem, os 187.56 dólares desapareceriam. Você teria então dois clientes insatisfeitos e uma possível auditoria. Porém, como assegurar que, em caso de falha, esse dinheiro retorne para o cliente que o pagou e tudo volte ao estado em que se encontrava?

A resposta é usando uma *transação*. Com uma transação você pode tornar essa transferência atômica e executar uma reversão (`ROLLBACK`), se algo falhar, e uma confirmação (`COMMIT`), se tudo der certo.

Primeiro, chame o comando `BEGIN` ou `BEGIN TRANSACTION` (são comandos iguais):

```
BEGIN TRANSACTION;
```

Agora qualquer instrução `INSERT`, `UPDATE` e `DELETE` será registrada para que

possa ser cancelada, se necessário. Execute as duas instruções `INSERT`. As ações serão executadas ao mesmo tempo em que são registradas no “modo de transação”:

```
INSERT INTO ACCOUNT_ACTIVITY (ACCOUNT_ID,AMOUNT) VALUES (1563,-187.56);
```

```
INSERT INTO ACCOUNT_ACTIVITY (ACCOUNT_ID,AMOUNT) VALUES (3067,187.56);
```

Se tudo der certo e não ocorrerem erros, você poderá chamar `COMMIT` ou seu alias, `END TRANSACTION`, para finalizar as instruções `INSERT`. A transferência foi executada com sucesso.

Agora começaremos outra transação para fazer mais uma transferência:

```
BEGIN TRANSACTION;
```

No entanto, dessa vez vamos invalidá-la. Digamos que fizéssemos outra transferência entre essas duas contas:

```
INSERT INTO ACCOUNT_ACTIVITY (ACCOUNT_ID,AMOUNT) VALUES (1563,-121.36);
```

```
INSERT INTO ACCOUNT_ACTIVITY (ACCOUNT_ID,AMOUNT) VALUES (121.36);
```

A primeira instrução SQL será bem-sucedida, mas a segunda está incorreta e causará erro. Está faltando o valor de `ACCOUNT_ID` e agora temos 121.36 dólares no limbo.

Felizmente, estamos no “modo de transação”. Podemos pressionar um botão desfazer e chamar `ROLLBACK`. Isso desfará tudo desde a última instrução `COMMIT` ou `BEGIN TRANSACTION` que chamamos. A primeira `INSERT` será desfeita e os 121.36 dólares retornarão à conta 1563.

No caso de uma conexão de banco de dados falhar, uma instrução SQL ser composta incorretamente ou uma regra de validação impedir a execução de uma série de atualizações, as transações são uma maneira de assegurar que os dados não fiquem corrompidos em seu banco de dados. Você vai querer usá-las principalmente com processos automatizados ou jobs maiores que demandem a inserção, a atualização e a exclusão de um grande volume de registros vulneráveis.

# Sobre o autor

**Thomas Nield** é um profissional da área de análise de negócios que trabalha no gerenciamento de receitas da Southwest Airlines. No começo de sua carreira, apaixonou-se por tecnologia e comprou vários livros para dominar a programação em Java, C# e o design de bancos de dados. Ele adora compartilhar o que aprende e ensinar novos conjuntos de habilidades para outras pessoas, mesmo quando não trabalham em TI. Também aprecia tornar conteúdo técnico inteligível e relevante para quem não conhece o assunto ou não se sente à vontade com ele.

# Colofão

O animal da capa de *Introdução à linguagem SQL* é um sapo-corredor (*Epidalea calamita*). Ele pertence à família Bufonidae e pode ser encontrado em ambientes arenosos, em charnecas e nos litorais, onde há pouca cobertura de grama, por toda a Europa Ocidental.

