Ben Forta

SQL em 10 Minutos por Dia

novatec

SQL em 10 Minutos por Dia

Ben Forta

SAMS
Novatec

Editor: Rubens Prates
Tradução: Cláudio José Adas
Revisão gramatical: Tássia Carvalho
Imagem da capa: mickyteam/Shutterstock

ISBN do ebook: 978-65-86057-45-4

ISBN do impresso: 978-65-86057-44-7

Histórico de impressões:

Março/2021 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

Sumário

Sobre o autor

Ben Forta é diretor sênior de Iniciativas Educacionais da Adobe e tem três décadas de experiência no setor de computadores nas áreas de desenvolvimento e marketing de produto, suporte, treinamento. Ele é o autor do campeão de vendas *Sams Teach Yourself SQL in 10 Minutes* [*SQL em 10 Minutos por Dia*] (incluindo títulos derivados sobre MariaDB, MySQL, SQL Server T-SQL e Oracle PL/SQL), *Learning Regular Expressions* [*Aprendendo expressões regulares*], bem como livros sobre Java, Windows e muito mais. Ele tem ampla experiência em projeto e desenvolvimento de banco de dados, implementou banco de dados para vários programas de software comerciais e sites altamente bem-sucedidos, e é um palestrante e colunista frequente sobre desenvolvimento de aplicações e tecnologias de Internet. Ben vive em Oak Park, Michigan, com a esposa, Marcy, e os filhos. Fique à vontade para enviar-lhe um e-mail (em inglês) no endereço `ben@forta.com` e visitar seu site no endereço `http://forta.com`.

Agradecimentos

Obrigado à equipe da Sams por todos esses anos de apoio, dedicação e incentivo. Nas últimas duas décadas criamos mais de 40 livros juntos, mas este pequeno volume é de longe o meu favorito, e agradeço a vocês pela liberdade criativa para desenvolvê-lo da forma como achei melhor.

Obrigado aos revisores da Amazon.com que sugeriram a inclusão dos Desafios, uma novidade nesta quinta edição.

Obrigado aos milhares de vocês que enviaram comentários sobre as primeiras quatro edições (em inglês) deste livro. Felizmente, a maior parte deles foi positiva e tudo foi levado em consideração. Os aprimoramentos e as alterações nesta edição são uma resposta direta aos seus comentários, que continuo recebendo com satisfação.

Obrigado às dezenas de faculdades e universidades que incluíram este livro em seus currículos de TI e ciências da computação. Ser incluído e ganhar a confiança de professores dessa maneira é imensamente gratificante e ao mesmo tempo nos torna humildes.

E, finalmente, obrigado a quase meio milhão de vocês que compraram as edições anteriores (e as derivadas) deste livro, tornando-o não apenas meu título mais vendido, mas também o livro mais vendido sobre o assunto. Seu apoio contínuo é o maior elogio que um autor pode receber.

– Ben Forta

Introdução

SQL é a linguagem de banco de dados mais usada. Seja você um desenvolvedor de aplicações, um administrador de banco de dados, um projetista de aplicações web, um desenvolvedor de aplicativos móveis ou um usuário de ferramentas populares de relatório de dados, um bom conhecimento prático de SQL é uma parte importante da interação com os bancos de dados.

Este livro nasceu por necessidade. Eu vinha ensinando o desenvolvimento de aplicativos web há vários anos e os alunos estavam constantemente pedindo recomendações de livros sobre SQL. Existem muitos livros sobre SQL no mercado. Alguns são realmente muito bons. Mas todos eles têm algo em comum: para a maioria dos usuários, eles fornecem informação demais. Em vez de ensinar o próprio SQL, a maioria dos livros ensina tudo, desde design e normalização de banco de dados até teoria de banco de dados relacional e questões administrativas. E, embora esses sejam todos tópicos importantes, eles não são de interesse para a maioria de nós, que apenas precisa aprender SQL.

Assim, não encontrando um único livro que eu me sentisse confortável em recomendar, transformei essa experiência em sala de aula no livro que você tem em mãos. *SQL em 10 Minutos por Dia* lhe ensinará o SQL que você precisa conhecer, começando com a consulta simples de dados e avançando para tópicos mais complexos, incluindo o uso de junção (*join*) de dados, subconsultas, procedimentos armazenados (*stored procedures*), cursores, gatilhos (*triggers*) e restrições (*constraints*) de tabela. Você aprenderá de forma metódica, sistemática e simples – por meio de lições que levarão 10 minutos ou menos para serem concluídas.

Agora, em sua quinta edição, este livro ensinou SQL a quase meio milhão de usuários falantes de inglês e foi traduzido para mais de uma dúzia de outros idiomas, de modo a ajudar os usuários em todo o mundo.

Uma novidade desta edição é a inclusão de desafios específicos da lição no final de cada lição 2-18. Eles oferecem uma chance de usar o SQL que você aprendeu e aplicá-lo a diferentes cenários e problemas. As respostas para cada desafio estão no Apêndice E.

Agora é a sua vez. Vá para a lição 1 e comece a trabalhar. Você escreverá SQL de classe mundial em pouco tempo.

A quem se destina o livro "SQL em 10 Minutos por Dia?

Este livro é destinado a você se

- For iniciante em SQL.
- Quiser aprender rapidamente como tirar o máximo proveito do SQL.
- Quiser aprender como usar SQL no desenvolvimento de seus próprios aplicativos.
- Quiser ser produtivo com rapidez e facilidade em SQL sem precisar chamar alguém para obter ajuda.

SGBDs abordados neste livro

Na maioria das vezes, o SQL ensinado neste livro será aplicado a qualquer SGBD (Sistema de Gerenciamento de Banco de Dados). No entanto, como todas as implementações SQL não são criadas da mesma forma, os SGBDs a seguir são explicitamente abordados (e instruções ou notas específicas são incluídas quando necessário):

- IBM DB2 (incluindo DB2 na nuvem)
- Microsoft SQL Server (incluindo o Microsoft SQL Server Express)
- MariaDB
- MySQL
- Oracle (incluindo o Oracle Express)
- PostgreSQL
- SQLite

Exemplos de bancos de dados (ou scripts SQL para criar os exemplos de bancos de dados) estão disponíveis para todos esses SGBDs no site do

autor em http://forta.com/books/0135182794.[1]

Convenções usadas neste livro

Este livro usa fontes diferentes para diferenciar o código e o português comum, e para ajudá-lo a identificar conceitos importantes.

O texto digitado e o texto que deve aparecer na tela são apresentados em fonte monoespaçada.

```
Ele terá essa aparência para imitar como o texto aparece em sua tela.
```

O texto que compõe o código de programação não tem cor. Mas a maioria das ferramentas usadas para criar e editar código (em todas as linguagens de programação, incluindo SQL) exibe código em cores. A razão para isso é facilitar a leitura de longas sequências de código e ajudar a identificar erros de digitação e erros (quando as cores não combinam ou não parecem certas, você sabe que algo está errado). O código SQL ao longo deste livro é impresso em cores com diferentes cores usadas para instruções SQL, cláusulas, sequências de caracteres, números, comentários e assim por diante. Lembre-se de que não há um código padrão de cores e que ferramentas diferentes usam esquemas de cores diferentes. Portanto, as cores que você vê em seu próprio editor ao testar os exemplos podem não corresponder exatamente ao que está no livro.

Esta seta (➥) no início de uma linha de código significa que uma única linha de código é muito longa para caber na página impressa. Continue digitando todos os caracteres depois de ➥ como se eles fizessem parte da linha anterior.

NOTA:

Uma Nota apresenta informações interessantes relacionadas à discussão em questão.

DICA:

Uma Dica oferece conselhos ou ensina uma maneira mais fácil de fazer algo.

CUIDADO:

Um Cuidado alerta sobre potenciais problemas e ajuda a evitar o desastre.

NOVO TERMO:

Um Novo Termo oferece definições claras de termos novos e essenciais.

Entrada q

O ícone Entrada identifica código que você pode digitar. Geralmente ele aparece ao lado de uma listagem.

Saída q

O ícone Saída destaca a saída gerada pela execução de um programa. Geralmente ele aparece depois de uma listagem.

Análise q

O ícone Análise alerta você para a análise de um programa linha a linha feita pelo autor.

---

1 N.T.: No Apêndice A você encontra a explicação detalhada das tabelas utilizadas ao longo do livro.

LIÇÃO 1

Entendendo SQL

Nesta lição, você aprenderá exatamente o que é SQL e o que ele fará por você.

Noções básicas de banco de dados

O fato de você estar lendo um livro sobre SQL indica que, de alguma forma, você precisa interagir com os bancos de dados. SQL é uma linguagem usada para fazer exatamente isso; portanto, antes de analisar o próprio SQL, é importante que entenda alguns conceitos básicos sobre bancos de dados e tecnologias de banco de dados.

Quer esteja ciente disso ou não, você usa bancos de dados o tempo todo. Cada vez que seleciona um contato em seu celular ou um nome do seu catálogo de endereços de e-mail, você está usando um banco de dados. Se realiza uma pesquisa no Google, está usando um banco de dados. Ao fazer login na sua rede no trabalho, está validando seu nome e senha em um banco de dados. Mesmo quando usa seu cartão de banco em um caixa eletrônico, está usando bancos de dados para verificação de PIN e consulta de saldo.

No entanto, embora todos usemos bancos de dados o tempo todo, ainda resta muita confusão sobre o que exatamente é um banco de dados. Isso é especialmente verdade porque pessoas diferentes usam os mesmos termos de banco de dados para significar coisas diferentes. Portanto, um bom lugar para começar nosso estudo é com uma lista e uma explicação dos termos mais importantes de banco de dados.

DICA: Revendo conceitos básicos

A seguir, é apresentada uma breve visão geral de alguns conceitos básicos de banco de dados. A intenção é refrescar sua memória se você já possui alguma experiência em banco de dados ou fornecer o básico absoluto se é iniciante em bancos de dados. Compreender os bancos de dados é uma parte importante do domínio do SQL, e você

deveria encontrar um bom livro sobre os fundamentos de banco de dados para aprimorar seu conhecimento sobre o assunto, se necessário.

Bancos de dados

O termo *banco de dados* é usado de várias maneiras diferentes, mas, para nossos propósitos (e, de fato, da perspectiva do SQL), um banco de dados é uma coleção de dados armazenados de alguma maneira organizada. A maneira mais simples de pensar nisso é imaginar um banco de dados como um armário de arquivos. O armário de arquivos é simplesmente um local físico para armazenar dados, independentemente do que são ou como estão organizados.

NOVO TERMO: Banco de dados

Um container (geralmente um arquivo ou um conjunto de arquivos) para armazenar dados organizados.

CUIDADO: O uso incorreto causa confusão

As pessoas com frequência usam o termo *banco de dados* para se referir ao software de banco de dados que estão executando. Isso é incorreto e uma fonte de muita confusão. O software de banco de dados é na verdade chamado de *Sistema de Gerenciamento de Banco de Dados* (ou SGBD). O banco de dados é o container criado e manipulado via SGDB, e exatamente o que é o banco de dados e que forma ele assume variam de um banco de dados para o outro.

Tabelas

Quando você armazena informações em seu armário de arquivos, não apenas as joga em uma gaveta. Em vez disso, você cria arquivos dentro do armário de arquivos e depois arquiva dados relacionados em arquivos específicos.

No universo do banco de dados, esse arquivo é chamado de tabela. Uma *tabela* é um arquivo estruturado que pode armazenar dados de um tipo específico. Uma tabela pode conter uma lista de clientes, um catálogo de produtos ou qualquer outra lista de informações.

NOVO TERMO: Tabela

Uma lista estruturada de dados de um tipo específico.

O importante é que os dados armazenados na tabela sejam um mesmo tipo de dados ou uma lista. Você nunca armazenaria uma lista de clientes e uma

lista de pedidos na mesma tabela do banco de dados. Isso dificultaria a busca e o acesso subsequentes. Em vez disso, você criaria duas tabelas, uma para cada lista.

Toda tabela em um banco de dados tem um nome que a identifica. Esse nome é sempre único – ou seja, nenhuma outra tabela nesse banco de dados pode ter o mesmo nome.

> **NOTA: Nomes de tabelas**
>
> O que torna um nome de tabela único é na verdade uma combinação de várias coisas, incluindo o nome do banco de dados e o nome da tabela. Alguns bancos de dados também usam o nome do proprietário do banco de dados como parte do nome único. Isso significa que, embora não seja possível usar o mesmo nome de tabela duas vezes no mesmo banco de dados, você definitivamente pode reutilizar nomes de tabelas em bancos de dados diferentes.

As tabelas têm características e propriedades que definem como os dados são armazenados nelas. Isso inclui informações sobre quais dados podem ser armazenados, como eles são separados, como as informações individuais são nomeadas e muito mais. Esse conjunto de informações que descreve uma tabela é conhecido como um *esquema*, e esquemas são usados para descrever tabelas específicas em um banco de dados, bem como bancos de dados inteiros (e o relacionamento entre suas tabelas, se houver).

> **NOVO TERMO: Esquema (schema)**
>
> Informações sobre o layout e as propriedades do banco de dados e das tabelas.

Colunas e Datatypes

As tabelas são compostas de colunas. Uma *coluna* contém uma informação específica dentro de uma tabela.

> **NOVO TERMO: Coluna**
>
> Um único campo em uma tabela. Todas as tabelas são compostas de uma ou mais colunas.

A melhor maneira de entender isso é visualizar tabelas de banco de dados como grades, um pouco como planilhas. Cada coluna na grade contém uma informação específica. Em uma tabela de clientes, por exemplo, uma coluna contém o número do cliente, outra contém o nome do cliente e o endereço,

cidade, estado e CEP são armazenados em suas próprias colunas.

> **DICA: Separando os dados**
>
> É extremamente importante separar os dados em várias colunas corretamente. Por exemplo, cidade, estado e CEP (ou código postal) devem sempre ser colunas separadas. Quando você os separa, torna-se possível classificar ou filtrar dados por colunas específicas (por exemplo, para encontrar todos os clientes em um estado específico ou em uma cidade específica). Se cidade e estado forem combinados em uma coluna, seria extremamente difícil classificar ou filtrar por estado.
>
> Quando você separa os dados, o nível de granularidade depende de você e de seus requisitos específicos. Por exemplo, os endereços geralmente são armazenados com o nome da rua e o número da casa juntos. Não há problema, a menos que um dia você precise classificar os dados pelo nome da rua; nesse caso, é preferível separar o nome da rua e o número da casa.

Cada coluna em um banco de dados tem um datatype associado. Um *datatype* define que tipo de dados a coluna pode conter. Por exemplo, se a coluna deve conter um número (talvez o número de itens em um pedido), o datatype será numérico. Se a coluna deve conter datas, texto, notas, valores monetários etc., o datatype apropriado será usado para especificar isso.

> **NOVO TERMO: Datatype**
>
> Um tipo de dados permitidos. Cada coluna da tabela possui um datatype associado que restringe (ou permite) dados específicos nessa coluna.

Os datatypes restringem o tipo de dados que podem ser armazenados em uma coluna (por exemplo, impedindo a entrada de caracteres alfabéticos em um campo numérico). Os datatypes também ajudam a classificar os dados corretamente e desempenham um papel importante na otimização do uso do disco. Dessa forma, é preciso dar atenção especial à escolha do datatypes correto quando as tabelas são criadas.

> **CUIDADO: Compatibilidade de datatypes**
>
> Os datatypes e seus nomes são uma das principais fontes de incompatibilidade do SQL. Embora a maioria dos datatypes básicos seja suportada de maneira consistente, muitos datatypes mais avançados não são. E pior, ocasionalmente, você perceberá que o mesmo datatype é referido por nomes diferentes em diferentes SGBDs. Não há muito o que fazer a respeito, mas é importante ter isso em mente ao criar esquemas de tabela.

Linhas

Os dados em uma tabela são armazenados em linhas; cada registro salvo é

armazenado em sua própria *linha*. Mais uma vez, visualizando uma tabela como uma grade no estilo de planilha, as colunas verticais na grade são as colunas da tabela e as linhas horizontais são as linhas da tabela.

Por exemplo, uma tabela de clientes pode armazenar um cliente por linha. O número de linhas na tabela é o número de registros nela.

NOVO TERMO: Linha

Um registro em uma tabela.

NOTA: Registros ou linhas?

Você pode ouvir os usuários se referindo aos registros do banco de dados ao se referir às linhas. Na maioria das vezes, os dois termos são usados de forma intercambiável, mas linha é tecnicamente o termo correto.

Chaves primárias

Cada linha em uma tabela deve ter alguma coluna (ou conjunto de colunas) que a identifique de forma única. Uma tabela que contém clientes pode usar uma coluna de número de cliente para essa finalidade, enquanto uma tabela que contém pedidos pode usar a ID do pedido. Uma tabela de lista de funcionários pode usar uma ID de funcionário. Uma tabela contendo uma lista de livros pode usar o ISBN para esse fim.

NOVO TERMO: Chave primária

Uma coluna (ou conjunto de colunas) cujos valores identificam de forma única cada linha em uma tabela.

Essa coluna (ou conjunto de colunas) que identifica de forma única cada linha de uma tabela é chamada de *chave primária*. A chave primária é usada para se referir a uma linha específica. Sem uma chave primária, atualizar ou excluir linhas específicas em uma tabela se torna extremamente difícil, pois não há uma maneira segura e garantida de se referir apenas às linhas a serem afetadas.

DICA: Sempre defina chaves primárias

Embora as chaves primárias não sejam realmente necessárias, a maioria dos projetistas de banco de dados garante que todas as tabelas criadas tenham uma chave primária, para que a manipulação futura de dados seja possível e gerenciável.

Qualquer coluna em uma tabela pode ser definida como a chave primária, desde que atenda às seguintes condições:

- Duas linhas não podem ter o mesmo valor de chave primária.
- Cada linha deve ter um valor na(s) coluna(s) da chave primária. (Portanto, valores NULL não são permitidos)
- Os valores nas colunas da chave primária nunca devem ser modificados ou atualizados.
- Os valores da chave primária nunca devem ser reutilizados. (Se uma linha for excluída de uma tabela, sua chave primária não pode ser atribuída a nenhuma nova linha no futuro.)

As chaves primárias são geralmente definidas em uma única coluna dentro de uma tabela. Mas isso não é obrigatório e várias colunas podem ser usadas juntas como chave primária. Quando várias colunas são usadas, as regras listadas devem se aplicar a todas as colunas e os valores de todas as colunas juntas devem ser únicos (colunas individuais não precisam ter valores únicos).

Existe um outro tipo muito importante de chave chamada chave estrangeira, mas isso será abordado na *Lição 12 – Juntando tabelas*.

O que é SQL?

SQL (pronunciado como as letras S-Q-L) é uma abreviação de Structured Query Language (Linguagem de Consulta Estruturada). SQL é uma linguagem projetada especificamente para a comunicação com bancos de dados.

Diferente de outras linguagens (linguagens faladas como português ou linguagens de programação como Java, C ou Python), o SQL é composto de poucas palavras. Isso é deliberado. O SQL foi projetado para fazer uma coisa e fazê-la bem – oferecer uma maneira simples e eficiente de ler e gravar dados de um banco de dados.

Quais as vantagens do SQL?

- SQL não é uma linguagem proprietária usada por fornecedores específicos de bancos de dados. Quase todos os principais SGBDs suportam SQL, portanto, aprender essa linguagem única permitirá que você interaja com praticamente todos os bancos de dados que encontrar.

- SQL é fácil de aprender. As instruções são todas compostas de palavras descritivas em inglês e não existem muitas delas.
- Apesar de sua aparente simplicidade, o SQL é uma linguagem muito poderosa e, usando e combinando inteligentemente seus elementos de linguagem, você pode executar operações de banco de dados muito complexas e sofisticadas.

E, com isso, vamos aprender SQL.

> **NOTA: Extensões SQL**
>
> Muitos fornecedores de SGBD estenderam seu suporte ao SQL adicionando declarações ou instruções à linguagem. O objetivo dessas extensões é fornecer funcionalidade adicional ou maneiras simplificadas de executar operações específicas. E, embora muitas vezes sejam extremamente úteis, essas extensões tendem a ser muito específicas de cada SGBD e raramente são suportadas por mais de um único fornecedor.
>
> O SQL padrão é governado pelo comitê de padrões ANSI e por isso é chamado de SQL padrão ANSI. Todos os principais SGBDs, mesmo aqueles com suas próprias extensões, suportam SQL padrão ANSI. As implementações individuais têm seus próprios nomes (PL-SQL, usado pela Oracle; Transact-SQL, usado pelo Microsoft SQL Server; e assim por diante).
>
> Na maioria das vezes, o SQL ensinado neste livro é o SQL padrão ANSI. Nas raras vezes em que o SQL específico do SGBD for usado, haverá uma observação.

Experimente você mesmo

Como em qualquer linguagem, a melhor maneira de aprender SQL é experimentá-lo você mesmo. Para isso, você precisará de um banco de dados e de uma aplicação para testar suas instruções SQL.

Todas as lições deste livro usam instruções SQL reais e tabelas de banco de dados reais, e você deve ter acesso a um SGBD para acompanhar as lições.

> **DICA: Qual SGBD você deve usar?**
>
> Você precisa ter acesso a um SGBD para acompanhar as lições. Mas qual deve usar?
>
> A boa notícia é que o SQL que você aprenderá neste livro é relevante para todos os principais SGBDs. Dessa forma, sua escolha de SGBD deve basear-se principalmente na conveniência e na simplicidade.
>
> Existem basicamente duas maneiras de proceder. Você pode instalar um SGBD (e o software cliente de apoio) em seu próprio computador; isso proporcionará o melhor acesso e controle. Mas, para muitos, a parte mais complicada de começar a aprender

SQL é instalar e configurar um SGBD. A alternativa é acessar um SGBD remoto (ou baseado na nuvem); assim não há nada a ser gerenciado e instalado.

Existem muitas opções se você decidir instalar seu próprio SGBD. Aqui estão algumas sugestões:

- O MySQL (ou seu derivado MariaDB) é uma escolha muito boa, pois é gratuito, suportado em todos os principais sistemas operacionais, fácil de instalar e um dos SGBDs mais populares em uso. O MySQL vem com uma ferramenta de linha de comando para inserir seu SQL, mas é melhor usar o MySQL Workbench opcional, então faça o download dele também (geralmente é uma instalação separada).
- Os usuários do Windows podem querer usar o Microsoft SQL Server Express. Essa versão gratuita do popular e poderoso SQL Server inclui um cliente amigável chamado SQL Server Management Studio.

A alternativa é usar um SGDB remoto:

- Se você está aprendendo a usar o SQL no trabalho, seu empregador pode ter um SGBD que você possa usar. Se essa for uma opção, você provavelmente receberá o próprio login no SGBD e uma ferramenta para conectar-se ao SGBD para inserir e testar seu SQL.
- SGBDs baseados em nuvem são instâncias de SGBDs executadas em servidores virtuais, efetivamente oferecendo os benefícios de seu próprio SGBD sem a necessidade de realmente instalar um localmente. Todos os principais fornecedores de serviços em nuvem (incluindo Google, Amazon e Microsoft) oferecem SGBDs na nuvem. Infelizmente, no momento em que escrevo este livro, configurá-los (incluindo configurar o acesso remoto seguro) não é trivial e em geral envolve mais trabalho do que instalar seu próprio SGBD localmente. As exceções são o Live SQL da Oracle e o Db2 on Cloud da IBM, que oferece uma versão gratuita que inclui uma interface web. Basta digitar seu SQL no navegador e você está pronto para começar.

Você encontrará links para todas as opções mencionadas aqui na página do livro na web e, à medida que as opções do SGBD evoluírem, essa página será atualizada com dicas e sugestões.

Depois de ter acesso a um SGBD, o *Apêndice A – Exemplos de scripts de tabela*, explica o que são as tabelas de exemplo e fornece detalhes sobre como obtê-las (ou criá-las) para que você siga as instruções de cada lição.

Além disso, a partir da Lição 2, você encontrará *Desafios* após a seção *Resumo*. Eles oferecem a oportunidade de pegar seu conhecimento SQL recém-adquirido e aplicá-lo para resolver problemas não mencionados explicitamente nas lições. Para verificar suas soluções (ou se você ficar

travado e precisar de ajuda), visite a página do livro na web.

Resumo

Nesta primeira lição, você aprendeu o que é SQL e por que ele é útil. Como o SQL é usado para interagir com bancos de dados, você também revisou algumas terminologias básicas de banco de dados.

LIÇÃO 2

Obtendo dados

Nesta lição, você aprenderá como usar a importante instrução SELECT *para obter uma ou mais colunas de dados de uma tabela.*

Instrução SELECT

Conforme explicado na *Lição 1 – Entendendo SQL*, as instruções SQL são compostas de termos simples em inglês. Esses termos são chamados de palavras-chave e cada instrução SQL é composta de uma ou mais palavras-chaves. A instrução SQL que você provavelmente usará com mais frequência é a instrução SELECT. Seu objetivo é obter informações de uma ou mais tabelas.

NOVO TERMO: Palavra-chave

Uma palavra reservada que faz parte da linguagem SQL. Nunca nomeie uma tabela ou coluna usando uma palavra-chave. O *Apêndice D – Palavras reservadas do SQL*, lista algumas das palavras reservadas mais comuns.

Para usar SELECT a fim de obter dados da tabela, você deve especificar, no mínimo, duas informações – o que deseja selecionar e de onde deseja selecioná-las.

NOTA: Acompanhando os exemplos

Os exemplos de instruções SQL (e a saída dos exemplos) ao longo das lições deste livro usam um conjunto de arquivos de dados descritos no Apêndice A, "Exemplos de scripts de tabela". Se você deseja acompanhar e testar os exemplos (recomendo fortemente que faça isso), consulte o Apêndice A, que contém instruções sobre como baixar ou criar esses arquivos de dados.

DICA: Use o banco de dados correto

SGBDs permitem que você trabalhe com vários bancos de dados (o armário de arquivos na analogia da Lição 1). Ao instalar os exemplos de tabela (conforme o Apêndice A), você foi aconselhado a instalá-las em um novo banco de dados. Se você fez isso, certifique-se de selecionar esse banco de dados antes de continuar, assim como fez quando criou e preencheu os exemplos de tabelas. À medida que você trabalhar nessas lições, se

encontrar erros sobre tabelas desconhecidas, provavelmente estará no banco de dados errado.

Obtendo colunas individuais

Começaremos com uma instrução SQL SELECT simples, como mostrado a seguir:

Entrada ▼

```
SELECT prod_name
FROM Products;
```

Análise ▼

O exemplo anterior usa a instrução SELECT para obter uma única coluna chamada `prod_name` da tabela `Products`. O nome da coluna desejada é especificado logo após a palavra-chave SELECT e a palavra-chave FROM especifica o nome da tabela da qual obter os dados. A saída dessa instrução é mostrada a seguir:

Saída ▼

```
prod_name
--------------------
Fish bean bag toy
Bird bean bag toy
Rabbit bean bag toy
8 inch teddy bear
12 inch teddy bear
18 inch teddy bear
Raggedy Ann
King doll
Queen doll
```

Dependendo do SGBD e do cliente que você está usando, também pode ver uma mensagem informando quantas linhas foram obtidas e o tempo de processamento. Por exemplo, a linha de comando do MySQL exibirá algo como:

```
9 rows in set (0.01 sec)
```

NOTA: Dados não classificados

Se você mesmo tentou fazer essa consulta, pode ter descoberto que os dados foram exibidos em uma ordem diferente da mostrada aqui. Se for esse o caso, não se preocupe –

ela está funcionando exatamente como deveria. Se os resultados da consulta não forem explicitamente classificados (falaremos disso na próxima lição), eles serão retornados sem nenhuma ordem significativa. Pode ser a ordem em que os dados foram adicionados à tabela, mas pode não ser. Desde que sua consulta retorne o mesmo número de linhas, ela está funcionando corretamente.

Uma instrução SELECT simples, semelhante à que acabamos de usar, retorna todas as linhas em uma tabela. Os dados não são filtrados (para obter um subconjunto dos resultados), nem classificados. Discutiremos esses tópicos nas próximas lições.

DICA: Encerrando instruções

Múltiplas instruções SQL devem ser separadas por ponto e vírgula (o caractere ;). A maioria dos SGBDs não exige que um ponto e vírgula seja especificado após instruções únicas. Mas, se o seu SGBD específico reclamar, talvez seja necessário adicionar esse caractere. Obviamente, você sempre pode adicionar um ponto e vírgula, se desejar. Não fará mal nenhum, mesmo que não seja de fato necessário.

NOTA: Instrução SQL e a distinção entre maiúsculas e minúsculas

É importante observar que as instruções SQL *não* diferenciam maiúsculas de minúsculas, portanto `SELECT` é o mesmo que `select`, que é o mesmo que `Select`. Muitos desenvolvedores SQL acham que usar letras maiúsculas para todas as palavras-chave SQL e letras minúsculas para nomes de colunas e tabelas facilita a leitura e a depuração do código. No entanto, lembre-se de que, embora a linguagem SQL não diferencie maiúsculas de minúsculas, os nomes de tabelas, colunas e valores podem diferenciar (isso depende do seu SGBD e de como ele está configurado).

DICA: Uso de espaços em branco

Todo o espaço em branco extra em uma instrução SQL é ignorado quando essa instrução é processada. As instruções SQL podem ser especificadas em uma linha longa ou divididas em várias linhas. Portanto, as três instruções a seguir são funcionalmente idênticas:

```
SELECT prod_name
FROM Products;

SELECT prod_name FROM Products;

SELECT
prod_name
FROM
Products;
```

A maioria dos desenvolvedores SQL acha que dividir instruções em várias linhas facilita a leitura e a depuração.

Obtendo múltiplas colunas

Para obter múltiplas colunas de uma tabela, a mesma instrução SELECT é usada. A única diferença é que vários nomes de colunas devem ser especificados após a palavra SELECT e cada coluna deve ser separada por vírgula.

> **DICA: Cuidado com as vírgulas**
>
> Ao selecionar múltiplas colunas, certifique-se de especificar uma vírgula entre cada nome de coluna, mas não após o nome da última coluna. Fazer isso gerará um erro.

A seguinte instrução SELECT obtém três colunas da tabela `Products`:

Entrada ▼

```
SELECT prod_id, prod_name, prod_price
FROM Products;
```

Análise ▼

Assim como no exemplo anterior, essa declaração usa a instrução SELECT para obter dados da tabela `Products`. Nesse exemplo, três nomes de colunas são especificados, cada um separado por uma vírgula. A saída dessa instrução é mostrada a seguir:

Saída ▼

```
prod_id      prod_name              prod_price
---------    --------------------   ----------
BNBG01       Fish bean bag toy            3.49
BNBG02       Bird bean bag toy            3.49
BNBG03       Rabbit bean bag toy          3.49
BR01         8 inch teddy bear            5.99
BR02         12 inch teddy bear           8.99
BR03         18 inch teddy bear          11.99
RGAN01       Raggedy Ann                  4.99
RYL01        King doll                    9.49
RYL02        Queen dool                   9.49
```

> **NOTA: Apresentação dos dados**
>
> As instruções SQL geralmente retornam dados brutos, não formatados, e diferentes SGBDs e clientes podem exibir os dados de maneira diferente (com alinhamento ou casas decimais diferentes, por exemplo). A formatação de dados é um problema de apresentação, não um problema de consulta. Portanto, a apresentação é normalmente especificada na aplicação

que exibe os dados. Dados reais obtidos (sem formatação fornecida pela aplicação) raramente são usados.

Obtendo todas as colunas

Além de poder especificar as colunas desejadas (uma ou mais, como visto anteriormente), as instruções SELECT também podem solicitar todas as colunas sem precisar listá-las individualmente. Isso é feito usando o caractere curinga asterisco (*) no lugar dos nomes reais das colunas, como mostrado a seguir:

Entrada ▼

```
SELECT *
FROM Products;
```

Análise ▼

Quando um curinga (*) é especificado, todas as colunas da tabela são retornadas. A ordem das colunas normalmente, mas nem sempre, será a ordem física na qual as colunas aparecem na definição da tabela. No entanto, os dados SQL raramente são exibidos como estão. (Em geral, eles são retornados a uma aplicação que formata ou apresenta os dados conforme necessário). Assim, isso não deve representar um problema.

ATENÇÃO: Usando caracteres curinga

Como regra, é melhor não usar o caractere curinga * a menos que você realmente precise de todas as colunas da tabela. Embora o uso de caracteres curinga possa economizar o tempo e o esforço necessários para listar explicitamente as colunas desejadas, obter colunas desnecessárias geralmente diminui o desempenho da consulta e da aplicação.

DICA: Obtendo colunas desconhecidas

Há uma grande vantagem em usar caracteres curinga. Como você não especifica explicitamente os nomes das colunas (porque o asterisco obtém todas as colunas), é possível obter colunas cujos nomes são desconhecidos.

Obtendo linhas distintas

Como vimos, a instrução SELECT retorna todas as linhas correspondentes. Mas e se você não quiser todas as ocorrências de todos os valores? Por exemplo, suponha que você queira a ID do fornecedor de todos os fornecedores com produtos em sua tabela Products:

Entrada ▼

```
SELECT vend_id
FROM Products;
```

Saída ▼

```
vend_id
----------
BRS01
BRS01
BRS01
DLL01
DLL01
DLL01
DLL01
FNG01
FNG01
```

A instrução `SELECT` retornou nove linhas (mesmo que haja apenas três fornecedores distintos nessa lista) porque há nove produtos listados na tabela `Products`. Então, como obter uma lista de valores distintos?

A solução é usar a palavra-chave `DISTINCT`, que, como o próprio nome indica, instrui o banco de dados a retornar apenas valores distintos.

Entrada ▼

```
SELECT DISTINCT vend_id
FROM Products;
```

Análise ▼

`SELECT DISTINCT vend_id` diz ao SGBD que retorne somente linhas `vend_id` distintas e, portanto, apenas três linhas são retornadas, como visto na saída a seguir. Se usada, a palavra-chave `DISTINCT` deve ser colocada diretamente ANTES dos nomes das colunas.

Saída ▼

```
vend_id
----------
BRS01
DLL01
FNG01
```

ATENÇÃO: Não é possível ser parcialmente `DISTINCT`

A palavra-chave DISTINCT se aplica a todas as colunas, não apenas àquela que ela precede. Se você especificar SELECT DISTINCT vend_id, prod_price, seis das nove linhas serão obtidas porque as colunas especificadas combinadas produziram seis combinações exclusivas. Para ver a diferença, teste estas duas instruções e compare os resultados:

```
SELECT DISTINCT vend_id, prod_price FROM Products;
```

```
SELECT vend_id, prod_price FROM Products;
```

Limitando os resultados

As instruções SELECT retornam todas as linhas correspondentes, possivelmente todas as linhas da tabela especificada. E se você quiser retornar apenas a primeira linha ou um número definido de linhas? Isso é possível, mas infelizmente é uma daquelas situações em que nem todas as implementações do SQL são criadas da mesma forma.

No Microsoft SQL Server, você pode usar a palavra-chave TOP para limitar o número máximo de entradas, como mostrado a seguir:

Entrada ▼

```
SELECT TOP 5 prod_name
FROM Products;
```

Saída ▼

```
prod_name
----------------
8 inch teddy bear
12 inch teddy bear
18 inch teddy bear
Fish bean bag toy
Bird bean bag toy
```

Análise ▼

O exemplo anterior usa a instrução SELECT TOP 5 para obter apenas as primeiras cinco linhas.

Se você estiver usando o DB2, poderá usar o SQL exclusivo para esse SGBD, como o exemplo a seguir:

Entrada ▼

```
SELECT prod_name
FROM Products
```

```
FETCH FIRST 5 ROWS ONLY;
```

Análise ▼

`FETCH FIRST 5 ROWS ONLY` faz exatamente o que a expressão sugere. Se você estiver usando o Oracle, precisará contar as linhas com base em `ROWNUM` (um contador de número de linhas) como a seguir:

Entrada ▼

```
SELECT prod_name
FROM Products
WHERE ROWNUM <=5;
```

Se você estiver usando MySQL, MariaDB, PostgreSQL ou SQLite, poderá usar a cláusula `LIMIT` da seguinte maneira:

Entrada ▼

```
SELECT prod_name
FROM Products
LIMIT 5;
```

Análise ▼

O exemplo anterior usa a instrução `SELECT` para obter uma única coluna. `LIMIT 5` instrui os SGBDs suportados a retornar não mais que cinco linhas. A saída dessa instrução é mostrada no código a seguir.

Para obter as próximas cinco linhas, especifique onde começar e o número de linhas a serem obtidas, como mostrado a seguir:

Entrada ▼

```
SELECT prod_name
FROM Products
LIMIT 5 OFFSET 5;
```

Análise ▼

`LIMIT 5 OFFSET 5` instrui os SGBDs suportados a retornar cinco linhas começando da linha `5`. O primeiro número é o número de linhas a serem obtidas e o segundo é onde começar. A saída dessa instrução é mostrada no código a seguir:

Saída ▼

```
prod_name
```

```
------------------
Rabbit bean bag toy
Raggedy Ann
King doll
Queen doll
```

Portanto, `LIMIT` especifica o número de linhas a serem retornadas. `LIMIT` com um `OFFSET` especifica onde começar. No nosso exemplo, existem apenas nove produtos na tabela `Products`, portanto, `LIMIT 5 OFFSET 5` retornou apenas quatro linhas (já que não havia uma quinta).

> **ATENÇÃO: Linha 0**
>
> A primeira linha obtida é a linha `0`, e não a linha `1`. Dessa forma, `LIMIT 1 OFFSET 1` obterá a segunda linha, e não a primeira.

> **DICA: Atalho do MySQL, MariaDB e SQLite**
>
> MySQL, MariaDB e SQLite suportam uma versão abreviada do `LIMIT 4 OFFSET 3`, permitindo combiná-los como `LIMIT 3,4`. Usando essa sintaxe, o valor antes da vírgula "`,`" é o `OFFSET` e o valor depois da vírgula "`,`" é o `LIMIT` (sim, eles estão invertidos, então tome cuidado).

> **NOTA: Nem TODO SQL é criado igual**
>
> Incluí esta seção sobre a limitação de resultados apenas por um motivo – para demonstrar que, embora o SQL seja geralmente bastante consistente nas implementações, você não pode confiar que isso sempre ocorrerá. Embora as instruções muito básicas tendam a ser muito portáveis, as mais complexas tendem a ser menos portáveis. Lembre-se disso ao procurar soluções SQL para problemas específicos.

Usando comentários

Como vimos, as instruções SQL são instruções processadas pelo seu SGBD. Mas e se você quiser incluir um texto que não deseja processar e executar? Por que você faria isso? Aqui estão algumas razões:

- As instruções SQL que temos usado aqui são muito curtas e muito simples. Mas, à medida que suas instruções SQL crescem (em tamanho e complexidade), você deve incluir comentários descritivos (para sua própria referência futura ou para quem precisar trabalhar no projeto a seguir). Esses comentários precisam ser incorporados aos scripts SQL, mas obviamente eles não são destinados a serem realmente processados pelo SGBD. (Para um exemplo disso, consulte

os arquivos `create.sql` e `populate.sql` usados no *Apêndice B – Sintaxe das instruções SQL*).

- O mesmo vale para os cabeçalhos na parte superior de um arquivo SQL (aquele que está salvando instruções SQL talvez para uso futuro), geralmente contendo uma descrição e notas, e talvez até informações de contato do programador. (Esse caso de uso também é visto nos arquivos `.sql` do Apêndice B).
- Outro uso importante para comentários é interromper temporariamente a execução do código SQL. Se você está trabalhando com uma instrução SQL longa e deseja testar apenas parte dela, pode *comentar* parte do código, para que o SGBD o considere um comentário e o ignore.

A maioria dos SGBDs suporta várias formas de sintaxe de comentário. Começaremos com comentários na própria linha:

Entrada ▼

```
SELECT prod_name   -- isto é um comentário
FROM Products;
```

Análise ▼

Os comentários podem ser incorporados na linha usando `--` (dois hifens). Qualquer texto na mesma linha após o `--` é considerado texto de comentário, tornando essa uma boa opção para descrever colunas em uma instrução `CREATE TABLE`, por exemplo.

Aqui está outra forma de comentário incorporado na linha (embora menos comumente suportado):

Entrada ▼

```
# Isto é um comentário
SELECT prod_name
FROM Products;
```

Análise ▼

Um `#` no início de uma linha faz com que a linha inteira seja considerada um comentário. Você pode ver esse formato de comentário usado nos scripts `create.sql` e `populate.sql`.

Você também pode criar comentários com várias linhas e comentários que param e iniciam em qualquer lugar do script:

Entrada ▼

```
/* SELECT prod_name, vend_id
FROM Products; */
SELECT prod_name
FROM Products;
```

Análise ▼

`/*` inicia um comentário e `*/` encerra esse comentário. Qualquer coisa entre `/*` e `*/` é texto do comentário. Esse tipo de comentário é geralmente usado para *comentar* código, como visto neste exemplo. Aqui, duas instruções `SELECT` são definidas, mas a primeira não será executada porque foi transformada em comentário.

Resumo

Nesta lição, você aprendeu como usar a instrução SQL `SELECT` para obter uma única coluna da tabela, múltiplas colunas da tabela e todas as colunas da tabela. Você também aprendeu como retornar valores distintos e como comentar seu código. E, infelizmente, você também foi apresentado ao fato de que SQL mais complexo tende a ser menos portável. Em seguida, você aprenderá a classificar os dados obtidos.

Desafios

1. Escreva uma instrução SQL para obter todas as IDs de cliente (`cust_id`) da tabela `Customers`.
2. A tabela `OrderItems` contém todos os itens pedidos (e alguns foram pedidos várias vezes). Escreva uma instrução SQL para obter uma lista dos produtos (`prod_id`) pedidos (não todos os pedidos, apenas uma lista de produtos distintos). Aqui vai uma dica: o resultado deve ser a exibição de sete linhas exclusivas.
3. Escreva uma instrução SQL que obtenha todas as colunas da tabela `Customers` e uma instrução `SELECT` alternativa que obtenha apenas a ID do cliente. Use comentários para comentar uma das instruções `SELECT`

de modo que a outra seja executada. (E, claro, teste ambas as instruções.)

DICA: Onde estão as respostas?

As respostas dos desafios estão no Apêndice E.

LIÇÃO 3

Classificando dados obtidos

Nesta lição você aprenderá como usar a cláusula `ORDER BY` *da instrução* `SELECT` *para classificar os dados obtidos, conforme necessário.*

Classificando dados

Como você aprendeu na última lição, a instrução SQL a seguir retorna uma única coluna de uma tabela de banco de dados. Mas observe a saída. Os dados parecem ser exibidos em nenhuma ordem específica.

Entrada ▼

```
SELECT prod_name
FROM Products;
```

Saída ▼

```
prod_name
--------------------
Fish bean bag toy
Bird bean bag toy
Rabbit bean bag toy
8 inch teddy bear
12 inch teddy bear
18 inch teddy bear
Raggedy Ann
King doll
Queen doll
```

Na verdade, os dados obtidos não serão exibidos em uma mera ordem aleatória. Se não forem classificados, os dados geralmente serão exibidos na ordem em que aparecem nas tabelas correspondentes. Essa pode ser a ordem em que os dados foram adicionados às tabelas inicialmente. No entanto, se dados foram atualizados ou excluídos posteriormente, a ordem será afetada pela maneira como o SGBD reutiliza o espaço de armazenamento recuperado. O resultado é que você não pode (e não

deve) confiar na ordem de classificação se não a controlar explicitamente. A teoria de projeto de banco de dados relacional afirma que a sequência de dados obtidos não pode ser assumida como sendo significativa se uma ordem não foi especificada explicitamente.

NOVO TERMO: Cláusula

As instruções SQL são compostas de cláusulas, algumas necessárias e outras opcionais. Uma cláusula geralmente consiste em uma palavra-chave e dados fornecidos. Um exemplo disso é a cláusula `FROM` da instrução `SELECT`, que você viu na última lição.

Para classificar explicitamente os dados obtidos usando uma instrução `SELECT`, você usa a cláusula `ORDER BY`. `ORDER BY` recebe o nome de uma ou mais colunas usadas para classificar a saída. Observe o exemplo a seguir:

Entrada ▼

```
SELECT prod_name
FROM Products
ORDER BY prod_name;
```

Análise ▼

Essa instrução é idêntica à instrução anterior, exceto que ela também especifica uma cláusula `ORDER BY` instruindo o software SGBD a classificar os dados pela coluna `prod_name`. Os resultados são os seguintes:

Saída ▼

```
prod_name
--------------------
12 inch teddy bear
18 inch teddy bear
8 inch teddy bear
Bird bean bag toy
Fish bean bag toy
King doll
Queen doll
Rabbit bean bag toy
Raggedy Ann
```

CUIDADO: Posição da cláusula `ORDER BY`

Ao especificar uma cláusula `ORDER BY`, certifique-se de que ela seja a última cláusula na sua instrução `SELECT`. Se ela não for a última cláusula, um erro será gerado.

DICA: Classificando por colunas não selecionadas

Embora na maioria das vezes as colunas usadas em uma cláusula `ORDER BY` sejam as selecionadas para exibição, isso não é realmente necessário. É perfeitamente legítimo classificar dados por uma coluna que não é obtida.

Classificando por múltiplas colunas

Geralmente, é necessário classificar os dados por mais de uma coluna. Por exemplo, se você estiver exibindo uma lista de funcionários, convém exibi-la classificada por sobrenome e nome (primeiro pelo sobrenome e depois, em cada sobrenome, classifique pelo nome). Esse tipo de classificação é útil se houver vários funcionários com o mesmo sobrenome.

Para classificar por múltiplas colunas, basta especificar os nomes das colunas separados por vírgulas (assim como você faz ao selecionar múltiplas colunas).

O código a seguir obtém três colunas e classifica os resultados por duas delas – primeiro pelo preço e depois pelo nome.

Entrada ▼

```
SELECT prod_id, prod_price, prod_name
FROM Products
ORDER BY prod_price, prod_name;
```

Saída ▼

```
prod_id    prod_price    prod_name
-------    ----------    --------------------
BNBG02     3.4900        Bird bean bag toy
BNBG01     3.4900        Fish bean bag toy
BNBG03     3.4900        Rabbit bean bag toy
RGAN01     4.9900        Raggedy Ann
BR01       5.9900        8 inch teddy bear
BR02       8.9900        12 inch teddy bear
RYL01      9.4900        King doll
RYL02      9.4900        Queen doll
BR03       11.9900       18 inch teddy bear
```

É importante entender que, ao classificar por múltiplas colunas, a sequência de classificação é exatamente a especificada. Em outras palavras, usando a saída no exemplo anterior, os produtos são

classificados pela coluna `prod_name` apenas quando várias linhas tiverem o mesmo valor de `prod_price`. Se todos os valores na coluna `prod_price` fossem distintos, nenhum dado teria sido classificado por `prod_name`.

Classificando pela posição da coluna

Além de poder especificar a ordem de classificação usando nomes de colunas, a cláusula `ORDER BY` também suporta a ordem especificada pela posição relativa da coluna. A melhor maneira de entender isso é ver um exemplo:

Entrada ▼

```
SELECT prod_id, prod_price, prod_name
FROM Products
ORDER BY 2, 3;
```

Saída ▼

```
prod_id    prod_price    prod_name
-------    ----------    --------------------
BNBG02     3.4900        Bird bean bag toy
BNBG01     3.4900        Fish bean bag toy
BNBG03     3.4900        Rabbit bean bag toy
RGAN01     4.9900        Raggedy Ann
BR01       5.9900        8 inch teddy bear
BR02       8.9900        12 inch teddy bear
RYL01      9.4900        King doll
RYL02      9.4900        Queen doll
BR03       11.9900       18 inch teddy bear
```

Análise ▼

Como você pode ver, a saída é idêntica à da consulta anterior. A diferença aqui está na cláusula `ORDER BY`. Em vez de especificar os nomes das colunas, você especifica as posições relativas das colunas selecionadas na lista `SELECT`. A expressão `ORDER BY 2` significa classificar pela segunda coluna na lista `SELECT`, a coluna `prod_price`. `ORDER BY 2, 3` significa classificar por `prod_price` e depois por `prod_name`.

A principal vantagem dessa técnica é que ela evita digitar novamente os nomes das colunas. Mas existem algumas desvantagens também. Primeiro, não listar explicitamente os nomes das colunas aumenta a

probabilidade de você especificar por engano a coluna errada. Segundo, é muito fácil reordenar dados por engano ao fazer alterações na lista SELECT (esquecendo de fazer as alterações correspondentes na cláusula ORDER BY). E, por fim, obviamente você não pode usar essa técnica ao classificar por colunas que não estão na lista SELECT.

> **DICA: Classificando por colunas não selecionadas**
>
> Essa técnica não pode ser usada ao classificar por colunas que não aparecem na lista SELECT. No entanto, você pode misturar e combinar nomes reais de colunas e posições relativas de colunas em uma única instrução, se necessário.

Especificando a direção da classificação

A classificação dos dados não se limita à ordem de classificação crescente (de A a Z). Embora essa seja a ordem de classificação padrão, a cláusula ORDER BY também pode ser usada para classificar em ordem decrescente (de Z a A). Para classificar por ordem decrescente, especifique a palavra-chave DESC.

O exemplo a seguir classifica os produtos por preço em ordem decrescente (primeiro o mais caro):

Entrada ▼

```
SELECT prod_id, prod_price, prod_name
FROM Products
ORDER BY prod_price DESC;
```

Saída ▼

```
prod_id    prod_price    prod_name
-------    ----------    --------------------
BR03       11.9900       18 inch teddy bear
RYL01      9.4900        King doll
RYL02      9.4900        Queen doll
BR02       8.9900        12 inch teddy bear
BR01       5.9900        8 inch teddy bear
RGAN01     4.9900        Raggedy Ann
BNBG01     3.4900        Fish bean bag toy
BNBG02     3.4900        Bird bean bag toy
BNBG03     3.4900        Rabbit bean bag toy
```

Mas e se você estiver classificando por múltiplas colunas? O exemplo a

seguir classifica os produtos em ordem decrescente (primeiro o mais caro), mais o nome do produto:

Entrada ▼

```
SELECT prod_id, prod_price, prod_name
FROM Products
ORDER BY prod_price DESC, prod_name;
```

Saída ▼

```
prod_id    prod_price    prod_name
-------    ----------    --------------------
BR03       11.9900       18 inch teddy bear
RYL01      9.4900        King doll
RYL02      9.4900        Queen doll
BR02       8.9900        12 inch teddy bear
BR01       5.9900        8 inch teddy bear
RGAN01     4.9900        Raggedy Ann
BNBG02     3.4900        Bird bean bag toy
BNBG01     3.4900        Fish bean bag toy
BNBG03     3.4900        Rabbit bean bag toy
```

Análise ▼

A palavra-chave `DESC` só se aplica ao nome da coluna que a precede diretamente. No exemplo anterior, `DESC` foi especificado para a coluna `prod_price`, mas não para a coluna `prod_name`. Portanto, a coluna `prod_price` é classificada em ordem decrescente, mas a coluna `prod_name` (dentro de cada preço) ainda é classificada em ordem crescente padrão.

CUIDADO: Classificando em ordem decrescente em relação a múltiplas colunas

Se você deseja classificar em ordem decrescente em relação a múltiplas colunas, verifique se cada coluna possui sua própria palavra-chave `DESC`.

Vale ressaltar que `DESC` é a abreviação de `DESCENDING` e as duas palavras-chave podem ser usadas. O oposto de `DESC` é `ASC` (ou `ASCENDING`), que pode ser especificado para classificar os dados em ordem crescente. Na prática, no entanto, `ASC` geralmente não é usado porque a ordem crescente é a sequência padrão (e é assumida se nem `ASC` e nem `DESC` forem especificados).

DICA: Distinção entre maiúsculas e minúsculas e ordens de classificação

Ao classificar dados textuais, A é o mesmo que a? E a vem antes de B ou depois de Z? Essas não são questões teóricas e as respostas dependem de como o banco de dados está configurado.

Na ordem de classificação do *dicionário* A é tratado da mesma forma que a, e esse é o comportamento padrão para a maioria dos SGBDs. No entanto, a maioria dos bons SGBDs permite que os administradores de banco de dados alterem esse comportamento, se necessário. (Se o seu banco de dados contiver muitos caracteres de idiomas estrangeiros, isso poderá ser necessário.)

O importante aqui é que, se você precisar classificar em uma ordem alternativa, poderá não conseguir fazê-lo com uma simples cláusula ORDER BY. Pode ser necessário entrar em contato com o administrador do banco de dados.

Resumo

Nesta lição, você aprendeu a classificar os dados obtidos usando a cláusula ORDER BY da instrução SELECT. Essa cláusula, que deve ser a última na instrução SELECT, pode ser usada para classificar dados por uma ou mais colunas, conforme necessário.

Desafios

1. Escreva uma instrução SQL para obter todos os nomes de clientes (cust_name) da tabela Customers e exiba os resultados classificados de Z a A.
2. Escreva uma instrução SQL para obter a ID do cliente (cust_id) e o número do pedido (order_num) da tabela Orders e classifique os resultados primeiro pela ID do cliente e depois pela data do pedido em ordem cronológica inversa.
3. Nossa loja fictícia obviamente prefere vender itens mais caros e em grande quantidade. Escreva uma instrução SQL para exibir a quantidade e o preço (item_price) da tabela OrderItems, classificando primeiro pelas quantidades maiores e pelo preço mais alto.
4. O que há de errado com a seguinte instrução SQL? (tente descobrir sem executá-la):

```
SELECT vend_name,
FROM Vendors
ORDER vend_name DESC;
```

LIÇÃO 4

Filtrando dados

Nesta lição, você aprenderá como usar a cláusula WHERE *da instrução* SELECT *para especificar condições de pesquisa.*

Usando a cláusula WHERE

As tabelas de banco de dados em geral contêm grandes quantidades de dados e você raramente precisa obter todas as linhas em uma tabela. Geralmente, você deseja extrair um subconjunto dos dados da tabela, conforme necessário para operações ou relatórios específicos. Obter apenas os dados desejados envolve a especificação de *critérios de pesquisa*, também conhecidos como *condição do filtro*.

Dentro de uma instrução SELECT, os dados são filtrados especificando-se critérios de pesquisa na cláusula WHERE. A cláusula WHERE é especificada logo após o nome da tabela (a cláusula FROM) da seguinte maneira:

Entrada q

```
SELECT prod_name, prod_price
FROM Products
WHERE prod_price = 3.49;
```

Análise q

Essa instrução obtém duas colunas da tabela Products, mas, em vez de retornar todas as linhas, somente as linhas com um valor de prod_price igual a 3.49 são retornadas, da seguinte maneira:

Saída q

```
prod_name           prod_price
------------------  ----------
Fish bean bag toy   3.49
Bird bean bag toy   3.49
Rabbit bean bag toy 3.49
```

Esse exemplo usa um teste de igualdade simples: ele verifica se uma coluna possui um valor especificado e filtra os dados de acordo. Mas o SQL permite fazer mais do que apenas testar a igualdade.

> **DICA: Quantos zeros?**
>
> Ao testar os exemplos desta lição, você poderá ver os resultados exibidos como 3.49, 3.490, 3.4900 e assim por diante. Esse comportamento tende a ser um pouco específico do SGBD, pois está vinculado aos tipos de dados usados e ao seu comportamento padrão. Portanto, se seu resultado for um pouco diferente do meu, não se preocupe; afinal, 3.49 e 3.4900 são matematicamente idênticos de qualquer maneira.

> **DICA: Filtragem pelo SQL x Filtragem pela aplicação**
>
> Os dados também podem ser filtrados no nível da aplicação cliente, não no SGBD, mas por qualquer ferramenta ou aplicação usada para obter os dados do SGBD. Para isso, a instrução SQL SELECT obtém mais dados do que o necessário para a aplicação cliente, e o código do cliente percorre os dados retornados para extrair apenas as linhas necessárias.
>
> Como regra, essa prática é fortemente desencorajada. Os bancos de dados são otimizados para executar a filtragem de maneira rápida e eficiente. Fazer com que a aplicação cliente (ou a linguagem de desenvolvimento) execute o trabalho do banco de dados afetará drasticamente o desempenho da aplicação e criará aplicações que não podem escalar adequadamente. Além disso, se os dados forem filtrados no cliente, o servidor precisará enviar dados desnecessários pelas conexões de rede, resultando em um desperdício de uso da largura de banda da rede.

> **CUIDADO: Posição da cláusula WHERE**
>
> Ao usar as cláusulas ORDER BY e WHERE, certifique-se de que ORDER BY venha depois de WHERE. Caso contrário, um erro será gerado. (Veja a *Lição 3 – Classificando dados obtidos*, para mais informações sobre como usar ORDER BY.)

Operadores da cláusula WHERE

A primeira cláusula WHERE analisada testa a igualdade – determinando se uma coluna contém um valor específico. O SQL suporta toda uma gama de operadores condicionais, conforme listado na Tabela 4.1.

Tabela 4.1 – Operadores da cláusula WHERE

Operador	Descrição
=	Igualdade
<>	Não igualdade
!=	Não igualdade

<	Menor que
<=	Menor que ou igual a
!<	Não menor que
>	Maior que
>=	Maior que ou igual a
!>	Não maior que
BETWEEN	Entre dois valores especificados
IS NULL	É um valor NULL

CUIDADO: Compatibilidade do operador

Alguns dos operadores listados na Tabela 4.1 são redundantes; por exemplo, <> é o mesmo que !=. !< (não menor que) tem o mesmo efeito que >= (maior que ou igual a). Nem todos esses operadores são suportados por todos os SGBDs. Consulte a documentação de seu SGBD para determinar exatamente o que ele suporta.

Verificando um único valor

Já vimos um exemplo de teste de igualdade. Vamos dar uma olhada em alguns exemplos para demonstrar o uso de outros operadores.

Este primeiro exemplo lista todos os produtos que custam menos de $ 10:

Entrada q

```
SELECT prod_name, prod_price
FROM Products
WHERE prod_price < 10;
```

Saída q

```
prod_name           prod_price
------------------  ----------
Fish bean bag toy   3.49
Bird bean bag toy   3.49
Rabbit bean bag toy 3.49
8 inch teddy bear   5.99
12 inch teddy bear  8.99
Raggedy Ann         4.99
King doll           9.49
Queen doll          9.49
```

A próxima instrução obtém todos os produtos que custam $ 10 ou menos (embora o resultado seja o mesmo do exemplo anterior, porque não há itens com um preço de exatamente $ 10):

Entrada q

```
SELECT prod_name, prod_price
FROM Products
WHERE prod_price <= 10;
```

Verificando não correspondências

Este próximo exemplo lista todos os produtos não fabricados pelo fornecedor DLL01:

Entrada q

```
SELECT vend_id, prod_name
FROM Products
WHERE vend_id <> 'DLL01';
```

Saída q

```
vend_id      prod_name
----------   ------------------
BRS01        8 inch teddy bear
BRS01        12 inch teddy bear
BRS01        18 inch teddy bear
FNG01        King doll
FNG01        Queen doll
```

> **DICA: Quando usar aspas**
>
> Se você observar atentamente as condições usadas nas cláusulas WHERE anteriores, notará que alguns valores estão entre aspas simples e outros não. As aspas simples são usadas para delimitar uma string. Se você estiver comparando um valor com uma coluna cujo tipo de dados é string, as aspas delimitadoras são necessárias. As aspas não são usadas para delimitar valores usados com colunas numéricas.

A seguir temos o mesmo exemplo, exceto que esse usa o operador != em vez de <>:

Entrada q

```
SELECT vend_id, prod_name
FROM Products
WHERE vend_id != 'DLL01';
```

> **CUIDADO: != ou <>?**
>
> Geralmente você pode usar != e <> de forma intercambiável. No entanto, nem todos os SGBDs suportam ambas as formas de operador de não igualdade. Se estiver em dúvida, consulte a documentação de seu SGBD.

Verificando um intervalo de valores

Para verificar um intervalo de valores, você pode usar o operador BETWEEN. Sua sintaxe é um pouco diferente da sintaxe de outros operadores da cláusula WHERE porque ela requer dois valores: o início e o fim do intervalo. O operador BETWEEN pode ser usado, por exemplo, para verificar todos os produtos que custam entre $ 5 e $ 10 ou todas as datas que caem entre as datas de início e término especificadas.

O exemplo a seguir demonstra o uso do operador BETWEEN obtendo todos os produtos com um preço entre $ 5 e $ 10:

Entrada q

```
SELECT prod_name, prod_price
FROM Products
WHERE prod_price BETWEEN 5 AND 10;
```

Saída q

```
prod_name            prod_price
-------------------  ----------
8 inch teddy bear    5.99
12 inch teddy bear   8.99
King doll            9.49
Queen doll           9.49
```

Análise q

Como visto neste exemplo, quando BETWEEN é usado, dois valores devem ser especificados – o limite inferior e o limite superior do intervalo desejado. Os dois valores também devem ser separados pela palavra-chave AND. BETWEEN busca a correspondência com todos os valores no intervalo, incluindo os valores inicial e final especificados.

Verificando a ausência de valor

Quando uma tabela é criada, o projetista da tabela pode especificar se colunas individuais podem ou não conter nenhum valor. Quando uma coluna não contém valor, dizemos que ela contém um valor NULL.

NOVO TERMO: NULL

Nenhum valor, em oposição a um campo contendo 0, ou uma string vazia ou apenas espaços.

Para determinar se um valor é NULL, você não pode simplesmente verificar se o valor = NULL. Em vez disso, a instrução SELECT possui uma cláusula WHERE especial que pode ser usada para verificar colunas com valores NULL –- a cláusula IS NULL. A sintaxe se parece com:

Entrada q

```
SELECT prod_name
FROM Products
WHERE prod_price IS NULL;
```

Essa instrução retorna uma lista de todos os produtos que não têm preço (um campo prod_price vazio, não um preço de 0) e, como não há nenhum, nenhum dado é retornado. A tabela Customers, no entanto, contém colunas com valores NULL – a coluna cust_email conterá NULL se um cliente não tiver um endereço de e-mail cadastrado:

Entrada q

```
SELECT cust_name
FROM Customers
WHERE cust_email IS NULL;
```

Saída q

```
cust_name
----------
Kids Place
The Toy Store
```

> **DICA: Operadores específicos do SGBD**
>
> Muitos SGBDs estendem o conjunto padrão de operadores, fornecendo opções avançadas de filtragem. Consulte a documentação de seu SGBD para obter mais informações.

> **CUIDADO: NULL e nenhuma correspondência**
>
> Espera-se que, ao filtrar para selecionar todas as linhas que não possuem um valor específico, as linhas com um NULL sejam retornadas. Mas elas não são retornadas. NULL é mesmo estranho e as linhas com NULL na coluna do filtro não são retornadas quando se procuram correspondências ou não correspondências.

Resumo

Nesta lição, você aprendeu como filtrar os dados retornados usando a cláusula WHERE da instrução SELECT. Você aprendeu como testar a condição

de igualdade, não igualdade, maior que e menor que, e intervalos de valores, bem como valores NULL.

Desafios

1. Escreva uma instrução SQL para obter a ID do produto (prod_id) e o nome do produto (prod_name) da tabela Products, retornando apenas produtos com um preço de 9.49.
2. Escreva uma instrução SQL para obter a ID do produto (prod_id) e o nome do produto (prod_name) da tabela Products, retornando apenas produtos com um preço de 9 ou mais.
3. Agora vamos combinar as lições 3 e 4. Escreva uma instrução SQL para obter a lista exclusiva de números de pedidos (order_num) da tabela OrderItems, que contém 100 ou mais itens.
4. Mais um. Escreva uma instrução SQL que retorne o nome do produto (prod_name) e o preço do produto (prod_price) da tabela Products para todos os produtos com preços entre 3 e 6. Ah, e classifique os resultados por preço. (Existem várias soluções para este desafio e voltaremos a ele na próxima lição, mas você pode resolvê-lo usando o que aprendeu até agora.)

LIÇÃO 5

Filtragem avançada de dados

Nesta lição, você aprenderá como combinar cláusulas `WHERE` *para criar condições de pesquisa poderosas e sofisticadas. Você também aprenderá como usar os operadores* `NOT` *e* `IN`.

Combinando cláusulas WHERE

Todas as cláusulas `WHERE` introduzidas na *Lição 4 – Filtrando dados*, filtram os dados usando um único critério. Para maior controle do filtro, o SQL permite especificar várias cláusulas `WHERE`, as quais podem ser usadas de duas maneiras: como cláusulas `AND` ou como cláusulas `OR`.

NOVO TERMO: Operador

Uma palavra-chave especial usada para juntar ou alterar cláusulas dentro de uma cláusula `WHERE`. Também conhecidos como operadores lógicos.

Usando o operador AND

Para filtrar por mais de uma coluna, use o operador `AND` para acrescentar condições à sua cláusula `WHERE`. O código a seguir demonstra isso:

Entrada ▼

```
SELECT prod_id, prod_price, prod_name
FROM Products
WHERE vend_id = 'DLL01' AND prod_price <= 4;
```

Análise ▼

A instrução SQL acima obtém o nome e o preço do produto para todos os produtos fabricados pelo fornecedor `DLL01`, desde que o preço seja $ 4 ou menos. A cláusula `WHERE` nesta instrução `SELECT` é composta de duas condições, e a palavra-chave `AND` é usada para uni-las. `AND` instrui o software do sistema de gerenciamento de banco de dados a retornar apenas linhas que atendam a todas as condições especificadas. Se um produto é fabricado pelo fornecedor `DLL01`, mas custa mais de $ 4, ele não é

retornado. Da mesma forma, os produtos que custam menos de $ 4 e são fabricados por um fornecedor diferente do especificado não devem ser retornados. A saída gerada por esta instrução SQL é a seguinte:

Saída ▼

```
prod_id    prod_price    prod_name
-------    ----------    --------------------
BNBG02     3.4900        Bird bean bag toy
BNBG01     3.4900        Fish bean bag toy
BNBG03     3.4900        Rabbit bean bag toy
```

> **NOVO TERMO:** AND
>
> Uma palavra-chave usada em uma cláusula WHERE para especificar que apenas as linhas que correspondem a todas as condições especificadas devem ser obtidas.

O exemplo usado acima continha uma única cláusula AND e, portanto, era composto de duas condições de filtro. Também podem ser usadas condições de filtro adicionais, cada uma separada por uma palavra-chave AND.

> **NOTA: Nenhuma cláusula ORDER BY especificada**
>
> Para economizar espaço (e sua digitação), omiti a cláusula ORDER BY em muitos desses exemplos. Dessa forma, é perfeitamente possível que sua saída não corresponda exatamente à saída exibida neste livro. Embora o número de linhas retornadas deva sempre ser o mesmo, sua ordem pode variar. Claro, sinta-se à vontade para adicionar uma cláusula ORDER BY, se desejar; ela precisa vir depois da cláusula WHERE.

Usando o operador OR

O operador OR é exatamente o oposto do operador AND. O operador OR instrui o software do sistema de gerenciamento de banco de dados a retornar linhas que correspondam a qualquer uma das condições. De fato, a maioria dos melhores SGBDs nem avaliará a segunda condição em uma cláusula OR WHERE se a primeira condição já tiver sido atendida. (Se a primeira condição tiver sido atendida, a linha será retornada independentemente da segunda condição).

Observe a seguinte instrução SELECT:

Entrada ▼

```
SELECT prod_id, prod_price, prod_name
FROM Products
```

```
WHERE vend_id = 'DLL01' OR vend_id = 'BRS01';
```

Análise ▼

A instrução SQL acima obtém o nome e o preço do produto para qualquer produto fabricado por um dos dois fornecedores especificados. O operador `OR` diz ao SGBD que atenda a qualquer uma das duas condições, não a ambas. Se um operador `AND` fosse usado aqui, nenhum dado seria retornado (pois criaria uma cláusula `WHERE` que não corresponderia a nenhuma linha). A saída gerada por essa instrução SQL é a seguinte:

Saída ▼

```
prod_name              prod_price
-------------------    ----------
Fish bean bag toy      3.4900
Bird bean bag toy      3.4900
Rabbit bean bag toy    3.4900
8 inch teddy bear      5.9900
12 inch teddy bear     8.9900
18 inch teddy bear     11.9900
Raggedy Ann            4.9900
```

NOVO TERMO: OR

Uma palavra-chave usada em uma cláusula WHERE para especificar que qualquer linha que corresponda a uma das condições especificadas deve ser retornada.

Entendendo a ordem de avaliação

Cláusulas `WHERE` podem conter qualquer número de operadores `AND` e `OR`. A combinação dos dois permite executar filtragem sofisticada e complexa.

Mas a combinação de operadores `AND` e `OR` apresenta um problema interessante. Para demonstrar isso, veja um exemplo. Você precisa de uma lista de todos os produtos que custam $ 10 ou mais fabricados pelos fornecedores `DLL01` e `BRS01`. A seguinte instrução `SELECT` usa uma combinação de operadores `AND` e `OR` para criar uma cláusula `WHERE`:

Entrada ▼

```
SELECT prod_name, prod_price
FROM Products
WHERE vend_id = 'DLL01' OR vend_id = 'BRS01'
```

```
      AND prod_price >= 10;
```

Saída ▼

```
prod_name              prod_price
-------------------    ----------
Fish bean bag toy      3.4900
Bird bean bag toy      3.4900
Rabbit bean bag toy    3.4900
18 inch teddy bear     11.9900
Raggedy Ann            4.9900
```

Análise ▼

Veja os resultados acima. Quatro das linhas retornadas têm preços inferiores a $ 10 – portanto, obviamente, as linhas não foram filtradas conforme planejado. Por que isso aconteceu? A resposta é a ordem da avaliação. O SQL (como a maioria das linguagens) processa os operadores `AND` antes dos operadores `OR`. Quando o SQL vê a cláusula `WHERE` acima, ele lê *todos os produtos que custam $ 10 ou mais fabricados pelo fornecedor* `BRS01` *e todos os produtos fabricados pelo fornecedor* `DLL01`, *independentemente do preço*. Em outras palavras, como `AND` está acima na ordem da avaliação, os operadores errados foram unidos.

A solução para esse problema é usar parênteses para agrupar explicitamente operadores relacionados. Dê uma olhada na seguinte instrução `SELECT` e na saída:

Entrada ▼

```
SELECT prod_name, prod_price
FROM Products
WHERE (vend_id = 'DLL01' OR vend_id = 'BRS01')
      AND prod_price >= 10;
```

Saída ▼

```
prod_name              prod_price
-------------------    ----------
18 inch teddy bear     11.9900
```

Análise ▼

A única diferença entre essa instrução `SELECT` e a anterior é que, nessa instrução, as primeiras duas condições da cláusula `WHERE` estão entre

parênteses. Como os parênteses têm uma ordem de avaliação mais alta que os operadores AND ou OR, o SGBD primeiro filtra a condição OR entre esses parênteses. A instrução SQL se torna então *qualquer produto fabricado pelo fornecedor* DLL01 *ou pelo fornecedor* BRS01 *que custa $ 10 ou mais*, exatamente o que queremos.

DICA: Usando parênteses nas cláusulas WHERE

Sempre que você escrever cláusulas WHERE que usam os operadores AND e OR, use parênteses para agrupar explicitamente os operadores. Nunca confie na ordem de avaliação padrão, mesmo que ela seja exatamente o que você deseja. Não há desvantagem em usar parênteses e é sempre melhor eliminar qualquer ambiguidade.

Usando o operador IN

O operador IN é usado para especificar um intervalo de condições, sendo que qualquer uma delas pode ser correspondida. IN recebe uma lista delimitada por vírgula de valores válidos, todos entre parênteses. O exemplo a seguir demonstra isso:

Entrada ▼

```
SELECT prod_name, prod_price
FROM Products
WHERE vend_id  IN ('DLL01','BRS01')
ORDER BY prod_name;
```

Saída ▼

```
prod_name              prod_price
-------------------    ----------
12 inch teddy bear     8.9900
18 inch teddy bear     11.9900
8 inch teddy bear      5.9900
Bird bean bag toy      3.4900
Fish bean bag toy      3.4900
Rabbit bean bag toy    3.4900
Raggedy Ann            4.9900
```

Análise ▼

A instrução SELECT obtém todos os produtos fabricados pelo fornecedor DLL01 e pelo fornecedor BRS01. O operador IN é seguido por uma lista delimitada por vírgula de valores válidos e a lista inteira deve estar entre

parênteses.

Se você está pensando que o operador IN alcança o mesmo objetivo que o operador OR, está certo. A seguinte instrução SQL realiza exatamente o mesmo que o exemplo anterior:

Entrada ▼

```
SELECT prod_name, prod_price
FROM Products
WHERE vend_id  = 'DLL01' OR vend_id = 'BRS01'
ORDER BY prod_name;
```

Saída ▼

```
prod_name              prod_price
------------------     ----------
12 inch teddy bear     8.9900
18 inch teddy bear     11.9900
8 inch teddy bear      5.9900
Bird bean bag toy      3.4900
Fish bean bag toy      3.4900
Rabbit bean bag toy    3.4900
Raggedy Ann            4.9900
```

Por que usar o operador IN? As vantagens são:

- Quando você trabalha com longas listas de opções válidas, a sintaxe do operador IN é muito mais limpa e fácil de ler.
- A ordem da avaliação é mais fácil de gerenciar quando IN é usado em conjunto com outros operadores AND e OR.
- Os operadores IN quase sempre são executados mais rapidamente do que as listas de operadores OR (embora você não veja nenhuma diferença de desempenho com listas muito curtas como as que estamos usando aqui).
- A maior vantagem do IN é que o operador IN pode conter outra instrução SELECT, permitindo criar cláusulas WHERE altamente dinâmicas. Você verá isso em detalhes na *Lição 11 – Trabalhando com subconsultas*.

NOVO TERMO: IN

Uma palavra-chave usada em uma cláusula WHERE para especificar uma lista de valores a serem correspondidos usando uma comparação OR.

Usando o operador NOT

O operador NOT da cláusula WHERE tem apenas uma função: NOT nega qualquer condição que venha a seguir. Como NOT nunca é usado por si só (sempre é usado em conjunto com outro operador), sua sintaxe é um pouco diferente da sintaxe de todos os outros operadores. Ao contrário de outros operadores, a palavra-chave NOT pode ser usada antes da coluna a ser filtrada, não apenas depois.

NOVO TERMO: NOT

Uma palavra-chave usada em uma cláusula WHERE para negar uma condição.

O exemplo a seguir demonstra o uso do NOT. Para listar os produtos fabricados por todos os fornecedores, exceto o fornecedor DLL01, você pode escrever o seguinte:

Entrada ▼

```
SELECT prod_name
FROM Products
WHERE NOT vend_id = 'DLL01'
ORDER BY prod_name;
```

Saída ▼

```
prod_name
------------------
12 inch teddy bear
18 inch teddy bear
8 inch teddy bear
King doll
Queen doll
```

Análise ▼

O NOT aqui nega a condição que vem depois dele; portanto, em vez de corresponder vend_id a DLL01, o SGBD corresponde vend_id a qualquer coisa que não seja DLL01.

O exemplo anterior também poderia ter sido realizado usando o operador <> da seguinte maneira:

Entrada ▼

```
SELECT prod_name
FROM Products
WHERE vend_id  <> 'DLL01'
ORDER BY prod_name;
```

Saída ▼

```
prod_name
-----------------
12 inch teddy bear
18 inch teddy bear
8 inch teddy bear
King doll
Queen doll
```

Análise ▼

Por que usar NOT? Bem, para cláusulas WHERE simples, como as mostradas aqui, realmente não há vantagem em usar NOT. NOT é útil em cláusulas mais complexas. Por exemplo, o uso de NOT em conjunto com um operador IN facilita a localização de todas as linhas que não correspondem a uma lista de critérios.

NOTA: NOT no MariaDB

O MariaDB suporta o uso de NOT para negar as cláusulas IN, BETWEEN e EXISTS. Isso é diferente da maioria dos SGBDs que permitem usar NOT para negar quaisquer condições.

Resumo

Esta lição continuou do ponto em que a última lição parou e ensinou como combinar cláusulas WHERE com os operadores AND e OR. Você também aprendeu a gerenciar explicitamente a ordem da avaliação e como usar os operadores IN e NOT.

Desafios

1. Escreva uma instrução SQL para obter o nome do fornecedor (vend_name) da tabela Vendors, retornando apenas fornecedores na Califórnia (isso requer filtragem por país [USA] e estado [CA]; afinal, pode haver uma Califórnia fora dos EUA). Aqui está uma dica: o filtro

requer strings correspondentes.

2. Escreva uma instrução SQL para encontrar todos os pedidos que incluem pelo menos 100 dos itens `BR01`, `BR02` ou `BR03`. Você deve retornar o número do pedido (`order_num`), a ID do produto (`prod_id`) e a quantidade da tabela `OrderItems`, filtrando pela ID e pela quantidade do produto. Aqui está uma dica: dependendo de como você escreve seu filtro, talvez precise prestar atenção especial à ordem da avaliação.
3. Agora, vamos revisitar um desafio da lição anterior. Escreva uma instrução SQL que retorne o nome (`prod_name`) e o preço (`prod_price`) do produto da tabela `Products` para todos os produtos com preços entre `3` e `6`. Use um operador `AND` e classifique os resultados por preço.
4. O que há de errado com a seguinte instrução SQL? (Tente descobrir sem executá-la.)

```
SELECT vend_name
FROM Vendors
ORDER BY vend_name
WHERE vend_country = 'USA' AND vend_state = 'CA';
```

LIÇÃO 6

Usando a filtragem curinga

Nesta lição, você aprenderá o que são curingas, como são usados e como executar pesquisas curinga usando o operador LIKE *para filtragem sofisticada de dados obtidos.*

Usando o operador LIKE

Todos os operadores anteriores que estudamos filtram valores conhecidos. Seja comparando um ou mais valores, testando valores maiores ou menores que os conhecidos ou verificando um intervalo de valores, o denominador comum é que os valores usados na filtragem são conhecidos.

Mas filtrar dados dessa maneira nem sempre funciona. Por exemplo, como você poderia procurar por todos os produtos que contêm o texto "bean bag" no nome do produto? Isso não pode ser feito com operadores de comparação simples; esse é um trabalho para pesquisa com curingas. Usando curingas, você pode criar padrões de pesquisa que podem ser comparados com seus dados. Neste exemplo, se você deseja encontrar todos os produtos que contêm as palavras "bean bag", é possível construir um padrão de pesquisa curinga, permitindo encontrar esse texto "bean bag" em qualquer lugar do nome do produto.

> **NOVO TERMO: Curingas**
>
> Caracteres especiais usados para comparar partes de um valor.

> **NOVO TERMO: Padrão de pesquisa**
>
> Uma condição de pesquisa composta de texto literal, curingas ou qualquer combinação desses itens.

Os curingas são, na verdade, caracteres com significados especiais nas cláusulas WHERE e o SQL suporta vários tipos de curingas diferentes.

Para usar curingas nas cláusulas de pesquisa, use o operador LIKE, que diz ao SGBD que o seguinte padrão de pesquisa deve ser comparado usando

uma correspondência curinga em vez de uma correspondência de igualdade direta.

> **NOVO TERMO: Predicado**
>
> Quando um operador não é um operador? Quando ele é um “predicado”. Tecnicamente, LIKE é um predicado, não um operador. O resultado é o mesmo. Esteja ciente desse termo, caso você o encontre na documentação ou nos manuais SQL.

A pesquisa curinga pode ser usada apenas com campos de texto (strings); você não pode usar curingas para pesquisar campos de tipos de dados que não sejam texto.

Sinal de porcentagem (%) como curinga

O curinga usado com mais frequência é o sinal de porcentagem (%). Dentro de uma string de pesquisa, % significa compare qualquer número de ocorrências de qualquer caractere. Por exemplo, para encontrar os produtos que começam com a palavra Fish, você pode usar o seguinte SELECT:

Entrada q

```
SELECT prod_id, prod_name
FROM Products
WHERE prod_name LIKE 'Fish%';
```

Saída q

```
prod_id    prod_name
-------    ------------------
BNBG01     Fish bean bag toy
```

Análise q

Esse exemplo usa um padrão de pesquisa 'Fish%'. Quando essa cláusula é avaliada, qualquer valor que comece com Fish será obtido. O % diz ao SGBD que aceite qualquer caractere após a palavra Fish, independentemente de quantos caracteres existem.

> **NOTA: Distinção entre maiúsculas e minúsculas**
>
> Dependendo do nosso SGBD e de como ele está configurado, as pesquisas podem diferenciar maiúsculas de minúsculas. Nesse caso, 'fish%' não corresponderia a Fish bean bag toy.

Os curingas podem ser usados em qualquer lugar do padrão de pesquisa e vários curingas também podem ser usados. O exemplo a seguir usa dois

curingas, um em cada extremidade do padrão:

Entrada q

```
SELECT prod_id, prod_name
FROM Products
WHERE prod_name LIKE '%bean bag%';
```

Saída q

```
prod_id     prod_name
--------    --------------------
BNBG01      Fish bean bag toy
BNBG02      Bird bean bag toy
BNBG03      Rabbit bean bag toy
```

Análise q

O padrão de pesquisa `'% bean bag%'` significa compare a qualquer valor que contenha o texto `bean bag` em qualquer lugar dele, independentemente de qualquer caractere antes ou depois desse texto.

Os curingas também podem ser usados no meio de um padrão de pesquisa, embora isso raramente seja útil. O exemplo a seguir localiza todos os produtos que começam com `F` e terminam com `y`.

Entrada q

```
SELECT prod_name
FROM Products
WHERE prod_name LIKE 'F%y';
```

DICA: Pesquisando endereços de e-mail parciais

Há uma situação em que os curingas podem realmente ser úteis no meio de um padrão de pesquisa, isto é, no caso de pesquisar endereços de e-mail com base em um endereço parcial, como `WHERE email LIKE 'b%@forta.com'`.

É importante observar que, além de corresponder a um ou mais caracteres, `%` também corresponde a zero caracteres. `%` representa zero, um ou mais caracteres no local especificado no padrão de pesquisa.

NOTA: Observe os espaços à direita

Alguns SGBDs preenchem o conteúdo do campo com espaços. Por exemplo, se uma coluna espera 50 caracteres e o texto armazenado é "Fish bean bag toy" (17 caracteres), 33 espaços podem ser adicionados ao texto para preencher completamente a coluna. Esse preenchimento em geral não tem impacto real nos dados e em como eles são usados, mas pode afetar negativamente a instrução SQL que acabamos de usar. A

cláusula WHERE prod_name LIKE 'F%y' só encontrará uma correspondência com prod_name se ele começar com F e terminar com y, e se o valor for preenchido com espaços, então ele não terminará com y e o produto "Fish bean bag toy" não será obtido. Uma solução simples para esse problema é adicionar um segundo % ao padrão de pesquisa. 'F% y%' também fará a correspondência com os caracteres (ou espaços) após o y. Uma solução melhor seria remover os espaços usando funções, como você aprenderá na *Lição 8 – Usando Funções de Manipulação de Dados*.

CUIDADO: Atenção com o valor NULL

Pode parecer que o curinga % corresponde a qualquer coisa, mas há uma exceção: NULL. Nem a cláusula WHERE prod_name LIKE '%' corresponderá a uma linha com o valor NULL como o nome do produto.

Sublinhado (_) como curinga

Outro curinga útil é o sublinhado (_). O sublinhado é usado como o caractere %, mas, em vez de corresponder a vários caracteres, o sublinhado corresponde apenas a um único caractere.

NOTA: Curingas no DB2

O curinga _ não é suportado pelo DB2.

Veja este exemplo:

Entrada q

```
SELECT prod_id, prod_name
FROM Products
WHERE prod_name LIKE '__ inch teddy bear';
```

NOTA: Observe os espaços à direita

Como no exemplo anterior, pode ser necessário adicionar um curinga ao padrão para que este exemplo funcione.

Saída q **t**

```
prod_id    prod_name
--------   --------------------
BR02       12 inch teddy bear
BR03       18 inch teddy bear
```

Análise q

O padrão de pesquisa usado na cláusula WHERE especificou dois curingas seguidos por texto literal. Os resultados mostrados são as únicas linhas que correspondem ao padrão de pesquisa: o sublinhado corresponde a 12 na primeira linha e 18 na segunda linha. O produto "8 inch teddy bear" não

correspondeu ao padrão de pesquisa porque o padrão de pesquisa exigia duas correspondências curinga, não apenas uma. Por outro lado, o SELECT seguinte usa o curinga % e retorna três produtos correspondentes:

Entrada q

```
SELECT prod_id, prod_name
FROM Products
WHERE prod_name LIKE '% inch teddy bear';
```

Saída q

```
prod_id     prod_name
--------    --------------------
BR01        8 inch teddy bear
BR02        12 inch teddy bear
BNR3        18 inch teddy bear
```

Ao contrário do %, que pode corresponder a zero caracteres, o _ sempre corresponde a um caractere – nem mais nem menos.

Colchetes como curinga ([])

Os colchetes ([]) como curinga são usados para especificar um conjunto de caracteres e qualquer um deles deve corresponder a um caractere na posição especificada (o local do curinga).

> **NOTA: Conjuntos não são comumente suportados**
>
> Ao contrário dos curingas descritos até o momento, o uso de [] para criar conjuntos não é suportado por todos os SGBDs. Os conjuntos são suportados no SQL Server, mas não no MySQL, Oracle, DB2 e SQLite. Consulte a documentação de seu SGBD para determinar se os conjuntos são suportados.

Por exemplo, para localizar todos os contatos cujos nomes começam com a letra J ou a letra M, você pode fazer o seguinte:

Entrada q

```
SELECT cust_contact
FROM Customers
WHERE cust_contact LIKE '[JM]%'
ORDER BY cust_contact;
```

Saída q

```
cust_contact
-----------------
```

```
Jim Jones
John Smith
Michelle Green
```

Análise q

A cláusula WHERE nessa instrução é '[JM]%'. Esse padrão de pesquisa usa dois curingas diferentes. O [JM] corresponde a qualquer nome de contato que comece com qualquer uma das letras entre colchetes e corresponde apenas a um único caractere. Portanto, qualquer nome com mais de um caractere não será retornado. O curinga % após o [JM] corresponde a qualquer número de caracteres após o primeiro caractere, retornando os resultados desejados.

Esse curinga pode ser negado prefixando os caracteres com ^ (o caractere de circunflexo). Por exemplo, o seguinte exemplo corresponde a qualquer nome de contato que não comece com a letra J ou a letra M (o oposto do exemplo anterior):

Entrada q

```
SELECT cust_contact
FROM Customers
WHERE cust_contact LIKE '[^JM]%'
ORDER BY cust_contact;
```

Obviamente, você pode obter o mesmo resultado usando o operador NOT. A única vantagem de usar ^ é que ele pode simplificar a sintaxe se você estiver usando várias cláusulas WHERE:

Entrada q

```
SELECT cust_contact
FROM Customers
WHERE NOT cust_contact LIKE '[JM]%'
ORDER BY cust_contact;
```

Dicas para usar curingas

Como você pode ver, os curingas do SQL são extremamente poderosos. Mas esse poder tem um preço: as pesquisas com curinga levam muito mais tempo para serem processadas do que quaisquer outros tipos de pesquisa discutidos previamente. Aqui estão algumas regras para se ter em mente ao usar curingas:

- Não use curingas em excesso. Se outro operador de pesquisa servir, use-o.
- Ao usar curingas, tente não os usar no início do padrão de pesquisa, a menos que seja absolutamente necessário. Os padrões de pesquisa que começam com curingas são de processamento mais lento.
- Preste muita atenção ao posicionamento dos símbolos curinga. Se eles forem posicionados incorretamente, talvez você não retorne os dados que pretendia.

Dito isso, os curingas são uma ferramenta de pesquisa importante e útil, e você os usará com frequência.

Resumo

Nesta lição, você aprendeu o que são curingas e como usar curingas SQL nas suas cláusulas WHERE. Também aprendeu que os curingas devem ser usados com cuidado e nunca devem ser usados em excesso.

Desafios

1. Escreva uma instrução SQL para obter o nome do produto (prod_name) e a descrição (prod_desc) da tabela Products, retornando apenas os produtos em que a palavra toy esteja na descrição.
2. Agora vamos inverter. Escreva uma instrução SQL para obter o nome do produto (prod_name) e a descrição (prod_desc) da tabela Products, retornando apenas os produtos em que a palavra toy não esteja na descrição. E, desta vez, classifique os resultados por nome do produto.
3. Escreva uma instrução SQL para obter o nome do produto (prod_name) e a descrição (prod_desc) da tabela Products, retornando apenas produtos em que as palavras toy e carrots apareçam na descrição. Existem algumas maneiras de fazer isso, mas, para este desafio, use AND e duas comparações LIKE.
4. Este próximo é um pouco mais complicado. Não mostrei essa sintaxe especificamente, mas veja se você consegue descobrir de qualquer maneira com base no que aprendeu até agora. Escreva uma instrução

SQL para obter o nome do produto (prod_name) e a descrição (prod_desc) da tabela Products, retornando apenas produtos em que as palavras toy e carrots apareçam na descrição na seguinte ordem (a palavra toy antes da palavra carrots). Aqui está uma dica: você só precisará de um LIKE com três símbolos % para fazer isso.

LIÇÃO 7

Criando campos calculados

Nesta lição, você aprenderá o que são os campos calculados, como criá-los e como usar aliases para se referir a eles de dentro de sua aplicação.