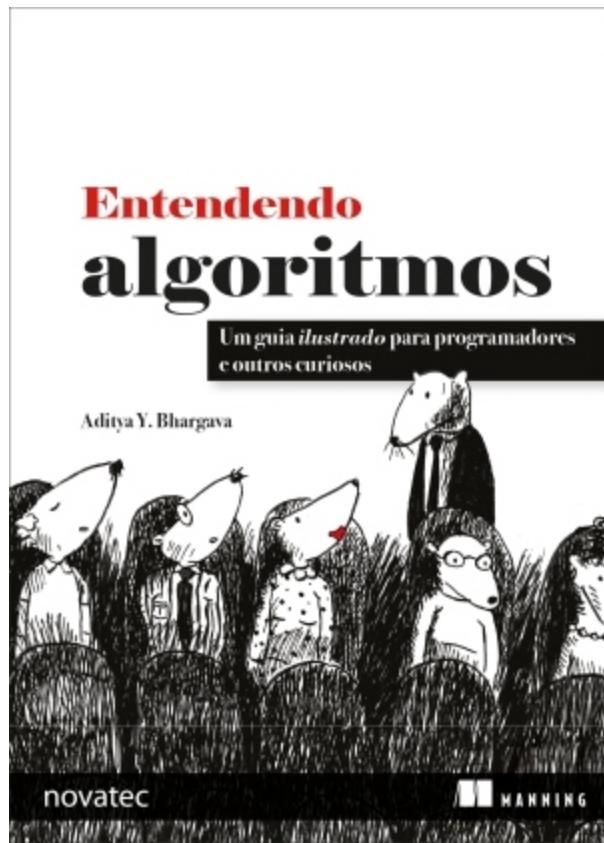
Uma característica que identifica os sapos-corredores é a linha amarela que cruza o meio de seu dorso. O tamanho dos adultos varia de 50 a 70 mm, com as fêmeas sendo maiores que os machos. A coloração em geral é marrom, creme ou verde e eles são cobertos por verrugas, como seus irmãos, os sapos-comuns. Outra característica distintiva são suas pernas posteriores mais curtas, o que o torna mais hábil para correr que para saltar ou caminhar, como é o caso dos outros sapos.

Como ocorre com os sapos mais comuns, a dieta do sapo-corredor é composta por insetos invertebrados como besouros, aranhas e minhocas. Eles são caçadores noturnos, logo, as refeições são todas feitas à noite. Os que vivem em dunas arenosas também comem pequenos crustáceos, como o camarão de água-doce. Os sapos-corredores liberam toxinas de sua pele, o que evita que se tornem boas refeições, mas pássaros como as garças-reais-europeias e as cobras-de-água-de-colar podem devorá-los sem problemas.

Os rituais de acasalamento do sapo-corredor também são noturnos. Os machos têm um alto e distinto chamado de acasalamento que sinaliza às fêmeas para irem para águas próximas, quentes e rasas (porque os sapos-corredores são péssimos nadadores). Geralmente isso ocorre entre os meses de abril e julho, com as fêmeas colocando de 1.500 a 7.500 ovos. Cerca de uma semana após a fertilização os ovos se transformam em girinos que, após 3 a 8 semanas, se transformam em pequenos sapinhos.

Muitos dos animais das capas da O'Reilly estão em perigo de extinção; todos eles são importantes para o mundo. Para saber mais sobre como

você pode ajudar, acesse [animals.oreilly.com](http://animals.oreilly.com). A imagem da capa foi tirada de *Johnson's Natural History*.



# Entendendo Algoritmos

Bhargava, Aditya Y.

9788575226629

264 páginas

[Compre agora e leia](#)

Um guia ilustrado para programadores e outros curiosos. Um algoritmo nada mais é do que um procedimento passo a passo para a resolução de um problema. Os algoritmos que você mais utilizará como um programador já foram descobertos, testados e provados.