Compreendendo campos calculados

Os dados armazenados nas tabelas de um banco de dados muitas vezes não estão disponíveis no formato exato que suas aplicações necessitam. Aqui estão alguns exemplos:

- Você precisa exibir um campo contendo o nome de uma empresa junto com a localização dela, mas essas informações são armazenadas em colunas separadas da tabela.
- Cidade, estado e CEP são armazenados em colunas separadas (como deveriam estar), mas o programa de impressão de etiquetas postais precisa que eles sejam obtidos como um campo corretamente formatado.
- Os dados da coluna estão em maiúsculas e minúsculas, e seu relatório precisa de todos os dados apresentados em maiúsculas.
- Uma tabela OrderItems armazena o preço e a quantidade do item, mas não o preço expandido (preço multiplicado pela quantidade) de cada item. Para imprimir faturas, você precisa desse preço expandido.
- Você precisa de valores totais, valores médios ou outros cálculos com base nos dados da tabela.

Em cada um desses exemplos, os dados armazenados na tabela não são exatamente o que sua aplicação precisa. Em vez de obter os dados como eles estão e, então, reformatá-los dentro de sua aplicação cliente ou relatório, o que você de fato quer é obter dados convertidos, calculados ou reformatados diretamente do banco de dados.

É aqui que entram os campos calculados. Ao contrário de todas as colunas que obtivemos nas lições até agora, campos calculados não existem

realmente nas tabelas de banco de dados. Em vez disso, um campo calculado é criado durante a execução dentro de uma instrução SELECT.

> **NOVO TERMO: Campo**
>
> Essencialmente significa a mesma coisa que *coluna* e muitas vezes esses termos são usados de modo intercambiável, embora as colunas de banco de dados sejam tipicamente chamadas de *colunas* e o termo *campos* seja geralmente usado em conjunto com campos calculados.

É importante notar que apenas o banco de dados sabe quais colunas em uma instrução SELECT são colunas reais da tabela e quais são campos calculados. Da perspectiva de um cliente (por exemplo, sua aplicação), os dados de um campo calculado são retornados da mesma forma que dados de qualquer outra coluna.

> **DICA: Formatação no cliente x formatação no servidor**
>
> Muitas das conversões e reformatações que podem ser realizadas dentro de instruções SQL também podem ser realizadas diretamente em sua aplicação cliente. No entanto, como regra geral, é muito mais rápido realizar essas operações no servidor do banco de dados do que as executar dentro da aplicação cliente.

Concatenando campos

Para demonstrar como trabalhar com campos calculados, vamos começar com um exemplo simples – criando um título composto de duas colunas.

A tabela Vendors contém informações de nome e endereço do fornecedor. Imagine que você está gerando um relatório de fornecedor e precisa listar o local dele como parte do nome do fornecedor, no formato nome (local).

O relatório quer um único valor e os dados na tabela são armazenados em duas colunas: vend_name e vend_country. Além disso, você precisa colocar vend_country entre parênteses, e definitivamente esses caracteres não são armazenados na tabela do banco de dados. A instrução SELECT que retorna os nomes e os locais do fornecedor é simples, mas como você criaria esse valor combinado?

> **NOVO TERMO: Concatenar**
>
> Juntar valores (anexando-os) para formar um único valor longo.

A solução é concatenar as duas colunas. Nas instruções SELECT você pode

concatenar colunas usando um operador especial. Dependendo do SGBD que você está usando, esse operador pode ser um sinal de mais (+) ou duas barras verticais (||). E, no caso do MySQL e do MariaDB, uma função especial deve ser usada como mostrado a seguir.

> **NOTA: + ou || ?**
>
> O SQL Server usa + para concatenação. O DB2, o Oracle, o PostgreSQL e o SQLite suportam ||. Consulte a documentação de seu SGBD para obter mais detalhes.

Aqui está um exemplo usando o sinal de mais:

Entrada q

```
SELECT vend_name + '(' + vend_country + ')'
FROM Vendors
ORDER BY vend_name;
```

Saída q

```
---------------------------------------------------
Bear Emporium              (USA         )
Bears R Us                 (USA         )
Doll House Inc.            (USA         )
Fun and Games              (England     )
Furball Inc.               (USA         )
Jouets et ours             (France      )
```

A seguir temos a mesma instrução, mas usando a sintaxe ||:

Entrada q

```
SELECT vend_name || '(' || vend_country || ')'
FROM Vendors
ORDER BY vend_name;
```

Saída q

```
---------------------------------------------------
Bear Emporium              (USA         )
Bears R Us                 (USA         )
Doll House Inc.            (USA         )
Fun and Games              (England     )
Furball Inc.               (USA         )
Jouets et ours             (France      )
```

E aqui está o que você precisa fazer se estiver usando MySQL ou MariaDB:

Entrada q

```
SELECT Concat(vend_name, ' (', vend_country, ')')
FROM Vendors
ORDER BY vend_name;
```

Análise q

As instruções SELECT anteriores concatenam os seguintes elementos:

- O nome armazenado na coluna vend_name.
- Uma string contendo um espaço e um parêntese de abertura.
- O país armazenado na coluna vend_country.
- Uma string contendo o parêntese de fechamento.

Como você pode ver na saída mostrada anteriormente, a instrução SELECT retorna uma única coluna (um campo calculado) contendo todos esses quatro elementos como uma unidade.

Observe novamente a saída retornada pela instrução SELECT. As duas colunas que são incorporadas ao campo calculado são preenchidas com espaços. Muitos bancos de dados (embora nem todos) salvam os valores de texto preenchidos de acordo com a largura da coluna; portanto, seus próprios resultados podem não conter esses espaços irrelevantes. Para retornar os dados formatados corretamente, remova esses espaços de preenchimento. Isso pode ser feito usando a função SQL RTRIM() da seguinte maneira:

Entrada q

```
SELECT RTRIM(vend_name) + ' (' + RTRIM(vend_country) + ')'
FROM Vendors
ORDER BY vend_name;
```

Saída q

```
---------------------------------------------------------
Bear Emporium (USA)
Bears R Us (USA)
Doll House Inc. (USA)
Fun and Games (England)
Furball Inc. (USA)
Jouets et ours (France)
```

A seguir temos a mesma instrução, mas usando a sintaxe ||:

Entrada q

```
SELECT RTRIM(vend_name) || ' (' || RTRIM(vend_country) || ')'
FROM Vendors
ORDER BY vend_name;
```

Saída q

```
-----------------------------------------------------------
Bear Emporium (USA)
Bears R Us (USA)
Doll House Inc. (USA)
Fun and Games (England)
Furball Inc. (USA)
Jouets et ours (France)
```

Análise q

A função RTRIM() remove todos os espaços à direita de um valor. Quando você usa RTRIM(), as colunas individuais são todas ajustadas corretamente.

> **NOTA: As funções TRIM**
>
> A maioria dos SGBDs suporta RTRIM() (que, como acabamos de ver, remove espaços à direita de uma string), bem como LTRIM(), que remove espaços à esquerda de uma string, e TRIM(), que remove espaços tanto à direita quanto à esquerda.

Usando aliases

A instrução SELECT usada para concatenar o campo de endereço funciona bem, como visto na saída anterior. Mas qual é o nome dessa nova coluna calculada? Bem, a verdade é que ela não tem nome; é simplesmente um valor. Embora isso seja bom se você estiver apenas visualizando os resultados em uma ferramenta de consulta SQL, uma coluna sem nome não pode ser usada em uma aplicação cliente, porque não há como fazer referência a essa coluna.

Para resolver esse problema, o SQL suporta aliases de coluna. Um alias é apenas isso, um nome alternativo para um campo ou valor. Os aliases são atribuídos com a palavra-chave AS. Dê uma olhada na seguinte instrução SELECT:

Entrada q

```
SELECT RTRIM(vend_name) + ' (' + RTRIM(vend_country) + ')'
 AS vend_title
```

```
FROM Vendors
ORDER BY vend_name;
```

Saída q

```
vend_title
------------------------------------------------------------
Bear Emporium (USA)
Bears R Us (USA)
Doll House Inc. (USA)
Fun and Games (England)
Furball Inc. (USA)
Jouets et ours (France)
```

A seguir temos a mesma instrução, mas usando a sintaxe ||:

Entrada q

```
SELECT RTRIM(vend_name) || ' (' || RTRIM(vend_country) || ')'
 AS vend_title
FROM Vendors
ORDER BY vend_name;
```

E aqui está o equivalente para uso com MySQL e MariaDB:

Entrada q

```
SELECT Concat(RTrim(vend_name), ' (',
      RTrim(vend_country), ')') AS vend_title
FROM Vendors
ORDER BY vend_name;
```

Análise q

A própria instrução SELECT é a mesma usada no trecho de código anterior, exceto que aqui o campo calculado é seguido pelo texto AS vend_title. Isso instrui o SQL a criar um campo calculado chamado vend_title contendo o cálculo especificado. Como você pode ver na saída, os resultados são os mesmos de antes, mas a coluna agora é denominada vend_title e qualquer aplicação cliente pode se referir a essa coluna pelo nome, como faria com qualquer coluna real da tabela.

NOTA: AS é muitas vezes opcional

O uso da palavra-chave AS é opcional em muitos SGBDs, mas o uso é considerado uma prática recomendada.

DICA: Outros usos para aliases

Os aliases também têm outros usos. Alguns usos comuns incluem renomear uma coluna se o nome real da coluna da tabela contiver caracteres ilegais (por exemplo, espaços) e expandir os nomes das colunas se os nomes originais forem ambíguos ou facilmente mal interpretados.

CUIDADO: Nomes de aliases

Aliases podem ser palavras simples ou strings completas. Se o último for usado, a string deve ser colocada entre aspas. Essa prática é válida, mas fortemente desencorajada. Embora os nomes com várias palavras sejam realmente bastante legíveis, eles criam todos os tipos de problemas para muitas aplicações clientes – tanto que um dos usos mais comuns dos aliases é renomear nomes de colunas com várias palavras para nomes com uma única palavra (como explicado anteriormente).

NOTA: Colunas derivadas

Às vezes, os aliases são chamados de *colunas derivadas*, portanto, independentemente do termo que você encontrar, eles têm o mesmo significado.

Executando cálculos matemáticos

Outro uso frequente para campos calculados é executar cálculos matemáticos nos dados obtidos. Vamos observar um exemplo. A tabela Orders contém todos os pedidos recebidos, e a tabela OrderItems contém os itens individuais em cada pedido. A seguinte instrução SQL obtém todos os itens do pedido 20008:

Entrada q

```
SELECT prod_id, quantity, item_price
FROM OrderItems
WHERE order_num = 20008;
```

Saída q

```
prod_id      quantity      item_price
----------   -----------   ---------------------
RGAN01       5             4.9900
BR03         5             11.9900
BNBG01       10            3.4900
BNBG02       10            3.4900
BNBG03       10            3.4900
```

A coluna item_price contém o preço unitário de cada item em um pedido. Para expandir o preço do item (preço do item multiplicado pela quantidade

pedida), basta fazer o seguinte:

Entrada q

```
SELECT prod_id,
    quantity,
    item_price,
    quantity*item_price AS expanded_price
FROM OrderItems
WHERE order_num = 20008;
```

Saída q

```
prod_id    quantity     item_price     expanded_price
---------- ----------   ------------   -----------------
RGAN01     5            4.9900         24.9500
BR03       5            11.9900        59.9500
BNBG01     10           3.4900         34.9000
BNBG02     10           3.4900         34.9000
BNBG03     10           3.4900         34.9000
```

Análise q

A coluna expanded_price mostrada na saída anterior é um campo calculado; o cálculo é simplesmente quantity*item_price. A aplicação cliente agora pode usar essa nova coluna calculada da mesma forma que faria com qualquer outra coluna.

O SQL suporta os operadores matemáticos básicos listados na Tabela 7.1. Além disso, você pode usar parênteses para estabelecer a ordem de precedência. Consulte a *Lição 5 – Filtragem avançada de dados*, para obter uma explicação sobre precedência.

DICA: Como testar os cálculos

SELECT oferece uma ótima maneira de testar e experimentar com funções e cálculos. Embora SELECT seja usado geralmente para obter dados de uma tabela, a cláusula FROM pode ser omitida para simplesmente acessar e trabalhar com expressões. Por exemplo, SELECT 3 * 2; retornaria 6, SELECT Trim(' abc '); retornaria abc e SELECT Curdate(); usa a função Curdate() para retornar a data e a hora atuais (no MySQL e no MariaDB, por exemplo). Você entendeu: use SELECT para testar conforme necessário.

Tabela 7.1 – Operadores matemáticos SQL

Operador	Descrição
+	Adição

-	Subtração
*	Multiplicação
/	Divisão

Resumo

Nesta lição, você aprendeu o que são campos calculados e como criá-los. Usou exemplos demonstrando o uso de campos calculados para concatenação de strings e operações matemáticas. Além disso, aprendeu a criar e usar aliases para que sua aplicação possa se referir a campos calculados.

Desafios

1. Um uso comum para aliases é renomear campos da coluna da tabela nos resultados obtidos (talvez para atender às necessidades específicas de relatório ou do cliente). Escreva uma instrução SQL que obtém vend_id, vend_name, vend_address e vend_city da tabela Vendors, renomeando vend_name para vname, vend_city para vcity e vend_address para vaddress. Classifique os resultados por nome do fornecedor (você pode usar o nome original ou o valor renomeado).
2. Nosso exemplo de loja está fazendo uma liquidação e todos os produtos têm 10% de desconto. Escreva uma instrução SQL que retorne prod_id, prod_price e sale_price da tabela Products. sale_price é um campo calculado que contém o preço de venda. Aqui está uma dica: você pode multiplicar por 0,9 para obter 90% do valor original (e, portanto, 10% de desconto).

LIÇÃO 8

Usando funções de manipulação de dados

Nesta lição, você aprenderá o que são funções, quais tipos de funções os SGBDs suportam e como usar essas funções. Também aprenderá por que o uso da função SQL pode ser muito problemático.

Compreendendo funções

Como quase qualquer outra linguagem de computador, o SQL suporta o uso de funções para manipular dados. Funções são operações geralmente executadas em dados, em geral para facilitar a conversão e manipulação, e são uma parte importante da sua caixa de ferramentas SQL.

Um exemplo de função é `RTRIM()`, que usamos na última lição para remover espaços do final de uma string.

O problema com as funções

Antes de continuar com esta lição e testar os exemplos, lembre-se de que, infelizmente, o uso de funções SQL pode ser altamente problemático.

Ao contrário das instruções SQL (por exemplo, `SELECT`), que na maioria das vezes são suportadas por todos os SGBDs igualmente, as funções tendem a ser muito específicas de cada SGBD. Na verdade, poucas funções são suportadas de maneira idêntica por todos os principais SGBDs. Embora todos os tipos de funcionalidade estejam geralmente disponíveis em cada SGBD, os nomes ou a sintaxe das funções podem diferir bastante. Para demonstrar o quão problemático isso pode ser, a Tabela 8.1 lista três funções comumente necessárias e sua sintaxe, conforme empregadas por vários SGBDs:

Tabela 8.1 – Diferenças da função por SGBD

Função	Sintaxe
Extrair	O DB2, o Oracle, o PostgreSQL e o SQLite usam `SUBSTR()`. O MariaDB, o MySQL

parte de uma string	e o SQL Server usam `SUBSTRING()`.
Conversão de tipo de dados	O Oracle usa várias funções, uma para cada tipo de conversão. O DB2, o PostgreSQL e o SQL Server usam `CAST()`. O MariaDB, o MySQL e o SQL Server usam `CONVERT()`.
Obter a data atual	O DB2 e o PostgreSQL usam `CURRENT_DATE`. O MariaDB e o MySQL usam `CURDATE()`. O Oracle usa `SYSDATE`. O SQL Server usa `GETDATE()`. O SQLite usa `DATE()`.

Como você pode ver, ao contrário das instruções SQL, as funções SQL não são *portáveis*. Isso significa que o código que você escreve para uma implementação SQL específica pode não funcionar em outra implementação.

NOVO TERMO: Portável

Código escrito para ser executado em vários sistemas diferentes.

Com a portabilidade de código em mente, alguns programadores SQL optam por não usar nenhum recurso específico da implementação. Embora essa seja uma visão um tanto nobre e idealista, nem sempre é do melhor interesse do desempenho da aplicação. Se você optar por não usar essas funções, fará com que o código de sua aplicação trabalhe mais, pois ela deve usar outros métodos para fazer o que o SGBD poderia ter feito com mais eficiência.

DICA: Você deve usar funções?

Então agora você está tentando decidir se deve ou não usar funções. Bem, essa decisão é sua, e não há escolha certa ou errada. Se você decidir usar funções, certifique-se de comentar bem o seu código, para que posteriormente (você ou outro desenvolvedor) saiba exatamente para qual implementação SQL você estava escrevendo.

Usando funções

A maioria das implementações SQL suporta os seguintes tipos de funções:

- As funções de texto são usadas para manipular strings de texto (por exemplo, remover ou inserir espaços nos valores e converter valores para maiúsculas e minúsculas).
- As funções numéricas são usadas para executar operações

matemáticas em dados numéricos (por exemplo, retornar números absolutos e executar cálculos algébricos).

- As funções de data e hora são usadas para manipular valores de data e hora e para extrair componentes específicos desses valores (por exemplo, retornar diferenças entre datas e verificar a validade da data).
- As funções de formatação são usadas para gerar saídas amigáveis ao usuário (por exemplo, exibir datas nos idiomas e formatos locais ou moedas com os símbolos e posicionamento de vírgula corretos).
- As funções do sistema retornam informações específicas para o SGBD usado (por exemplo, retornar informações de login do usuário).

Na última lição, você viu uma função usada como parte de uma lista de colunas em uma instrução `SELECT`, mas isso nem todas as funções podem fazer. Você pode usar funções em outras partes da instrução `SELECT` (por exemplo, na cláusula `WHERE`), bem como em outras instruções SQL (mais sobre isso nas lições posteriores).

Funções de manipulação de texto

Você já viu um exemplo de funções de manipulação de texto: na última lição, a função `RTRIM()` foi usada para remover espaços em branco no final de um valor da coluna. Aqui está outro exemplo, desta vez usando a função `UPPER()`:

Entrada ▼

```
SELECT vend_name, UPPER(vend_name) AS vend_name_upcase
FROM Vendors
ORDER BY vend_name;
```

Saída ▼

```
vend_name                 vend_name_upcase
------------------------  ----------------
Bear Emporium             BEAR EMPORIUM
Bears R Us                BEARS R US
Doll House Inc.           DOLL HOUSE INC.
Fun and Games             FUN AND GAMES
Furball Inc.              FURBALL INC.
Jouets et ours            JOUETS ET OURS
```

Como você pode ver, `UPPER()` converte texto em maiúsculas e, portanto, nesse exemplo, cada fornecedor é listado duas vezes – primeiro exatamente como armazenado na tabela `Vendors` e depois convertido em maiúsculas como coluna `vend_name_upcase`.

DICA: MAIÚSCULAS, minúsculas, MisturaDeMaiúsculasEMinúsculas

Como já deve estar claro, as funções SQL não diferenciam maiúsculas de minúsculas; portanto, você pode usar `upper()`, `UPPER()`, `Upper()` ou `substr()`, `SUBSTR()`, `SubStr()` e assim por diante. O uso de maiúsculas ou minúsculas é uma preferência do usuário, então faça o que quiser, mas seja consistente e não fique alterando os estilos no seu código; isso torna o SQL realmente difícil de ler.

A Tabela 8.2 lista algumas funções de manipulação de texto comumente usadas.

Tabela 8.2 – Funções de manipulação de texto comumente usadas

Função	Descrição
`LEFT()` (ou use a função substring)	Retorna caracteres da esquerda da string
`LENGTH()` (também `DATALENGTH()` ou `LEN()`)	Retorna o comprimento de uma string
`LOWER()`	Converte o texto para minúsculas
`LTRIM()`	Remove espaços em branco da esquerda da string
`RIGHT()` (ou use a função substring)	Retorna caracteres da direita da string
`RTRIM()`	Remove espaços em branco da direita da string
`SUBSTR()` ou `SUBSTRING()`	Extrai parte de uma string (conforme observado na Tabela 8.1)
`SOUNDEX()`	Retorna o valor `SOUNDEX` de uma string
`UPPER()`	Converte o texto para maiúsculas

Um item da Tabela 8.2 requer explicações adicionais. `SOUNDEX` é um algoritmo que converte qualquer string de texto em um padrão alfanumérico que descreve a representação fonética desse texto. `SOUNDEX` leva em consideração caracteres e sílabas com sons semelhantes, permitindo que strings sejam comparadas pela maneira como soam, e não como são digitadas. Embora `SOUNDEX` não seja um conceito SQL, a maioria dos SGBDs oferece suporte para `SOUNDEX`.

NOTA: Suporte a SOUNDEX

SOUNDEX() não é suportado pelo PostgreSQL e, portanto, o exemplo a seguir não funcionará nesse SGBD.

Além disso, ele só estará disponível no SQLite se a opção de compilação SQLITE_SOUNDEX for usada quando o SQLite for criado e, como essa não é a opção de compilação padrão, a maioria das implementações do SQLite não suportará SOUNDEX().

Aqui está um exemplo usando a função SOUNDEX(). O cliente Kids Place está na tabela Customers e tem um contato chamado Michelle Green. Mas e se isso fosse um erro de digitação e o contato realmente se chamasse Michael Green? Obviamente, procurar pelo nome correto do contato não retornaria dados, como mostrado aqui:

Entrada ▼

```
SELECT cust_name, cust_contact
FROM Customers
WHERE cust_contact = 'Michael Green';
```

Saída▼

```
cust_name                      cust_contact
-------------------------      --------------------------
```

Agora tente a mesma pesquisa usando a função SOUNDEX() para achar a correspondência de todos os nomes de contatos que soam semelhantes a Michael Green:

Entrada ▼

```
SELECT cust_name, cust_contact
FROM Customers
WHERE SOUNDEX(cust_contact) = SOUNDEX('Michael Green');
```

Saída ▼

```
cust_name                      cust_contact
-------------------------      --------------------------
Kids Place                     Michelle Green
```

Análise ▼

Nesse exemplo, a cláusula WHERE usa a função SOUNDEX() para converter o valor da coluna cust_contact e da string de pesquisa nos valores SOUNDEX. Como Michael Green e Michelle Green soam parecido, seus valores SOUNDEX são equivalentes e, portanto, WHERE filtrou corretamente os dados desejados.

Funções de manipulação de data e hora

A data e a hora são armazenadas em tabelas usando datatypes e cada SGBD usa as próprias variedades especiais. Os valores de data e hora são armazenados em formatos especiais para que sejam classificados ou filtrados de forma rápida e eficiente, além de economizar espaço de armazenamento físico.

O formato interno usado para armazenar datas e horas geralmente não tem utilidade para suas aplicações e, portanto, as funções de data e hora quase sempre são usadas para ler, expandir e manipular esses valores. Por isso, as funções de manipulação de data e hora são algumas das funções mais importantes da linguagem SQL. Infelizmente, elas também tendem a ser as mais inconsistentes e menos portáveis.

Para demonstrar o uso de uma função de manipulação de data, aqui está um exemplo simples. A tabela `Orders` contém todos os pedidos junto com uma data de pedido. Para obter todos os pedidos feitos em um ano específico, você precisa filtrar por data do pedido, mas não o valor inteiro da data, apenas a parte do ano. Obviamente, é necessário extrair o ano da data completa.

Para obter uma lista de todos os pedidos feitos em 2020 no SQL Server, faça o seguinte:

Entrada ▼

```
SELECT order_num
FROM Orders
WHERE DATEPART(yy, order_date) = 2020;
```

Saída ▼

```
order_num
-----------
20005
20006
20007
20008
20009
```

Análise ▼

Esse exemplo usa a função DATEPART(), que, como o nome sugere, retorna uma parte de uma data. DATEPART() recebe dois parâmetros: a parte a ser retornada e a data da qual retornar. Em nosso exemplo, DATEPART() especifica yy como a parte desejada e retorna apenas o ano da coluna order_date. Comparando isso com 2020, a cláusula WHERE pode filtrar apenas os pedidos daquele ano.

Esta é a versão do PostgreSQL, que usa uma função semelhante chamada DATE_PART():

Entrada ▼

```
SELECT order_num
FROM Orders
WHERE DATE_PART('year', order_date) = 2020;
```

O Oracle não tem função DATEPART(), mas existem várias outras funções de manipulação de data que podem ser usadas para realizar a mesma operação. Aqui está um exemplo:

Entrada ▼

```
SELECT order_num
FROM Orders
WHERE EXTRACT(year FROM order_date) = 2020;
```

Análise ▼

Nesse exemplo, a função EXTRACT() é usada para extrair parte da data com year especificando qual parte da data deve ser extraída. O valor retornado é então comparado a 2020.

> **DICA: O PostgreSQL oferece suporte a Extract()**
>
> O PostgreSQL também suporta a função Extract(), portanto, essa técnica funcionará (além de usar DatePart() como visto anteriormente).

Outra maneira de realizar essa mesma tarefa é usar o operador BETWEEN.

Entrada ▼

```
SELECT order_num
FROM Orders
WHERE order_date BETWEEN to_date('2020-01-01', 'yyyy-mm-dd')
 AND to_date('2020-12-31', 'yyyy-mm-dd');
```

Análise ▼

Nesse exemplo, a função `to_date()` do Oracle é usada para converter duas strings em datas. Uma contém a data de 1º de janeiro de 2020 e a outra contém a data de 31 de dezembro de 2020. Um operador `BETWEEN` padrão é usado para localizar todos os pedidos entre essas duas datas. É importante observar que esse mesmo código não funcionaria com o SQL Server porque ele não oferece suporte à função `to_date()`. No entanto, se você substituiu `to_date()` por `DATEPART()`, poderia realmente usar esse tipo de instrução.

O DB2, o MySQL e o MariaDB têm todos os tipos de funções de manipulação de data, mas não `DATEPART()`. Os usuários do DB2, MySQL e MariaDB podem usar uma função chamada `YEAR()` para extrair o ano de uma data:

Entrada ▼

```
SELECT order_num
FROM Orders
WHERE YEAR(order_date) = 2020;
```

O SQLite é um pouco mais complicado:

Entrada ▼

```
SELECT order_num
FROM Orders
WHERE strftime('%Y', order_date) = '2020';
```

O exemplo mostrado aqui extraiu e usou parte de uma data (o ano). Para filtrar por um mês específico, você pode usar o mesmo processo, especificando um operador `AND` e comparações de ano e mês.

Os SGBDs geralmente oferecem muito mais do que simples extração de partes de data. A maioria tem funções para comparar datas, realizar aritmética com base em datas, formatar datas e muito mais. Mas, como você viu, as funções de manipulação de data e hora são particularmente específicas do SGBD. Consulte a documentação do seu SGBD para obter a lista das funções de manipulação de data e hora que ele suporta.

Funções de manipulação numérica

As funções de manipulação numérica fazem exatamente isso – manipulam dados numéricos. Essas funções tendem a ser usadas principalmente para cálculos algébricos, trigonométricos ou geométricos e, portanto, não são tão frequentemente usadas como strings ou funções de manipulação de data e hora.

O irônico é que, de todas as funções encontradas nos SGBDs principais, as funções numéricas são as mais uniformes e consistentes. A Tabela 8.3 lista algumas das funções de manipulação numérica mais comumente usadas.

Tabela 8.3 – Funções de manipulação numérica comumente usadas

Função	Descrição
`ABS()`	Retorna o valor absoluto de um número
`COS()`	Retorna o cosseno trigonométrico de um ângulo especificado
`EXP()`	Retorna o valor exponencial de um número específico
`PI()`	Retorna o valor de PI
`SIN()`	Retorna o seno trigonométrico de um ângulo especificado
`SQRT()`	Retorna a raiz quadrada de um número especificado
`TAN()`	Retorna a tangente trigonométrica de um ângulo especificado

Consulte a documentação do seu SGBD para obter a lista das funções de manipulação numérica que ele suporta.

Resumo

Nesta lição, você aprendeu como usar as funções de manipulação de dados do SQL. Também aprendeu que, embora essas funções sejam extremamente úteis na formatação, manipulação e filtragem de dados, os detalhes da função são muito inconsistentes de uma implementação SQL para a outra.

Desafios

1. Nossa loja agora está online e contas de clientes estão sendo criadas. Todos os usuários precisam de um login, e o login padrão será uma combinação de seu nome e cidade. Escreva uma instrução SQL que

retorne a ID do cliente (`cust_id`), o nome do cliente (`cust_name`) e o `user_login`, que está todo em maiúsculas e composto dos dois primeiros caracteres do contato do cliente (`cust_contact`) e dos três primeiros caracteres da cidade do cliente (`cust_city`). Portanto, por exemplo, meu login (Ben Forta morando em Oak Park) seria `BEOAK`. Dica: para este desafio você usará funções, concatenação e alias.

2. Escreva uma instrução SQL para retornar o número do pedido (`order_num`) e a data do pedido (`order_date`) para todos os pedidos feitos em janeiro de 2020, classificados por data do pedido. Você deve ser capaz de descobrir isso com base no que aprendeu até agora, mas sinta-se à vontade para consultar a documentação do SGBD conforme necessário.

LIÇÃO 9

Resumindo dados

Nesta lição, você aprenderá o que são as funções de agregação do SQL e como usá-las para resumir os dados da tabela.

Usando funções de agregação

Muitas vezes é necessário resumir dados sem realmente obter todos eles, e o SQL fornece funções especiais para esse fim. Usando essas funções, as consultas SQL são frequentemente usadas para obter dados para fins de análise e geração de relatórios. Exemplos desse tipo de obtenção de dados são:

- Determinar o número de linhas em uma tabela (ou o número de linhas que atendem a alguma condição ou contêm um valor específico).
- Obter a soma de um conjunto de linhas em uma tabela.
- Encontrar os valores mais altos, mais baixos e médios em uma coluna de tabela (seja para todas as linhas ou para linhas específicas).

Em cada um desses exemplos, você quer um resumo dos dados em uma tabela, não os dados propriamente ditos. Portanto, retornar os dados reais da tabela seria um desperdício de tempo e de recursos de processamento (sem mencionar a largura de banda). Para repetir, tudo o que você realmente quer é a informação do resumo.

Para facilitar esse tipo de obtenção, o SQL possui um conjunto de cinco *funções de agregação*, listadas na Tabela 9.1. Essas funções permitem a você executar todos os tipos de obtenção que acabamos de enumerar. Você ficará aliviado ao saber que, ao contrário das funções de manipulação de dados da última lição, as funções de agregação do SQL são suportadas de forma consistente pelas principais implementações do SQL.

NOVO TERMO: Funções de agregação

Funções que operam em um conjunto de linhas para calcular e retornar um único valor.

Tabela 9.1 – Funções de agregação do SQL

Função	Descrição
`AVG()`	Retorna o valor médio de uma coluna
`COUNT()`	Retorna o número de linhas em uma coluna
`MAX()`	Retorna o valor mais alto de uma coluna
`MIN()`	Retorna o valor mais baixo de uma coluna
`SUM()`	Retorna a soma dos valores de uma coluna

O uso de cada uma dessas funções é explicado nas seções a seguir.

Função AVG()

`AVG()` é usada para retornar o valor médio de uma coluna específica, contando tanto o número de linhas na tabela quanto a soma de seus valores. `AVG()` pode ser usada para retornar o valor médio de todas as colunas ou de colunas ou linhas específicas.

Este primeiro exemplo usa `AVG()` para retornar o preço médio de todos os produtos na tabela `Products`:

Entrada ▼

```
SELECT AVG(prod_price) AS avg_price
FROM Products;
```

Saída ▼

```
avg_price
-------------
6.823333
```

Análise ▼

A instrução `SELECT` anterior retorna um único valor, `avg_price`, que contém o preço médio de todos os produtos na tabela `Products`. `avg_price` é um alias como explicado na *Lição 7 – Criando campos calculados*.

`AVG()` também pode ser usada para determinar o valor médio de colunas ou linhas específicas. O exemplo a seguir retorna o preço médio dos

produtos oferecidos por um fornecedor específico:

Entrada ▼

```
SELECT AVG(prod_price) AS avg_price
FROM Products
WHERE vend_id = 'DLL01';
```

Saída ▼

```
avg_price
-----------
3.8650
```

Análise ▼

Esta instrução SELECT difere da anterior apenas pelo fato de conter uma cláusula WHERE. A cláusula WHERE filtra apenas produtos com um vendor_id de DLL01 e, portanto, o valor retornado em avg_price é a média apenas dos produtos desse fornecedor.

> **CUIDADO: Somente colunas individuais**
>
> AVG() só pode ser usada para determinar a média de uma coluna numérica específica e esse nome de coluna deve ser especificado como o parâmetro da função. Para obter o valor médio de várias colunas, você deve usar várias funções AVG(). A exceção a isso é ao retornar um único valor que é calculado a partir de várias colunas, conforme será explicado posteriormente nesta lição.

> **NOTA: Valores NULL**
>
> As linhas da coluna contendo valores NULL são ignoradas pela função AVG().

Função COUNT()

COUNT() faz exatamente isto – conta. Usando COUNT(), você pode determinar o número de linhas em uma tabela ou o número de linhas que correspondem a um critério específico.

COUNT() pode ser usada de duas maneiras:

- Use COUNT(*) para contar o número de linhas em uma tabela, quer as colunas contenham valores ou valores NULL.
- Use COUNT(column) para contar o número de linhas que possuem valores em uma coluna específica, ignorando os valores NULL.

Este primeiro exemplo retorna o número total de clientes na tabela

Customers:

Entrada ▼

```
SELECT COUNT(*) AS num_cust
FROM Customers;
```

Saída ▼

```
num_cust
--------
5
```

Análise ▼

Neste exemplo, `COUNT(*)` é usado para contar todas as linhas, independentemente dos valores. A contagem é retornada em `num_cust`.

O exemplo a seguir conta apenas os clientes com um endereço de e-mail:

Entrada ▼

```
SELECT COUNT(cust_email) AS num_cust
FROM Customers;
```

Saída ▼

```
num_cust
--------
3
```

Análise ▼

Esta instrução `SELECT` usa `COUNT(cust_email)` para contar apenas as linhas com um valor na coluna `cust_email`. Neste exemplo, `cust_email` é 3 (significando que apenas 3 dos 5 clientes têm endereços de e-mail).

> **NOTA: Valores `NULL`**
>
> As linhas da coluna com valores `NULL` são ignoradas pela função `COUNT()` se um nome de coluna for especificado, mas não se o asterisco (`*`) for usado.

Função MAX()

`MAX()` retorna o valor mais alto em uma coluna especificada. `MAX()` requer que o nome da coluna seja especificado, como visto aqui:

Entrada ▼

```
SELECT MAX(prod_price) AS max_price
```

```
FROM Products;
```

Saída ▼

```
max_price
----------
11.9900
```

Análise ▼

Aqui `MAX()` retorna o preço do item mais caro na tabela `Products`.

> **DICA: Usando `MAX()` com dados não numéricos**
>
> Embora `MAX()` seja normalmente usada para encontrar os valores numéricos ou de data mais altos, muitos (mas não todos) SGBDs permitem que ela seja usada para retornar o valor mais alto em qualquer coluna, incluindo colunas de texto. Quando usada com dados textuais, `MAX()` retorna a linha que seria a última se os dados fossem classificados por essa coluna.

> **NOTA: Valores `NULL`**
>
> As linhas da coluna com valores `NULL` são ignoradas pela função `MAX()`.

Função MIN()

`MIN()` faz exatamente o oposto de `MAX()` – retorna o valor mais baixo em uma coluna especificada. Como a função `MAX()`, `MIN()` requer que o nome da coluna seja especificado, como visto aqui:

Entrada ▼

```
SELECT MIN(prod_price) AS min_price
FROM Products;
```

Saída ▼

```
min_price
----------
3.4900
```

Análise ▼

Aqui `MIN()` retorna o preço do item menos caro na tabela `Products`.

> **DICA: Usando `MIN()` com dados não numéricos**
>
> Embora `MIN()` seja normalmente usada para encontrar os valores numéricos ou de data mais baixos, muitos (mas não todos) SGBDs permitem que ela seja usada para retornar o valor mais baixo em qualquer coluna, incluindo colunas de texto. Quando usada com dados

textuais, MIN() retorna a linha que seria a primeira se os dados fossem classificados por essa coluna.

NOTA: Valores NULL

As linhas da coluna com valores NULL são ignoradas pela função MIN().

Função SUM()

SUM() é usada para retornar a soma (total) dos valores em uma coluna específica.

Aqui está um exemplo para demonstrar isso. A tabela OrderItems contém os itens reais em um pedido e cada item tem uma quantidade associada. O número total de itens pedidos (a soma de todos os valores de quantidade) pode ser obtido da seguinte forma:

Entrada ▼

```
SELECT SUM(quantity) AS items_ordered
FROM OrderItems
WHERE order_num = 20005;
```

Saída ▼

```
items_ordered
----------
200
```

Análise ▼

A função SUM(quantity) retorna a soma de todas as quantidades de itens em um pedido, e a cláusula WHERE garante que apenas os itens certos do pedido sejam incluídos.

SUM() também pode ser usada para totalizar os valores calculados. Neste próximo exemplo, o valor total do pedido é obtido totalizando item_price*quantity para cada item:

Entrada ▼

```
SELECT SUM(item_price*quantity) AS total_price
FROM OrderItems
WHERE order_num = 20005;
```

Saída ▼

```
total_price
```

```
----------
1648.0000
```

Análise ▼

A função `SUM(item_price*quantity)` retorna a soma de todos os preços expandidos em um pedido e a cláusula `WHERE` garante que apenas os itens certos do pedido sejam incluídos.

> **DICA: Executando cálculos em várias colunas**
>
> Todas as funções de agregação podem ser usadas para realizar cálculos em várias colunas usando os operadores matemáticos padrão, conforme mostrado no exemplo.

> **NOTA: Valores `NULL`**
>
> As linhas da coluna com valores `NULL` são ignoradas pela função `SUM()`.

Agregação com valores distintos

As cinco funções de agregação podem ser usadas de duas maneiras:

- Para realizar cálculos em todas as linhas, especifique o argumento `ALL` ou não especifique nenhum argumento (porque `ALL` é o comportamento padrão).
- Para incluir apenas valores exclusivos, especifique o argumento `DISTINCT`.

> **DICA: `ALL` é o argumento padrão**
>
> O argumento `ALL` não precisa ser especificado porque ele é o comportamento padrão. Se `DISTINCT` não for especificado, `ALL` será assumido.

O exemplo a seguir usa a função `AVG()` para retornar o preço médio dos produtos oferecidos por um fornecedor específico. É a mesma instrução `SELECT` usada anteriormente, mas aqui o argumento `DISTINCT` é usado para que a média leve em consideração apenas preços exclusivos:

Entrada ▼

```
SELECT AVG(DISTINCT prod_price) AS avg_price
FROM Products
WHERE vend_id = 'DLL01';
```

Saída ▼

```
avg_price
```

```
----------
4.2400
```

Análise ▼

Como você pode ver, neste exemplo `avg_price` é mais alto quando `DISTINCT` é usado porque há vários itens com o mesmo preço mais baixo. Excluí-los aumenta o preço médio.

> **CUIDADO: Não usar `DISTINCT` com `COUNT(*)`**
>
> `DISTINCT` só pode ser usado com `COUNT()` se um nome de coluna for especificado. `DISTINCT` não pode ser usado com `COUNT(*)`. Da mesma forma, `DISTINCT` deve ser usado com um nome de coluna, e não com um cálculo ou expressão.

> **DICA: Usando `DISTINCT` com `MIN()` e `MAX()`**
>
> Embora `DISTINCT` possa tecnicamente ser usado com `MIN()` e `MAX()`, não há realmente nenhum sentido em fazer isso. Os valores mínimo e máximo em uma coluna serão os mesmos, independentemente de serem incluídos ou não apenas valores distintos.

> **NOTA: Argumentos de agregação adicionais**
>
> Além dos argumentos `DISTINCT` e `ALL` mostrados aqui, alguns SGBDs oferecem suporte a argumentos adicionais, como `TOP` e `TOP PERCENT` que permitem realizar cálculos em subconjuntos de resultados de consulta. Consulte a documentação de seu SGBD para determinar exatamente quais argumentos estão disponíveis.

Combinando funções de agregação

Todos os exemplos de funções de agregação usados até agora envolveram uma única função. Mas, na verdade, as instruções `SELECT` podem conter tantas funções de agregação quantas forem necessárias. Veja este exemplo:

Entrada ▼

```
SELECT COUNT(*) AS num_items,
       MIN(prod_price) AS price_min,
       MAX(prod_price) AS price_max,
       AVG(prod_price) AS price_avg
FROM Products;
```

Saída ▼

```
num_items    price_min         price_max          price_avg
----------   ---------------   ---------------    ---------
9            3.4900            11.9900            6.823333
```

Análise ▼

Aqui, uma única instrução SELECT executa quatro cálculos de agregação em uma etapa e retorna quatro valores (o número de itens na tabela `Products` e os preços mais altos, mais baixos e médios de produtos).

> **CUIDADO: Nomeando aliases**
>
> Ao especificar nomes de alias para conter os resultados de uma função de agregação, tente não usar o nome de uma coluna real na tabela. Embora não haja nada realmente ilegal em fazer isso, muitas implementações de SQL não suportam isso e gerarão mensagens de erro obscuras se você o fizer.

Resumo

As funções de agregação são usadas para resumir os dados. O SQL oferece suporte a cinco funções de agregação, que podem ser usadas de várias maneiras para retornar apenas os resultados necessários. Essas funções são projetadas para serem altamente eficientes e em geral retornam resultados muito mais rapidamente do que você mesmo poderia calculá-los em sua própria aplicação cliente.

Desafios

1. Escreva uma instrução SQL para determinar o número total de itens vendidos (usando a coluna `quantity` em `OrderItems`).
2. Modifique a instrução que você acabou de criar para determinar o número total de produtos com o id `BR01` (`prod_id`) vendidos.
3. Escreva uma instrução SQL para determinar o preço (`prod_price`) do item mais caro na tabela `Products` que não custe mais que `10`. Nomeie o campo calculado `max_price`.

LIÇÃO 10

Agrupando dados

Nesta lição, você aprenderá como agrupar dados para que possa resumir subconjuntos de conteúdo de tabela. Isso envolve duas novas cláusulas da instrução `SELECT`*: a cláusula* `GROUP BY` *e a cláusula* `HAVING`.

Compreendendo o agrupamento de dados

Na última lição, você aprendeu que as funções de agregação do SQL podem ser usadas para resumir dados. Essas funções permitem contar linhas, calcular somas e médias, e obter valores altos e baixos sem ter de retornar todos os dados.

Todos os cálculos até agora foram executados em todos os dados em uma tabela ou em dados que correspondiam a uma cláusula `WHERE` específica. Como um lembrete, o exemplo a seguir retorna o número de produtos oferecidos pelo fornecedor `DLL01`:

Entrada ▼

```
SELECT COUNT(*) AS num_prods
FROM Products
WHERE vend_id ='DLL01';
```

Saída ▼

```
num_prods
-----------
4
```

Mas e se você quisesse retornar o número de produtos oferecidos por fornecedor? Ou produtos oferecidos por fornecedores que oferecem um único produto ou apenas aqueles que oferecem mais de 10 produtos?

É aqui que os grupos começam a fazer sentido. O agrupamento permite dividir os dados em conjuntos lógicos para que você realize cálculos de agregação em cada grupo.

Criando grupos

Os grupos são criados usando a cláusula `GROUP BY` em sua instrução `SELECT`. A melhor maneira de entender isso é observar um exemplo:

Entrada ▼

```
SELECT vend_id, COUNT(*) AS num_prods
FROM Products
GROUP BY vend_id;
```

Saída ▼

```
vend_id  num_prods
-------  ---------
BRS01    3
DLL01    4
FNG01    2
```

Análise ▼

A instrução `SELECT` acima especifica duas colunas, `vend_id`, que contém a ID do fornecedor de um produto, e `num_prods`, que é um campo calculado (criado usando a função `COUNT(*)`). A cláusula `GROUP BY` instrui o SGBD a classificar os dados e agrupá-los por `vend_id`. Isso faz com que `num_prods` seja calculado uma vez por `vend_id` em vez de uma vez para a tabela inteira. Como você pode ver na saída, o fornecedor `BRS01` tem `3` produtos listados, o fornecedor `DLL01` tem `4` produtos listados e o fornecedor `FNG01` tem `2` produtos listados.

Como você usou `GROUP BY`, não foi necessário especificar cada grupo a ser avaliado e calculado. Isso foi feito automaticamente. A cláusula `GROUP BY` instrui o SGBD a agrupar os dados e, em seguida, realizar a agregação em cada grupo em vez de em todo o conjunto de resultados.

Antes de usar `GROUP BY`, aqui estão algumas regras importantes sobre seu uso que você precisa saber:

- As cláusulas `GROUP BY` podem conter quantas colunas você quiser. Isso permite aninhar grupos, fornecendo um controle mais granular sobre como os dados são agrupados.
- Se você tiver grupos aninhados em sua cláusula `GROUP BY`, os dados são

resumidos no último grupo especificado. Em outras palavras, todas as colunas especificadas são avaliadas juntas quando o agrupamento é estabelecido (portanto, você não obterá dados para cada nível de coluna individual).

- Cada coluna listada em `GROUP BY` deve ser uma coluna obtida ou uma expressão válida (mas não uma função de agregação). Se uma expressão for usada em `SELECT`, essa mesma expressão deverá ser especificada em `GROUP BY`. Aliases não podem ser usados.
- A maioria das implementações SQL não permite colunas `GROUP BY` com tipos de dados de tamanho variável (como campos de texto ou memorando).
- Além das instruções de cálculo de agregação, todas as colunas em sua instrução `SELECT` devem estar presentes na cláusula `GROUP BY`.
- Se a coluna de agrupamento contiver uma linha com um valor `NULL`, `NULL` será retornado como um grupo. Se houver várias linhas com valores `NULL`, todas elas serão agrupadas.
- A cláusula `GROUP BY` deve vir depois de qualquer cláusula `WHERE` e antes de qualquer cláusula `ORDER BY`.

DICA: A cláusula `ALL`

Algumas implementações SQL (como o Microsoft SQL Server) suportam uma cláusula opcional `ALL` em `GROUP BY`. Essa cláusula pode ser usada para retornar todos os grupos, mesmo aqueles que não possuem linhas correspondentes (nesse caso, a agregação retornaria `NULL`). Consulte a documentação do seu SGBD para ver se ele suporta `ALL`.

CUIDADO: Especificando colunas por posição relativa

Algumas implementações SQL permitem especificar colunas `GROUP BY` pela posição na lista `SELECT`. Por exemplo, `GROUP BY 2,1` pode significar *agrupe pela segunda coluna selecionada e depois pela primeira*. Embora essa sintaxe abreviada seja conveniente, ela não é compatível com todas as implementações SQL. Seu uso também é arriscado, pois é altamente suscetível à introdução de erros ao editar instruções SQL.

Filtrando grupos

Além de poder agrupar dados usando `GROUP BY`, o SQL também permite filtrar quais grupos incluir e quais excluir. Por exemplo, você pode desejar uma lista de todos os clientes que fizeram pelo menos dois pedidos. Para

obter esses dados, filtre com base no grupo completo, não em linhas individuais.

Você já viu a cláusula WHERE em ação (que foi introduzida na *Lição 4 – Filtrando dados*). Mas WHERE não funciona aqui porque WHERE filtra linhas específicas, não grupos. Na verdade, WHERE não tem ideia do que seja um grupo.

Então, o que usar em vez de WHERE? O SQL fornece ainda outra cláusula para esse propósito: a cláusula HAVING. HAVING é muito semelhante a WHERE. Na verdade, todos os tipos de cláusulas WHERE que você aprendeu até agora também podem ser usados com HAVING. A única diferença é que WHERE filtra linhas e HAVING filtra grupos.

DICA: HAVING suporta todos os operadores da cláusula WHERE

Na *Lição 4 – Filtrando dados* e na *Lição 5 – Filtragem avançada de dados*, você aprendeu sobre as condições da cláusula WHERE (incluindo condições curinga e cláusulas com vários operadores). Todas as técnicas e opções que você aprendeu sobre WHERE podem ser aplicadas a HAVING. A sintaxe é idêntica; apenas a palavra-chave é diferente.

Então, como filtrar grupos? Veja o seguinte exemplo:

Entrada ▼

```
SELECT cust_id, COUNT(*) AS orders
FROM Orders
GROUP BY cust_id
HAVING COUNT(*) >= 2;
```

Saída ▼

```
cust_id    orders
---------- -----------
1000000001 2
```

Análise ▼

As primeiras três linhas desta instrução SELECT são semelhantes às instruções vistas anteriormente. A linha final adiciona uma cláusula HAVING que filtra esses grupos com um COUNT(*)>= 2 – dois ou mais pedidos.

Como você pode ver, uma cláusula WHERE não funcionou aqui porque a filtragem é baseada no valor agregado do grupo, não nos valores de linhas específicas.

NOTA: A diferença entre HAVING e WHERE

Esta é outra maneira de ver a questão: WHERE filtra antes que os dados sejam agrupados e HAVING filtra depois que os dados são agrupados. Essa é uma distinção importante; linhas que são eliminadas por uma cláusula WHERE não serão incluídas no grupo. Isso pode alterar os valores calculados, que por sua vez podem afetar quais grupos são filtrados com base no uso desses valores na cláusula HAVING.

Portanto, será necessário usar as cláusulas WHERE e HAVING em uma instrução? Na verdade, sim. Suponha que você queira filtrar ainda mais a instrução para que ela retorne todos os clientes que fizeram dois ou mais pedidos nos últimos 12 meses. Para isso, pode adicionar uma cláusula WHERE que filtra apenas os pedidos feitos nos últimos 12 meses. Em seguida, você adiciona uma cláusula HAVING para filtrar apenas os grupos com duas ou mais linhas.

Para demonstrar isso melhor, veja o exemplo a seguir, que lista todos os fornecedores que têm dois ou mais produtos com preços de 4 ou mais:

Entrada ▼

```
SELECT vend_id, COUNT(*) AS num_prods
FROM Products
WHERE prod_price >= 4
GROUP BY vend_id
HAVING COUNT(*) >= 2;
```

Saída ▼

```
vend_id  num_prods
-------  -----------
BRS01    3
FNG01    2
```

Análise ▼

Esta instrução merece uma explicação. A primeira linha é um SELECT básico usando uma função de agregação – muito parecido com os exemplos até agora. A cláusula WHERE filtra todas as linhas com um prod_price de pelo menos 4. Os dados são agrupados por vend_id e, em seguida, uma cláusula HAVING filtra apenas os grupos com uma contagem de 2 ou mais. Sem a cláusula WHERE, uma linha extra teria sido obtida (fornecedor DLL01 que vende quatro produtos, todos com preços abaixo de 4), conforme visto aqui:

Entrada ▼

```
SELECT vend_id, COUNT(*) AS num_prods
FROM Products
GROUP BY vend_id
HAVING COUNT(*) >= 2;
```

Saída ▼

```
vend_id  num_prods
-------  -----------
BRS01    3
DLL01    4
FNG01    2
```

> **NOTA: Usando HAVING e WHERE**
>
> HAVING é tão semelhante a WHERE que a maioria dos SGBDs as tratam como a mesma coisa se nenhum GROUP BY for especificado. No entanto, você mesmo deve fazer essa distinção. Use HAVING apenas em conjunto com as cláusulas GROUP BY. Use WHERE para filtragem padrão em nível de linha.

Agrupando e classificando

É importante entender que GROUP BY e ORDER BY são muito diferentes, embora frequentemente realizem a mesma coisa. A Tabela 10.1 resume as diferenças entre elas.

Tabela 10.1 – ORDER BY x GROUP BY

ORDER BY	GROUP BY
Classifica a saída gerada.	Agrupa linhas. No entanto, a saída pode não estar na ordem do grupo.
Quaisquer colunas (mesmo colunas não selecionadas) podem ser usadas.	Apenas colunas selecionadas ou colunas de expressões podem ser usadas, e cada expressão de coluna selecionada deve ser usada.
Nunca é necessário.	Obrigatório se estiver usando colunas (ou expressões) com funções de agregação.

A primeira diferença listada na Tabela 10.1 é extremamente importante. Na maioria das vezes, você descobrirá que os dados agrupados usando GROUP BY estarão realmente na ordem do grupo. Mas nem sempre é o caso e isso não é de fato exigido pelas especificações SQL. Além disso, mesmo que o seu SGBD específico, realmente, sempre classifique os dados pela

cláusula GROUP BY especificada, você pode mesmo querer que eles sejam classificados de forma diferente. Só porque você agrupa os dados de uma maneira (para obter valores agregados específicos do grupo) não significa que deseja que a saída seja classificada da mesma maneira. Você deve sempre fornecer também uma cláusula ORDER BY explícita, mesmo que ela seja idêntica à cláusula GROUP BY.

> **DICA: Não se esqueça de ORDER BY**
>
> Como regra, sempre que usar uma cláusula GROUP BY, você também deve especificar uma cláusula ORDER BY. Essa é a única maneira de garantir que os dados sejam classificados corretamente. Nunca confie em GROUP BY para classificar seus dados.

Para demonstrar o uso de GROUP BY e ORDER BY, vamos ver um exemplo. A seguinte instrução SELECT é semelhante àquelas vistas anteriormente. Ela obtém o número do pedido e o número de itens pedidos para todos os pedidos contendo três ou mais itens:

Entrada ▼

```
SELECT order_num, COUNT(*) AS items
FROM OrderItems
GROUP BY order_num
HAVING COUNT(*) >= 3;
```

Saída ▼

```
order_num  items
---------  -----
20006      3
20007      5
20008      5
20009      3
```

Para classificar a saída por número de itens pedidos, basta adicionar uma cláusula ORDER BY, como exibido a seguir:

Entrada ▼

```
SELECT order_num, COUNT(*) AS items
FROM OrderItems
GROUP BY order_num
HAVING COUNT(*) >= 3
ORDER BY items, order_num;
```

Saída ▼

```
order_num  items
---------  -----
20006      3
20009      3
20007      5
20008      5
```

Análise ▼

Neste exemplo, a cláusula `GROUP BY` é usada para agrupar os dados por número de pedido (a coluna `order_num`) para que a função `COUNT(*)` retorne o número de itens em cada pedido. A cláusula `HAVING` filtra os dados para que apenas pedidos com três ou mais itens sejam retornados. Por fim, a saída é classificada usando a cláusula `ORDER BY`.

Ordem das cláusulas da instrução SELECT

Este é provavelmente um bom momento para revisar a ordem em que as cláusulas da instrução `SELECT` devem ser especificadas. A Tabela 10.2 lista todas as cláusulas que aprendemos até agora, na ordem em que devem ser usadas.

Tabela 10.2 – Cláusulas da instrução `SELECT` *e sua sequência*

Cláusula	Descrição	Obrigatória
`SELECT`	Colunas ou expressões a serem retornadas	Sim
`FROM`	Tabela da qual obter os dados	Apenas se estiver selecionando dados de uma tabela
`WHERE`	Filtragem de nível de linha	Não
`GROUP BY`	Especificação do grupo	Apenas se estiver calculando agregados por grupo
`HAVING`	Filtragem de nível de grupo	Não
`ORDER BY`	Ordem de classificação de saída	Não

Resumo

Na *Lição 9 – Resumindo dados*, você aprendeu como usar as funções de

agregação do SQL para realizar cálculos de resumo em seus dados. Nesta lição, aprendeu como usar a cláusula `GROUP BY` para realizar esses cálculos em grupos de dados, retornando resultados para cada grupo. Viu também como usar a cláusula `HAVING` para filtrar grupos específicos. Além disso, também aprendeu a diferença entre `ORDER BY` e `GROUP BY` e entre `WHERE` e `HAVING`.

Desafios

1. A tabela `OrderItems` contém os itens individuais para cada pedido. Escreva uma instrução SQL que retorne o número de linhas (como `order_lines`) para cada número de pedido (`order_num`) e classifique os resultados por `order_lines`.
2. Escreva uma instrução SQL que retorne um campo chamado `cheapest_item`, que contém o item de menor custo para cada fornecedor (usando `prod_price` na tabela `Products`) e classifique os resultados do menor para o maior custo.
3. É importante identificar os melhores clientes, portanto, escreva uma instrução SQL para retornar o número do pedido (`order_num` na tabela `OrderItems`) para todos os pedidos de pelo menos 100 itens.
4. Outra forma de determinar os melhores clientes é em função de quanto eles gastaram. Escreva uma instrução SQL para retornar o número do pedido (`order_num` na tabela `OrderItems`) para todos os pedidos com um preço total de pelo menos 1000. Dica: para este desafio você precisará calcular e somar o total (`item_price` multiplicado por `quantity`). Classifique os resultados por número de pedido.
5. O que há de errado com a seguinte instrução SQL? (Tente descobrir sem executá-la.)

```
SELECT order_num, COUNT(*) AS items
FROM OrderItems
GROUP BY items
HAVING COUNT(*) >= 3
ORDER BY items, order_num;
```

LIÇÃO 11

Trabalhando com subconsultas

Nesta lição, você aprenderá o que são subconsultas e como usá-las.

Compreendendo subconsultas

As instruções SELECT são consultas SQL. Todas as instruções SELECT que vimos até agora são consultas simples – instruções únicas que obtêm dados de tabelas individuais de banco de dados.

> **NOVO TERMO: Consulta**
>
> Qualquer instrução SQL. No entanto, o termo geralmente é usado para se referir às instruções SELECT.

O SQL também permite que você crie subconsultas – consultas que são incorporadas a outras consultas. Por que fazer isso? A melhor maneira de entender esse conceito é observar alguns exemplos:

Filtrando por subconsulta

As tabelas de banco de dados usadas em todas as lições deste livro são tabelas relacionais. (Consulte o *Apêndice A – Exemplos de scripts de tabela*para obter uma descrição de cada uma das tabelas e seus relacionamentos.) Os pedidos são armazenados em duas tabelas. A tabela Orders armazena uma única linha para cada pedido contendo o número do pedido, ID do cliente e data do pedido. Os itens de pedidos individuais são armazenados na tabela OrderItems relacionada. A tabela Orders não armazena informações do cliente. Ela armazena apenas uma ID de cliente. As informações reais do cliente são armazenadas na tabela Customers.

Agora suponha que você queira uma lista de todos os clientes que solicitaram o item RGAN01. O que você teria de fazer para obter essas informações? Estas são as etapas:

1. Obtenha os números de pedido de todos os pedidos contendo o item

RGAN01.

2. Obtenha a ID do cliente de todos os clientes que têm pedidos listados nos números de pedido retornados na etapa anterior.
3. Obtenha as informações do cliente para todas as IDs de cliente retornadas na etapa anterior.

Cada uma dessas etapas pode ser executada como uma consulta separada. Fazendo isso, você usa os resultados retornados por uma instrução SELECT para preencher a cláusula WHERE da próxima instrução SELECT.

Você também pode usar subconsultas para combinar todas as três consultas em uma única instrução.

A primeira instrução SELECT deve ser autoexplicativa por ora. Ela obtém a coluna order_num para todos os itens do pedido com um prod_id igual a RGAN01. A saída lista os dois pedidos que contêm esse item:

Entrada q

```
SELECT order_num
FROM OrderItems
WHERE prod_id = 'RGAN01';
```

Saída q

```
order_num
-----------
20007
20008
```

Agora que sabemos quais pedidos contêm o item desejado, a próxima etapa é obter as IDs de cliente associados a esses números de pedido, 20007 e 20008. Usando a cláusula IN descrita na *Lição 5 – Filtragem avançada de dados*, você pode criar uma instrução SELECT da seguinte maneira:

Entrada q

```
SELECT cust_id
FROM Orders
WHERE order_num IN (20007,20008);
```

Saída q

```
cust_id
----------
```

```
1000000004
1000000005
```

Agora, combine as duas consultas transformando a primeira (aquela que retornou os números dos pedidos) em uma subconsulta. Dê uma olhada na seguinte instrução SELECT:

Entrada q

```
SELECT cust_id
FROM Orders
WHERE order_num IN (SELECT order_num
                    FROM OrderItems
                    WHERE prod_id = 'RGAN01');
```

Saída q

```
cust_id
----------
1000000004
1000000005
```

Análise q

As subconsultas são sempre processadas começando com a instrução SELECT mais interna e prosseguindo de dentro para fora. Quando a instrução SELECT anterior é processada, o SGBD realmente executa duas operações.

Primeiro, ele executa a seguinte subconsulta:

```
SELECT order_num FROM orderitems WHERE prod_id='RGAN01'
```

Essa consulta retorna os dois números de pedido 20007 e 20008. Esses dois valores são então passados para a cláusula WHERE da consulta externa no formato delimitado por vírgulas exigido pelo operador IN. A consulta externa agora se torna

```
SELECT cust_id FROM orders WHERE order_num IN (20007,20008)
```

Como você pode ver, a saída está correta e é exatamente igual à saída retornada pela cláusula WHERE embutida no código anterior.

DICA: Formatando seu SQL

As instruções SELECT contendo subconsultas podem ser difíceis de ler e depurar, especialmente à medida que sua complexidade aumenta. Separar as consultas em várias linhas e recuar as linhas apropriadamente, conforme mostrado aqui, pode simplificar muito trabalhar com subconsultas.

A propósito, é aqui que a codificação por cores também se torna inestimável e os melhores clientes SGBD de fato codificam SQL por cores exatamente por esse motivo.

Agora você tem as IDs de todos os clientes que solicitaram o item RGAN01. A próxima etapa é obter as informações do cliente para cada uma dessas IDs de cliente. Aqui está a instrução SQL para obter as duas colunas:

Entrada q

```
SELECT cust_name, cust_contact
FROM Customers
WHERE cust_id IN (1000000004,1000000005);
```

Em vez de codificar explicitamente essas IDs de cliente, você pode transformar esta cláusula WHERE em outra subconsulta:

Entrada q

```
SELECT cust_name, cust_contact
FROM Customers
WHERE cust_id IN (SELECT cust_id
                  FROM Orders
                  WHERE order_num IN (SELECT order_num
                                      FROM OrderItems
                                      WHERE prod_id = 'RGAN01'));
```

Saída q

```
cust_name                      cust_contact
-----------------------------  --------------------
Fun4All                        Denise L. Stephens
The Toy Store                    Kim Howard
```

Análise q

Para executar a instrução SELECT anterior, na verdade o SGBD teve de executar três instruções SELECT. A subconsulta mais interna retornou uma lista de números de pedido que foram então usados como a cláusula WHERE para a subconsulta acima dela. Essa subconsulta retornou uma lista de IDs de cliente que foram usadas como a cláusula WHERE para a consulta de nível superior. A consulta de nível superior realmente retornou os dados desejados.

Como você pode ver, o uso de subconsultas em uma cláusula WHERE permite que você escreva instruções SQL extremamente poderosas e flexíveis. Não há limite imposto ao número de subconsultas que podem ser

aninhadas, embora na prática você descobrirá que o desempenho vai dizer quando você estiver aninhando muitas subconsultas.

CUIDADO: Apenas coluna única

As instruções de subconsulta SELECT só podem retornar uma única coluna. A tentativa de obter várias colunas retornará um erro.

CUIDADO: Subconsultas e desempenho

O código mostrado aqui funciona e alcança o resultado desejado. No entanto, usar subconsultas nem sempre é a maneira mais eficiente de executar esse tipo de obtenção de dados. Mais sobre isso na *Lição 12 – Juntando tabelas* na qual você retomará esse mesmo exemplo.

Usando subconsultas como campos calculados

Outra maneira de usar subconsultas é criando campos calculados. Suponha que você queira exibir o número total de pedidos feitos por cliente em sua tabela Customers. Os pedidos são armazenados na tabela Orders junto com a ID do cliente apropriado.

Para realizar essa operação, siga estas etapas:

1. Obtenha a lista de clientes da tabela Customers.
2. Para cada cliente obtido, conte o número de pedidos associados na tabela Orders.

Como você aprendeu nas duas lições anteriores, é possível usar SELECT COUNT(*) para contar linhas em uma tabela e, usando uma cláusula WHERE para filtrar uma ID de cliente específico, é possível contar apenas os pedidos desse cliente. Por exemplo, o código a seguir conta o número de pedidos feitos pelo cliente 1000000001:

Entrada q

```
SELECT COUNT(*) AS orders
FROM Orders
WHERE cust_id = 1000000001;
```

Para executar esse cálculo COUNT(*) para cada cliente, use COUNT* como uma subconsulta. Observe o seguinte código:

Entrada q

```
SELECT cust_name,
      cust_state,
      (SELECT COUNT(*)
       FROM Orders
       WHERE Orders.cust_id = Customers.cust_id) AS orders
FROM Customers
ORDER BY cust_name;
```

Saída q

```
cust_name              cust_state    orders
-----------------------   ----------    ------
Fun4All                IN          1
Fun4All                AZ          1
Kids Place             OH          0
The Toy Store          IL          1
Village Toys           MI          2
```

Análise q

Esta instrução SELECT retorna três colunas para cada cliente na tabela Customers: cust_name, cust_state e orders. Orders é um campo calculado definido por uma subconsulta fornecida entre parênteses. Essa subconsulta é executada uma vez para cada cliente obtido. No exemplo anterior, a subconsulta é executada cinco vezes porque cinco clientes foram obtidos.

A cláusula WHERE na subconsulta é um pouco diferente das cláusulas WHERE usadas anteriormente porque usa nomes de coluna totalmente qualificados; em vez de apenas um nome de coluna (cust_id), ela especifica a tabela e o nome da coluna (como Orders.cust_id e Customers.cust_id). A seguinte cláusula WHERE diz ao SQL que compare cust_id na tabela Orders com aquele que está sendo obtido atualmente da tabela Customers:

```
WHERE Orders.cust_id = Customers.cust_id
```

Esta sintaxe – o nome da tabela e o nome da coluna separados por um ponto – deve ser usada sempre que houver ambiguidade possível sobre os nomes das colunas. Neste exemplo, existem duas colunas cust_id, uma em Customers e outra em Orders. Sem qualificar totalmente os nomes das colunas, o SGDB assume que você está comparando o cust_id na tabela Orders com ele mesmo. Porque

```
SELECT COUNT(*) FROM Orders WHERE cust_id = cust_id
```

sempre retornará o número total de pedidos na tabela Orders, os resultados não serão o que você esperava:

Entrada q

```
SELECT cust_name,
      cust_state,
      (SELECT COUNT(*)
       FROM Orders
       WHERE cust_id = cust_id) AS orders
FROM Customers
ORDER BY cust_name;
```

Saída q

```
cust_name                cust_state    orders
-------------------------   ----------    ------
Fun4All                  IN          5
Fun4All                  AZ          5
Kids Place               OH          5
The Toy Store              IL          5
Village Toys             MI          5
```

Embora as subconsultas sejam extremamente úteis na construção desse tipo de instrução SELECT, deve-se tomar cuidado para qualificar corretamente os nomes de coluna ambíguos.

CUIDADO: Nomes de colunas totalmente qualificados

Você acabou de ver um motivo muito importante para usar nomes de coluna totalmente qualificados. Sem a especificidade extra, os resultados errados foram retornados porque o SGBD entendeu mal o que você pretendia. Às vezes, a ambiguidade causada pela presença de nomes de coluna conflitantes fará com que o SGBD gere um erro. Por exemplo, isso pode ocorrer se sua cláusula WHERE ou ORDER BY especificou um nome de coluna que estava presente em várias tabelas. Uma boa regra é que, se você estiver trabalhando com mais de uma tabela em uma instrução SELECT, use nomes de coluna totalmente qualificados para evitar qualquer ambiguidade.

DICA: As subconsultas nem sempre podem ser a melhor opção

Conforme explicado anteriormente nesta lição, embora o exemplo de código mostrado aqui funcione, geralmente não é a maneira mais eficiente de realizar esse tipo de obtenção de dados. Você voltará a este exemplo quando aprender sobre junções (joins) nas próximas duas lições.

Resumo

Nesta lição, você aprendeu o que são subconsultas e como usá-las. Os usos mais comuns para subconsultas estão nos operadores IN da cláusula WHERE e para preencher colunas calculadas. Você viu exemplos de ambos os tipos de operações.

Desafios

1. Usando uma subconsulta, retorne uma lista de clientes que compraram itens com preço de 10 ou mais. Recomenda-se usar a tabela OrderItems para encontrar os números de pedidos correspondentes (order_num) e, em seguida, a tabela Orders para obter a ID do cliente (cust_id) para esses pedidos.
2. Você precisa saber as datas em que o produto BR01 foi encomendado. Escreva uma instrução SQL que usa uma subconsulta para determinar quais pedidos (em OrderItems) continham itens com prod_id igual a BR01 e então retorne a ID do cliente (cust_id) e a data do pedido (order_date) para cada pedido da tabela Orders. Classifique os resultados pela data do pedido.
3. Agora vamos tornar as coisas um pouco mais desafiadoras. Atualize o desafio anterior para retornar o e-mail do cliente (cust_email na tabela Customers) para todos os clientes que compraram itens com um prod_id igual a BR01. Dica: isso envolve a instrução SELECT, a mais interna retornando order_num da tabela OrderItems, e a do meio retornando cust_id da tabela Orders.
4. Precisamos de uma lista de IDs de clientes com o valor total de seus pedidos. Escreva uma instrução SQL para retornar a ID do cliente (cust_id na tabela Orders) e total_ordered usando uma subconsulta para retornar o total de pedidos para cada cliente. Classifique os resultados por valor gasto de maior para o menor. Dica: você usou SUM() para calcular os totais do pedido anteriormente.
5. Mais um. Escreva uma instrução SQL que obtenha todos os nomes de produtos (prod_name) da tabela Products, junto com uma coluna calculada chamada quant_sold contendo o número total vendido deste item (obtido usando uma subconsulta e a função SUM(quantity) na tabela OrderItems).

LIÇÃO 12

Juntando tabelas

Nesta lição, você aprenderá o que são junções (joins), por que elas são usadas e como criar instruções SELECT *usando-as.*

Compreendendo junções (joins)

Um dos recursos mais poderosos do SQL é a capacidade de juntar tabelas dinamicamente durante as consultas de obtenção de dados. Junções são uma das operações mais importantes que você pode executar usando a instrução SELECT do SQL, e um bom entendimento de junções e da sintaxe de junções é uma parte extremamente importante do aprendizado de SQL.

Antes de usar junções com eficácia, você deve compreender as tabelas relacionais e os fundamentos do projeto de banco de dados relacional. O que se segue não é uma cobertura completa do assunto, mas deve ser o suficiente para você começar a entender o tema.

Compreendendo as tabelas relacionais

A melhor maneira de entender as tabelas relacionais é examinar um exemplo do mundo real baseado nos dados que você usou nas lições até agora.

Suponha que você tenha uma tabela de banco de dados contendo uma lista de produtos, com cada produto em sua própria linha. O tipo de informação que você armazenaria com cada produto incluiria uma descrição e o preço, junto com informações do fornecedor sobre a empresa que cria o produto.

Agora, suponha que você tenha vários produtos criados pelo mesmo fornecedor. Onde você armazenaria as informações do fornecedor (coisas como nome do fornecedor, endereço e informações de contato)? Não seria recomendável armazenar esses dados junto com os produtos por vários motivos:

- Como as informações do fornecedor são as mesmas para cada produto que o fornecedor produz, repetir as informações para cada produto é um desperdício de tempo e de espaço de armazenamento.
- Se as informações do fornecedor forem alteradas (por exemplo, se o fornecedor mudar ou as informações de contato mudarem), você precisará atualizar todas as ocorrências das informações do fornecedor.
- Quando os dados são repetidos (ou seja, as informações do fornecedor são usadas com cada produto), há uma grande probabilidade de os dados não serem inseridos de forma idêntica todas as vezes. Dados inconsistentes são extremamente difíceis de usar em relatórios.

O importante é que ter várias ocorrências dos mesmos dados nunca é uma coisa boa, e esse princípio é a base para o projeto do banco de dados relacional. As tabelas relacionais são projetadas para que as informações sejam divididas em várias tabelas, uma para cada tipo de dados. As tabelas estão relacionadas entre si por meio de valores comuns (e, portanto, o termo *relacional* em um projeto relacional).

Em nosso exemplo, você pode criar duas tabelas – uma para informações do fornecedor e outra para informações do produto. A tabela Vendors contém todas as informações do fornecedor, uma linha da tabela por fornecedor, junto com um identificador exclusivo para cada fornecedor. Esse valor, chamado de *chave primária*, pode ser uma ID de fornecedor ou qualquer outro valor único.

A tabela Products armazena apenas informações do produto e nenhuma informação específica do fornecedor além da ID do fornecedor (a chave primária da tabela Vendors). Essa chave relaciona a tabela Vendors à tabela Products, e usar essa ID do fornecedor permite que você use a tabela Vendors para encontrar os detalhes sobre o fornecedor adequado.

O que isso faz por você? Considere o seguinte:

- As informações do fornecedor nunca são repetidas e, portanto, tempo e espaço não são desperdiçados.
- Se as informações do fornecedor forem alteradas, você pode atualizar um único registro, aquele na tabela Vendors. Os dados nas tabelas

relacionadas não mudam.

- Como nenhum dado é repetido, os dados usados são obviamente consistentes, tornando os relatórios e a manipulação de dados muito mais simples.

O resultado final é que os dados relacionais podem ser armazenados com eficiência e ser facilmente manipulados. Por isso, os bancos de dados relacionais *escalam* muito melhor do que os não relacionais.

NOVO TERMO: **Escalar**

Capacidade de lidar com uma carga crescente sem falhar. Diz-se que um banco de dados ou aplicação bem projetados *escala bem*.

Por que usar junções?

Como acabamos de explicar, separar os dados em várias tabelas permite um armazenamento mais eficiente, manipulação mais fácil e maior escalabilidade. Mas esses benefícios têm um preço.

Se os dados forem armazenados em várias tabelas, como você pode obter esses dados com uma única instrução SELECT?

A resposta é usar uma junção. Simplificando, uma junção é um mecanismo usado para associar, ou juntar, tabelas em uma instrução SELECT (e, portanto, o nome *junção*). Usando uma sintaxe especial, você pode juntar várias tabelas para que um único conjunto de saída seja retornado e a junção associe as linhas corretas em cada tabela dinamicamente.

NOTA: **Usando ferramentas interativas do SGBD**

Entenda que uma junção não é uma entidade física; em outras palavras, ela não existe nas tabelas reais do banco de dados. Uma junção é criada pelo SGBD conforme necessário e persiste durante a execução da consulta.

Muitos SGBDs fornecem interfaces gráficas que podem ser usadas para definir relacionamentos de tabelas interativamente. Essas ferramentas podem ser inestimáveis para ajudar a manter a integridade referencial. Ao usar tabelas relacionais, é importante que apenas dados válidos sejam inseridos nas colunas relacionais. Voltando ao exemplo, se uma ID inválida de fornecedor for armazenada na tabela Products, esses produtos ficarão inacessíveis porque não estarão relacionados a nenhum fornecedor. Para evitar que isso ocorra, você pode instruir o banco de dados a permitir apenas valores válidos (os presentes na tabela Vendors) na coluna de ID do fornecedor na tabela Products. Integridade referencial significa que o SGBD impõe

regras de integridade de dados. E essas regras são frequentemente gerenciadas por meio de interfaces fornecidas pelo SGBD.

Criando uma junção

Criar uma junção é simples. Você deve especificar todas as tabelas a serem incluídas e como elas se relacionam entre si. Veja o seguinte exemplo:

Entrada q

```
SELECT vend_name, prod_name, prod_price
FROM Vendors, Products
WHERE Vendors.vend_id = Products.vend_id;
```

Saída q

```
vend_name            prod_name            prod_price
-------------------  -------------------  ----------
Doll House Inc.       Fish bean bag toy     3.4900
Doll House Inc.       Bird bean bag toy     3.4900
Doll House Inc.       Rabbit bean bag toy   3.4900
Bears R Us           8 inch teddy bear     5.9900
Bears R Us           12 inch teddy bear    8.9900
Bears R Us           18 inch teddy bear    11.9900
Doll House Inc.       Raggedy Ann          4.9900
Fun and Games         King doll            9.4900
Fun and Games         Queen doll           9.4900
```

Análise q

Vamos dar uma olhada no código anterior. A instrução SELECT começa da mesma maneira que todas as instruções que vimos até agora, especificando as colunas a serem obtidas. A grande diferença aqui é que duas das colunas especificadas (prod_name e prod_price) estão em uma tabela, enquanto a outra (vend_name) está em outra tabela.

Agora observe a cláusula FROM. Ao contrário de todas as instruções SELECT anteriores, esta tem duas tabelas listadas na cláusula FROM, Vendors e Products. Estes são os nomes das duas tabelas que estão sendo associadas nessa instrução SELECT. As tabelas são associadas corretamente com uma cláusula WHERE que instrui o SGBD a encontrar a correspondência entre vend_id na tabela Vendors e vend_id na tabela Products.

Você notará que as colunas são especificadas como Vendors.vend_id e

Products.vend_id. Esse nome de coluna totalmente qualificado é necessário aqui porque, se você apenas especificasse vend_id, o SGBD não poderia dizer a qual coluna vend_id você está se referindo. (Existem duas delas, uma em cada tabela.) Como você pode ver na saída anterior, uma única instrução SELECT retorna dados de duas tabelas diferentes.

> **CUIDADO: Nomes de colunas totalmente qualificados**
>
> Conforme observado na lição anterior, você deve usar o nome de coluna totalmente qualificado (tabela e coluna separadas por um ponto) sempre que houver uma possível ambiguidade sobre a qual coluna você está se referindo. A maioria dos SGBDs retornará uma mensagem de erro se você se referir a um nome de coluna ambíguo sem qualificá-lo totalmente com um nome de tabela.

A importância da cláusula WHERE

Pode parecer estranho usar uma cláusula WHERE para definir o relacionamento de junção, mas, na verdade, há um bom motivo para isso. Lembre-se, quando as tabelas são associadas em uma instrução SELECT, esse relacionamento é construído dinamicamente. Não há nada nas definições da tabela do banco de dados que possa instruir o SGBD sobre como juntar as tabelas. Você tem de fazer isso sozinho. Quando junta duas tabelas, o que você realmente faz é emparelhar cada linha da primeira tabela com cada linha da segunda tabela. A cláusula WHERE atua como um filtro para incluir apenas as linhas que correspondem à condição de filtro especificada – a condição de junção, nesse caso. Sem a cláusula WHERE, todas as linhas da primeira tabela serão emparelhadas com todas as linhas da segunda tabela, independentemente de estarem logicamente juntas ou não.

> **NOVO TERMO: Produto cartesiano**
>
> Os resultados retornados por um relacionamento de tabela sem uma condição de junção. O número de linhas obtidas será o número de linhas da primeira tabela multiplicado pelo número de linhas da segunda tabela.

Para entender isso, observe a seguinte instrução SELECT e sua saída:

Entrada q

```
SELECT vend_name, prod_name, prod_price
FROM Vendors, Products;
```

Saída q

```
vend_name       prod_name              prod_price
```

Bears R Us	8 inch teddy bear	5.99
Bears R Us	12 inch teddy bear	8.99
Bears R Us	18 inch teddy bear	11.99
Bears R Us	Fish bean bag toy	3.49
Bears R Us	Bird bean bag toy	3.49
Bears R Us	Rabbit bean bag toy	3.49
Bears R Us	Raggedy Ann	4.99
Bears R Us	King doll	9.49
Bears R Us	Queen doll	9.49
Bear Emporium	8 inch teddy bear	5.99
Bear Emporium	12 inch teddy bear	8.99
Bear Emporium	18 inch teddy bear	11.99
Bear Emporium	Fish bean bag toy	3.49
Bear Emporium	Bird bean bag toy	3.49
Bear Emporium	Rabbit bean bag toy	3.49
Bear Emporium	Raggedy Ann	4.99
Bear Emporium	King doll	9.49
Bear Emporium	Queen doll	9.49
Doll House Inc.	8 inch teddy bear	5.99
Doll House Inc.	12 inch teddy bear	8.99
Doll House Inc.	18 inch teddy bear	11.99
Doll House Inc.	Fish bean bag toy	3.49
Doll House Inc.	Bird bean bag toy	3.49
Doll House Inc.	Rabbit bean bag toy	3.49
Doll House Inc.	Raggedy Ann	4.99
Doll House Inc.	King doll	9.49
Doll House Inc.	Queen doll	9.49
Furball Inc.	8 inch teddy bear	5.99
Furball Inc.	12 inch teddy bear	8.99
Furball Inc.	18 inch teddy bear	11.99
Furball Inc.	Fish bean bag toy	3.49
Furball Inc.	Bird bean bag toy	3.49
Furball Inc.	Rabbit bean bag toy	3.49
Furball Inc.	Raggedy Ann	4.99
Furball Inc.	King doll	9.49
Furball Inc.	Queen doll	9.49
Fun and Games	8 inch teddy bear	5.99
Fun and Games	12 inch teddy bear	8.99
Fun and Games	18 inch teddy bear	11.99
Fun and Games	Fish bean bag toy	3.49
Fun and Games	Bird bean bag toy	3.49
Fun and Games	Rabbit bean bag toy	3.49

```
Fun and Games      Raggedy Ann          4.99
Fun and Games      King doll            9.49
Fun and Games      Queen doll           9.49
Jouets et ours     8 inch teddy bear    5.99
Jouets et ours     12 inch teddy bear   8.99
Jouets et ours     18 inch teddy bear   11.99
Jouets et ours     Fish bean bag toy    3.49
Jouets et ours     Bird bean bag toy    3.49
Jouets et ours     Rabbit bean bag toy  3.49
Jouets et ours     Raggedy Ann          4.99
Jouets et ours     King doll            9.49
Jouets et ours     Queen doll           9.49
```

Análise q

Como você pode ver na saída anterior, o produto cartesiano raramente é o que você deseja. Os dados retornados aqui são a correspondência de cada produto com cada fornecedor, incluindo produtos com o fornecedor incorreto (e até mesmo fornecedores sem nenhum produto).

CUIDADO: Não se esqueça da cláusula WHERE

Certifique-se de que todas as suas junções tenham cláusulas WHERE; caso contrário, o SGBD retornará muito mais dados do que você deseja. Da mesma forma, certifique-se de que suas cláusulas WHERE estejam corretas. Uma condição de filtro incorreta fará com que o SGBD retorne dados incorretos.

DICA: Junções cruzadas

Às vezes, você ouvirá o tipo de junção que retorna um produto cartesiano ser chamado de *junção cruzada*.

Junções internas

A junção que você tem usado até agora é chamada de *equijoin* – uma junção baseada no teste de igualdade entre duas tabelas. Esse tipo de junção também é chamado de *junção interna*. Na verdade, você pode usar uma sintaxe ligeiramente diferente para essas junções, especificando o tipo de junção explicitamente. A seguinte instrução SELECT retorna exatamente os mesmos dados de um exemplo anterior:

Entrada q

```
SELECT vend_name, prod_name, prod_price
FROM Vendors
INNER JOIN Products ON Vendors.vend_id = Products.vend_id;
```

Análise q

O SELECT na instrução é o mesmo que a instrução SELECT anterior, mas a cláusula FROM é diferente. Aqui, a relação entre as duas tabelas faz parte da cláusula FROM especificada como INNER JOIN. Nessa sintaxe, a condição de junção é especificada usando a cláusula especial ON em vez de uma cláusula WHERE. A condição real passada para a cláusula ON é a mesma que seria passada para a cláusula WHERE.

Consulte a documentação de seu SGBD para ver qual sintaxe é preferida.

> **NOTA: A sintaxe "certa"**
>
> De acordo com a especificação SQL padrão ANSI, o uso da sintaxe INNER JOIN é preferível à sintaxe de equijoins simples usada anteriormente. Na verdade, os puristas do SQL tendem a olhar para a sintaxe simples com desdém. Dito isso, os SGBDs realmente suportam os formatos mais simples e padrão, portanto, minha recomendação é que você dedique um tempo para entender os dois formatos, mas use aquele com o qual se sentir mais confortável.

Juntando várias tabelas

O SQL não impõe limite ao número de tabelas que podem ser associadas em uma instrução SELECT. As regras básicas para criar a junção permanecem as mesmas. Primeiro, liste todas as tabelas e, em seguida, defina a relação entre cada uma. Aqui está um exemplo:

Entrada q

```
SELECT prod_name, vend_name, prod_price, quantity
FROM OrderItems, Products, Vendors
WHERE Products.vend_id = Vendors.vend_id
 AND OrderItems.prod_id = Products.prod_id
 AND order_num = 20007;
```

Saída q

```
prod_name           vend_name        prod_price  quantity
---------------     -------------    ----------  --------
18 inch teddy bear  Bears R Us       11.9900     50
Fish bean bag toy   Doll House Inc.  3.4900      100
Bird bean bag toy   Doll House Inc.  3.4900      100
Rabbit bean bag toy Doll House Inc.  3.4900      100
Raggedy Ann         Doll House Inc.  4.9900      50
```

Análise q

Esse exemplo exibe os itens do pedido número 20007. Os itens do pedido são armazenados na tabela OrderItems. Cada produto é armazenado por sua ID de produto, que se refere a um produto na tabela Products. Os produtos são vinculados ao fornecedor apropriado na tabela Vendors pela ID do fornecedor, que é armazenada com cada registro de produto. A cláusula FROM aqui lista as três tabelas e a cláusula WHERE define ambas as condições de junção. Uma condição adicional WHERE é então usada para filtrar apenas os itens do pedido 20007.

CUIDADO: Considerações de desempenho

Os SGBDs processam as junções em tempo de execução, relacionando cada tabela conforme especificado. Esse processo pode consumir muitos recursos, portanto, tome cuidado para não juntar tabelas desnecessariamente. Quanto mais tabelas você juntar, mais degradará o desempenho.

CUIDADO: Número máximo de tabelas em uma junção

Embora seja verdade que o próprio SQL não restringe o número máximo de tabelas por junção, muitos SGBDs de fato têm restrições. Consulte a documentação de seu SGBD para determinar quais restrições existem, se houver.

Agora seria um bom momento para retornar ao seguinte exemplo da *Lição 11 – Trabalhando com subconsultas*. Como você deve se lembrar, essa instrução SELECT retorna a lista de clientes que solicitaram o produto RGAN01:

Entrada q

```
SELECT cust_name, cust_contact
FROM Customers
WHERE cust_id IN (SELECT cust_id
                  FROM Orders
                  WHERE order_num IN (SELECT order_num
                                      FROM OrderItems
                                      WHERE prod_id = 'RGAN01'));
```

Conforme mencionado na *Lição 11 – Trabalhando com subconsultas*, as subconsultas nem sempre são a maneira mais eficiente de executar operações SELECT complexas e, conforme prometido, aqui está a mesma consulta usando junções:

Entrada q

```
SELECT cust_name, cust_contact
```

```
FROM Customers, Orders, OrderItems
WHERE Customers.cust_id = Orders.cust_id
 AND OrderItems.order_num = Orders.order_num
 AND prod_id = 'RGAN01';
```

Saída q

```
cust_name                cust_contact
-------------------------  --------------------
Fun4All                  Denise L. Stephens
The Toy Store              Kim Howard
```

Análise q

Conforme explicado na *Lição 11 – Trabalhando com subconsultas*, retornar os dados necessários nesta consulta requer o uso de três tabelas. Mas, em vez de usá-las dentro de subconsultas aninhadas, aqui duas junções são usadas para conectar as tabelas. Existem três condições da cláusula WHERE aqui. As duas primeiras conectam as tabelas na junção e a última filtra os dados do produto RGAN01.

> **DICA: Vale a pena experimentar**
>
> Como você pode ver, em geral há mais de uma maneira de executar qualquer operação SQL. E raramente existe um caminho certo ou errado definitivo. O desempenho pode ser afetado pelo tipo de operação, pelo SGBD que está sendo usado, pela quantidade de dados nas tabelas, pelo fato de índices e chaves estarem presentes ou não e por uma série de outros critérios. Portanto, geralmente vale a pena experimentar diferentes mecanismos de seleção para encontrar aquele que funciona melhor para você.

> **NOTA: Nomes de colunas juntadas**
>
> Em todos os exemplos apresentados aqui, as colunas que estão sendo juntadas têm o mesmo nome (cust_id em Customers e Orders, por exemplo). Ter colunas com nomes idênticos não é um requisito, e você frequentemente encontrará bancos de dados que usam convenções de nomenclatura diferentes. Criei as tabelas dessa maneira para tornar os exemplos mais simples e claros.

Resumo

As junções (joins) são um dos recursos mais importantes e poderosos do SQL e usá-las de maneira eficaz requer um conhecimento básico de projeto de banco de dados relacional. Nesta lição, você aprendeu alguns dos fundamentos de projeto de banco de dados relacional como uma introdução para aprender sobre junções. Você também aprendeu como criar uma

equijoin (também conhecida como junção interna), que é a forma de junção mais comumente usada. Na próxima lição, aprenderá como criar outros tipos de junções.