Se você quer entendê-los, mas se recusa a estudar páginas e mais páginas de provas, este é o livro certo. Este guia cativante e completamente ilustrado torna simples aprender como utilizar os principais algoritmos nos seus programas. O livro Entendendo Algoritmos apresenta uma abordagem agradável para esse tópico essencial da ciência da computação. Nele, você aprenderá como aplicar algoritmos comuns nos problemas de programação enfrentados diariamente. Você começará com tarefas básicas como a ordenação e a pesquisa. Com a prática, você enfrentará problemas mais complexos, como a compressão de dados e a inteligência artificial. Cada exemplo é apresentado em detalhes e inclui diagramas e códigos completos em Python. Ao final deste livro, você terá dominado algoritmos amplamente aplicáveis e saberá quando e onde utilizá-los. O que este livro inclui A abordagem de algoritmos de pesquisa, ordenação e algoritmos gráficos Mais de 400 imagens com descrições detalhadas Comparações de desempenho entre algoritmos Exemplos de código em Python Este livro de fácil leitura e repleto de imagens é destinado a programadores autodidatas, engenheiros ou pessoas que gostariam de recordar o assunto.

[Compre agora e leia](#)



# Estruturas de dados e algoritmos com JavaScript

Groner, Loiane

9788575227282

408 páginas

[Compre agora e leia](#)

Uma estrutura de dados é uma maneira particular de organizar dados em um computador com o intuito de usar os recursos de

modo eficaz. As estruturas de dados e os algoritmos são a base de todas as soluções para qualquer problema de programação. Com este livro, você aprenderá a escrever códigos complexos e eficazes usando os recursos mais recentes da ES 2017. O livro Estruturas de dados e algoritmos com JavaScript começa abordando o básico sobre JavaScript e apresenta a ECMAScript 2017, antes de passar gradualmente para as estruturas de dados mais importantes, como arrays, filas, pilhas e listas ligadas. Você adquirirá um conhecimento profundo sobre como as tabelas hash e as estruturas de dados para conjuntos funcionam, assim como de que modo as árvores e os mapas hash podem ser usados para buscar arquivos em um disco rígido ou para representar um banco de dados. Este livro serve como um caminho para você mergulhar mais fundo no JavaScript. Você também terá uma melhor compreensão de como e por que os grafos – uma das estruturas de dados mais complexas que há – são amplamente usados em sistemas de navegação por GPS e em redes sociais. Próximo ao final do livro, você descobrirá como todas as teorias apresentadas podem ser aplicadas para solucionar problemas do mundo real, trabalhando com as próprias redes de computador e com pesquisas no Facebook. Você aprenderá a:

- declarar, inicializar, adicionar e remover itens de arrays, pilhas e filas;
- criar e usar listas ligadas, duplamente ligadas e ligadas circulares;
- armazenar elementos únicos em tabelas hash, dicionários e conjuntos;
- explorar o uso de árvores binárias e árvores binárias de busca;
- ordenar estruturas de dados usando algoritmos como bubble sort, selection sort, insertion sort, merge sort e quick sort;
- pesquisar elementos em estruturas de dados usando ordenação sequencial e busca binária

[Compre agora e leia](#)



# Fundamentos de HTML5 e CSS3

Silva, Maurício Samy

9788575227084

304 páginas

[Compre agora e leia](#)

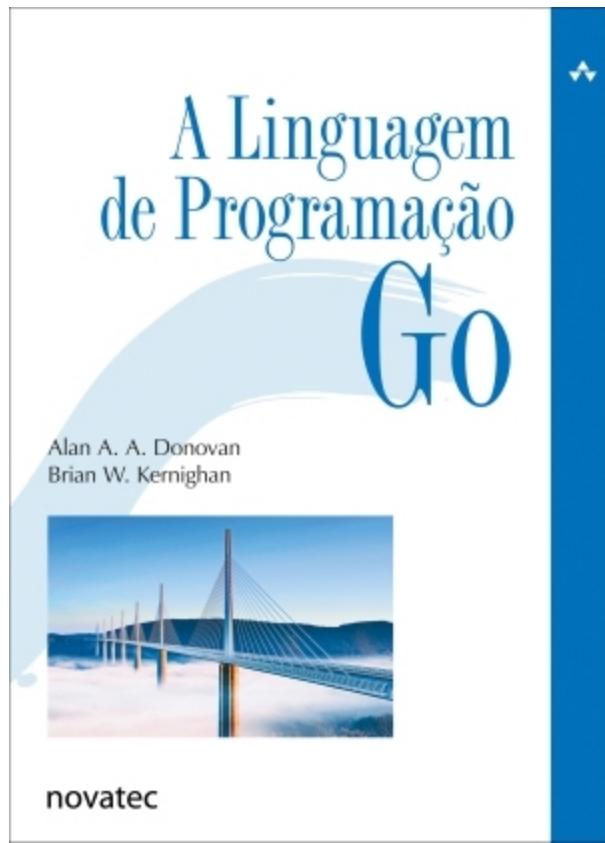
Fundamentos de HTML5 e CSS3 tem o objetivo de fornecer aos iniciantes e estudantes da área de desenvolvimento web conceitos básicos e fundamentos da marcação HTML e estilização CSS, para a criação de sites, interfaces gráficas e aplicações para a web.