Desafios

1. Escreva uma instrução SQL para retornar o nome do cliente (cust_name) da tabela Customers e números de pedidos relacionados (order_num) da tabela Orders, classificando o resultado por nome do cliente e, em seguida, pelo número do pedido. Na verdade, tente este desafio duas vezes – uma vez usando a sintaxe simples de equijoin e uma vez usando um INNER JOIN.
2. Vamos tornar o desafio anterior mais útil. Além de retornar o nome do cliente e o número do pedido, adicione uma terceira coluna chamada OrderTotal contendo o preço total de cada pedido. Há duas maneiras de fazer isso: você pode criar a coluna OrderTotal usando uma subconsulta na tabela OrderItems ou pode juntar a tabela OrderItems às tabelas existentes e usar uma função de agregação. Aqui está uma dica: observe onde você precisa usar nomes de coluna totalmente qualificados.
3. Vamos retornar ao Desafio 2 da *Lição 11 – Trabalhando com subconsultas*. Escreva uma instrução SQL que obtenha as datas quando o produto BR01 foi pedido, mas desta vez use uma junção e a sintaxe de equijoin simples. O resultado deve ser idêntico ao da Lição 11.
4. Esse foi divertido; vamos tentar novamente. Recrie o SQL que você escreveu para a*Lição 11 – Trabalhando com subconsultas*, Desafio 3, mas desta vez usando a sintaxe INNER JOIN padrão ANSI. O código que você escreveu lá usou duas subconsultas aninhadas. Para recriá-lo, você precisará de duas instruções INNER JOIN, cada uma formatada como o exemplo INNER JOIN mostrado anteriormente nesta lição. E não se esqueça da cláusula WHERE para filtrar por prod_id.
5. Mais uma vez, e para tornar as coisas mais divertidas, combinaremos junções, funções de agregação e agrupamentos. Está pronto? Na *Lição 10 – Agrupando dados*, lancei a você um desafio para encontrar todos os números de pedido com um valor de 1000 ou mais. Aqueles resultados

são úteis, mas o que seria ainda mais útil são os nomes dos clientes que fizeram pedidos de pelo menos esse valor. Portanto, escreva uma instrução SQL que use junções para retornar o nome do cliente (cust_name) da tabela Customers e o preço total de todos os pedidos da tabela OrderItems. Aqui está uma dica: para juntar essas tabelas, você também precisará incluir a tabela Orders (porque Customers não está relacionada diretamente a OrderItems, Customers está relacionada a Orders e Orders está relacionada a OrderItems). Não se esqueça de GROUP BY e HAVING, e classifique os resultados por nome do cliente. Você pode usar equijoin simples ou a sintaxe INNER JOIN padrão ANSI para este desafio. Ou, se estiver se sentindo corajoso, tente escrever dos dois jeitos.

LIÇÃO 13

Criação junções avançadas

Nesta lição, você aprenderá tudo sobre tipos de junção adicionais – o que são e como usá-las. Também aprenderá como usar aliases de tabela e como usar funções de agregação com tabelas juntadas.

Usando aliases de tabela

Antes de examinarmos tipos adicionais de junções (joins), precisamos revisitar os aliases. Na *Lição 7 – Criando campos calculados*, você aprendeu como usar aliases para se referir a colunas obtidas de tabela. A sintaxe para criar um alias para uma coluna (no SQL Server) é esta:

Entrada q

```
SELECT RTRIM(vend_name) + ' (' + RTRIM(vend_country) + ')'
     AS vend_title
FROM Vendors
ORDER BY vend_name;
```

Além de usar aliases para nomes de colunas e campos calculados, o SQL também permite criar aliases para nomes de tabelas. Existem duas razões principais para isso:

- Para encurtar a sintaxe SQL.
- Para permitir vários usos da mesma tabela em uma única instrução SELECT.

Dê uma olhada na instrução SELECT a seguir: Ela é basicamente a mesma instrução de um exemplo usado na lição anterior, mas foi modificada para usar aliases:

Entrada q

```
SELECT cust_name, cust_contact
FROM Customers AS C, Orders AS O, OrderItems AS OI
WHERE C.cust_id = O.cust_id
 AND OI.order_num = O.order_num
```

```
AND prod_id = 'RGAN01';
```

Análise q

Você notará que todas as três tabelas nas cláusulas FROM possuem aliases. Customers AS C estabelece C como um alias para Customers e assim por diante. Essa abordagem permite que você use a abreviatura C em vez do texto completo Customers. Neste exemplo, os aliases de tabela foram usados apenas na cláusula WHERE, mas os aliases não se limitam apenas à cláusula WHERE. Você pode usar aliases na lista SELECT, na cláusula ORDER BY e em qualquer outra parte da instrução.

> **CUIDADO: Não existe AS no Oracle**
>
> O Oracle não suporta a palavra-chave AS ao criar aliases de tabelas. Para usar aliases no Oracle, basta especificar o alias sem AS (portanto, Customers C em vez de Customers AS C).

Também é importante notar que os aliases de tabela são usados apenas durante a execução da consulta. Ao contrário dos aliases de coluna, os aliases de tabela nunca são retornados ao cliente.

Usando diferentes tipos de junção

Até agora, você usou apenas junções simples conhecidas como junções internas ou *equijoins*. Agora você verá três tipos de junção adicionais: a autojunção, a junção natural e a junção externa.

Autojunções

Conforme mencionado anteriormente, um dos principais motivos para usar aliases de tabela é poder se referir à mesma tabela mais de uma vez em uma única instrução SELECT. Um exemplo vai demonstrar isso.

Suponha que você queira enviar uma correspondência a todos os contatos do cliente que trabalham para a mesma empresa para a qual Jim Jones trabalha. Essa consulta exige que você primeiro descubra para qual empresa Jim Jones trabalha e, em seguida, quais clientes trabalham para essa empresa. A seguir veremos uma maneira de abordar esse problema:

Entrada q

```
SELECT cust_id, cust_name, cust_contact
```

```
FROM Customers
WHERE cust_name = (SELECT cust_name
                   FROM Customers
                   WHERE cust_contact = 'Jim Jones');
```

Saída q

```
cust_id      cust_name        cust_contact
--------     --------------   --------------
1000000003   Fun4All          Jim Jones
1000000004   Fun4All          Denise L. Stephens
```

Análise q

Esta primeira solução usa subconsultas. A instrução SELECT interna faz uma obtenção simples para retornar o cust_name da empresa para a qual Jim Jones trabalha. Esse nome é o usado na cláusula WHERE da consulta externa para obter todos os funcionários que trabalham para essa empresa. (Você aprendeu tudo sobre subconsultas na *Lição 11 – Trabalhando com subconsultas*. Consulte essa lição para obter mais informações.)

Agora observe a mesma consulta usando uma junção:

Entrada q

```
SELECT c1.cust_id, c1.cust_name, c1.cust_contact
FROM Customers AS c1, Customers AS c2
WHERE c1.cust_name = c2.cust_name
 AND c2.cust_contact = 'Jim Jones';
```

Saída q

```
cust_id      cust_name      cust_contact
-------      -----------    --------------
1000000003   Fun4All        Jim Jones
1000000004   Fun4All        Denise L. Stephens
```

> **DICA: Não existe AS no Oracle**
>
> Usuários do Oracle, lembrem-se de descartar o AS.

Análise q

As duas tabelas necessárias nesta consulta são na verdade a mesma tabela e, portanto, a tabela Customers aparece na cláusula FROM duas vezes. Embora isso seja perfeitamente legal, qualquer referência à tabela Customers seria ambígua porque o SGBD não sabe a qual tabela Customers você está se

referindo.

Para resolver esse problema, são usados aliases de tabela. A primeira ocorrência de Customers tem um alias C1 e a segunda tem um alias C2. Agora, esses aliases podem ser usados como nomes de tabela. A instrução SELECT, por exemplo, usa o prefixo C1 para indicar explicitamente o nome completo das colunas desejadas. Do contrário, o SGBD retornaria um erro porque há duas de cada coluna denominadas cust_id, cust_name e cust_contact. Ele não consegue saber qual você deseja. (Mesmo que sejam iguais.) A cláusula WHERE primeiro junta as tabelas e, em seguida, filtra os dados por cust_contact na segunda tabela para retornar apenas os dados desejados.

DICA: Autojunções em vez de subconsultas

As autojunções costumam ser usadas para substituir instruções usando subconsultas que obtêm dados da mesma tabela da instrução externa. Embora o resultado final seja o mesmo, muitos SGBDs processam junções muito mais rapidamente do que subconsultas. Normalmente, vale a pena experimentar ambas as abordagens para determinar qual tem melhor desempenho.

Junções naturais

Sempre que as tabelas são juntadas, pelo menos uma coluna aparecerá em mais de uma tabela (as colunas sendo usadas para criar a junção). As junções padrão (as junções internas que você aprendeu na última lição) retornam todos os dados, até mesmo várias ocorrências da mesma coluna. Uma junção natural simplesmente elimina essas várias ocorrências de modo que apenas uma de cada coluna seja retornada.

Como ela faz isso? A resposta é que ela não faz – você é que faz. Uma junção natural é aquela na qual você seleciona apenas colunas que são únicas. Normalmente, isso é feito usando um curinga (SELECT *) para uma tabela e subconjuntos explícitos das colunas para todas as outras tabelas. A seguir temos um exemplo:

Entrada q

```
SELECT C.*, O.order_num, O.order_date,
    OI.prod_id, OI.quantity, OI.item_price
FROM Customers AS C, Orders AS O,
   OrderItems AS OI
WHERE C.cust_id = O.cust_id
```

```
AND OI.order_num = O.order_num
AND prod_id = 'RGAN01';
```

> **DICA: Não existe AS no Oracle**
>
> Usuários do Oracle, lembrem-se de descartar o AS.

Análise q

Neste exemplo, um curinga é usado apenas para a primeira tabela. Todas as outras colunas são listadas explicitamente para que nenhuma coluna duplicada seja obtida.

A verdade é que cada junção interna que você criou até agora é, na verdade, uma junção natural, e provavelmente você nunca precisará de uma junção interna que não seja natural.

Junções externas

A maioria das junções relaciona linhas em uma tabela com linhas em outra tabela. Mas, ocasionalmente, você deseja incluir linhas que não possuem linhas relacionadas. Por exemplo, você pode usar junções para realizar as seguintes tarefas:

- Contar quantos pedidos foram feitos por cliente, incluindo clientes que ainda não fizeram um pedido.
- Listar todos os produtos com quantidades do pedido, incluindo produtos não pedidos por ninguém.
- Calcular tamanhos médios de venda, levando em consideração os clientes que ainda não fizeram um pedido.

Em cada um desses exemplos, a junção inclui linhas de tabela que não possuem linhas associadas na tabela relacionada. Esse tipo de junção é chamado de junção externa.

> **CUIDADO: Diferenças de sintaxe**
>
> É importante observar que a sintaxe usada para criar uma junção externa pode variar ligeiramente entre as diferentes implementações de SQL. As várias formas de sintaxe descritas na seção a seguir cobrem a maioria das implementações, mas consulte a documentação de seu SGBD para verificar sua sintaxe antes de continuar.

A seguinte instrução SELECT é uma junção interna simples. Ela obtém uma lista de todos os clientes e seus pedidos:

Entrada q

```
SELECT Customers.cust_id, Orders.order_num
FROM Customers
 INNER JOIN Orders ON Customers.cust_id = Orders.cust_id;
```

A sintaxe de junção externa é semelhante. Para obter uma lista dos clientes, incluindo aqueles que não fizeram pedidos, você pode fazer o seguinte:

Entrada q

```
SELECT Customers.cust_id, Orders.order_num
FROM Customers
 LEFT OUTER JOIN Orders ON Customers.cust_id = Orders.cust_id;
```

Saída q

```
cust_id       order_num
----------    ---------
1000000001    20005
1000000001    20009
1000000002    NULL
1000000003    20006
1000000004    20007
1000000005    20008
```

Análise q

Como a junção interna vista na última lição, esta instrução SELECT usa as palavras-chave OUTER JOIN para especificar o tipo de junção (em vez de especificá-lo na cláusula WHERE). Mas, ao contrário das junções internas, que relacionam linhas em ambas as tabelas, as junções externas também incluem linhas sem linhas relacionadas. Ao usar a sintaxe OUTER JOIN, você deve usar as palavras-chave RIGHT ou LEFT para especificar a tabela da qual incluir todas as linhas (RIGHT para aquela à direita de OUTER JOIN e LEFT para aquela da esquerda). O exemplo anterior usa LEFT OUTER JOIN para selecionar todas as linhas da tabela à esquerda na cláusula FROM (a tabela Customers). Para selecionar todas as linhas da tabela à direita, use um RIGHT OUTER JOIN como mostrado no próximo exemplo:

Entrada q

```
SELECT Customers.cust_id, Orders.order_num
FROM Customers
 RIGHT OUTER JOIN Orders ON Customers.cust_id = Orders.cust_id;
```

CUIDADO: Junções externos no SQLite

O SQLite suporta LEFT OUTER JOIN, mas não RIGHT OUTER JOIN. Felizmente, se você precisa da funcionalidade RIGHT OUTER JOIN no SQLite, há uma solução muito simples, como será explicado na próxima dica.

DICA: Tipos de junção externa

Lembre-se de que sempre há duas formas básicas de junções externas – a junção externa esquerda e a junção externa direita. A única diferença entre elas é a ordem das tabelas que elas estão relacionando. Em outras palavras, uma junção externa esquerda pode ser transformada em uma junção externa direita simplesmente invertendo a ordem das tabelas na cláusula FROM ou WHERE. Assim, os dois tipos de junção externa podem ser usados indistintamente e a decisão sobre qual deles usar é baseada puramente na conveniência.

Existe uma outra variante da junção externa, que tende a ser raramente usada. A junção externa completa obtém todas as linhas de ambas as tabelas e relaciona aquelas que podem ser relacionadas. Ao contrário de uma junção externa esquerda ou uma junção externa direita, que inclui linhas não relacionadas de uma única tabela, a junção externa completa inclui linhas não relacionadas de ambas as tabelas. A sintaxe para uma junção externa completa é a seguinte:

Entrada q

```
SELECT Customers.cust_id, Orders.order_num
FROM Customers
 FULL OUTER JOIN Orders ON Customers.cust_id = Orders.cust_id;
```

CUIDADO: Suporte a FULL OUTER JOIN

A sintaxe FULL OUTER JOIN não é suportada pelo MariaDB, MySQL ou SQLite.

Usando junções com funções de agregação

Como você aprendeu na *Lição 9 – Resumindo dados*, as funções de agregação são usadas para resumir dados. Embora todos os exemplos de funções de agregação até agora apenas resumiram dados de uma única tabela, essas funções também podem ser usadas com junções.

Para demonstrar isso, vejamos um exemplo. Você deseja obter uma lista de todos os clientes e o número de pedidos que cada um fez. O código a seguir usa a função COUNT() para isso:

Entrada q

```
SELECT Customers.cust_id,
      COUNT(Orders.order_num) AS num_ord
FROM Customers
 INNER JOIN Orders ON Customers.cust_id = Orders.cust_id
GROUP BY Customers.cust_id;
```

Saída q

```
cust_id       num_ord
----------    --------
1000000001    2
1000000003    1
1000000004    1
1000000005    1
```

Análise q

Esta instrução SELECT usa INNER JOIN para relacionar as tabelas Customers e Orders entre si. A cláusula GROUP BY agrupa os dados por cliente e, portanto, a chamada de função COUNT(Orders.order_num) conta o número de pedidos para cada cliente e o retorna como num_ord.

As funções de agregação podem ser usadas com a mesma facilidade com outros tipos de junção. Veja um exemplo a seguir:

Entrada q

```
SELECT Customers.cust_id,
      COUNT(Orders.order_num) AS num_ord
FROM Customers
 LEFT OUTER JOIN Orders ON Customers.cust_id = Orders.cust_id
GROUP BY Customers.cust_id;
```

Saída q

```
cust_id       num_ord
----------    -------
1000000001    2
1000000002    0
1000000003    1
1000000004    1
1000000005    1
```

Análise q

Este exemplo usa uma junção externa esquerda para incluir todos os clientes, mesmo aqueles que não fizeram pedidos. Os resultados mostram

que o cliente 1000000002 com 0 pedidos está incluído desta vez, ao contrário de quando INNER JOIN foi usado.

Usando junções e condições de junção

Antes de encerrar nossa discussão sobre junções, acho que vale a pena resumir alguns pontos-chave sobre junções e seu uso:

- Preste muita atenção ao tipo de junção que está sendo usado. Na maioria das vezes, você desejará uma junção interna, mas também há usos válidos para junções externas.
- Verifique a documentação de seu SGBD para obter a sintaxe de junção exata que ele suporta. (A maioria dos SGBDs usa uma das formas de sintaxe descritas nessas duas lições.)
- Certifique-se de usar a condição de junção correta (independentemente da sintaxe usada), ou você retornará dados incorretos.
- Certifique-se de sempre fornecer uma condição de junção, ou você acabará com o produto cartesiano.
- Você pode incluir várias tabelas em uma junção e até mesmo ter diferentes tipos de junção para cada uma. Embora isso seja legal e frequentemente útil, certifique-se de testar cada junção separadamente antes de testá-las juntas. Isso tornará a solução de problemas muito mais simples.

Resumo

Esta lição foi uma continuação da última lição sobre junções. Começou ensinando como e por que usar aliases e, em seguida, continuou com uma discussão sobre os diferentes tipos de junção e várias formas de sintaxe usadas com cada um. Você também aprendeu como usar funções de agregação com junções e algumas coisas importantes que deve e não deve fazer ao trabalhar com junções.

Desafios

1. Escreva uma instrução SQL usando INNER JOIN para obter o nome do

cliente (cust_name em Customers) e todos os números de pedido (order_num em Orders) para cada um.

2. Modifique a instrução SQL que você acabou de criar para listar todos os clientes, mesmo aqueles sem pedidos.
3. Use um OUTER JOIN para juntar as tabelas Products e OrderItems, retornando uma lista classificada de nomes de produtos (prod_name) e os números de pedido (order_num) associados a cada um.
4. Modifique a instrução SQL criada no desafio anterior para que ela retorne um número total de pedidos para cada item (em oposição aos números dos pedidos).
5. Escreva uma instrução SQL para listar os fornecedores (vend_id em Vendors) e o número de produtos que eles têm disponíveis, incluindo fornecedores sem produtos. Recomenda-se usar OUTER JOIN e a função de agregação COUNT() para contar o número de produtos para cada fornecedor na tabela Products. Preste atenção: a coluna vend_id aparece em várias tabelas, portanto, sempre que você se referir a ela, será necessário qualificá-la totalmente.

LIÇÃO 14

Combinando consultas

Nesta lição, você aprenderá a usar o operador `UNION` *para combinar várias instruções* `SELECT` *em um único conjunto de resultados.*

Compreendendo as consultas combinadas

A maioria das consultas SQL contém uma única instrução `SELECT` que retorna dados de uma ou mais tabelas. O SQL também permite que você execute várias consultas (várias instruções `SELECT`) e retorne os resultados como um único conjunto de resultados de consulta. Essas consultas combinadas em geral são conhecidas como *uniões* ou *consultas compostas*.

Existem basicamente dois cenários em que você usaria consultas combinadas:

- Para retornar dados estruturados de forma semelhante de tabelas diferentes em uma única consulta.
- Para realizar várias consultas em uma única tabela, retornando os dados como uma única consulta.

DICA: Combinando consultas e várias condições `WHERE`

Na maior parte das vezes, combinar duas consultas à mesma tabela tem o mesmo resultado que uma única consulta com várias condições da cláusula `WHERE`. Em outras palavras, qualquer instrução `SELECT` com várias cláusulas `WHERE` também pode ser especificada como uma consulta combinada, como você verá na seção a seguir.

Criando consultas combinadas

As consultas SQL são combinadas usando o operador `UNION`. Usando `UNION` você pode especificar várias instruções `SELECT` e seus resultados podem ser combinados em um único conjunto de resultados.

Usando UNION

Usar `UNION` é bastante simples. Basta especificar cada instrução `SELECT` e

colocar a palavra-chave UNION entre cada uma.

Vejamos um exemplo. Você precisa de um relatório sobre todos os seus clientes em Illinois, Indiana e Michigan. Também deseja incluir todos os locais de Fun4All, independentemente do estado. Claro, você pode criar uma cláusula WHERE para isso, mas desta vez usará uma cláusula UNION.

Conforme explicado, criar a cláusula UNION envolve escrever várias instruções SELECT. Primeiro, observe as instruções individuais:

Entrada ▼

```
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_state IN ('IL','IN','MI');
```

Saída ▼

```
cust_name      cust_contact    cust_email
-----------    -------------   ------------
Village Toys   John Smith      sales@villagetoys.com
Fun4All        Jim Jones       jjones@fun4all.com
The Toy Store  Kim Howard      NULL
```

Entrada ▼

```
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_name = 'Fun4All';
```

Saída ▼

```
cust_name      cust_contact          cust_email
------------   -----------           -------------
Fun4All        Jim Jones             jjones@fun4all.com
Fun4All        Denise L. Stephens    dstephens@fun4all.com
```

Análise ▼

O primeiro SELECT obtém todas as linhas em Illinois, Indiana e Michigan, passando as abreviações desses estados para a cláusula IN. O segundo SELECT usa um teste de igualdade simples para encontrar todos os locais Fun4All. Você notará que uma linha aparece em ambas as saídas, pois atende às duas condições WHERE.

Para combinar essas duas instruções, faça o seguinte:

Entrada ▼

```
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_state IN ('IL','IN','MI')
UNION
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_name = 'Fun4All';
```

Saída ▼

```
cust_name      cust_contact          cust_email
-----------    -----------           ----------------
Fun4All        Denise L. Stephens    dstephens@fun4all.com
Fun4All        Jim Jones             jjones@fun4all.com
Village Toys   John Smith            sales@villagetoys.com
The Toy Store  Kim Howard            NULL
```

Análise ▼

As instruções anteriores são compostas de ambas as instruções `SELECT` anteriores separadas pela palavra-chave `UNION`. `UNION` instrui o SGBD a executar ambas as instruções `SELECT` e combinar a saída em um único conjunto de resultados de consulta.

Como ponto de referência, aqui está a mesma consulta usando várias cláusulas `WHERE` em vez de `UNION`:

Entrada ▼

```
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_state IN ('IL','IN','MI') OR cust_name = 'Fun4All';
```

Em nosso exemplo simples, `UNION` pode realmente ser mais complicado do que usar uma cláusula `WHERE`. Mas com condições de filtragem mais complexas, ou se os dados estiverem sendo obtidos de várias tabelas (e não apenas de uma única tabela), a cláusula `UNION` poderia ter tornado o processo muito mais simples.

DICA: Limites da cláusula `UNION`

Não há limite padrão no SQL para o número de instruções `SELECT` que podem ser combinadas com instruções `UNION`. No entanto, é melhor consultar a documentação do

SGBD para garantir que ele não tenha nenhuma restrição própria quanto ao número máximo de instruções.

CUIDADO: Problemas de desempenho

A maioria dos SGBDs bons usa um otimizador de consulta interno para combinar as instruções `SELECT` antes mesmo de elas serem processadas. Em teoria, isso significa que, de uma perspectiva de desempenho, não deve haver nenhuma diferença real entre o uso de várias condições da cláusula `WHERE` ou `UNION`. Digo em teoria porque, na prática, a maioria dos otimizadores de consulta nem sempre faz um trabalho tão bom quanto deveria. Sua melhor aposta é testar os dois métodos para ver qual funcionará melhor para você.

Regras da cláusula UNION

Como você pode ver, as uniões são muito fáceis de usar. Mas existem algumas regras que definem exatamente quais podem ser combinadas:

- Uma cláusula `UNION` deve ser composta de duas ou mais instruções `SELECT`, cada uma separada pela palavra-chave `UNION` (portanto, se estiver combinando quatro instruções `SELECT`, você usará três palavras-chave `UNION`).
- Cada consulta em um `UNION` deve conter as mesmas colunas, expressões ou funções de agregação (e alguns SGBDs até exigem que as colunas sejam listadas na mesma ordem).
- Os tipos de dados da coluna devem ser compatíveis. Eles não precisam ter o mesmo nome ou exatamente o mesmo tipo, mas devem ser de um tipo que o SGBD possa converter implicitamente (por exemplo, diferentes tipos numéricos ou diferentes tipos de data).

NOTA: Nomes de colunas da cláusula `UNION`

Se as instruções `SELECT` combinadas com `UNION` tiverem nomes de coluna diferentes, qual nome será realmente retornado? Por exemplo, se uma instrução contivesse `SELECT prod_name` e a próxima usasse `SELECT productname`, qual seria o nome da coluna combinada retornada?

A resposta é que o primeiro nome é usado, portanto, em nosso exemplo, a coluna combinada seria chamada de `prod_name`, embora o segundo `SELECT` use um nome diferente. Isso também significa que você pode usar um alias no primeiro nome para definir o nome da coluna retornada conforme necessário.

Esse comportamento tem outro efeito colateral interessante. Como o primeiro conjunto de nomes de coluna é usado, apenas esses nomes podem ser especificados durante a classificação. Novamente, em nosso exemplo, você poderia usar `ORDER BY prod_name` para classificar os resultados combinados, mas `ORDER BY productname` exibiria uma mensagem de erro porque não há coluna `productname` nos resultados combinados.

Além dessas regras básicas e restrições, as uniões podem ser usadas para qualquer tarefa de obtenção de dados.

Incluindo ou eliminando linhas duplicadas

Volte para a seção anterior intitulada "Usando UNION" e observe o exemplo de instruções SELECT usadas. Você notará que, quando executada individualmente, a primeira instrução SELECT retorna três linhas e a segunda instrução SELECT retorna duas linhas. No entanto, quando as duas instruções SELECT são combinadas com UNION, apenas quatro linhas são retornadas, não cinco.

A cláusula UNION remove automaticamente quaisquer linhas duplicadas do conjunto de resultados da consulta (em outras palavras, ela se comporta da mesma forma que várias condições de cláusula WHERE em um único SELECT). Como há um local Fun4All em Indiana, essa linha foi retornada por ambas as instruções SELECT. Quando a cláusula UNION foi usada, a linha duplicada foi eliminada.

Esse é o comportamento padrão de UNION, mas você pode alterá-lo se desejar. Se quiser realmente que todas as ocorrências de todas as correspondências sejam retornadas, pode usar UNION ALL em vez de UNION.

Veja o seguinte exemplo:

Entrada ▼

```
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_state IN ('IL','IN','MI')
UNION ALL
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_name = 'Fun4All';
```

Saída ▼

```
cust_name      cust_contact          cust_email
-----------    -------------         ------------
Village Toys   John Smith            sales@villagetoys.com
Fun4All        Jim Jones             jjones@fun4all.com
The Toy Store  Kim Howard            NULL
Fun4All        Jim Jones             jjones@fun4all.com
```

```
Fun4All        Denise L. Stephens     dstephens@fun4all.com
```

Análise ▼

Quando você usa `UNION ALL`, o SGBD não elimina duplicatas. Portanto, o exemplo anterior retorna cinco linhas, uma delas ocorrendo duas vezes.

> **DICA: `UNION` X `WHERE`**
>
> No início desta lição, eu disse que `UNION` quase sempre realiza a mesma coisa que várias condições `WHERE`. `UNION ALL` é a forma de `UNION` que realiza o que não pode ser feito com as cláusulas `WHERE`. Se você realmente quiser todas as ocorrências de correspondências para cada condição (incluindo duplicatas), deve usar `UNION ALL`, e não `WHERE`.

Classificando resultados de consulta combinados

A saída da instrução `SELECT` é classificada usando a cláusula `ORDER BY`. Ao combinar consultas com uma cláusula `UNION`, você pode usar apenas uma cláusula `ORDER BY` e ela deve ocorrer após a instrução `SELECT` final. Há muito pouco sentido em classificar parte de um conjunto de resultados de uma maneira e parte de outra, portanto, várias cláusulas `ORDER BY` não são permitidas.

O exemplo a seguir classifica os resultados retornados pela cláusula `UNION` usada anteriormente:

Entrada ▼

```
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_state IN ('IL','IN','MI')
UNION
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_name = 'Fun4All'
ORDER BY cust_name, cust_contact;
```

Saída ▼

```
cust_name      cust_contact         cust_email
-----------    ------------         -------------
Fun4All        Denise L. Stephens   dstephens@fun4all.com
Fun4All        Jim Jones            jjones@fun4all.com
The Toy Store  Kim Howard           NULL
Village Toys   John Smith           sales@villagetoys.com
```

Análise ▼

Este UNION usa uma única cláusula ORDER BY após a instrução SELECT final. Mesmo que ORDER BY pareça fazer parte apenas da última instrução SELECT, o SGBD vai de fato usá-lo para classificar todos os resultados retornados por todas as instruções SELECT.

NOTA: Outros tipos de UNION

Alguns SGBDs suportam dois tipos adicionais de UNION. EXCEPT (às vezes chamado de MINUS) pode ser usado para obter apenas as linhas que existem na primeira tabela, mas não na segunda, e INTERSECT pode ser usado para obter apenas as linhas que existem em ambas as tabelas. Na prática, entretanto, esses tipos de UNION raramente são usados porque os mesmos resultados podem ser obtidos usando junções.

DICA: Trabalhando com várias tabelas

Para simplificar, todos os exemplos nesta lição usaram UNION para combinar várias consultas à mesma tabela. Na prática, UNION é realmente útil quando você precisa combinar dados de várias tabelas, mesmo tabelas com nomes de coluna incompatíveis; nesse caso, você pode combinar UNION com aliases para obter um único conjunto de resultados.

Resumo

Nesta lição, você aprendeu como combinar instruções SELECT com o operador UNION. Usando UNION você pode retornar os resultados de várias consultas como uma consulta combinada, incluindo ou excluindo duplicatas. O uso de UNION pode simplificar muito as cláusulas WHERE complexas e a obtenção de dados de várias tabelas.

Desafios

1. Escreva uma instrução SQL que combine duas instruções SELECT que obtém a ID do produto (prod_id) e quantity da tabela OrderItems, uma filtrando por linhas com uma quantidade de exatamente 100 e a outra filtrando por produtos com uma ID que começa com BNBG. Classifique os resultados por ID do produto.
2. Reescreva a instrução SQL que você acabou de criar para usar uma única instrução SELECT.
3. Este é um pouco sem sentido, eu sei, mas reforça uma observação

anterior nesta lição. Escreva uma instrução SQL que retorne e combine o nome do produto (`prod_name`) de `Products` e o nome do cliente (`cust_name`) de `Customers` e classifique o resultado pelo nome do produto.

4. O que há de errado com a seguinte instrução SQL? (Tente descobrir sem executá-la.)

```
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_state  = 'MI'
ORDER BY cust_name;
UNION
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_state = 'IL'ORDER BY cust_name;
```

LIÇÃO 15

Inserindo dados

Nesta lição, você aprenderá como inserir dados em tabelas usando a instrução `INSERT` *do SQL.*

Compreendendo a inserção de dados

`SELECT` é sem dúvida a instrução SQL usada com mais frequência (é por isso que as últimas 14 lições foram dedicadas a ela). Mas existem três outras instruções SQL usadas com frequência que você deve aprender. A primeira é `INSERT`. (Você chegará às outras duas na próxima lição.)

Como o próprio nome sugere, `INSERT` é usado para inserir (adicionar) linhas em uma tabela de banco de dados. Insert pode ser usado de várias maneiras:

- Inserindo uma única linha completa
- Inserindo uma única linha parcial
- Inserindo os resultados de uma consulta

Vamos agora examinar cada uma dessas maneiras.

> **DICA: `INSERT` e segurança do sistema**
>
> O uso da instrução `INSERT` pode exigir privilégios de segurança especiais em SGBDs do tipo cliente/servidor. Antes de tentar usar `INSERT`, certifique-se de ter os privilégios de segurança adequados para isso.

Inserindo linhas completas

A maneira mais simples de inserir dados em uma tabela é usar a sintaxe `INSERT` básica, que requer que você especifique o nome da tabela e os valores a serem inseridos na nova linha. Aqui está um exemplo:

Entrada ▼

```
INSERT INTO Customers
VALUES(1000000006,
```

```
        'Toy Land',
        '123 Any Street',
        'New York',
        'NY',
        '11111',
        'USA',
        NULL,
        NULL);
```

Análise ▼

O exemplo anterior insere um novo cliente na tabela `Customers`. Os dados a serem armazenados em cada coluna da tabela são especificados na cláusula `VALUES` e um valor deve ser fornecido para cada coluna. Se uma coluna não tiver valor (por exemplo, as colunas `cust_contact` e `cust_email`), o valor `NULL` deve ser usado (assumindo que a tabela permite que nenhum valor seja especificado para essa coluna). As colunas devem ser preenchidas na ordem em que aparecem na definição da tabela.

> **DICA: A palavra-chave `INTO`**
>
> Em algumas implementações de SQL, a palavra-chave `INTO` após `INSERT` é opcional. No entanto, é uma boa prática fornecer essa palavra-chave, mesmo que ela não seja necessária. Isso garantirá que seu código SQL seja portável entre SGBDs.

Embora essa sintaxe seja realmente simples, ela não é nada segura e geralmente deve ser evitada a todo custo. A instrução SQL é altamente dependente da ordem em que as colunas são definidas na tabela. Ela também depende da disponibilidade imediata de informações sobre esse pedido. Mesmo se as informações estiverem disponíveis, não há garantia de que as colunas estarão exatamente na mesma ordem na próxima vez em que a tabela for reconstruída. Portanto, escrever instruções SQL que dependem de uma ordem de coluna específica é muito inseguro. Se você fizer isso, algo inevitavelmente dará errado em algum ponto.

A maneira mais segura (e infelizmente mais complicada) de escrever a instrução `INSERT` é a seguinte:

Entrada ▼

```
INSERT INTO Customers(cust_id,
                      cust_name,
                      cust_address,
```

```
                    cust_city,
                    cust_state,
                    cust_zip,
                    cust_country,
                    cust_contact,
                    cust_email)
VALUES(1000000006,
       'Toy Land',
       '123 Any Street',
       'New York',
       'NY',
       '11111',
       'USA',
       NULL,
       NULL);
```

Análise ▼

Este exemplo faz exatamente a mesma coisa que a instrução INSERT anterior, mas desta vez os nomes das colunas são declarados explicitamente entre parênteses após o nome da tabela. Quando a linha for inserida, o SGBD corresponderá cada item na lista de colunas com o valor apropriado na lista VALUES. A primeira entrada em VALUES corresponde ao primeiro nome de coluna especificado. O segundo valor corresponde ao nome da segunda coluna e assim por diante.

Como os nomes das colunas são fornecidos, VALUES deve corresponder aos nomes das colunas especificadas na ordem em que são especificados, e não necessariamente na ordem em que as colunas aparecem na tabela real. A vantagem disso é que, mesmo se o layout da tabela for alterado, a instrução INSERT ainda funcionará corretamente.

> **NOTA: Não é possível INSERT (inserir) o mesmo registro duas vezes**
>
> Se você tentou as duas versões deste exemplo, descobriu que a segunda gerou um erro porque já existia um cliente com uma ID 1000000006. Conforme discutido na *Lição 1 – Entendendo SQL*, os valores da chave primária devem ser únicos e, como cust_id é a chave primária, o SGBD não permitirá que você insira duas linhas com o mesmo valor cust_id. O mesmo vale para o próximo exemplo. Para tentar as outras instruções INSERT, você precisa excluir a primeira linha adicionada (como será mostrado na próxima lição). Ou não, porque a linha foi inserida e você pode continuar as lições sem excluí-la.

A instrução INSERT a seguir preenche todas as colunas de linha (como

antes), mas em uma ordem diferente. Como os nomes das colunas são especificados, a inserção funcionará corretamente:

Entrada ▼

```
INSERT INTO Customers(cust_id,
                      cust_contact,
                      cust_email,
                      cust_name,
                      cust_address,
                      cust_city,
                      cust_state,
                      cust_zip)
VALUES(1000000006,
       NULL,
       NULL,
       'Toy Land',
       '123 Any Street',
       'New York',
       'NY',
       '11111');
```

DICA: Sempre use uma lista de colunas

Como regra, nunca use `INSERT` sem explicitamente especificar a lista de colunas. Isso aumentará muito a probabilidade de que seu SQL continue a funcionar no caso de ocorrerem alterações na tabela.

CUIDADO: Use `VALUES` com cuidado

Independentemente da sintaxe `INSERT` que está sendo usada, o número correto de `VALUES` deve ser especificado. Se nenhum nome de coluna for fornecido, um valor deve estar presente para cada coluna da tabela. Se forem fornecidos nomes de coluna, um valor deve estar presente para cada coluna listada. Se nenhum valor estiver presente, uma mensagem de erro será gerada e a linha não será inserida.

Inserindo linhas parciais

Como acabei de explicar, a maneira recomendada de usar `INSERT` é especificar explicitamente os nomes das colunas da tabela. Usando essa sintaxe, você também pode omitir colunas. Isso significa que você fornece valores apenas para algumas colunas, mas não para outras.

Veja o seguinte exemplo:

Entrada ▼

```
INSERT INTO Customers(cust_id,
                      cust_name,
                      cust_address,
                      cust_city,
                      cust_state,
                      cust_zip,
                      cust_country)
VALUES(1000000006,
       'Toy Land',
       '123 Any Street',
       'New York',
       'NY',
       '11111',
       'USA');
```

Análise ▼

Nos exemplos fornecidos anteriormente nesta lição, não foram fornecidos valores para duas das colunas, `cust_contact` e `cust_email`. Isso significa que não há razão para incluir essas colunas na instrução `INSERT`. Esta instrução `INSERT`, portanto, omite as duas colunas e os dois valores correspondentes.

CUIDADO: Omitindo colunas

Você pode omitir colunas de uma operação `INSERT` se a definição da tabela assim permitir. Uma das seguintes condições deve ser verdadeira:

- A coluna é definida para permitir valores `NULL` (nenhum valor).
- Um valor padrão é especificado na definição da tabela. Isso significa que o valor padrão será usado se nenhum valor for especificado.

CUIDADO: Omitindo valores exigidos

Se você omitir um valor de uma tabela que não permite valores `NULL` e não tem um valor padrão, o SGBD vai gerar uma mensagem de erro e a linha não será inserida.

Inserindo dados obtidos

`INSERT` é geralmente usado para adicionar uma linha a uma tabela usando valores especificados. Há outra forma de `INSERT` que pode ser usada para inserir o resultado de uma instrução `SELECT` em uma tabela. Isso é conhecido como `INSERT SELECT` e, como o nome sugere, é composto de uma instrução `INSERT` e uma instrução `SELECT`.

Suponha que você queira mesclar uma lista de clientes de outra tabela à

sua tabela Customers. Em vez de ler uma linha por vez e inseri-la com INSERT, você pode fazer o seguinte:

Entrada ▼

```
INSERT INTO Customers(cust_id,
                      cust_contact,
                      cust_email,
                      cust_name,
                      cust_address,
                      cust_city,
                      cust_state,
                      cust_zip,
                      cust_country)
SELECT cust_id,
       cust_contact,
       cust_email,
       cust_name,
       cust_address,
       cust_city,
       cust_state,
       cust_zip,
       cust_country
FROM CustNew;
```

> **NOTA: Instruções necessárias para o próximo exemplo**
>
> O exemplo a seguir importa dados de uma tabela chamada CustNew para a tabela Customers. Para testar esse exemplo, crie e preencha a tabela CustNew primeiro. O formato da tabela CustNew deve ser o mesmo da tabela Customers descrita no *Apêndice A – Exemplos de scripts de tabela*. Ao preencher CustNew, certifique-se de não usar valores cust_id que já foram usados em Customers. (A operação INSERT subsequente falhará se os valores da chave primária forem duplicados.)

Análise ▼

Este exemplo usa INSERT SELECT para importar todos os dados de CustNew para Customers. Em vez de listar os VALUES a serem inseridos, a instrução SELECT obtém os valores da tabela CustNew. Cada coluna em SELECT corresponde a uma coluna na lista de colunas especificadas. Quantas linhas esta instrução vai inserir? Isso depende de quantas linhas existem na tabela CustNew. Se a tabela estiver vazia, nenhuma linha será inserida (e nenhum erro será gerado porque a operação ainda é válida). Se a tabela

contiver, de fato, dados, todos esses dados serão inseridos em `Customers`.

> **DICA: Nomes de colunas em `INSERT SELECT`**
>
> Este exemplo usa os mesmos nomes de coluna em ambas as instruções `INSERT` e `SELECT` para simplificar. Mas não há exigência de que os nomes das colunas sejam os mesmos. Na verdade, o SGBD nem mesmo presta atenção aos nomes das colunas retornados pela instrução `SELECT`. Em vez disso, a posição da coluna é usada, portanto, a primeira coluna na instrução `SELECT` (independentemente de seu nome) será usada para preencher a primeira coluna da tabela especificada e assim por diante.

A instrução `SELECT` usada em um `INSERT SELECT` pode incluir uma cláusula `WHERE` para filtrar os dados a serem inseridos.

> **DICA: Inserindo várias linhas**
>
> `INSERT` geralmente insere apenas uma única linha. Para inserir várias linhas, você deve executar várias instruções `INSERT`. A exceção a esta regra é `INSERT SELECT`, que pode ser usada para inserir várias linhas com uma única instrução; tudo o que a instrução `SELECT` retornar será inserido pelo `INSERT`.

Copiando de uma tabela para outra

Há outra forma de inserção de dados que não usa a instrução `INSERT`. Para copiar o conteúdo de uma tabela para uma tabela totalmente nova (uma que é criada dinamicamente), você pode usar a instrução `CREATE SELECT` (ou `SELECT INTO` se estiver usando o SQL Server).

> **NOTA: Não suportado pelo DB2**
>
> O DB2 não suporta o uso de `CREATE SELECT` conforme descrito aqui.

Ao contrário de `INSERT SELECT`, que acrescenta dados a uma tabela existente, `CREATE SELECT` copia os dados para uma nova tabela (e, dependendo do SGBD usado, pode sobrescrever a tabela se ela já existir).

O exemplo a seguir demonstra o uso de `CREATE SELECT`:

Entrada ▼

```
CREATE TABLE CustCopy AS SELECT * FROM Customers;
```

Se estiver usando o SQL Server, use esta sintaxe:

Entrada ▼

```
SELECT * INTO CustCopy FROM Customers;
```

Análise ▼

Esta instrução SELECT cria uma nova tabela chamada CustCopy e copia todo o conteúdo da tabela Customers para ela. Como SELECT * foi usado, todas as colunas na tabela Customers serão criadas (e preenchidas) na tabela CustCopy. Para copiar apenas um subconjunto das colunas disponíveis, você pode especificar nomes de coluna explícitos em vez do caractere curinga *.

Aqui estão alguns pontos a serem considerados ao usar SELECT INTO:

- Qualquer opção e cláusula da instrução SELECT pode ser usada, incluindo WHERE e GROUP BY.
- As junções podem ser usadas para inserir dados de várias tabelas.
- Os dados só podem ser inseridos em uma única tabela, independentemente das tabelas de onde os dados foram obtidos.

DICA: Fazendo cópias de tabelas

A técnica descrita aqui é uma ótima maneira de fazer cópias de tabelas antes de experimentar novas instruções SQL. Ao fazer uma cópia primeiro, você poderá testar seu SQL nessa cópia em vez de nos dados ativos.

NOTA: Mais exemplos

Está procurando mais exemplos de uso de INSERT? Veja os exemplos de scripts de preenchimento de tabela descritos no *Apêndice A – Exemplos de scripts de tabela*.

Resumo

Nesta lição, você aprendeu como inserir linhas em uma tabela de banco de dados usando INSERT. Aprendeu várias maneiras de usar INSERT e por que a especificação de coluna explícita é preferida. Também aprendeu como usar INSERT SELECT para importar linhas de outra tabela e como usar SELECT INTO para exportar linhas para uma nova tabela. Na próxima lição, você aprenderá como usar UPDATE e DELETE para manipular ainda mais os dados da tabela.

Desafios

1. Usando INSERT e as colunas especificadas, adicione-se à tabela Customers. Liste explicitamente as colunas que você está adicionando e apenas as que você precisa.
2. Faça cópias de backup de suas tabelas Orders e OrderItems.

LIÇÃO 16

Atualizando e excluindo dados

Nesta lição, você aprenderá como usar as instruções `UPDATE` *e* `DELETE` *para permitir que você manipule ainda mais os dados da tabela.*

Atualizando dados

Para atualizar (modificar) dados em uma tabela, você usa a instrução `UPDATE`. `UPDATE` pode ser usada de duas maneiras:

- Para atualizar linhas específicas em uma tabela.
- Para atualizar todas as linhas em uma tabela.

Agora, observaremos cada um desses usos.

> **CUIDADO: Não omita a cláusula `WHERE`**
>
> Cuidado especial deve ser tomado ao usar `UPDATE` porque é muito fácil atualizar erroneamente todas as linhas em sua tabela. Leia toda esta seção sobre `UPDATE` antes de usar essa instrução.

> **DICA: `UPDATE` e segurança**
>
> O uso da instrução `UPDATE` pode exigir privilégios de segurança especiais em SGBDs do tipo cliente/servidor. Antes de tentar usar `UPDATE`, certifique-se de ter os privilégios de segurança adequados para isso.

A instrução `UPDATE` é muito fácil de usar – alguns diriam que é fácil demais. O formato básico de uma instrução `UPDATE` é composto de três partes:

- A tabela a ser atualizada.
- Os nomes das colunas e seus novos valores.
- A condição do filtro que determina quais linhas devem ser atualizadas.

Vamos observar um exemplo simples. O cliente `1000000005` não tem endereço de e-mail registrado e agora tem um endereço, portanto, esse registro precisa ser atualizado. A seguinte instrução realiza essa atualização:

Entrada ▼

```
UPDATE Customers
SET cust_email = 'kim@thetoystore.com'
WHERE cust_id = 1000000005;
```

A instrução `UPDATE` sempre começa com o nome da tabela que está sendo atualizada. Neste exemplo, é a tabela `Customers`. O comando `SET` é então usado para atribuir o novo valor a uma coluna. Conforme usada aqui, a cláusula `SET` configura a coluna `cust_email` com o valor especificado:

```
  SET cust_email = 'kim@thetoystore.com'
```

A instrução `UPDATE` termina com uma cláusula `WHERE` que informa ao SGBD qual linha atualizar. Sem uma cláusula `WHERE`, o SGBD atualizaria todas as linhas na tabela `Customers` com esse novo endereço de e-mail – definitivamente esse não é o resultado desejado.

Atualizar várias colunas requer uma sintaxe ligeiramente diferente:

Entrada ▼

```
UPDATE Customers
SET cust_contact = 'Sam Roberts',
    cust_email = 'sam@toyland.com'
WHERE cust_id = 1000000006;
```

Ao atualizar várias colunas, você usa apenas um único comando `SET` e cada par `coluna = valor` é separado por uma vírgula. (Nenhuma vírgula é especificada após a última coluna.) Neste exemplo, as colunas `cust_contact` e `cust_email` serão atualizadas para o cliente `1000000006`.

DICA: Usando subconsultas em uma instrução `UPDATE`

Subconsultas podem ser usadas em instruções `UPDATE`, permitindo que você atualize colunas com dados obtidos com uma instrução `SELECT`. Consulte a *Lição 11 – Trabalhando com subconsultas* para obter mais informações sobre subconsultas e seus usos.

DICA: A palavra-chave `FROM`

Algumas implementações SQL suportam uma cláusula `FROM` na instrução `UPDATE` que pode ser usada para atualizar as linhas em uma tabela com dados de outra tabela. Consulte a documentação do seu SGBD para ver se ele suporta esse recurso.

Para excluir o valor de uma coluna, você pode defini-lo como `NULL` (assumindo que a tabela está definida para permitir valores `NULL`). Você pode fazer isso da seguinte maneira:

Entrada ▼

```
UPDATE Customers
SET cust_email = NULL
WHERE cust_id = 1000000005;
```

Aqui, a palavra-chave `NULL` é usada para não salvar valor algum na coluna `cust_email`. Isso é muito diferente de salvar uma string vazia. Uma string vazia (especificada como `''`) é um valor, enquanto `NULL` significa que não há nenhum valor.

Excluindo dados

Para excluir (remover) dados de uma tabela, use a instrução `DELETE`. `DELETE` pode ser usada de duas maneiras:

- Para excluir linhas específicas de uma tabela.
- Para excluir todas as linhas de uma tabela.

Vamos examinar agora cada uma dessas maneiras.

> **CUIDADO: Não omita a cláusula `WHERE`**
>
> Cuidado especial deve ser tomado ao usar `DELETE` porque é muito fácil excluir erroneamente todas as linhas de sua tabela. Leia toda esta seção sobre `DELETE` antes de usar essa instrução.

> **DICA: `DELETE` e segurança**
>
> O uso da instrução `DELETE` pode exigir privilégios de segurança especiais em SGBDs do tipo cliente/servidor. Antes de tentar usar `DELETE`, certifique-se de ter os privilégios de segurança adequados para isso.

Já afirmei que `UPDATE` é muito fácil de usar. A boa (e má) notícia é que `DELETE` é ainda mais fácil de usar.

A seguinte instrução exclui uma única linha da tabela `Customers` (a linha que você adicionou na última lição):

Entrada ▼

```
DELETE FROM Customers
WHERE cust_id = 1000000006;
```

Esta instrução deve ser autoexplicativa. `DELETE FROM` requer que você especifique o nome da tabela da qual os dados serão excluídos. A cláusula `WHERE` filtra quais linhas devem ser excluídas. Neste exemplo, apenas o

cliente `1000000006` será excluído. Se a cláusula `WHERE` fosse omitida, essa instrução teria excluído todos os clientes da tabela!

> **DICA: Chaves estrangeiras são suas amigas**
>
> As junções foram introduzidas na *Lição 12 – Juntando tabelas* e, como você aprendeu então, para juntar duas tabelas, você simplesmente precisa de campos comuns em ambas as tabelas. Mas também pode fazer com que o SGBD imponha o relacionamento usando chaves estrangeiras. (Elas são definidas no *Apêndice A – Exemplos de scripts de tabela*) Quando as chaves estrangeiras estão presentes, o SGBD as usa para impor a integridade referencial. Por exemplo, se você tentar inserir um novo produto na tabela `Products`, o SGBD não permitirá que você insira esse produto com uma ID de fornecedor desconhecido porque a coluna `vend_id` está conectada à tabela `Vendors` como uma chave estrangeira. Então, o que isso tem a ver com `DELETE`? Bem, um bom efeito colateral do uso de chaves estrangeiras para garantir a integridade referencial é que o SGBD geralmente evita a exclusão de linhas que são necessárias para um relacionamento. Por exemplo, se você tentou excluir um produto de `Products` que foi usado em pedidos existentes em `OrderItems`, essa instrução `DELETE` geraria um erro e seria cancelada. Essa é outra razão para sempre definir suas chaves estrangeiras.

> **DICA: A palavra-chave `FROM`**
>
> Em algumas implementações SQL, a palavra-chave `FROM` após `DELETE` é opcional. No entanto, é uma boa prática sempre fornecer essa palavra-chave, mesmo que ela não seja necessária. Isso garantirá que seu código SQL seja portável entre SGBDs.

`DELETE` não aceita nomes de coluna ou caracteres curinga. `DELETE` exclui linhas inteiras, não colunas. Para excluir colunas específicas, você usa uma instrução `UPDATE`.

> **NOTA: Conteúdo da tabela, não tabelas**
>
> A instrução `DELETE` exclui linhas de tabelas, até mesmo todas as linhas de tabelas. Mas `DELETE` nunca exclui a própria tabela.

> **DICA: Exclusões mais rápidas**
>
> Se você realmente deseja excluir todas as linhas de uma tabela, não use `DELETE`. Em vez disso, use a instrução `TRUNCATE TABLE`, que realiza a mesma coisa, mas faz isso muito mais rápido (porque as alterações de dados não são registradas).

Diretrizes para atualizar e excluir dados

As instruções `UPDATE` e `DELETE` usadas na seção anterior têm cláusulas `WHERE` e há um bom motivo para isso. Se você omitir a cláusula `WHERE`, `UPDATE` ou `DELETE` será aplicado a todas as linhas da tabela. Em outras palavras, se você executar `UPDATE` sem uma cláusula `WHERE`, todas as linhas da tabela

serão atualizadas com os novos valores. Da mesma forma, se você executar DELETE sem uma cláusula WHERE, todo o conteúdo da tabela será excluído.

Aqui estão algumas diretrizes importantes que muitos programadores de SQL seguem:

- Nunca execute UPDATE ou DELETE sem uma cláusula WHERE, a menos que você realmente pretenda atualizar e excluir todas as linhas.
- Certifique-se de que cada tabela tenha uma chave primária (consulte a *Lição 12 – Juntando tabelas* se você se esqueceu do que se trata) e use-a como a cláusula WHERE sempre que possível. (Você pode especificar chaves primárias individuais, vários valores ou intervalos de valores.)
- Antes de usar uma cláusula WHERE com um UPDATE ou DELETE, teste-a com a instrução SELECT para ter certeza de que ela está filtrando os registros corretos; é muito fácil escrever cláusulas WHERE incorretas.
- Use a integridade referencial imposta pelo banco de dados (consulte a *Lição 12 – Juntando tabelas* para informações sobre esse tópico) de modo que o SGBD não permita a exclusão de linhas que contenham dados em outras tabelas relacionadas a elas.
- Alguns SGBDs permitem que os administradores de banco de dados imponham restrições para evitar a execução de UPDATE ou DELETE sem uma cláusula WHERE. Se o seu SGBD oferecer suporte a esse recurso, considere usá-lo.

Resumindo, o SQL não possui um botão Desfazer. Tenha muito cuidado ao usar UPDATE e DELETE, ou você se verá atualizando e excluindo os dados errados.

Resumo

Nesta lição, você aprendeu como usar as instruções UPDATE e DELETE para manipular os dados em suas tabelas. Aprendeu a sintaxe de cada uma dessas instruções, bem como os perigos inerentes que elas representam. Também aprendeu por que as cláusulas WHERE são tão importantes nas instruções UPDATE e DELETE e recebeu orientações que devem ser seguidas

para ajudar a garantir que os dados não sejam danificados inadvertidamente.

Desafios

1. As abreviações dos estados dos EUA devem sempre estar em maiúsculas. Escreva uma instrução SQL para atualizar todos os endereços dos EUA, tanto os estados do fornecedor (`vend_state` em `Vendors`) como estados do cliente (`cust_state` em `Customers`), de modo que as abreviações fiquem em maiúsculas.
2. O Desafio 1 da *Lição 15 – Inserindo dados* pediu que você se adicionasse à tabela `Customers`. Agora exclua a si mesmo. Certifique-se de usar uma cláusula `WHERE` (e teste-a com um `SELECT` antes de usá-la em `DELETE`) ou você excluirá todos os clientes!

LIÇÃO 17

Criando e manipulando tabelas

Nesta lição, você aprenderá os fundamentos da criação, alteração e exclusão de tabelas.

Criando tabelas

O SQL não é usado apenas para manipulação de dados da tabela. Em vez disso, pode ser usado para realizar todas as operações de banco de dados e tabelas, incluindo a criação e manipulação das próprias tabelas.

Geralmente, existem duas maneiras de criar tabelas de banco de dados:

- A maioria dos SGBDs vem com uma ferramenta de administração que você pode usar para criar e gerenciar tabelas de banco de dados interativamente.
- As tabelas também podem ser manipuladas diretamente com instruções SQL.

Para criar tabelas programaticamente, use CREATE TABLE. É importante observar que, ao usar ferramentas de gerenciamento interativas, você está, na verdade, usando instruções SQL. Em vez de você escrever essas instruções, no entanto, a interface gera e executa o SQL de forma transparente para você (o mesmo vale para alterações em tabelas existentes).

> **CUIDADO: Diferenças de sintaxe**
>
> A sintaxe exata de CREATE TABLE pode variar de uma implementação SQL para outra. Certifique-se de consultar a documentação do SGBD para obter mais informações sobre exatamente quais sintaxes e recursos ele suporta.

A cobertura completa de todas as opções disponíveis ao criar tabelas está além do escopo desta lição, mas aqui estão os fundamentos. Recomendo que você reveja a documentação de seu SGBD para obter mais informações e detalhes.

NOTA: Exemplos específicos do SGBD

Para obter exemplos de instruções CREATE TABLE específicas do SGBD, consulte os exemplos de scripts de criação de tabela descritos no *Apêndice A – Exemplos de scripts de tabela*.

Fundamentos de criação de tabelas

Para criar uma tabela usando CREATE TABLE, você deve especificar as seguintes informações:

- O nome da nova tabela especificado após as palavras-chave CREATE TABLE.
- O nome e a definição das colunas da tabela separados por vírgulas.
- Alguns SGBDs exigem que você também especifique o local da tabela (como em qual banco de dados específico ela deve ser criada).

A seguinte instrução SQL cria a tabela Products usada ao longo deste livro:

Entrada q

```
CREATE TABLE Products
(
   prod_id      CHAR(10)       NOT NULL,
   vend_id      CHAR(10)       NOT NULL,
   prod_name    CHAR(254)      NOT NULL,
   prod_price   DECIMAL(8,2)   NOT NULL,
   prod_desc    VARCHAR(1000)  NULL
);
```

Análise q

Como você pode ver na instrução acima, o nome da tabela é especificado imediatamente após as palavras-chave CREATE TABLE. A definição real da tabela (todas as colunas) está entre parênteses. As próprias colunas são separadas por vírgulas. Essa tabela específica é composta de cinco colunas. Cada definição de coluna começa com o nome da coluna (que deve ser exclusivo na tabela), seguido pelo tipo de dados da coluna. (Consulte a *Lição 1 – Entendendo SQL*, para uma explicação dos tipos de dados. Além disso, o *Apêndice C – Usando datatypes do SQL*, lista os tipos de dados comumente usados e sua compatibilidade.) A instrução inteira é encerrada com um ponto-e-vírgula após o parêntese de fechamento.

Mencionei anteriormente que a sintaxe CREATE TABLE varia muito de um

SGBD para outro, e o script simples anterior demonstra isso. Embora a instrução funcione como está na maioria dos SGBDs, para o DB2 o NULL deve ser removido da coluna final. É por isso que tive de criar um script de criação de tabela SQL diferente para cada SGBD (conforme explicado no *Apêndice A – Exemplos de scripts de tabela*).

> **DICA: Formatação da instrução**
>
> Como você deve se lembrar, espaços em branco são ignorados nas instruções SQL. As instruções podem ser digitadas em uma longa linha ou divididas em várias linhas. Não faz diferença alguma. Isso permite que você formate seu SQL da maneira mais adequada. A instrução CREATE TABLE anterior é um bom exemplo de formatação de instrução SQL: o código é especificado em várias linhas, com as definições de coluna recuadas para facilitar a leitura e edição. Formatar seu SQL dessa maneira é totalmente opcional, mas altamente recomendado.

> **DICA: Substituindo tabelas existentes**
>
> Quando você cria uma nova tabela, o nome da tabela especificado não deve existir; caso contrário, você gerará um erro. Para evitar a substituição acidental, o SQL requer que você primeiro remova manualmente uma tabela (consulte as seções posteriores para obter detalhes) e, em seguida, recrie-a, em vez de apenas a substituir.

Trabalhando com valores NULL

Na *Lição 4 – Filtrando dados*, você aprendeu que os valores NULL são um não valor ou a ausência de um valor. Uma coluna que permite valores NULL também permite que linhas sejam inseridas sem nenhum valor nessa coluna. Uma coluna que não permite valores NULL não aceita linhas sem valor; em outras palavras, essa coluna sempre será necessária quando as linhas forem inseridas ou atualizadas.

Cada coluna da tabela é uma coluna NULL ou uma coluna NOT NULL e esse estado é especificado na definição da tabela no momento da criação. Veja o seguinte exemplo:

Entrada q

```
CREATE TABLE Orders
(
   order_num     INTEGER     NOT NULL,
   order_date    DATETIME    NOT NULL,
   cust_id       CHAR(10)    NOT NULL
);
```

Análise q

Essa instrução cria a tabela Orders usada em todo este livro. A tabela Orders contém três colunas: o número do pedido, a data do pedido e a ID do cliente. Todas as três colunas são obrigatórias e, portanto, cada uma contém a palavra-chave NOT NULL. Isso evitará a inserção de colunas sem valor. Se alguém tentar inserir uma coluna sem valor, um erro será retornado e a inserção falhará.

O próximo exemplo cria uma tabela com uma mistura de colunas NULL e NOT NULL:

Entrada q

```
CREATE TABLE Vendors
(
   vend_id       CHAR(10)    NOT NULL,
   vend_name     CHAR(50)    NOT NULL,
   vend_address  CHAR(50)    ,
   vend_city     CHAR(50)    ,
   vend_state    CHAR(5)     ,
   vend_zip      CHAR(10)    ,
   vend_country  CHAR(50)
);
```

Análise q

Essa instrução cria a tabela Vendors usada em todo este livro. As colunas ID e nome do fornecedor são obrigatórias e, portanto, especificadas como NOT NULL. Todas as cinco colunas restantes permitem valores NULL e, portanto, NOT NULL não é especificado. NULL é a configuração padrão, portanto, se NOT NULL não for especificado, NULL será assumido.

> **CUIDADO: Especificando NULL**
>
> A maioria dos SGBDs trata a ausência de NOT NULL como se significasse NULL. No entanto, nem todos o fazem. Alguns SGBDs requerem a palavra-chave NULL e vão gerar um erro se ela não for especificada. Consulte a documentação de seu SGBD para obter informações completas sobre a sintaxe.

> **DICA: Chaves primárias e valores NULL**
>
> Na *Lição 1 – Entendendo SQL*, você aprendeu que as chaves primárias são colunas cujos valores identificam de forma única cada linha em uma tabela. Apenas colunas que não permitem valores NULL podem ser usadas em chaves primárias. As colunas que permitem nenhum valor não podem ser usadas como identificadores únicos.

CUIDADO: Compreendendo o valor NULL

Não confunda valores NULL com strings vazias. Um valor NULL é a ausência de um valor; não é uma string vazia. Se você especificasse '' (duas aspas simples sem nada entre elas), isso seria permitido em uma coluna NOT NULL. Uma string vazia é um valor válido; ela não é equivalente a nenhum valor. Valores NULL são especificados com a palavra-chave NULL, e não com uma string vazia.

Especificando valores padrão

O SQL permite que você especifique os valores padrão a serem usados se nenhum valor for especificado quando uma linha for inserida. Os valores padrão são especificados usando a palavra-chave DEFAULT nas definições de coluna na instrução CREATE TABLE.

Veja o seguinte exemplo:

Entrada q

```
CREATE TABLE OrderItems
(
   order_num     INTEGER        NOT NULL,
   order_item    INTEGER        NOT NULL,
   prod_id       CHAR(10)       NOT NULL,
   quantity      INTEGER        NOT NULL    DEFAULT 1,
   item_price    DECIMAL(8,2)   NOT NULL
);
```

Análise q

Essa instrução cria a tabela OrderItems que contém os itens individuais que constituem um pedido. (O próprio pedido é armazenado na tabela Orders.) A coluna quantity contém a quantidade de cada item em um pedido. Neste exemplo, adicionar o texto DEFAULT 1 à descrição da coluna instrui o SGBD a usar uma quantidade de 1 se nenhuma quantidade for especificada.

Os valores padrão são frequentemente usados para armazenar valores em colunas de data ou timestamp. Por exemplo, a data do sistema pode ser usada como uma data padrão especificando a função ou variável usada para se referir à data do sistema. Por exemplo, os usuários do MySQL podem especificar DEFAULT CURRENT_DATE(), enquanto os usuários do Oracle podem especificar DEFAULT SYSDATE e os usuários do SQL Server podem especificar DEFAULT GETDATE(). Infelizmente, o comando usado para obter a data do sistema é diferente em quase todos os SGBDs. A Tabela 17.1 lista a

sintaxe de alguns SGBDs. Se o seu não estiver listado aqui, consulte a documentação de seu SGBD.

Tabela 17.1 – Obtendo a data do sistema

SGBD	Função/Variável
DB2	CURRENT_DATE
MySQL	CURRENT_DATE() ou Now()
Oracle	SYSDATE
PostgreSQL	CURRENT_DATE
SQL Server	GETDATE()
SQLite	date('now')

DICA: Usando DEFAULT em vez de valores NULL

Muitos desenvolvedores de banco de dados usam valores DEFAULT em vez de colunas NULL, especialmente em colunas que serão usadas em cálculos ou agrupamentos de dados.

Atualizando tabelas

Para atualizar as definições da tabela, você usa a instrução ALTER TABLE. Embora todos os SGBDs suportem ALTER TABLE, o que eles permitem alterar varia drasticamente de um para outro. Aqui estão alguns pontos a considerar ao usar ALTER TABLE:

- Idealmente, as tabelas nunca devem ser alteradas depois que elas contiverem dados. Você deve gastar tempo suficiente antecipando as necessidades futuras durante o processo de projeto da tabela para que mudanças extensivas não sejam necessárias mais tarde.
- Todos os SGBDs permitem adicionar colunas às tabelas existentes, embora alguns restrinjam os tipos de dados que podem ser adicionados (assim como o uso de NULL e DEFAULT).
- Muitos SGBDs não permitem que você remova ou altere colunas em uma tabela.
- A maioria dos SGBDs permite que você renomeie colunas.
- Muitos SGBDs restringem os tipos de alterações que você pode fazer em colunas que estão preenchidas e impõem menos restrições em colunas

não preenchidas.

Como você pode ver, fazer alterações nas tabelas existentes não é simples nem consistente. Certifique-se de consultar a documentação de seu SGBD para determinar exatamente o que você pode alterar.

Para alterar uma tabela usando ALTER TABLE, especifique as seguintes informações:

- O nome da tabela a ser alterada após as palavras-chave ALTER TABLE. (A tabela deve existir; caso contrário, um erro será gerado.)
- A lista de alterações a serem feitas.

Como adicionar colunas a uma tabela existente é praticamente a única operação suportada por todos os SGBDs, vou usar isso como um exemplo:

Entrada q

```
ALTER TABLE Vendors
ADD vend_phone CHAR(20);
```

Análise q

Esta instrução adiciona uma coluna chamada vend_phone à tabela Vendors. O datatype deve ser especificado.

Outras operações ALTER – por exemplo, alterar ou descartar colunas, ou adicionar restrições ou chaves – usam uma sintaxe semelhante.

Observe que o exemplo a seguir não funcionará com todos os SGBDs:

Entrada q

```
ALTER TABLE Vendors
DROP COLUMN vend_phone;
```

Alterações complexas da estrutura de tabela geralmente requerem um processo manual envolvendo estas etapas:

1. Crie uma nova tabela com o novo layout da coluna.
2. Use a instrução INSERT SELECT (ver *Lição 15 – Inserindo dados*, para obter detalhes desta instrução) para copiar os dados da tabela antiga para a nova tabela. Use funções de conversão e campos calculados, se necessário.
3. Verifique se a nova tabela contém os dados desejados.

4. Renomeie a antiga tabela (ou exclua-a, se você for realmente corajoso).
5. Renomeie a nova tabela com o nome anteriormente usado pela tabela antiga.
6. Recrie quaisquer triggers, stored procedures, índices e chaves estrangeiras conforme necessário.

NOTA: ALTER TABLE e o SQLite

O SQLite limita as operações que podem ser realizadas usando ALTER TABLE. Uma das limitações mais importantes é que ele não suporta o uso de ALTER TABLE para definir chaves primárias e estrangeiras; elas devem ser especificadas no momento inicial de criação da tabela com a instrução CREATE TABLE.

CUIDADO: Use ALTER TABLE com cuidado

Use ALTER TABLE com muito cuidado e certifique-se de ter um conjunto completo de backups (esquema e dados) antes de continuar. As alterações na tabela do banco de dados não podem ser desfeitas e, se você adicionar colunas desnecessárias, talvez não consiga removê-las. Da mesma forma, se você descartar uma coluna necessária, poderá perder os dados dessa coluna.

Excluindo tabelas

Excluir tabelas (na verdade, remover a tabela inteira, não apenas o conteúdo) é muito fácil – alguns diriam que é fácil demais. As tabelas são excluídas usando a instrução DROP TABLE:

Entrada q

```
DROP TABLE CustCopy;
```

Análise q

Esta instrução exclui a tabela CustCopy. (Você a criou na *Lição 15 – Inserindo dados*.) Não há confirmação e a operação não pode ser desfeita. Executar a instrução removerá permanentemente a tabela.

DICA: Usando regras relacionais para evitar exclusão acidental

Muitos SGBDs permitem que você aplique regras que evitam o descarte de tabelas relacionadas a outras tabelas. Quando essas regras são aplicadas, se você emitir uma instrução DROP TABLE em uma tabela que faz parte de um relacionamento, o SGBD bloqueia a operação até que o relacionamento seja removido. É uma boa ideia ativar essas opções, se disponíveis, para evitar o descarte acidental das tabelas necessárias.

Renomeando tabelas

Renomear tabelas é uma operação suportada de maneira diferente por cada SGBD. Não existe um padrão rígido e rápido para esta operação. Os usuários do DB2, MariaDB, MySQL, Oracle e PostgreSQL podem usar a instrução RENAME. Os usuários do SQL Server podem usar o stored procedure sp_rename fornecido. O SQLite suporta a renomeação de tabelas por meio da instrução ALTER TABLE.

A sintaxe básica para todas as operações de renomeação requer que você especifique o nome antigo e um novo nome; no entanto, existem diferenças de implementação do SGBD. Consulte a documentação de seu próprio SGBD para obter detalhes sobre a sintaxe compatível.

Resumo

Nesta lição, você aprendeu várias novas instruções SQL. A instrução CREATE TABLE é usada para criar novas tabelas, ALTER TABLE é usada para alterar as colunas da tabela (ou outros objetos como restrições ou índices) e DROP TABLE é usada para excluir completamente uma tabela. Essas instruções devem ser usadas com extremo cuidado e somente após os backups terem sido feitos. Como a sintaxe exata de cada uma dessas instruções varia de um SGBD para outro, você deve consultar a documentação de seu próprio SGBD para obter mais informações.

Desafios

1. Adicione uma coluna de site (vend_web) à tabela Vendors. Você precisa de um campo de texto grande o suficiente para acomodar uma URL.
2. Use as instruções UPDATE para atualizar os registros de Vendor para incluir um site (você pode inventar qualquer endereço).

LIÇÃO 18

Usando visualizações (views)

Nesta lição, você aprenderá exatamente o que são visualizações (views), como funcionam e quando devem ser usadas. Você também verá como as visualizações podem ser usadas para simplificar algumas das operações SQL realizadas nas lições anteriores.

Compreendendo as visualizações

As visualizações são tabelas virtuais. Ao contrário das tabelas que contêm dados, as visualizações simplesmente contêm consultas que retornam dados dinamicamente quando usadas.

> **NOTA: Visualizações no SQLite**
>
> O SQLite oferece suporte apenas para visualizações somente para leitura, portanto, as visualizações podem ser criadas e lidas, mas seu conteúdo não pode ser atualizado.

A melhor maneira de entender visualizações é observar um exemplo: Na *Lição 12 – Juntando tabelas*, você usou a seguinte instrução SELECT para obter dados de três tabelas:

Entrada q

```
SELECT cust_name, cust_contact
FROM Customers, Orders, OrderItems
WHERE Customers.cust_id = Orders.cust_id
 AND OrderItems.order_num = Orders.order_num
 AND prod_id = 'RGAN01';
```

Essa consulta foi usada para obter os clientes que haviam pedido um produto específico. Qualquer pessoa que necessite desses dados deve compreender a estrutura da tabela, bem como a maneira de criar a consulta e juntar as tabelas. Para obter os mesmos dados para outro produto (ou para vários produtos), você teria de modificar a última cláusula WHERE.

Agora imagine que você possa agrupar toda a consulta em uma tabela virtual chamada ProductCustomers. Você poderia simplesmente fazer o seguinte

para obter os mesmos dados:

Entrada q

```
SELECT cust_name, cust_contact
FROM ProductCustomers
WHERE prod_id = 'RGAN01';
```

É aqui que as visualizações começam a fazer sentido. ProductCustomers é uma visualização e, como uma visualização, ela não contém colunas ou dados. Em vez disso, ela contém uma consulta – a mesma consulta usada anteriormente para juntar as tabelas corretamente.

> **DICA: Consistência entre SGBDs**
>
> Você ficará aliviado ao saber que a sintaxe de criação de visualização é suportada de forma bastante consistente por todos os principais SGBDs.

Por que usar visualizações

Você já viu um uso para visualizações. Aqui estão alguns outros usos comuns:

- Para reutilizar instruções SQL.
- Para simplificar operações SQL complexas. Depois que a consulta é escrita, ela pode ser reutilizada facilmente, sem precisar saber os detalhes da própria consulta subjacente.
- Para expor partes de uma tabela em vez de tabelas completas.
- Para proteger os dados. Os usuários podem ter acesso a subconjuntos específicos de tabelas em vez de tabelas inteiras.
- Para alterar a formatação e representação dos dados. As visualizações podem retornar os dados formatados e apresentados de forma diferente de suas tabelas subjacentes.

Na maioria das vezes, depois que as visualizações são criadas, elas podem ser usadas da mesma forma que as tabelas. Você pode executar operações SELECT, filtrar e classificar dados, juntar visualizações a outras visualizações ou tabelas e, possivelmente, até mesmo adicionar e atualizar dados. (Há algumas restrições em relação a esse último item. Falaremos mais sobre isso em breve.)

O importante a se lembrar é que as visualizações são apenas isso –

visualizações de dados armazenados em outros lugares. As visualizações em si não contêm dados, de modo que os dados que elas retornam são obtidos de outras tabelas. Quando os dados são adicionados ou alterados nessas tabelas, as visualizações retornarão os dados alterados.

CUIDADO: Problemas de desempenho

Como as visualizações não contêm dados, qualquer obtenção de dados necessária para executar uma consulta deve ser processada toda vez que a visualização for usada. Se você criar visualizações complexas com várias junções e filtros, ou se aninhar visualizações, talvez descubra que o desempenho se degrada dramaticamente. Certifique-se de testar a execução antes de implantar aplicações que usam amplamente as visualizações.

Regras e restrições das visualizações

Antes de você mesmo criar visualizações, esteja ciente de algumas restrições. Infelizmente, as restrições tendem a ser muito específicas do SGBD, então verifique a documentação de seu próprio SGBD antes de prosseguir.

Aqui estão algumas das regras e restrições mais comuns que regem a criação e o uso de visualizações:

- Como tabelas, as visualizações devem ter nomes únicos. (Elas não podem ser nomeadas com o nome de qualquer outra tabela ou visualização.)
- Não há limite para o número de visualizações que podem ser criadas.
- Para criar visualizações, você deve ter acesso de segurança. Esse nível de acesso geralmente é concedido pelo administrador do banco de dados.
- As visualizações podem ser aninhadas; ou seja, uma visualização pode ser construída usando uma consulta que obtém dados de outra visualização. O número exato de níveis aninhados permitidos varia de SGBD para SGBD. (Aninhar visualizações pode degradar seriamente o desempenho da consulta, por isso teste com cuidado antes de usar esse recurso em ambientes de produção.)
- Muitos SGBDs proíbem o uso da cláusula ORDER BY em consultas de visualização.

- Alguns SGBDs exigem que cada coluna retornada seja nomeada; isso exigirá o uso de aliases se as colunas forem campos calculados. (Veja a *Lição 7 – Criando campos calculados*, para obter mais informações sobre aliases de coluna.)
- As visualizações não podem ser indexadas, nem podem ter triggers ou valores padrão associados a elas.
- Alguns SGBDs, como o SQLite, tratam as visualizações como consultas somente de leitura, o que significa que você pode obter dados das visualizações, mas não gravar dados de volta para as tabelas subjacentes. Consulte a documentação de seu SGBD para obter mais detalhes.
- Alguns SGBDs permitem criar visualizações que não permitem que linhas sejam inseridas ou atualizadas se essa inserção ou atualização fizer com que essa linha não faça mais parte da visualização. Por exemplo, se você tiver uma visualização que obtém apenas clientes com endereços de e-mail, atualizar um cliente para remover seu endereço de e-mail fará com que esse cliente saia da visualização. Esse é o comportamento padrão e é permitido, mas, dependendo do seu SGBD, você pode ser capaz de impedir que isso ocorra.

DICA: Consulte a documentação de seu SGBD

Essa é uma longa lista de regras, e a documentação de seu próprio SGBD provavelmente conterá regras adicionais também. Vale a pena gastar um tempo para entender quais restrições você deve respeitar antes de criar visualizações.

Criando visualizações

Então, agora que você sabe o que são as visualizações (e as regras e restrições que as governam), vamos observar a criação de visualizações.

As visualizações são criadas usando a instrução CREATE VIEW. Como CREATE TABLE, a instrução CREATE VIEW só pode ser usada para criar uma visualização que não existe.

NOTA: Renomeando visualizações

Para remover uma visualização, você deve usar a instrução DROP. A sintaxe é simplesmente DROP VIEW nome_visualização;

Para substituir (ou atualizar) uma visualização, você deve primeiro DROP (excluí-la) e, em seguida, recriá-la.

Usando visualizações para simplificar junções complexas

Um dos usos mais comuns de visualizações é esconder SQL complexo, e isso muitas vezes envolve junções (joins). Observe a seguinte instrução:

Entrada q

```
CREATE VIEW ProductCustomers AS
SELECT cust_name, cust_contact, prod_id
FROM Customers, Orders, OrderItems
WHERE Customers.cust_id = Orders.cust_id
 AND OrderItems.order_num = Orders.order_num;
```

Análise q

Esta instrução cria uma visualização chamada ProductCustomers, que junta três tabelas para retornar uma lista de todos os clientes que encomendaram qualquer produto. Se você fosse usar SELECT * FROM ProductCustomers, listaria todos os clientes que encomendaram algum item.

Para obter uma lista de clientes que encomendaram o produto RGAN01, você pode fazer o seguinte:

Entrada q

```
SELECT cust_name, cust_contact
FROM ProductCustomers
WHERE prod_id = 'RGAN01';
```

Saída q

```
cust_name      cust_contact
-------------- --------------
Fun4All        Denise L. Stephens
The Toy Store  Kim Howard
```

Análise q

Esta instrução obtém dados específicos da visualização usando uma cláusula WHERE. Quando o SGBD processa a solicitação, ele adiciona a cláusula WHERE especificada a qualquer cláusula WHERE existente na consulta da visualização para que os dados sejam filtrados corretamente.

Como você pode ver, as visualizações podem simplificar muito o uso de instruções SQL complexas. Usando visualizações, você pode escrever o SQL subjacente uma vez e depois o reutilizar conforme necessário.

> **DICA: Criando visualizações reutilizáveis**
>
> É uma boa ideia criar visualizações que não estejam vinculadas a dados específicos. Por exemplo, a visualização criada anteriormente retorna os clientes para todos os produtos, não apenas para o produto RGAN01 (para o qual a visualização foi criada pela primeira vez). Expandir o escopo da visualização permite que ela seja reutilizada, tornando-a ainda mais útil. Isso também elimina a necessidade de você criar e manter várias visualizações semelhantes.

Usando visualizações para reformatar dados obtidos

Como mencionado anteriormente, outro uso comum de visualizações é reformatar dados obtidos. A seguinte instrução SELECT do SQL Server (da *Lição 7 – Criando campos calculados*) retorna o nome e a localização do fornecedor em uma única coluna calculada combinada:

Entrada q

```
SELECT RTRIM(vend_name) + ' (' + RTRIM(vend_country) + ')'
    AS vend_title
FROM Vendors
ORDER BY vend_name;
```

Saída q

```
vend_title
------------------------------------------------------------
Bear Emporium (USA)
Bears R Us (USA)
Doll House Inc. (USA)
Fun and Games (England)
Furball Inc. (USA)
Jouets et ours (France)
```

A seguir, temos a mesma instrução, mas usando a sintaxe || (como explicado na *Lição 7 – Criando campos calculados*):

Entrada q

```
SELECT RTRIM(vend_name) || ' (' || RTRIM(vend_country) || ')'
    AS vend_title
FROM Vendors
ORDER BY vend_name;
```

Saída q

```
vend_title
------------------------------------------------------------
Bear Emporium (USA)
Bears R Us (USA)
Doll House Inc. (USA)
Fun and Games (England)
Furball Inc. (USA)
Jouets et ours (France)
```

Agora suponha que você precise regularmente de resultados nesse formato. Em vez de executar a concatenação cada vez que fosse necessário, você poderia criar uma visualização e usá-la. Para transformar essa instrução em uma visualização, pode fazer o seguinte:

Entrada q

```
CREATE VIEW VendorLocations AS
SELECT RTRIM(vend_name) + ' (' + RTRIM(vend_country) + ')'
    AS vend_title
FROM Vendors;
```

Aqui está a mesma instrução usando a sintaxe ||:

Entrada q

```
CREATE VIEW VendorLocations AS
SELECT RTRIM(vend_name) || ' (' || RTRIM(vend_country) || ')'
    AS vend_title
FROM Vendors;
```

Análise q

Esta instrução cria uma visualização usando exatamente a mesma consulta da instrução SELECT anterior. Para obter os dados para criar todas as etiquetas postais, basta fazer o seguinte:

Entrada q

```
SELECT * FROM VendorLocations;
```

Saída q

```
vend_title
------------------------------------------------------------
Bear Emporium (USA)
Bears R Us (USA)
```

Doll House Inc. (USA)
Fun and Games (England)
Furball Inc. (USA)
Jouets et ours (France)

> **NOTA: Todas as restrições se aplicam à instrução SELECT**
>
> No início desta lição, afirmei que a sintaxe usada para criar visualizações é bastante consistente entre SGBDs. Então por que várias versões das instruções? Uma visualização simplesmente envolve uma instrução SELECT e a sintaxe desse SELECT deve seguir todas as regras e restrições do SGBD que está sendo usado.

Usando visualizações para filtrar dados indesejados

As visualizações também são úteis para aplicar cláusulas WHERE comuns. Por exemplo, você pode querer definir uma visualização CustomerEMailList para que ela filtre os clientes sem endereços de e-mail. Para isso, pode usar a seguinte instrução:

Entrada q

```
CREATE VIEW CustomerEMailList AS
SELECT cust_id, cust_name, cust_email
FROM Customers
WHERE cust_email IS NOT NULL;
```

Análise q

Obviamente, ao enviar e-mails para uma lista de mailing, recomenda-se ignorar usuários que não têm endereço de e-mail. A cláusula WHERE filtra as linhas que têm valores NULL nas colunas cust_email para que elas não sejam obtidas.

A visualização CustomerEMailList agora pode ser usada como qualquer tabela:

Entrada q

```
SELECT *
FROM CustomerEMailList;
```

Saída q

```
cust_id     cust_name     cust_email
----------  ------------  --------------------
1000000001  Village Toys  sales@villagetoys.com
1000000003  Fun4All       jjones@fun4all.com
1000000004  Fun4All       dstephens@fun4all.com
```

NOTA: Cláusulas WHERE e cláusulas WHERE

Se uma cláusula WHERE for usada ao obter dados da visualização, os dois conjuntos de cláusulas (o da visualização e aquele passado para ela) serão combinados automaticamente.

Usando visualizações com campos calculados

As visualizações são excepcionalmente úteis para simplificar o uso de campos calculados. A seguinte instrução SELECT foi introduzida na *Lição 7 – Criando campos calculados*. Ela obtém os itens de pedido para um pedido específico, calculando o preço expandido para cada item:

Entrada q

```
SELECT prod_id,
    quantity,
    item_price,
    quantity*item_price AS expanded_price
FROM OrderItems
WHERE order_num = 20008;
```

Saída q

```
prod_id    quantity    item_price      expanded_price
--------   ---------   -----------     --------------
RGAN01     5           4.9900          24.9500
BR03       5           11.9900         59.9500
BNBG01     10          3.4900          34.9000
BNBG02     10          3.4900          34.9000
BNBG03     10          3.4900          34.9000
```

Para transformar isso em uma visualização, você pode fazer o seguinte:

Entrada q

```
CREATE VIEW OrderItemsExpanded AS
SELECT order_num,
    prod_id,
    quantity,
    item_price,
    quantity*item_price AS expanded_price
FROM OrderItems
```

Para obter os detalhes do pedido 20008(a saída anterior), faça o seguinte:

Entrada q

```
SELECT *
```

```
FROM OrderItemsExpanded
WHERE order_num = 20008;
```

Saída q

```
order_num  prod_id   quantity   item_price   expanded_price
---------  -------   ---------  ----------   --------------
20008      RGAN01    5          4.99         24.95
20008      BR03      5          11.99        59.95
20008      BNBG01    10         3.49         34.90
20008      BNBG02    10         3.49         34.90
20008      BNBG03    10         3.49         34.90
```

Como você pode ver, as visualizações são fáceis de criar e ainda mais fáceis de usar. Usadas corretamente, elas podem simplificar muito a manipulação de dados complexas.

Resumo

As visualizações são tabelas virtuais. Elas não contêm dados, mas, em vez disso, contêm consultas que obtêm dados conforme necessário. As visualizações fornecem um nível de encapsulamento em torno das instruções SELECT do SQL e podem ser usadas para simplificar a manipulação de dados, bem como para reformatar ou proteger dados subjacentes.

Desafios

1. Crie uma visualização chamada CustomersWithOrders que contém todas as colunas em Customers, mas inclui apenas aquelas que fizeram pedidos. Dica: você pode usar JOIN na tabela Orders para filtrar apenas os clientes que você deseja. Em seguida, use uma instrução SELECT para ter certeza de que você tem os dados certos.
2. O que há de errado com a seguinte instrução SQL? (Tente descobrir sem executá-la.)

```
CREATE VIEW OrderItemsExpanded AS
SELECT order_num,
    prod_id,
    quantity,
    item_price,
```

```
    quantity*item_price AS expanded_price
FROM OrderItems
ORDER BY order_num;
```

LIÇÃO 19

Trabalhando com stored procedures

Nesta lição, você aprenderá o que são stored procedures, por que e como são usadas. Também examinará a sintaxe básica para criá-las e usá-las.

Compreendendo stored procedures

A maioria das instruções SQL que usamos até agora são simples, pois usam uma única instrução em relação a uma ou mais tabelas. Nem todas as operações são tão simples. Frequentemente, várias instruções serão necessárias para executar uma operação completa. Por exemplo, considere o seguinte cenário:

- Para processar um pedido, verificações devem ser feitas para garantir que os itens estão em estoque.
- Se houver itens em estoque, eles precisam ser reservados para que não sejam vendidos a mais ninguém e a quantidade disponível deve ser reduzida para refletir a quantidade correta em estoque.
- Quaisquer itens sem estoque precisam ser encomendados; isso requer alguma interação com o fornecedor.
- O cliente precisa ser notificado sobre quais itens estão em estoque (e podem ser enviados imediatamente) e quais estão em espera.

Obviamente, esse não é um exemplo completo e está até mesmo além do escopo das tabelas de exemplo que usamos neste livro, mas será suficiente para ilustrar esse tópico. A execução desse processo requer muitas instruções SQL acessando muitas tabelas. Além disso, as instruções SQL exatas que precisam ser executadas e sua ordem não são fixas; elas podem (e vão) variar de acordo com quais itens estão em estoque e quais não estão.

Como você escreveria esse código? Pode escrever cada uma das instruções SQL individualmente e executar outras instruções condicionalmente com

base no resultado. Você teria de fazer isso sempre que esse processamento fosse necessário (e em todas as aplicações que precisassem dele).

Você pode criar uma stored procedure. Stored procedures são simplesmente coleções de uma ou mais instruções SQL salvas para uso futuro. Você pode pensar nelas como arquivos em lote, embora, na verdade, sejam mais do que isso.

NOTA: Não suportada no SQLite

O SQLite não suporta stored procedures.

NOTA: Há muito mais do que isso

As stored procedures são complexas e a cobertura completa do assunto requer mais espaço do que podemos dedicar neste livro. Na verdade, existem livros inteiros sobre o assunto. Esta lição não ensinará tudo o que você precisa saber sobre stored procedures. Em vez disso, a intenção é simplesmente apresentar o assunto para que você esteja familiarizado com o conceito de stored procedure e o que elas podem fazer. Dessa forma, os exemplos apresentados aqui fornecem apenas a sintaxe para Oracle e SQL Server.

Compreendendo por que usar stored procedures

Já que você sabe o que são stored procedures, por que as usar? Existem muitos motivos, mas aqui estão os principais:

- Para simplificar operações complexas (como visto no exemplo anterior), encapsulando processos em uma única unidade fácil de usar.
- Para garantir a consistência dos dados, sem a necessidade de criar uma série de etapas repetidamente. Se todos os desenvolvedores e aplicações usarem a mesma stored procedure, o mesmo código será usado por todos.
- Para evitar erros; esta é uma extensão do motivo anterior. Quanto mais etapas precisarem ser executadas, maior a probabilidade de introdução de erros. A prevenção de erros garante a consistência dos dados.
- Para simplificar o gerenciamento de mudanças. Se tabelas, nomes de colunas ou lógica de negócios (ou quase qualquer coisa) forem alterados, apenas o código da stored procedure precisará ser atualizado e ninguém mais precisará estar ciente de que as alterações foram feitas.

- Para garantir a segurança; esta é uma extensão do motivo anterior. Restringir o acesso aos dados subjacentes por meio de stored procedures reduz a chance de corrupção de dados (intencional ou não).
- Menos trabalho para processar o comando. Como as stored procedures geralmente são armazenadas em uma forma compilada, o SGBD tem menos trabalho. Isso resulta em melhor desempenho.
- Para escrever código mais poderoso e flexível. Existem elementos e recursos da linguagem SQL que estão disponíveis apenas em solicitações individuais. As stored procedures podem usá-los por esse motivo.