Maujor aborda as funcionalidades da HTML5 e das CSS3 de forma clara, em linguagem didática, mostrando vários exemplos práticos em funcionamento no site do livro. Mesmo sem conhecimento prévio, com este livro o leitor será capaz de:

- Criar um código totalmente semântico empregando os elementos da linguagem HTML5.
- Usar os atributos da linguagem HTML5 para criar elementos gráficos ricos no desenvolvimento de aplicações web.
- Inserir mídia sem dependência de plugins de terceiros ou extensões proprietárias.
- Desenvolver formulários altamente interativos com validação no lado do cliente utilizando atributos criados especialmente para essas finalidades.
- Conhecer os mecanismos de aplicação de estilos, sua sintaxe, suas propriedades básicas, esquemas de posicionamento, valores e unidades CSS3.
- Usar as propriedades avançadas das CSS3 para aplicação de

fundos, bordas, sombras, cores e opacidade. •Desenvolver layouts simples com uso das CSS3.

[Compre agora e leia](#)



# A Linguagem de Programação Go

Donovan, Alan A. A.

9788575226551

480 páginas

[Compre agora e leia](#)

A linguagem de programação Go é a fonte mais confiável para qualquer programador que queira conhecer Go. O livro mostra como escrever código claro e idiomático em Go para resolver problemas do mundo real. Esta obra não pressupõe conhecimentos prévios de

Go nem experiência com qualquer linguagem específica, portanto você a achará acessível, independentemente de se sentir mais à vontade com JavaScript, Ruby, Python, Java ou C++.

[Compre agora e leia](#)



# Pense em Python

Downey, Allen B.

9788575227503

312 páginas

[Compre agora e leia](#)

Se você quer aprender como programar, usar Python é uma ótima forma de começar. Este guia prático apresenta a linguagem passo a passo, começando com conceitos de programação básicos antes de chegar a funções, recursividade, estruturas de dados e design orientado a objeto. Esta edição e seu código de apoio foram atualizados para o Python 3. Com os exercícios em cada capítulo, você testará conceitos de programação conforme os aprende.

Pense em Python é ideal para estudantes de ensino médio e universitários ou para autodidatas, estudantes educados em casa e profissionais que precisam aprender fundamentos de programação. Os principiantes que queiram apenas ter uma noção básica, podem começar a usar Python em um navegador.

- Comece com o básico, incluindo sintaxe e semântica da linguagem.
- Tenha uma definição clara de cada conceito de programação.
- Aprenda sobre valores, variáveis, instruções, funções e estruturas de dados em uma progressão lógica.
- Descubra como trabalhar com arquivos e bancos de dados.
- Entenda objetos, métodos e programação orientada a objeto.
- Use técnicas de depuração para corrigir erros

de sintaxe, tempo de execução e semânticos. •Explore funções, estruturas de dados e algoritmos com uma série de estudos de caso. O código de exemplo deste livro é mantido em um repositório GitHub público, no qual os usuários podem carregá-lo e modificá-lo facilmente.

[Compre agora e leia](#)