Em outras palavras, existem três benefícios principais: simplicidade, segurança e desempenho. Obviamente, todos são extremamente importantes. Antes de transformar todo o seu código SQL em stored procedures, aqui estão as desvantagens:

- A sintaxe de stored procedure varia drasticamente de um SGBD para o outro. Na verdade, é quase impossível escrever stored procedures verdadeiramente portáveis. Dito isso, as próprias chamadas de stored procedure (seus nomes e como os dados são passados) podem ser mantidas relativamente portáveis para que, se você precisar mudar para outro SGBD, pelo menos o código da aplicação cliente não precise ser alterado.
- As stored procedures tendem a ser mais complexas de escrever do que as instruções SQL básicas, e escrevê-las requer um grau maior de habilidade e experiência. Como resultado, muitos administradores de banco de dados restringem os direitos de criação de stored procedure como uma medida de segurança (principalmente devido ao item anterior).

No entanto, stored procedures são muito úteis e devem ser usadas. Na verdade, a maioria dos SGBDs vem com vários tipos de stored procedures que são usadas para gerenciamento de banco de dados e tabelas. Consulte a documentação de seu SGBD para obter mais informações sobre o tema.

NOTA: Não consegue escrevê-las? Você ainda pode usá-las

A maioria dos SGBDs distingue a segurança e o acesso necessários para escrever stored procedures da segurança e o acesso necessários para executá-las. Isso é uma coisa boa; mesmo se você não puder (ou não quiser) escrever suas próprias stored procedures, ainda poderá executá-las quando apropriado.

Executando stored procedures

As stored procedures são executadas com muito mais frequência do que são escritas, portanto, começaremos por aí. A instrução SQL para executar uma stored procedure é simplesmente `EXECUTE`. É preciso informar para a instrução `EXECUTE` o nome da stored procedure e quaisquer parâmetros que precisem ser passados a ela. Dê uma olhada neste exemplo (você não pode realmente a executar porque a stored procedure `AddNewProduct` não existe):

Entrada ▼

```
EXECUTE AddNewProduct('JTS01',
                      'Stuffed Eiffel Tower',
                      6.49,
                      'Plush stuffed toy with
➥the text La Tour Eiffel in red white and blue');
```

Análise ▼

Aqui, uma stored procedure denominada `AddNewProduct` é executada; ela adiciona um novo produto à tabela `Products`. `AddNewProduct` recebe quatro parâmetros: a ID do fornecedor (a chave primária da tabela `Vendors`), o nome do produto, o preço e a descrição. Esses quatro parâmetros correspondem a quatro variáveis esperadas dentro da stored procedure (definidas como parte da própria stored procedure). A stored procedure adiciona uma nova linha à tabela `Products` e atribui esses atributos passados às colunas apropriadas.

Na tabela `Products` você notará que outra coluna precisa de um valor – a coluna `prod_id`, que é a chave primária da tabela. Por que esse valor não foi passado como um atributo para a stored procedure? Para garantir que as IDs sejam geradas corretamente, é mais seguro ter esse processo automatizado (e não depender dos usuários finais). É por isso que uma

stored procedure é usada neste exemplo. Isto é o que essa stored procedure faz:

- Valida os dados passados, garantindo que todos os quatro parâmetros tenham valores.
- Gera uma ID única para ser usada como chave primária.
- Insere o novo produto na tabela `Products`, armazenando a chave primária gerada e os dados passados nas colunas apropriadas.

Essa é a forma básica de execução da stored procedure. Dependendo do SGBD usado, outras opções de execução incluem o seguinte:

- Parâmetros opcionais, com valores padrão assumidos se um parâmetro não for fornecido.
- Parâmetros fora de ordem, especificados em pares `parâmetro = valor`.
- Parâmetros de saída, permitindo que a stored procedure atualize um parâmetro para uso na aplicação em execução.
- Dados obtidos por uma instrução `SELECT`.
- Códigos de retorno, permitindo que a stored procedure retorne um valor para a aplicação em execução.

Criando stored procedures

Como já explicado, escrever uma stored procedure não é trivial. Para lhe dar uma ideia do que está envolvido, vejamos um exemplo simples – uma stored procedure que conta o número de clientes em uma lista de mala direta que têm endereços de e-mail.

Aqui está a versão para Oracle:

Entrada ▼

```
CREATE PROCEDURE MailingListCount (
  ListCount OUT INTEGER
)
IS
v_rows INTEGER;
BEGIN
    SELECT COUNT(*) INTO v_rows
    FROM Customers
```

```
    WHERE NOT cust_email IS NULL;
    ListCount := v_rows;
END;
```

Análise ▼

Esta stored procedure usa um único parâmetro denominado `ListCount`. Em vez de passar um valor para a stored procedure, ela retorna um valor nesse parâmetro. A palavra-chave `OUT` é usada para especificar esse comportamento. O Oracle suporta parâmetros dos tipos `IN` (aqueles passados para stored procedures), `OUT` (aqueles retornados pelas stored procedures, como usamos aqui) e `INOUT` (aqueles usados para passar parâmetros de e para stored procedures). O código da stored procedure está inserido entre as instruções `BEGIN` e `END`, e aqui um simples `SELECT` é executado para obter os clientes com endereços de e-mail. Então, `ListCount` (o parâmetro de saída passado) é configurado com o número de linhas que foram obtidas.

Para invocar o exemplo do Oracle, você pode fazer o seguinte:

Entrada ▼

```
var ReturnValue NUMBER
EXEC MailingListCount(:ReturnValue);
SELECT ReturnValue;
```

Análise ▼

Este código declara uma variável para conter qualquer coisa que a stored procedure retornar, executa a stored procedure e usa um `SELECT` para exibir o valor retornado.

Esta é a versão para o Microsoft SQL Server:

Entrada ▼

```
CREATE PROCEDURE MailingListCount
AS
DECLARE @cnt INTEGER
SELECT @cnt = COUNT(*)
FROM Customers
WHERE NOT cust_email IS NULL;
RETURN @cnt;
```

Análise ▼

Esta stored procedure não aceita nenhum parâmetro. A aplicação chamadora obtém o valor usando o suporte ao código de retorno do SQL Server. Aqui, uma variável local chamada `@cnt` é declarada usando a instrução `DECLARE` (todas as variáveis locais no SQL Server são nomeadas começando com `@`). Essa variável é então usada na instrução `SELECT` para conter o valor retornado pela função `COUNT()`. Finalmente, a instrução `RETURN` é usada para retornar a contagem para a aplicação chamadora como `RETURN @cnt`.

Para invocar o exemplo para o SQL Server, você pode fazer o seguinte:

Entrada ▼

```
DECLARE @ReturnValue INT
EXECUTE @ReturnValue=MailingListCount;
SELECT @ReturnValue;
```

Análise ▼

Este código declara uma variável para conter qualquer coisa que a stored procedure retornar, executa a stored procedure e usa um `SELECT` para exibir o valor retornado.

Aqui está outro exemplo, desta vez para inserir um novo pedido na tabela `Orders`. Esse é um exemplo apenas para o SQL Server, mas demonstra alguns usos e técnicas úteis de stored procedures:

Entrada ▼

```
CREATE PROCEDURE NewOrder @cust_id CHAR(10)
AS
- Declara variável para o número do pedido
DECLARE @order_num INTEGER
- Obtém o número de pedido mais alta no momento
SELECT @order_num=MAX(order_num)
FROM Orders
- Determina o próximo número do pedido
SELECT @order_num=@order_num+1
- Insere novo pedido
INSERT INTO Orders(order_num, order_date, cust_id)
VALUES(@order_num, GETDATE(), @cust_id)
- Retorna o número do pedido
```

```
RETURN @order_num;
```

Análise ▼

Esta stored procedure cria um novo pedido na tabela `Orders`. Ela recebe um único parâmetro – a ID do cliente que está fazendo o pedido. As outras duas colunas da tabela, o número do pedido e a data do pedido, são geradas automaticamente na própria stored procedure. O código primeiro declara uma variável local para armazenar o número do pedido. Em seguida, o número de pedido mais alto atual é obtido (usando uma função `MAX()`) e incrementado (usando uma instrução `SELECT`). Em seguida, o pedido é inserido com uma instrução `INSERT` usando o número de pedido recém-gerado, a data atual do sistema (obtida usando a função `GETDATE()`) e a ID do cliente que foi passada como parâmetro. Finalmente, o número do pedido (que é necessário para processar itens de pedido) é retornado como `RETURN @order_num`. Observe que o código é comentado; isso deve sempre ser feito ao escrever stored procedures.

> **NOTA: Comente seu código**
>
> Todo código deve ser comentado e as stored procedures não são diferentes. Adicionar comentários não afetará de modo algum o desempenho, então não há desvantagem nisso (além do tempo necessário para escrevê-los).
>
> Os benefícios são inúmeros e incluem tornar mais fácil para os outros (e para você mesmo) entender o código e mais seguro fazer mudanças em uma data posterior.
>
> Como observado na *Lição 2 – Obtendo dados*, uma maneira comum de comentar código é precedê-lo com -- (dois hifens). Alguns SGBDs suportam outra sintaxe de comentários, mas todos suportam -- e assim é melhor usar essa sintaxe.

Aqui está uma versão bem diferente do mesmo código para o SQL Server:

Entrada ▼

```
CREATE PROCEDURE NewOrder @cust_id CHAR(10)
AS
- Insere novo pedido
INSERT INTO Orders(cust_id)
VALUES(@cust_id)
- Retorna o número do pedido
SELECT order_num = @@IDENTITY;
```

Análise ▼

Esta stored procedure também cria um novo pedido na tabela `Orders`.

Desta vez, o próprio SGBD gera o número do pedido. A maioria dos SGBDs suporta esse tipo de funcionalidade; o SQL Server refere-se a essas colunas de incremento automático como campos de identidade (outros SGBDs usam nomes como Número automático [Auto Number]ou Sequências [Sequences]). Mais uma vez um único parâmetro é passado – a ID do cliente que está fazendo o pedido. O número do pedido e a data do pedido não estão especificados; o SGBD usa um valor padrão para a data (a função `GETDATE()`), e o número do pedido é gerado automaticamente. Como você pode descobrir qual é a ID gerada? O SQL Server torna isso disponível na variável global `@@IDENTITY`, que é retornada à aplicação chamadora (desta vez usando uma instrução `SELECT`).

Como você pode ver, com stored procedures muitas vezes há muitas maneiras diferentes de realizar a mesma tarefa. O método escolhido muitas vezes será ditado pelos recursos do SGBD que você está usando.

Resumo

Nesta lição, você aprendeu o que são stored procedures e por que são usadas. Também aprendeu o básico sobre a execução de stored procedures e a sintaxe de criação, e viu algumas das maneiras pelas quais elas podem ser usadas. Usar stored procedures é um tema muito importante e está muito além do escopo de uma lição. Como você viu aqui, as stored procedures são implementadas de forma diferente em cada SGBD. Além disso, seu próprio SGBD provavelmente oferece alguma forma dessas funções, assim como outras não mencionadas aqui. Consulte a documentação de seu SGBD para obter mais detalhes.

LIÇÃO 20

Gerenciando o processamento de transações

Nesta lição, você aprenderá o que são transações e como usar instruções `COMMIT` *e* `ROLLBACK` *para gerenciar o processamento de transações.*

Compreendendo o processamento de transações

O processamento de transações é usado para manter a integridade do banco de dados, garantindo que lotes de operações SQL sejam executados completamente ou não sejam executados.

Conforme explicado na *Lição 12 – Juntando tabelas*, os bancos de dados relacionais são projetados para que os dados sejam armazenados em várias tabelas para facilitar a manipulação, o gerenciamento e a reutilização de dados. Sem entrar em detalhes sobre o como e o porquê do projeto de banco de dados relacional, considere que os esquemas de banco de dados bem projetados são relacionais até certo ponto.

As tabelas `Orders` que você tem usado nas últimas 19 lições são um bom exemplo disso. Os pedidos são armazenados em duas tabelas: A tabela `Orders` armazena os pedidos propriamente ditos e a tabela `OrderItems` armazena os itens individuais pedidos. Essas duas tabelas estão relacionadas entre si usando IDs exclusivas chamadas chaves primárias (conforme discutido na *Lição 1 – Entendendo SQL*). Essas tabelas, por sua vez, estão relacionadas a outras tabelas que contêm as informações do cliente e de produto.

O processo de adicionar um pedido ao sistema é o seguinte:

1. Verifique se o cliente já está no banco de dados. Caso contrário, adicione-o.
2. Obtenha a ID do cliente.
3. Adicione uma linha à tabela `Orders` associando-a à ID do cliente.
4. Obtenha a nova ID do pedido atribuído na tabela `Orders`.

5. Adicione uma linha à tabela `OrderItems` para cada item pedido, associando-a à tabela `Orders` pela ID obtida (e à tabela `Products` pela ID do produto).

Agora imagine que alguma falha do banco de dados (por exemplo, espaço em disco insuficiente, restrições de segurança, bloqueios de tabela) impeça que toda a sequência seja concluída. O que aconteceria com seus dados?

Bem, se a falha ocorreu depois que o cliente foi adicionado e antes de a tabela `Orders` ser adicionada, não há problema real. É perfeitamente válido ter clientes sem pedidos. Ao executar a sequência novamente, o registro do cliente inserido será obtido e usado. Você pode efetivamente continuar de onde parou.

Mas e se a falha ocorreu depois que a linha foi adicionada à tabela `Orders`, mas antes que as linhas fossem adicionadas à tabela `OrderItems`? Agora você teria um pedido vazio em seu banco de dados.

Pior, e se o sistema falhou durante a adição das linhas à tabela `OrderItems`? Agora você acabaria tendo um pedido parcial em seu banco de dados, mas não saberia disso.

Como você resolve esse problema? É aí que entra o processamento da transação. *Processamento de transações* é um mecanismo usado para gerenciar conjuntos de operações SQL que devem ser executadas em lote para garantir que os bancos de dados nunca contenham os resultados de operações parciais. Com o processamento de transações, você pode garantir que conjuntos de operações não sejam interrompidos durante o processamento – eles são executados em sua totalidade ou não são executados (a menos que seja explicitamente instruído de outra forma). Se nenhum erro ocorrer, todo o conjunto de instruções é salvo (gravado) nas tabelas do banco de dados. Se ocorrer um erro, então pode ocorrer uma reversão (desfazer) para restaurar o banco de dados a um estado conhecido e seguro.

Então, se olharmos para o mesmo exemplo, é assim que o processo funcionaria:

1. Verifique se o cliente já está no banco de dados; se não, adicione-o.
2. Confirme as informações do cliente.
3. Obtenha a ID do cliente.
4. Adicione uma linha à tabela `Orders`.
5. Se ocorrer uma falha ao adicionar a linha à tabela `Orders`, reverta a operação.
6. Obtenha a nova ID do pedido atribuído na tabela `Orders`.
7. Adicione uma linha à tabela `OrderItems` para cada item pedido.
8. Se ocorrer uma falha ao adicionar linhas à tabela `OrderItems`, reverta todas as linhas adicionadas à tabela `OrderItems` e a linha da tabela `Orders`.

Quando você está trabalhando com transações e processamento de transações, algumas palavras-chave continuam reaparecendo. Aqui estão os termos que você precisa conhecer:

- **Transação** – Um bloco de instruções SQL
- **Rollback** – O processo de desfazer instruções SQL especificadas
- **Commit** – Gravar instruções SQL não salvas nas tabelas do banco de dados
- **Ponto de salvamento** – Um marcador temporário em um conjunto de transações para o qual você pode executar um rollback (em oposição a desfazer uma transação inteira)

DICA: Quais instruções você pode desfazer?

O processamento de transação é usado para gerenciar as instruções `INSERT`, `UPDATE` e `DELETE`. Você não pode desfazer instruções `SELECT`. (Não haveria muito sentido em fazer isso de qualquer maneira.) Você não pode desfazer as operações `CREATE` ou `DROP`. Essas instruções podem ser usadas em um bloco de transação, mas, se você executar um rollback, elas não serão desfeitas.

Controle de transações

Agora que você sabe o que é o processamento de transações, vamos examinar o que está envolvido no gerenciamento de transações.

CUIDADO: Diferenças de implementação

A sintaxe exata usada para implementar o processamento de transações difere de um SGBD para outro. Consulte a documentação de seu SGBD antes de continuar.

O importante para gerenciar transações envolve dividir suas instruções SQL em trechos lógicos e declarar explicitamente quando os dados devem ser revertidos e quando não.

Alguns SGBDs exigem que você marque explicitamente o início e o fim dos blocos de transação. No SQL Server, por exemplo, você pode fazer o seguinte (substituindo ... pelo código real):

Entrada ▼

```
BEGIN TRANSACTION
...
COMMIT TRANSACTION
```

Análise ▼

Neste exemplo, qualquer código SQL entre as instruções `BEGIN TRANSACTION` e `COMMIT TRANSACTION` deve ser executado inteiramente ou não deve ser executado.

O código equivalente para MariaDB e MySQL é

Entrada ▼

```
START TRANSACTION
...
```

O Oracle usa esta sintaxe:

Entrada ▼

```
SET TRANSACTION
...
```

O PostgreSQL usa a sintaxe do SQL padrão ANSI:

Entrada ▼

```
BEGIN
...
```

Outros SGBDs usam variações dos exemplos anteriores. Você notará que a maioria das implementações não tem um fim explícito de transação. Em vez disso, a transação existe até que algo a encerre, geralmente um `COMMIT` para salvar as alterações ou um `ROLLBACK` para desfazê-las, como será

explicado a seguir.

Usando ROLLBACK

O comando ROLLBACK do SQL é usado para reverter (desfazer) instruções SQL, conforme visto na próxima instrução:

Entrada ▼

```
DELETE FROM Orders;
ROLLBACK;
```

Análise ▼

Neste exemplo, uma operação DELETE é executada e então desfeita usando uma instrução ROLLBACK. Embora não seja o exemplo mais útil, ele demonstra que, dentro de um bloco de transação, as operações DELETE (como as operações INSERT e UPDATE) nunca são finais.

Usando COMMIT

Normalmente, as instruções SQL são executadas e gravadas diretamente nas tabelas do banco de dados. Isso é conhecido como *commit implícito* – a operação commit (gravar ou salvar) ocorre automaticamente.

Dentro de um bloco de transação, entretanto, as instruções commit podem não ocorrer implicitamente. Isso também é específico do SGBD. Alguns SGBDs tratam o final da transação como um commit implícito; outros não o fazem.

Para forçar um commit explícito, você deve usar a instrução COMMIT. O seguinte é um exemplo do SQL Server:

Entrada ▼

```
BEGIN TRANSACTION
DELETE OrderItems WHERE order_num = 12345
DELETE Orders WHERE order_num = 12345
COMMIT TRANSACTION
```

Análise ▼

Neste exemplo do SQL Server, o número de pedido 12345 é removido inteiramente do sistema. Como isso envolve atualizar duas tabelas do banco de dados, Orders e OrderItems, um bloco da transação é usado para

garantir que o pedido não seja suprimido parcialmente. A instrução COMMIT final só grava a mudança se não ocorreu nenhum erro. Se o primeiro DELETE funcionou, mas o segundo falhou, o DELETE não será salvo.

Para realizar a mesma coisa no Oracle, você pode fazer o seguinte:

Entrada ▼

```
SET TRANSACTION
DELETE OrderItems WHERE order_num = 12345;
DELETE Orders WHERE order_num = 12345;
COMMIT;
```

Usando pontos de salvamento

Instruções ROLLBACK e COMMIT simples permitem que você grave ou desfaça uma transação inteira. Embora essa abordagem funcione para transações simples, transações mais complexas podem exigir commits ou rollbacks parciais.

Por exemplo, o processo de adicionar um pedido descrito anteriormente é uma única transação. Se ocorrer um erro, você só deseja reverter para o ponto anterior à linha da tabela Orders ter sido adicionada. Você não deseja reverter a adição à tabela Customers (se houver).

Para suportar a reversão de transações parciais, você deve ser capaz de colocar marcadores em locais estratégicos no bloco de transação. Então, se um rollback for necessário, você pode reverter para um dos marcadores.

No SQL, esses marcadores são chamados de *pontos de salvamento*. Para criar um ponto de salvamento no MariaDB, MySQL e Oracle, você usa a instrução SAVEPOINT, da seguinte maneira:

Entrada ▼

```
SAVEPOINT delete1;
```

No SQL Server, você faz o seguinte:

Entrada ▼

```
SAVE TRANSACTION delete1;
```

Cada ponto de salvamento recebe um nome exclusivo que o identifica para que, quando você fizer o rollback, o SGBD saiba para onde você está

revertendo. Para reverter para esse ponto de salvamento, faça o seguinte no SQL Server:

Entrada ▼

```
ROLLBACK TRANSACTION delete1;
```

No MariaDB, MySQL e Oracle, você pode fazer o seguinte:

Entrada ▼

```
ROLLBACK TO delete1;
```

O seguinte é um exemplo completo para o SQL Server:

Entrada ▼

```
BEGIN TRANSACTION
INSERT INTO Customers(cust_id, cust_name)
VALUES(1000000010, 'Toys Emporium');
SAVE TRANSACTION StartOrder;
INSERT INTO Orders(order_num, order_date, cust_id)
VALUES(20100,'2020/12/1',1000000010);
IF @@ERROR <> 0 ROLLBACK TRANSACTION StartOrder;
INSERT INTO OrderItems(order_num, order_item,
➥prod_id, quantity, item_price)
VALUES(20100, 1, 'BR01', 100, 5.49);
IF @@ERROR <> 0 ROLLBACK TRANSACTION StartOrder;
INSERT INTO OrderItems(order_num, order_item,
➥prod_id, quantity, item_price)
VALUES(20100, 2, 'BR03', 100, 10.99);
IF @@ERROR <> 0 ROLLBACK TRANSACTION StartOrder;
COMMIT TRANSACTION
```

Análise ▼

Aqui, quatro instruções `INSERT` são colocadas dentro de um bloco de transação. Um ponto de salvamento é definido após o primeiro `INSERT` para que, se qualquer uma das operações `INSERT` subsequentes falhar, a transação só seja revertida até esse ponto. No SQL Server, uma variável chamada `@@ ERROR` pode ser inspecionada para ver se uma operação foi bem-sucedida. (Outros SGBDs usam funções ou variáveis diferentes para retornar essa informação.) Se `@@ ERROR` retornar um valor diferente de `0`, ocorreu um erro e a transação será revertida para o ponto de salvamento. Se toda a transação for processada, um `COMMIT` será emitido para salvar os

dados.

DICA: Quanto mais pontos de salvamento, melhor

Você pode ter quantos pontos de salvamento desejar em seu código SQL, e quanto mais, melhor. Por quê? Porque quanto mais pontos de salvamento você tiver, mais flexibilidade você terá no gerenciamento de rollbacks exatamente conforme você precisar deles.

Resumo

Nesta lição, você aprendeu que as transações são blocos de instruções SQL que devem ser executadas em lote. Aprendeu que as instruções `COMMIT` e `ROLLBACK` são usadas para gerenciar explicitamente quando os dados são gravados e quando as operações são desfeitas. Também aprendeu que os pontos de salvamento fornecem um nível maior de controle sobre as operações de rollback. O processamento de transações é um tópico muito importante e está muito além do escopo de uma lição. Além disso, como você viu aqui, o processamento de transações é implementado de maneira diferente em cada SGBD. Dessa forma, você deve consultar a documentação de seu SGBD para obter mais detalhes.

LIÇÃO 21

Usando cursores

Nesta lição, você será apresentado aos cursores e como (e por que) usá-los.

Compreendendo cursores

As operações de obtenção de dados do SQL funcionam com conjuntos de linhas conhecidos como conjuntos de resultados. As linhas retornadas são todas as linhas que correspondem a uma instrução SQL – zero ou mais delas. Quando você usa instruções SELECT simples, não há como obter a primeira linha, a próxima linha ou as 10 linhas anteriores. Essa é a essência do funcionamento de um SGBD relacional.

NOVO TERMO: Conjunto de resultados

Os resultados obtidos por uma consulta SQL. }

Às vezes, você precisa avançar ou retroceder pelas linhas e uma ou mais de cada vez. É para isso que os cursores são usados. Um cursor é uma consulta de banco de dados armazenada no servidor SGBD – não uma instrução SELECT, mas o conjunto de resultados obtidos por essa instrução. Depois que o cursor é armazenado, as aplicações podem percorrer ou navegar para cima e para baixo nos dados conforme necessário.

NOTA: Suporte no SQLite

O SQLite oferece suporte a uma forma de cursores chamados passos. Os conceitos básicos descritos nesta lição se aplicam aos passos do SQLite, mas a sintaxe pode ser bem diferente.

Diferentes SGBDs oferecem suporte a diferentes opções e recursos de cursor. Alguns dos mais comuns são

- A capacidade de sinalizar um cursor como somente leitura para que os dados possam ser lidos, mas não atualizados ou excluídos
- A capacidade de controlar as operações direcionais que podem ser realizadas (para a frente, para trás, primeiro, último, posição absoluta,

posição relativa e assim por diante)

- A capacidade de sinalizar algumas colunas como editáveis e outras como não editáveis
- Especificação de escopo de forma a tornar o cursor acessível à solicitação específica que o criou (uma stored procedure, por exemplo) ou a todas as solicitações
- Instruir o SGBD a fazer uma cópia dos dados obtidos (em vez de apontar para os dados ativos na tabela) para que os dados não mudem entre o momento em que o cursor é aberto e o momento em que é acessado

Os cursores são usados principalmente por aplicações interativas nas quais os usuários precisam rolar para cima e para baixo nas telas de dados, pesquisando ou fazendo alterações.

Trabalhando com cursores

O uso de cursores envolve várias etapas distintas:

- Antes que um cursor seja usado, ele deve ser declarado (definido). Na verdade, esse processo não obtém nenhum dado, ele apenas define a instrução SELECT a ser usada e quaisquer opções de cursor.
- Depois de declarado, o cursor deve ser aberto para uso. Na verdade, esse processo obtém os dados usando a instrução SELECT definida anteriormente.
- Com o cursor preenchido com dados, linhas individuais podem ser buscadas (obtidas) conforme necessário.
- Quando terminar, o cursor deve ser fechado e possivelmente desalocado (dependendo do SGBD).

Uma vez que um cursor é declarado, ele pode ser aberto e fechado tão frequentemente quanto necessário. Uma vez aberto, as operações de busca podem ser executadas tão frequentemente quanto necessário.

Criando cursores

Os cursores são criados usando a instrução DECLARE, que difere de um

SGBD para o outro. DECLARE nomeia o cursor e recebe uma instrução SELECT completa com WHERE e outras cláusulas se necessário. Para demonstrar isso, criaremos um cursor que retorna todos os clientes sem endereços de e-mail, como parte de uma aplicação que permite a um operador fornecer endereços de e-mail ausentes.

Aqui está a versão para DB2, MariaDB, MySQL e SQL Server:

Entrada ▼

```
DECLARE CustCursor CURSOR
FOR
SELECT * FROM Customers
WHERE cust_email IS NULL;
```

Aqui está a versão para Oracle e PostgreSQL:

Entrada ▼

```
DECLARE CURSOR CustCursor
IS
SELECT * FROM Customers
WHERE cust_email IS NULL;
```

Análise ▼

Em ambas as versões, a instrução DECLARE é usada para definir e nomear o cursor – neste caso, CustCursor. A instrução SELECT define um cursor contendo todos os clientes sem endereço de e-mail (um valor NULL).

Agora que o cursor está definido, ele está pronto para ser aberto.

Usando cursores

Os cursores são abertos usando a instrução OPEN CURSOR, que é uma instrução tão simples que a maioria dos SGBDs suporta exatamente a mesma sintaxe:

Entrada ▼

```
OPEN CURSOR CustCursor
```

Análise ▼

Quando a instrução OPEN CURSOR é processada, a consulta é executada e os dados obtidos são armazenados para pesquisa e rolagem subsequentes.

Agora os dados do cursor podem ser acessados usando a instrução FETCH. FETCH especifica as linhas a serem obtidas, de onde elas serão obtidas e onde elas serão armazenadas (nomes de variáveis, por exemplo). O primeiro exemplo usa a sintaxe do Oracle para obter uma única linha do cursor (a primeira linha):

Entrada ▼

```
DECLARE TYPE CustCursor IS REF CURSOR
    RETURN Customers%ROWTYPE;
DECLARE CustRecord Customers%ROWTYPE
BEGIN
    OPEN CustCursor;
    FETCH CustCursor INTO CustRecord;
    CLOSE CustCursor;
END;
```

Análise ▼

Neste exemplo, FETCH é usado para obter a linha atual (ele começará na primeira linha automaticamente) e colocar em uma variável declarada chamada CustRecord. Nada é feito com os dados obtidos.

No próximo exemplo (novamente, usando a sintaxe do Oracle), os dados obtidos são repetidos da primeira linha à última:

Entrada ▼

```
DECLARE TYPE CustCursor IS REF CURSOR
    RETURN Customers%ROWTYPE;
DECLARE CustRecord Customers%ROWTYPE
BEGIN
    OPEN CustCursor;
    LOOP
    FETCH CustCursor INTO CustRecord;
    EXIT WHEN CustCursor%NOTFOUND;
       ...
    END LOOP;
    CLOSE CustCursor;
END;
```

Análise ▼

Como o exemplo anterior, este exemplo usa FETCH para obter a linha atual

e colocá-la em uma variável declarada chamada `CustRecord`. Ao contrário do exemplo anterior, a instrução `FETCH` aqui está dentro de um `LOOP` para que seja repetida várias vezes. O código `EXIT WHEN CustCursor% NOTFOUND` faz com que o processamento seja encerrado (saindo do loop) quando não houver mais linhas a serem buscadas. Este exemplo também não faz nenhum processamento real; no código do mundo real, você substituiria o marcador ... pelo seu próprio código.

Aqui está outro exemplo, desta vez usando a sintaxe do Microsoft SQL Server:

Entrada ▼

```
DECLARE @cust_id CHAR(10),
        @cust_name CHAR(50),
        @cust_address CHAR(50),
        @cust_city CHAR(50),
        @cust_state CHAR(5),
        @cust_zip CHAR(10),
        @cust_country CHAR(50),
        @cust_contact CHAR(50),
        @cust_email CHAR(255)
OPEN CustCursor
FETCH NEXT FROM CustCursor
    INTO @cust_id, @cust_name, @cust_address,
         @cust_city, @cust_state, @cust_zip,
         @cust_country, @cust_contact, @cust_email
   ...
WHILE @@FETCH_STATUS = 0
BEGIN

FETCH NEXT FROM CustCursor
        INTO @cust_id, @cust_name, @cust_address,
             @cust_city, @cust_state, @cust_zip,
             @cust_country, @cust_contact, @cust_email
...
END
CLOSE CustCursor
```

Análise ▼

Neste exemplo, as variáveis são declaradas para cada uma das colunas obtidas e as instruções `FETCH` obtêm uma linha e salvam os valores nessas

variáveis. Um loop `WHILE` é usado para percorrer as linhas, e a condição `WHILE @@ FETCH_STATUS = 0` faz com que o processamento seja encerrado (saindo do loop) quando não houver mais linhas a serem buscadas. Novamente este exemplo também não faz nenhum processamento real; no código do mundo real, você substituiria o marcador ... pelo seu próprio código.

Fechando cursores

Conforme já mencionado e visto nos exemplos anteriores, os cursores precisam ser fechados após serem usados. Além disso, alguns SGBDs (como o SQL Server) exigem que os recursos usados pelo cursor sejam desalocados explicitamente. Esta é a sintaxe do DB2, Oracle e PostgreSQL:

Entrada ▼

```
CLOSE CustCursor
```

Esta é a versão para o Microsoft SQL Server:

Entrada ▼

```
CLOSE CustCursor
DEALLOCATE CURSOR CustCursor
```

Análise ▼

A instrução `CLOSE` é usada para fechar cursores; uma vez que o cursor é fechado, ele não pode ser reutilizado sem ser aberto novamente. No entanto, um cursor não precisa ser declarado novamente para ser usado; uma instrução `OPEN` é suficiente.

Resumo

Nesta lição, você foi apresentado aos cursores, o que são e por que são usados. Seu próprio SGBD provavelmente oferece alguma forma dessa função, assim como outras não mencionadas aqui. Consulte a documentação de seu SGBD para obter mais detalhes.

LIÇÃO 22

Compreendendo os recursos SQL avançados

Nesta lição, você verá vários dos recursos avançados de manipulação de dados que evoluíram com o SQL: restrições, índices e triggers.

Compreendendo as restrições

O SQL evoluiu através de muitas versões para se tornar uma linguagem muito completa e poderosa. Muitos dos recursos mais poderosos são ferramentas sofisticadas que fornecem técnicas de manipulação de dados, como *restrições*.

Tabelas relacionais e integridade referencial foram ambas discutidas várias vezes em lições anteriores. Conforme expliquei nessas lições, os bancos de dados relacionais armazenam dados divididos em várias tabelas, cada uma delas armazenando dados relacionados. As chaves são usadas para criar referências de uma tabela para outra (daí o termo *integridade referencial*).

Para que os projetos de banco de dados relacional funcionem corretamente, você precisa de uma maneira de garantir que apenas dados válidos sejam inseridos nas tabelas. Por exemplo, se a tabela `Orders` armazenar informações do pedido e `OrderItems` armazenar detalhes do pedido, você deve garantir que quaisquer IDs de pedido referenciados em `OrderItems` existam em `Orders`. Da mesma forma, todos os clientes mencionados em `Orders` devem estar na tabela `Customers`.

Embora você possa realizar verificações antes de inserir novas linhas (faça um `SELECT` em outra tabela para se certificar de que os valores são válidos e presentes), é melhor evitar essa prática pelos seguintes motivos:

- Se as regras de integridade do banco de dados forem aplicadas no nível do cliente, cada cliente é obrigado a aplicar essas regras e, inevitavelmente, alguns clientes não o farão.

- Você também deve aplicar as regras nas operações UPDATE e DELETE.
- Executar verificações do lado do cliente é um processo demorado. Fazer com que o SGBD execute as verificações para você é muito mais eficiente.

NOVO TERMO: Restrições (constraints)

Regras que governam como os dados do banco de dados são inseridos ou manipulados.

Os SGBDs fazem valer a integridade referencial, impondo restrições às tabelas do banco de dados. A maioria das restrições é definida nas definições de tabela (usando CREATE TABLE ou ALTER TABLE conforme discutido na *Lição 17 – Criando e manipulando tabelas*).

CUIDADO: As restrições são específicas do SGBD

Existem vários tipos diferentes de restrições e cada SGBD fornece seu próprio nível de suporte para elas. Portanto, os exemplos mostrados aqui podem não funcionar como você os vê. Consulte a documentação de seu SGBD antes de continuar.

Chaves primárias

A *Lição 1 – Entendendo SQL*, discutiu brevemente as chaves primárias. Uma *chave primária* é uma restrição especial usada para garantir que os valores em uma coluna (ou conjunto de colunas) sejam únicos e nunca mudem – em outras palavras, uma coluna (ou colunas) em uma tabela cujos valores identificam exclusivamente cada linha na tabela. Isso facilita a manipulação direta e a interação com linhas individuais. Sem as chaves primárias, seria difícil usar com segurança UPDATE ou DELETE em linhas específicas sem afetar outras linhas.

Qualquer coluna em uma tabela pode ser estabelecida como a chave primária, desde que atenda às seguintes condições:

- Duas linhas não podem ter o mesmo valor de chave primária.
- Cada linha deve ter um valor de chave primária. (As colunas não devem permitir valores NULL.)
- A coluna que contém os valores da chave primária nunca pode ser modificada ou atualizada. (A maioria dos SGBDs não permite isso, mas, se o seu permitir, bem, não o faça!)
- Os valores da chave primária nunca podem ser reutilizados. Se uma

linha for excluída da tabela, sua chave primária não deve ser atribuída a nenhuma nova linha.

Uma maneira de definir as chaves primárias é criá-las da seguinte maneira:

Entrada q

```
CREATE TABLE Vendors
(
    vend_id        CHAR(10)    NOT NULL PRIMARY KEY,
    vend_name      CHAR(50)    NOT NULL,
    vend_address   CHAR(50)    NULL,
    vend_city      CHAR(50)    NULL,
    vend_state     CHAR(5)     NULL,
    vend_zip       CHAR(10)    NULL,
    vend_country   CHAR(50)    NULL
);
```

Análise q

No exemplo anterior, a palavra-chave `PRIMARY KEY` é adicionada à definição da tabela para que `vend_id` se torne a chave primária.

Entrada q

```
ALTER TABLE Vendors
ADD CONSTRAINT PRIMARY KEY (vend_id);
```

Análise q

Aqui, a mesma coluna é definida como a chave primária, mas a sintaxe `CONSTRAINT` é usada em seu lugar. Essa sintaxe pode ser usada nas instruções `CREATE TABLE` e `ALTER TABLE`.

> **NOTA: Chaves no SQLite**
>
> O SQLite não permite que as chaves sejam definidas usando `ALTER TABLE` e requer que elas sejam definidas como parte do `CREATE TABLE` inicial.

Chaves estrangeiras

Uma chave estrangeira é uma coluna em uma tabela cujos valores devem ser listados em uma chave primária em outra tabela. As chaves estrangeiras são uma parte extremamente importante para garantir a integridade referencial. Para entender as chaves estrangeiras, vejamos um

exemplo.

A tabela `Orders` contém uma única linha para cada pedido inserido no sistema. As informações reais do cliente são armazenadas na tabela `Customers`. Os pedidos na tabela `Orders` estão vinculados a linhas específicas na tabela `Customers` pela ID do cliente. A ID do cliente é a chave primária na tabela `Customers`; cada cliente possui uma ID única. O número do pedido é a chave primária na tabela `Orders`; cada pedido possui um número único.

Os valores na coluna de ID do cliente na tabela `Orders` não são necessariamente exclusivos. Se um cliente tiver vários pedidos, haverá várias linhas com a mesma ID de cliente (embora cada uma tenha um número de pedido diferente). Ao mesmo tempo, os únicos valores válidos na coluna ID do cliente em `Orders` são as IDs de clientes na tabela `Customers`.

Isso é o que uma chave estrangeira faz. Em nosso exemplo, uma chave estrangeira é definida na coluna de ID do cliente em `Orders` para que a coluna possa aceitar apenas valores que estão na chave primária da tabela `Customers`.

Aqui está uma maneira de definir essa chave estrangeira:

Entrada q

```
CREATE TABLE Orders
(
    order_num    INTEGER    NOT NULL PRIMARY KEY,
    order_date   DATETIME   NOT NULL,
    cust_id      CHAR(10)   NOT NULL REFERENCES Customers(cust_id)
);
```

Análise q

Aqui, a definição da tabela usa a palavra-chave `REFERENCES` para afirmar que quaisquer valores em `cust_id` devem estar em `cust_id` na tabela `Customers`.

O mesmo resultado pode ser obtido usando a sintaxe `CONSTRAINT` em uma instrução `ALTER TABLE`:

Entrada q

```
ALTER TABLE Orders
ADD CONSTRAINT
```

```
FOREIGN KEY (cust_id) REFERENCES Customers (cust_id);
```

DICA: Chaves estrangeiras podem ajudar a prevenir exclusão acidental

Conforme observado na *Lição 16 — Atualizando e excluindo dados*, além de ajudar a impor a integridade referencial, as chaves estrangeiras servem a outro propósito inestimável. Depois que uma chave estrangeira é definida, seu SGBD não permite a exclusão de linhas que possuem linhas relacionadas em outras tabelas. Por exemplo, você não tem permissão para excluir um cliente que tenha pedidos associados. A única maneira de excluir esse cliente é primeiro excluir os pedidos relacionados (o que, por sua vez, significa excluir os itens do pedido relacionados). Por exigirem essa exclusão metódica, as chaves estrangeiras podem ajudar a evitar a exclusão acidental de dados.

No entanto, alguns SGBDs suportam um recurso chamado *exclusão em cascata*. Se ativado, esse recurso exclui todos os dados relacionados quando uma linha é excluída de uma tabela. Por exemplo, se a exclusão em cascata estiver ativada e um cliente for excluído da tabela `Customers`, todas as linhas de pedido relacionadas serão excluídas automaticamente.

Restrições exclusivas

As restrições exclusivas são usadas para garantir que todos os dados em uma coluna (ou um conjunto de colunas) sejam únicos. Elas são semelhantes às chaves preliminares, mas há algumas diferenças importantes:

- A tabela pode conter várias restrições exclusivas, mas somente uma chave primária é permitida por tabela.
- Colunas com restrição exclusiva podem conter valores `NULL`.
- Colunas com restrição exclusiva podem ser modificadas ou atualizadas.
- Valores de coluna com restrição exclusiva podem ser reutilizados.
- Ao contrário das chaves primárias, restrições exclusivas não podem ser usadas para definir chaves estrangeiras.

Um exemplo do uso de restrições é uma tabela `Employees`. Cada funcionário tem um número de Seguro Social exclusivo, mas você não gostaria de usá-lo para a chave primária porque ele é muito longo (além do fato de que você pode não querer que essa informação esteja facilmente disponível). Portanto, cada funcionário também possui uma ID de funcionário exclusiva (uma chave primária), além de um número de Seguro Social.

Como a ID do funcionário é uma chave primária, você pode ter certeza de que ela é exclusiva. Você também pode querer que o SGBD garanta que cada número do Seguro Social também seja único (para garantir que um erro de digitação não resulte no uso do número de outra pessoa). Você pode fazer isso definindo uma restrição `UNIQUE` (exclusiva) na coluna do número do Seguro Social.

A sintaxe para restrições exclusivas é semelhante à de outras restrições. A palavra-chave `UNIQUE` está definida na definição da tabela ou uma palavra-chave `CONSTRAINT` separada é usada.

Restrições de verificação

As restrições de verificação são usadas para garantir que os dados em uma coluna (ou conjunto de colunas) atendam a um conjunto de critérios que você especificar. Os usos comuns disso são

- **Verificar valores mínimos ou máximos** – Por exemplo, evitando um pedido de `0` (zero) itens (mesmo que 0 seja um número válido).
- **Especificar intervalos** – Por exemplo, certificando-se de que uma data de envio é maior ou igual à data de hoje e não maior que um ano a partir de agora.
- **Permitir apenas valores específicos** – Por exemplo, permitindo apenas `M` ou `F` em um campo de gênero.

Em outras palavras, os datatypes (discutidos na *Lição 1 – Entendendo SQL*) restringem o tipo de dados que podem ser armazenados em uma coluna. As restrições de verificação impõem outras restrições nesse datatype, e elas podem ser inestimáveis para garantir que os dados inseridos em seu banco de dados sejam exatamente os que você deseja. Em vez de depender de aplicações clientes ou dos usuários para fazer isso direito, o próprio SGBD rejeitará qualquer coisa que seja inválida.

O exemplo a seguir aplica uma restrição de verificação à tabela `OrderItems` para garantir que todos os itens tenham uma quantidade maior que `0`:

Entrada q

```
CREATE TABLE OrderItems
(
```

```
    order_num     INTEGER    NOT NULL,
    order_item    INTEGER    NOT NULL,
    prod_id       CHAR(10)   NOT NULL,
    quantity      INTEGER    NOT NULL CHECK (quantity > 0),
    item_price    MONEY      NOT NULL
);
```

Análise q

Com esta restrição em vigor, qualquer linha inserida (ou atualizada) será verificada para garantir que a quantidade é maior que `0`.

Para verificar se uma coluna chamada `gender` contém apenas `M` ou `F`, você pode fazer o seguinte em uma instrução `ALTER TABLE`:

Entrada q

```
ADD CONSTRAINT CHECK (gender LIKE '[MF]');
```

DICA: Datatypes definidos pelo usuário

Alguns SGBDs permitem que você defina seus próprios datatypes. Esses são datatypes essencialmente simples com restrições de verificação (ou outras restrições) definidas. Por exemplo, você pode definir seu próprio datatype chamado `gender`, que é um datatype de texto de um único caractere com uma restrição de verificação que restringe seus valores a `M` ou `F` (e talvez `NULL` para Desconhecido). Você pode então usar esse datatype nas definições da tabela. A vantagem dos datatypes personalizados é que as restrições precisam ser aplicadas apenas uma vez (na definição do datatype) e são aplicadas automaticamente cada vez que o datatype é usado. Verifique a documentação de seu SGBD para determinar se datatypes definidos pelo usuário são suportados.

Compreendendo os índices

Os índices são usados para classificar dados logicamente para melhorar a velocidade das operações de pesquisa e classificação. A melhor maneira de entender os índices é visualizá-los no final de um livro (este livro, por exemplo).

Suponha que você queira encontrar todas as ocorrências da palavra *datatype* neste livro. A maneira simples de fazer isso seria ir para a página 1 e examinar cada linha de cada página em busca de correspondências. Embora isso funcione, obviamente não é uma solução viável. Examinar algumas páginas de texto pode ser viável, mas examinar um livro inteiro dessa maneira não. À medida que a quantidade de texto a ser pesquisado

aumenta, também aumenta o tempo que leva para localizar os dados desejados.

É por isso que os livros têm índices. Um índice é uma lista alfabética de palavras com referências a suas localizações no livro. Para pesquisar *datatype*, você encontra essa palavra no índice para determinar em quais páginas ela aparece. Então, você acessa essas páginas específicas para encontrar suas correspondências.

O que faz um índice funcionar? Simplesmente, é o fato de que ele está classificado corretamente. A dificuldade em encontrar palavras em um livro não é a quantidade de conteúdo que deve ser pesquisado, mas o fato de que o conteúdo não é classificado por palavra. Se o conteúdo for classificado como um dicionário, um índice não é necessário (é por isso que os dicionários não têm índices).

Os índices de banco de dados funcionam da mesma maneira. Os dados da chave primária são sempre classificados; isso é algo que o SGBD faz por você. Obter linhas específicas por chave primária, portanto, é sempre uma operação rápida e eficiente.

No entanto, pesquisar valores em outras colunas geralmente não é tão eficiente. Por exemplo, e se você quiser obter todos os clientes que moram em um estado específico? Como a tabela não é classificada por estado, o SGBD deve ler todas as linhas da tabela (começando na primeira linha) em busca de correspondências, assim como você faria se estivesse tentando encontrar palavras em um livro sem usar um índice.

A solução é usar um índice. Você pode definir um índice em uma ou mais colunas de modo que o SGBD mantenha uma lista classificada do conteúdo para seu próprio uso. Depois que um índice é definido, o SGBD o usa da mesma maneira que você usaria um índice de um livro. Ele pesquisa o índice classificado para encontrar a localização de quaisquer correspondências e, em seguida, obtém essas linhas específicas.

Mas, antes de sair criando dezenas de índices, tenha em mente o seguinte:

- Índices melhoram o desempenho das operações de obtenção, mas degradam o desempenho da inserção, modificação e exclusão de

dados. Quando essas operações são executadas, o SGBD deve atualizar o índice dinamicamente.

- Dados de índice podem ocupar muito espaço de armazenamento.
- Nem todos os dados são adequados para indexação. Dados que não são suficientemente exclusivos (estado, por exemplo) não se beneficiarão tanto da indexação quanto os dados que têm mais valores possíveis (nome ou sobrenome, por exemplo).
- Índices são usados para filtragem de dados e classificação de dados. Se você frequentemente classifica os dados em uma ordem específica, esses dados podem ser adequados para indexação.
- Várias colunas podem ser definidas em um índice (por exemplo, estado mais cidade). Esse índice será útil apenas quando os dados forem classificados na ordem "estado mais cidade". (Se você quiser classificar por cidade, esse índice não terá nenhuma utilidade.)

Não existe uma regra rígida quanto ao que deve ser indexado e quando. A maioria dos SGBDs fornece utilitários que você pode usar para determinar a eficácia dos índices e você deve usá-los regularmente.

Os índices são criados com a instrução `CREATE INDEX` (que varia drasticamente de um SGBD para outro). A instrução a seguir cria um índice simples na coluna de nome do produto da tabela `Products`:

Entrada q

```
CREATE INDEX prod_name_ind
ON Products (prod_name);
```

Análise q

Cada índice deve ter um nome exclusivo. Aqui, o nome `prod_name_ind` é definido após as palavras-chave `CREATE INDEX`. `ON` é usado para especificar a tabela que está sendo indexada e as colunas a serem incluídas no índice (apenas uma neste exemplo) são especificadas entre parênteses após o nome da tabela.

DICA: Revisitando índices

A eficácia do índice muda conforme os dados da tabela são adicionados ou alterados. Muitos administradores de banco de dados descobrem que o que antes era um conjunto

ideal de índices pode não ser tão ideal após vários meses de manipulação de dados. É sempre uma boa ideia revisitar os índices regularmente para ajustá-los conforme necessário.

Compreendendo os triggers

Triggers são stored procedures especiais executadas automaticamente quando ocorre uma atividade específica do banco de dados. Os triggers podem estar associados às operações `INSERT`, `UPDATE` e `DELETE` (ou qualquer combinação delas) em tabelas específicas.

Ao contrário das stored procedures (que são simplesmente instruções SQL armazenadas), os triggers são vinculados a tabelas individuais. Um trigger associado às operações `INSERT` na tabela `Orders` será executado somente quando uma linha for inserida na tabela `Orders`. Da mesma forma, um trigger nas operações `INSERT` e `UPDATE` na tabela `Customers` será executado apenas quando essas operações específicas ocorrerem nessa tabela.

Dentro dos triggers, seu código tem acesso ao seguinte:

- Todos os novos dados em operações `INSERT`.
- Todos os novos dados e dados antigos em operações `UPDATE`.
- Dados excluídos em operações `DELETE`.

Dependendo do SGBD usado, os triggers podem ser executados antes ou depois de uma operação especificada ser executada.

A seguir estão alguns usos comuns para triggers:

- Garantir a consistência dos dados; por exemplo, converter todos os nomes de estado em maiúsculas durante uma operação `INSERT` ou `UPDATE`.
- Executar ações em outras tabelas com base nas alterações em uma tabela; por exemplo, gravar um registro de rastreamento de auditoria em uma tabela de log cada vez que uma linha é atualizada ou excluída.
- Executar validação adicional e reversão de dados, se necessário; por exemplo, certificar-se de que o crédito disponível de um cliente não foi excedido e bloquear a inserção, caso isso tenha ocorrido.

- Calcular valores de coluna computados ou atualizar timestamps.

Como você provavelmente já deve esperar, a sintaxe de criação de trigger varia drasticamente de um SGBD para outro. Verifique sua documentação para mais detalhes.

O exemplo a seguir cria um trigger que converte a coluna `cust_state` na tabela `Customers` em maiúsculas em todas as operações `INSERT` e `UPDATE`.

Esta é a versão do SQL Server:

Entrada q

```
CREATE TRIGGER customer_state
ON Customers
FOR INSERT, UPDATE
AS
UPDATE Customers
SET cust_state = Upper(cust_state)
WHERE Customers.cust_id = inserted.cust_id;
```

Esta é a versão para Oracle e PostgreSQL:

Entrada q

```
CREATE TRIGGER customer_state
AFTER INSERT OR UPDATE
FOR EACH ROW
BEGIN
UPDATE Customers
SET cust_state = Upper(cust_state)
WHERE Customers.cust_id = :OLD.cust_id
END;
```

DICA: As restrições são mais rápidas do que os triggers

Como regra, as restrições são processadas mais rapidamente do que os triggers, portanto, sempre que possível, use as restrições.

Segurança de banco de dados

Não há nada mais valioso para uma organização do que seus dados, e os dados devem sempre ser protegidos de possíveis ladrões ou bisbilhoteiros casuais. Obviamente, ao mesmo tempo, os dados devem ser acessíveis aos usuários que precisam acessá-los, e assim a maioria dos DBMSs fornece aos administradores mecanismos para conceder ou restringir o acesso aos

dados.

A base de qualquer sistema de segurança é a autorização e a autenticação do usuário. Esse é o processo pelo qual um usuário é validado para garantir que ele é quem diz ser e que tem permissão para realizar a operação que está tentando realizar. Alguns SGBDs se integram à segurança do sistema operacional para isso, outros mantêm suas próprias listas de usuários e senhas e ainda há outros que se integram a servidores de serviços de diretório externos.

Aqui estão algumas operações que costumam ser protegidas:

- Acesso a recursos de administração de banco de dados (criação de tabelas, alteração ou eliminação de tabelas existentes e assim por diante).
- Acesso a bancos de dados ou tabelas específicas.
- O tipo de acesso (somente leitura, acesso a colunas específicas e assim por diante).
- Acesso a tabelas por meio de visualizações ou stored procedures apenas.
- Criação de vários níveis de segurança, permitindo assim vários graus de acesso e controle com base no login.
- Restrições na capacidade de gerenciar contas de usuário.

A segurança é gerenciada por meio das instruções `GRANT` e `REVOKE`, embora a maioria dos SGBDs forneça utilitários de administração interativos que usam as instruções `GRANT` e `REVOKE` internamente.

Resumo

Nesta lição, você aprendeu como usar alguns recursos SQL avançados. As restrições são uma parte importante para garantir a integridade referencial; os índices podem melhorar o desempenho da obtenção de dados; triggers podem ser usados para realizar o processamento antes ou depois da execução; e opções de segurança podem ser usadas para gerenciar o acesso aos dados. Seu próprio SGBD provavelmente oferece alguma forma desses recursos. Consulte a documentação de seu SGBD

para obter mais detalhes.

APÊNDICE A

Exemplos de scripts de tabela

Escrever instruções SQL requer um bom entendimento do projeto do banco de dados subjacente. Se você não sabe quais informações estão armazenadas em que tabela, como as tabelas estão relacionadas entre si e a real divisão de dados dentro de uma linha, é impossível escrever código SQL eficaz.

Recomenda-se fortemente que você de fato teste todos os exemplos em cada lição deste livro. Todas as lições usam um conjunto comum de arquivos de dados. Para ajudá-lo a entender melhor os exemplos e permitir que você acompanhe as lições, este apêndice descreve as tabelas usadas, seus relacionamentos e como construí-las (ou obtê-las)[1].

Entendendo os exemplos de tabelas

As tabelas utilizadas ao longo deste livro fazem parte de um sistema de entrada de pedidos usado por um distribuidor imaginário de brinquedos. As tabelas são usadas para executar várias tarefas:

- Gerenciar fornecedores
- Gerenciar catálogos de produtos
- Gerenciar listas de clientes
- Inserir pedidos de clientes

Fazer tudo isso funcionar requer cinco tabelas (que estão intimamente interconectadas como parte de um projeto de banco de dados relacional). Uma descrição de cada uma das tabelas aparece nas seguintes seções.

> **NOTA: Exemplos simplificados**
>
> As tabelas usadas aqui não estão completas. Um sistema de entrada de pedidos no mundo real teria de acompanhar muitos outros dados que não foram incluídos aqui (por exemplo, informações de pagamento e contabilidade, rastreamento de envios e muito mais). No entanto, essas tabelas demonstram os tipos de organização de dados

e relacionamentos que você encontrará na maioria das instalações reais. Você pode aplicar essas técnicas e tecnologias em seus próprios bancos de dados.

Descrições das tabelas

A seguir veremos uma descrição de cada uma das cinco tabelas, junto com o nome das colunas dentro de cada tabela e suas descrições.

Tabela Vendors

A tabela Vendors armazena os fornecedores cujos produtos são vendidos. Cada fornecedor tem um registro nesta tabela e essa ID do fornecedor (a coluna vend_id) é usada para estabelecer a correspondência entre produtos e fornecedores.

Tabela A.1 – Colunas da tabela Vendors

Coluna	Descrição
vend_id	ID única do fornecedor
vend_name	Nome do fornecedor
vend_address	Endereço do fornecedor
vend_city	Cidade do fornecedor
vend_state	Estado do fornecedor
vend_zip	CEP do fornecedor
vend_country	País do fornecedor

- Todas as tabelas devem ter chaves primárias definidas. Essa tabela deve usar vend_id como sua chave primária.

Tabela Products

A tabela Products contém o catálogo do produto, um produto por linha. Cada produto tem uma ID única (a coluna prod_id) e está relacionado a seu fornecedor por vend_id (a ID única do fornecedor).

Tabela A.2 – Colunas da tabela Products

Coluna	Descrição
prod_id	ID única do produto
vend_id	ID do fornecedor do produto (relacionada a vend_id na tabela Vendors)
prod_name	Nome do produto

Coluna	Descrição
prod_price	Preço do produto
prod_desc	Descrição do produto

- Todas as tabelas devem ter chaves primárias definidas. Essa tabela deve usar prod_id como sua chave primária.
- Para impor integridade referencial, uma chave estrangeira deve ser definida em vend_id relacionando-a a vend_id em VENDORS.

Tabela Customers

A tabela Customers armazena todas as informações do cliente. Cada cliente possui uma ID única (a coluna cust_id).

Tabela A.3 – Colunas da tabela Customers

Coluna	Descrição
cust_id	ID única do cliente
cust_name	Nome do cliente
cust_address	Endereço do cliente
cust_city	Cidade do cliente
cust_state	Estado do cliente
cust_zip	CEP do cliente
cust_country	País do cliente
cust_contact	Nome do contato do cliente
cust_email	Endereço de e-mail do contato do cliente

- Todas as tabelas devem ter chaves primárias definidas. Essa tabela deve usar cust_id como sua chave primária.

Tabela Orders

A tabela Orders armazena os pedidos do cliente (mas não os detalhes do pedido). Cada pedido é numerado exclusivamente (a coluna order_num). Os pedidos são associados aos clientes apropriados pela coluna cust_id (que se relaciona à ID exclusiva do cliente na tabela Customers).

Tabela A.4 – Colunas da tabela Orders

Coluna	Descrição

Coluna	Descrição
order_num	Número único do pedido
order_date	Data do pedido
cust_id	ID do cliente do pedido (se relaciona a cust_id na tabela Customers)

- Todas as tabelas devem ter chaves primárias definidas. Essa tabela deve usar order_num como sua chave primária.
- Para impor a integridade referencial, uma chave estrangeira deve ser definida em cust_id relacionando esse campo ao campo cust_id em CUSTOMERS.

Tabela OrderItems

A tabela OrderItems armazena os itens reais em cada pedido, uma linha por item por pedido. Para cada linha em Orders, há uma ou mais linhas em OrderItems. Cada item do pedido é identificado exclusivamente pelo número do pedido mais o item do pedido (primeiro item no pedido, segundo item no pedido e assim por diante). Os itens do pedido são associados ao seu pedido apropriado pela coluna order_num (que está relacionada à ID exclusiva do pedido em Orders). Além disso, cada item do pedido contém a ID do produto dos pedidos do item (que relaciona o item de volta à tabela Products).

Tabela A.5 – Colunas da tabela OrderItems

Coluna	Descrição
order_num	Número do pedido (relacionado a order_num na tabela Orders)
order_item	Número do item do pedido (sequencial dentro de um pedido)
prod_id	ID do produto (relacionada a prod_id na tabela Products)
quantity	Quantidade de itens
item_price	Preço do item

- Todas as tabelas devem ter chaves primárias definidas. Essa tabela deve usar order_num e order_item como suas chaves primárias.
- Para impor a integridade referencial, chaves estrangeiras devem ser definidas em order_num relacionando esse campo a order_num em Orders e prod_id relacionado a prod_id em Products.

Os administradores de banco de dados geralmente usam diagramas de

relacionamento para ajudar a demonstrar como as tabelas de banco de dados estão conectadas. Lembre-se de que são as chaves estrangeiras que definem esses relacionamentos, conforme observado nas descrições da tabela anterior. A Figura A.1 é o diagrama de relacionamento para as cinco tabelas descritas neste apêndice.

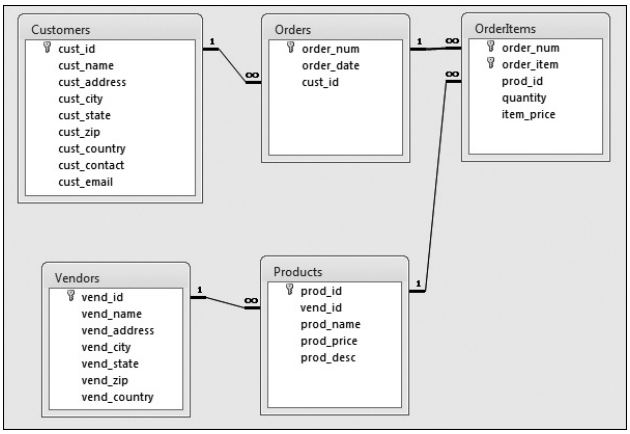

Figura A.1 – Diagrama de relacionamento dos exemplos de tabelas.

Obtendo exemplos de tabelas

Para acompanhar os exemplos, você precisa de um conjunto de tabelas preenchidas. Tudo o que você precisa para começar a trabalhar pode ser encontrado na página da web deste livro em http://forta.com/books/0135182794/.

Nessa página, você encontrará links para baixar scripts SQL para seu SGBD. Existem dois arquivos para cada um:

- create.txt contém as instruções SQL para criar as cinco tabelas de banco de dados (incluindo a definição de todas as chaves primárias e restrições de chave estrangeira).
- populate.txt contém as instruções INSERT usadas para preencher essas tabelas.

As instruções SQL nesses arquivos são muito específicas do SGBD, portanto, certifique-se de executar aquela específica para o seu próprio SGBD. Esses scripts são fornecidos para a conveniência dos leitores e nenhuma responsabilidade é assumida por problemas que possam resultar de seu uso.

Na época em que este livro foi lançado, os scripts estavam disponíveis para

- IBM DB2 (incluindo Db2 on Cloud)
- Microsoft SQL Server (incluindo Microsoft SQL Server Express)
- MariaDB
- MySQL
- Oracle (inclui Oracle Express)
- PostgreSQL
- SQLite

> **DICA: Arquivo de dados do SQLite**
>
> O SQLite armazena seus dados em um único arquivo. Você pode usar os scripts de criação e preenchimento para criar o próprio arquivo de dados do SQLite. Ou, para facilitar as coisas, pode baixar um arquivo pronto para uso na URL http://forta.com/books/0135182794/.

Outros SGBDs podem ser adicionados conforme necessário ou solicitado.

> **NOTA: Criar e depois preencher**
>
> Você deve executar os scripts de criação de tabela antes dos scripts de preenchimento de tabela. Não se esqueça de verificar se há mensagens de erro retornadas por esses scripts. Se os scripts de criação falharem, você precisará solucionar qualquer problema que possa existir antes de continuar com o preenchimento da tabela.

> **NOTA: Instruções específicas de configuração de SGBD**
>
> As etapas específicas usadas para configurar seu SGBD variam muito com base no SGBD usado. Ao baixar os scripts ou bancos de dados da página da web do livro, você encontrará um arquivo README que descreve as etapas de configuração e instalação específicas para SGBDs específicos.

---

1 N.T.: As tabelas e os scripts para a criação dessas tabelas estão disponíveis para download em https://forta.com/books/0135182794/.

APÊNDICE B

Sintaxe das instruções SQL

Para ajudá-lo a encontrar a sintaxe necessária quando você precisar dela, este apêndice lista a sintaxe para as operações SQL usadas com mais frequência. Cada instrução começa com uma breve descrição e, em seguida, a sintaxe apropriada é exibida. Para maior comodidade, você também encontrará referências cruzadas para as lições em que instruções específicas são ensinadas.

Ao ler a sintaxe da instrução, lembre-se do seguinte:

- O símbolo `|` é usado para indicar uma das várias opções, portanto `NULL|NOT NULL` significa especificar `NULL` ou `NOT NULL`.
- Palavras-chave ou cláusulas entre colchetes `[assim]` são opcionais.
- A sintaxe dos comandos listados a seguir funcionará com quase todos os SGBDs. Aconselhamos você a consultar a documentação do seu próprio SGBD para obter detalhes sobre a implementação de mudanças sintáticas específicas.

ALTER TABLE

`ALTER TABLE` é usada para atualizar o esquema de uma tabela existente. Para criar uma nova tabela, use `CREATE TABLE`. Consulte a *Lição 17 – Criando e manipulando tabelas*, para obter mais informações.

Entrada ▼

```
ALTER TABLE nome da tabela
(
  ADD|DROP  coluna  datatype  [NULL|NOT NULL]  [CONSTRAINTS],
  ADD|DROP  coluna  datatype  [NULL|NOT NULL]  [CONSTRAINTS],
    ...
);
```

COMMIT

COMMIT é usada para gravar uma transação no banco de dados. Consulte a *Lição 20 – Gerenciando o processamento de transações*, para obter mais informações.

Entrada ▼

```
COMMIT [TRANSACTION];
```

CREATE INDEX

CREATE INDEX é usada para criar um índice em uma ou mais colunas. Consulte a *Lição 22 – Compreendendo os recursos SQL avançados*, para obter mais informações.

Entrada ▼

```
CREATE INDEX nome do índice
ON nome da tabela (coluna, ...);
```

CREATE PROCEDURE

CREATE PROCEDURE é usada para criar uma stored procedure. Consulte a *Lição 19 – Trabalhando com stored procedures*, para obter mais informações. O Oracle usa uma sintaxe diferente, conforme descrito nessa lição.

Entrada ▼

```
CREATE PROCEDURE nome da procedure [parâmetros] [opções]
AS
instrução SQL;
```

CREATE TABLE

CREATE TABLE é usada para criar novas tabelas de banco de dados. Para atualizar o esquema de uma tabela existente, use ALTER TABLE. Consulte a *Lição 17 – Criando e manipulando tabelas* para obter mais informações.

Entrada ▼

```
CREATE TABLE nome da tabela
(
    coluna    datatype    [NULL|NOT NULL]    [CONSTRAINTS],
    coluna    datatype    [NULL|NOT NULL]    [CONSTRAINTS],
       ...
```

```
);
```

CREATE VIEW

`CREATE VIEW` é usada para criar uma nova visualização de uma ou mais tabelas. Consulte a *Lição 18 – Usando visualizações (views)*, para obter mais informações.

Entrada ▼

```
CREATE VIEW nome da visualização AS
SELECT colunas, ...
FROM tabelas, ...
[WHERE ...]
[GROUP BY ...]
[HAVING ...];
```

DELETE

`DELETE` exclui uma ou mais linhas de uma tabela. Consulte a *Lição 16 – Atualizando e excluindo dados*, para obter mais informações.

Entrada ▼

```
DELETE FROM nome da tabela
[WHERE ...];
```

DROP

`DROP` remove objetos de banco de dados permanentemente (tabelas, visualizações, índices e assim por diante). Consulte as lições 17 e 18 para obter mais informações.

Entrada ▼

```
DROP INDEX|PROCEDURE|TABLE|VIEW nome do índice|nome da procedure|nome da
  tabela|nome da visualização;
```

INSERT

`INSERT` adiciona uma única linha a uma tabela. Consulte a *Lição 15 – Inserindo dados*, para obter mais informações.

Entrada ▼

```
INSERT INTO nome da tabela [(colunas, ...)]
VALUES(valores, ...);
```

INSERT SELECT

`INSERT SELECT` insere os resultados de uma instrução `SELECT` em uma tabela. Consulte a *Lição 15 – Inserindo dados* para obter mais informações.

Entrada ▼

```
INSERT INTO nome da tabela [(colunas, ...)]
SELECT colunas, ... FROM nome da tabela, ...
[WHERE ...];
```

ROLLBACK

`ROLLBACK` é usada para desfazer um bloco de transação. Consulte a *Lição 20 – Gerenciando o processamento de transações* para obter mais informações.

Entrada ▼

```
ROLLBACK [TO nome do ponto de salvamento];
```

ou

Entrada ▼

```
ROLLBACK TRANSACTION;
```

SELECT

`SELECT` é usada para obter dados de uma ou mais tabelas (ou visualizações). Consulte a *Lição 2 – Obtendo dados*, a *Lição 3 – Classificando dados obtidos*, e a *Lição 4 – Filtrando dados*, para obter mais informações básicas. (As lições de 2 a 14 abordam aspectos da instrução `SELECT`.)

Entrada ▼

```
SELECT nome da coluna, ...
FROM nome da tabela, ...
[WHERE ...]
[UNION ...]
[GROUP BY ...]
[HAVING ...]
[ORDER BY ...];
```

UPDATE

`UPDATE` atualiza uma ou mais linhas em uma tabela. Consulte a *Lição 16 – Atualizando e excluindo dados* para obter mais informações.

Entrada ▼

```
UPDATE nome da tabela
SET nome da coluna = valor, ...
[WHERE ...];
```

APÊNDICE C

Usando datatypes do SQL

Conforme explicado na *Lição 1 – Entendendo SQL*, datatypes são essencialmente regras que definem quais dados podem ser armazenados em uma coluna e como esses dados são realmente armazenados.

Datatypes são usados por vários motivos:

- Datatypes permitem restringir o tipo de dados que podem ser armazenados em uma coluna. Por exemplo, uma coluna com datatype numérico só aceitará valores numéricos.
- Datatypes permitem um armazenamento interno mais eficiente. Números e valores de data-hora podem ser armazenados em um formato mais condensado do que as strings de texto.
- Datatypes permitem ordens de classificação alternativas. Se tudo for tratado como string, `1` vem antes de `10`, que vem antes de `2`. (Strings são classificadas em sequência de dicionário, um caractere de cada vez começando pela esquerda.) Como datatypes numéricos, os números seriam classificados corretamente.

Ao projetar tabelas, preste atenção nos datatypes que estão sendo usados. O uso de datatypes incorretos pode afetar seriamente sua aplicação. Alterar os datatypes das colunas preenchidas existentes não é uma tarefa trivial. (Além disso, fazê-lo pode resultar em perda de dados.)

Embora este apêndice não seja de forma alguma um tutorial completo sobre datatypes e como eles devem ser usados, ele explica os principais datatypes, para que eles são usados e problemas de compatibilidade dos quais você deve estar ciente.

CUIDADO: Nenhum SGBD é exatamente igual ao outro

Isso já foi dito antes, mas precisa ser dito de novo. Infelizmente, datatypes podem variar drasticamente de um SGBD para outro. Mesmo datatypes iguais podem significar coisas

diferentes para SGBDs diferentes. Certifique-se de consultar a documentação de seu SGBD para obter detalhes sobre exatamente o que ele suporta e como.

Datatypes do tipo string

Os datatypes mais comumente usados são strings. Eles armazenam strings: por exemplo, nomes, endereços, números de telefone e códigos postais. Há basicamente dois tipos de strings que você pode usar – strings de tamanho fixo e strings de tamanho variável (veja a Tabela C.1).

Strings de tamanho fixo são datatypes definidos para aceitarem um número fixo de caracteres e esse número é especificado quando a tabela é criada. Por exemplo, você pode permitir 30 caracteres em uma coluna de nome ou 11 caracteres em uma coluna de número de Seguridade Social (o número exato necessário para permitir dois traços). As colunas de tamanho fixo não reservam mais do que o número especificado de caracteres. Elas alocam também o espaço de armazenamento para tantos caracteres quanto especificados. Assim, se a string `Ben` for armazenada em um campo de 30 caracteres, todos os 30 caracteres são armazenados (e o texto pode ser preenchido com espaços conforme necessário).

Strings de comprimento variável armazenam texto de qualquer comprimento (o tamanho máximo varia de acordo com o datatype e o SGBD). Alguns tipos de datatypes de comprimento variável têm um comprimento fixo mínimo. Outros são totalmente variáveis. De qualquer forma, apenas os dados especificados são salvos (e nenhum dado extra é armazenado).

Se os datatypes de comprimento variável são tão flexíveis, por que você iria querer usar datatypes de comprimento fixo? A resposta é desempenho. Os SGBDs podem classificar e manipular colunas de comprimento fixo muito mais rapidamente do que classificar colunas de comprimento variável. Além disso, muitos SGBDs não permitem que você indexe colunas de comprimento variável (ou a parte variável de uma coluna). Isso também afeta drasticamente o desempenho. (Consulte a *Lição 22 – Compreendendo os recursos SQL avançados*, para obter mais informações sobre índices.)

Tabela C.1 – Datatypes do tipo string

Datatype	Descrição
`CHAR`	String de comprimento fixo de 1 a 255 caracteres. Seu tamanho deve ser especificado no momento da criação
`NCHAR`	Forma especial de `CHAR` projetado para oferecer suporte a caracteres multibyte ou Unicode. (As especificações exatas variam drasticamente de uma implementação para outra.)
`NVARCHAR`	Forma especial de `TEXT` projetado para oferecer suporte a caracteres multibyte ou Unicode. (As especificações exatas variam drasticamente de uma implementação para outra.)
`TEXT (também chamado de LONG ou MEMO ou VARCHAR)`	Texto de comprimento variável

DICA: Usando aspas

Independentemente da forma do datatype tipo string em uso, os valores da string devem sempre estar entre aspas simples.

CUIDADO: Quando valores numéricos não são valores numéricos

Você poderia pensar que números de telefone e CEP devem ser armazenados em campos numéricos (afinal de contas, eles só armazenam dados numéricos), mas fazê-lo não seria aconselhável. Se você armazenar o CEP 01234 em um campo numérico, o número 1234 seria salvo. Dessa forma, você perderia um dígito.

A regra básica a seguir é: Se o número é um número usado em cálculos (somas, médias e assim por diante), ele pertence em uma coluna com datatype numérico. Se ele for usado como uma string literal (que por acaso contém apenas dígitos), ele pertence em uma coluna do tipo string.

Datatypes numéricos

Os datatypes numéricos armazenam números. A maioria dos SGBDs oferece suporte a vários datatypes numéricos, cada um com um intervalo diferente de números que podem ser armazenados nele. Obviamente, quanto maior o intervalo suportado, mais espaço de armazenamento é necessário. Além disso, alguns datatypes numéricos suportam o uso de casas decimais (e números fracionários), enquanto outros suportam apenas números inteiros. A Tabela C.2 lista os usos comuns para vários datatypes, mas nem todos os SGBDs seguem as convenções de nomenclatura exatas e as descrições listadas aqui.

Tabela C.2 – Datatypes numéricos

Datatype	Descrição
`BIT`	Valor de bit único, `0` ou `1`, usado principalmente para flags do tipo ligado/desligado
`DECIMAL` (também chamado de `NUMERIC`)	Valores de ponto fixo ou flutuante com vários níveis de precisão
`FLOAT` (também chamado de `NUMBER`)	Valores de ponto flutuante
`INT` (também chamado de `INTEGER`)	Valor inteiro de 4 bytes que suporta números de -2147483648 a 2147483647
`REAL`	Valores de ponto flutuante de 4 bytes
`SMALLINT`	Valor inteiro de 2 bytes que suporta números de -32768 a 32767
`TINYINT`	Valor inteiro de 1 byte que suporta números de 0 a 255

DICA: Não usar aspas

Ao contrário das strings, os valores numéricos nunca devem ser colocados entre aspas.

DICA: Datatypes do tipo moeda

A maioria dos SGBDs oferece suporte a um datatype numérico especial para armazenar valores monetários. Normalmente chamados de `MONEY` ou `CURRENCY`, esses datatypes são essencialmente datatypes `DECIMAL` com intervalos específicos que os tornam adequados para armazenar valores de moeda.

Datatypes do tipo data e hora

Todos os SGBDs suportam datatypes projetados para o armazenamento de valores de data e hora (consulte a Tabela C.3). Como os valores numéricos, a maioria dos SGBDs oferece suporte a vários datatypes, cada um com diferentes intervalos e níveis de precisão.

Tabela C.3 – Datatypes do tipo data e hora

Datatype	Descrição
`DATE`	Valor da data
`DATETIME` (também conhecido como `TIMESTAMP`)	Valores de data e hora
`SMALLDATETIME`	Valores de data e hora com precisão de minuto (sem segundos ou milissegundos)

`TIME`	Valor da hora

CUIDADO: Especificando datas

Não existe uma maneira padrão de definir uma data que será compreendida por todos os SGBDs. A maioria das implementações entende formatos como `2020-12-30` ou `30 de dezembro de 2020`, mas mesmo esses podem ser problemáticos para alguns SGBDs. Certifique-se de consultar a documentação de seu SGBD para obter uma lista dos formatos de data que ele reconhece.

DICA: Datas ODBC

Como cada SGBD tem seu próprio formato para especificar datas, o ODBC criou um formato próprio que funcionará com todos os bancos de dados quando o ODBC estiver sendo usado. O formato ODBC é semelhante a `{d '2020-12-30'}` para datas, `{t '21: 46: 29 '}` para horas e `{ts' 2020-12-30 21:46:29 '}` para valores de data e hora. Se você estiver usando SQL via ODBC, certifique-se de que suas datas e horas estejam formatadas dessa maneira.

Datatypes binários

Datatypes binários são alguns dos datatypes menos compatíveis (e, felizmente, também alguns dos menos usados). Ao contrário de todos os datatypes explicados até agora, que têm usos muito específicos, os datatypes binários podem conter qualquer dado, até mesmo informações binárias, como imagens gráficas, multimídia e documentos de processador de texto (consulte a Tabela C.4).

Tabela C.4 – Datatypes binários

Datatype	Descrição
`BINARY`	Dados binários de comprimento fixo (o comprimento máximo pode variar de 255 bytes a 8.000 bytes, dependendo da implementação)
`LONG RAW`	Dados binários de comprimento variável de até 2 GB
`RAW (chamado de BINARY por algumas implementações)`	Dados binários de comprimento fixo de até 255 bytes
`VARBINARY`	Dados binários de comprimento variável (o comprimento máximo pode variar de 255 bytes a 8.000 bytes, dependendo da implementação)

NOTA: Comparando datatypes

Se você quiser ver um exemplo de comparações de banco de dados do mundo real, olhe os scripts de criação de tabela usados para construir os exemplos de tabelas neste livro (consulte o *Apêndice A – Exemplos de scripts de tabela*). Ao comparar os scripts usados para diferentes SGBDs, você verá em primeira mão o quão complexa é a tarefa de estabelecer correspondência entre datatypes.

APÊNDICE D

Palavras reservadas do SQL

O SQL é uma linguagem composta de palavras-chave – palavras especiais que são usadas para executar operações SQL. É preciso prestar especial atenção para não usar essas palavras-chave para nomear bancos de dados, tabelas, colunas e qualquer outro tipo de objeto. Dessa forma, essas palavras-chave são consideradas reservadas.

Este apêndice contém uma lista das palavras reservadas mais comuns encontradas nos principais SGBDs. Observe o seguinte:

- Palavras-chave tendem a ser muito específicas do SGBD e nem todas as palavras-chave a seguir são usadas por todos os SGBDs.
- Muitos SGBDs ampliaram a lista de palavras reservadas do SQL para incluir termos específicos de suas implementações. A maioria das palavras-chave específicas do SGBD não está incluída na lista a seguir.
- Para garantir compatibilidade e portabilidade futuras, recomenda-se evitar toda e qualquer palavra-chave, mesmo aquelas não reservadas por seu próprio SGBD.

ABORT	ABSOLUTE	ACTION	ACTIVE
ADD	AFTER	ALL	ALLOCATE
ALTER	ANALYZE	AND	ANY
ARE	AS	ASC	ASCENDING
ASSERTION	AT	AUTHORIZATION	AUTO
AUTO-INCREMENT	AUTOINC	AVG	BACKUP
BEFORE	BEGIN	BETWEEN	BIGINT
BINARY	BIT	BLOB	BOOLEAN
BOTH	BREAK	BROWSE	BULK
BY	BYTES	CACHE	CALL
CASCADE	CASCADED	CASE	CAST
CATALOG	CHANGE	CHAR	CHARACTER

CHARACTER_LENGTH	CHECK	CHECKPOINT	CLOSE
CLUSTER	CLUSTERED	COALESCE	COLLATE
COLUMN	COLUMNS	COMMENT	COMMIT
COMMITTED	COMPUTE	COMPUTED	CONDITIONAL
CONFIRM	CONNECT	CONNECTION	CONSTRAINT
CONSTRAINTS	CONTAINING	CONTAINS	CONTAINSTABLE
CONTINUE	CONTROLROW	CONVERT	COPY
COUNT	CREATE	CROSS	CSTRING
CUBE	CURRENT	CURRENT_DATE	CURRENT_TIME
CURRENT_TIMESTAMP	CURRENT_USER	CURSOR	DATABASE
DATABASES	DATE	DATETIME	DAY
DBCC	DEALLOCATE	DEBUG	DEC
DECIMAL	DECLARE	DEFAULT	DELETE
DENY	DESC	DESCENDING	DESCRIBE
DISCONNECT	DISK	DISTINCT	DISTRIBUTED
DIV	DO	DOMAIN	DOUBLE
DROP	DUMMY	DUMP	ELSE
ELSEIF	ENCLOSED	END	ERRLVL
ERROREXIT	ESCAPE	ESCAPED	EXCEPT
EXCEPTION	EXEC	EXECUTE	EXISTS
EXIT	EXPLAIN	EXTEND	EXTERNAL
EXTRACT	FALSE	FETCH	FIELD
FIELDS	FILE	FILLFACTOR	FILTER
FLOAT	FLOPPY	FOR	FORCE
FOREIGN	FOUND	FREETEXT	FREETEXTTABLE
FROM	FULL	FUNCTION	GENERATOR
GET	GLOBAL	GO	GOTO
GRANT	GROUP	HAVING	HOLDLOCK
HOUR	IDENTITY	IF	IN
INACTIVE	INDEX	INDICATOR	INFILE
INNER	INOUT	INPUT	INSENSITIVE
INSERT	INT	INTEGER	INTERSECT
INTERVAL	INTO	IS	ISOLATION
JOIN	KEY	KILL	LANGUAGE

LAST	LEADING	LEFT	LENGTH
LEVEL	LIKE	LIMIT	LINENO
LINES	LISTEN	LOAD	LOCAL
LOCK	LOGFILE	LONG	LOWER
MANUAL	MATCH	MAX	MERGE
MESSAGE	MIN	MINUTE	MIRROREXIT
MODULE	MONEY	MONTH	MOVE
NAMES	NATIONAL	NATURAL	NCHAR
NEXT	NEW	NO	NOCHECK
NONCLUSTERED	NONE	NOT	NULL
NULLIF	NUMERIC	OF	OFF
OFFSET	OFFSETS	ON	ONCE
ONLY	OPEN	OPTION	OR
ORDER	OUTER	OUTPUT	OVER
OVERFLOW	OVERLAPS	PAD	PAGE
PAGES	PARAMETER	PARTIAL	PASSWORD
PERCENT	PERM	PERMANENT	PIPE
PLAN	POSITION	PRECISION	PREPARE
PRIMARY	PRINT	PRIOR	PRIVILEGES
PROC	PROCEDURE	PROCESSEXIT	PROTECTED
PUBLIC	PURGE	RAISERROR	READ
READTEXT	REAL	REFERENCES	REGEXP
RELATIVE	RENAME	REPEAT	REPLACE
REPLICATION	REQUIRE	RESERV	RESERVING
RESET	RESTORE	RESTRICT	RETAIN
RETURN	RETURNS	REVOKE	RIGHT
ROLLBACK	ROLLUP	ROWCOUNT	RULE
SAVE	SAVEPOINT	SCHEMA	SECOND
SECTION	SEGMENT	SELECT	SENSITIVE
SEPARATOR	SEQUENCE	SESSION_USER	SET
SETUSER	SHADOW	SHARED	SHOW
SHUTDOWN	SINGULAR	SIZE	SMALLINT
SNAPSHOT	SOME	SORT	SPACE
SQL	SQLCODE	SQLERROR	STABILITY

STARTING	STARTS	STATISTICS	SUBSTRING
SUM	SUSPEND	TABLE	TABLES
TEMP	TEMPORARY	TEXT	TEXTSIZE
THEN	TIME	TIMESTAMP	TO
TOP	TRAILING	TRAN	TRANSACTION
TRANSLATE	TRIGGER	TRIM	TRUE
TRUNCATE	TYPE	UNCOMMITTED	UNION
UNIQUE	UNTIL	UPDATE	UPDATETEXT
UPPER	USAGE	USE	USER
USING	VALUE	VALUES	VARCHAR
VARIABLE	VARYING	VERBOSE	VIEW
VOLUME	WAIT	WAITFOR	WHEN
WHERE	WHILE	WITH	WORK
WRITE	WRITETEXT	XOR	YEAR
ZONE			

APÊNDICE E

Respostas dos desafios

Lição 2

Desafio 1

```
SELECT cust_id
FROM Customers;
```

Desafio 2

```
SELECT DISTINCT prod_id
FROM OrderItems;
```

Desafio 3

```
SELECT *
# SELECT cust_id
FROM Customers;
```

Lição 3

Desafio 1

```
SELECT cust_name
FROM Customers
ORDER BY cust_name DESC;
```

Desafio 2

```
SELECT cust_id, order_num
FROM Orders
ORDER BY cust_id, order_date DESC;
```

Desafio 3

```
SELECT quantity, item_price
FROM OrderItems
ORDER BY quantity DESC, item_price DESC;
```

Desafio 4

```
SELECT vend_name,
FROM Vendors
ORDER vend_name DESC;
```

Não deve haver uma vírgula após `vend_name` (uma vírgula é usada apenas

para separar várias colunas) e BY está faltando após ORDER.

Lição 4

Desafio 1

```
SELECT prod_id, prod_name
FROM Products
WHERE prod_price = 9.49;
```

Desafio 2

```
SELECT prod_id, prod_name
FROM Products
WHERE prod_price >= 9;
```

Desafio 3

```
SELECT DISTINCT order_num
FROM OrderItems
WHERE quantity >=100;
```

Desafio 4

```
SELECT prod_name, prod_price
FROM products
WHERE prod_price BETWEEN 3 AND 6
ORDER BY prod_price;
```

Lição 5

Desafio 1

```
SELECT vend_name
FROM Vendors
WHERE vend_country = 'USA' AND vend_state = 'CA';
```

Desafio 2

```
-- Solução 1
SELECT order_num, prod_id, quantity
FROM OrderItems
WHERE (prod_id='BR01' OR prod_id='BR02' OR prod_id='BR03')
 AND quantity >=100;

-- Solução 2
SELECT order_num, prod_id, quantity
FROM OrderItems
WHERE prod_id IN ('BR01','BR02','BR03')
 AND quantity >=100;
```

Desafio 3

```
SELECT prod_name, prod_price
FROM products
WHERE prod_price >= 3 AND prod_price <= 6
ORDER BY prod_price;
```

Desafio 4

```
SELECT vend_name
FROM Vendors
ORDER BY vend_name
WHERE vend_country = 'USA' AND vend_state = 'CA';
```

`ORDER BY` deve vir após qualquer cláusula `WHERE`.

Lição 6

Desafio 1

```
SELECT prod_name, prod_desc
FROM Products
WHERE prod_desc LIKE '%toy%';
```

Desafio 2

```
SELECT prod_name, prod_desc
FROM Products
WHERE NOT prod_desc LIKE '%toy%'
ORDER BY prod_name;
```

Desafio 3

```
SELECT prod_name, prod_desc
FROM Products
WHERE prod_desc LIKE '%toy%' AND prod_desc LIKE '%carrots%';
```

Desafio 4

```
SELECT prod_name, prod_desc
FROM Products
WHERE prod_desc LIKE '%toy%carrots%';
```

Lição 7

Desafio 1

```
SELECT vend_id,
        vend_name as vname,
        vend_address AS vaddress,
        vend_city AS vcity
FROM Vendors
ORDER BY vname;
```

Desafio 2

```
SELECT prod_id, prod_price,
       prod_price * 0.9 AS sale_price
FROM Products;
```

Lição 8

Desafio 1

```
-- DB2, PostgreSQL
SELECT cust_id, cust_name,
  UPPER(LEFT(cust_contact, 2)) ||
  UPPER(LEFT(cust_city, 3)) AS user_login
FROM customers;
-- Oracle, SQLite
SELECT cust_id, cust_name,
  UPPER(SUBSTR(cust_contact, 1, 2)) ||
  UPPER(SUBSTR(cust_city, 1, 3)) AS user_login
FROM customers;
-- MySQL
SELECT cust_id, cust_name,
  CONCAT(UPPER(LEFT(cust_contact, 2)),
  UPPER(LEFT(cust_city, 3))) AS user_login
FROM customers;
-- SQL Server
SELECT cust_id, cust_name,
  UPPER(LEFT(cust_contact, 2)) +
  UPPER(LEFT(cust_city, 3)) AS user_login
FROM customers;
```

Desafio 2

```
-- DB2, MariaDB, MySQL
SELECT order_num, order_date
FROM Orders
WHERE YEAR(order_date) = 2020 AND MONTH(order_date) = 1
ORDER BY order_date;
-- Oracle, PostgreSQL
SELECT order_num, order_date
FROM Orders
WHERE EXTRACT(year FROM order_date) = 2020 AND
      EXTRACT(month FROM order_date) = 1
ORDER BY order_date;
-- PostgreSQL
SELECT order_num, order_date
```

```
FROM Orders
WHERE DATE_PART('year', order_date) = 2020
 AND DATE_PART('month', order_date) = 1
ORDER BY order_num;
-- SQL Server
SELECT order_num, order_date
FROM Orders
WHERE DATEPART(yy, order_date) = 2020 AND
      DATEPART(mm, order_date) = 1
ORDER BY order_date;
-- SQLite
SELECT order_num
FROM Orders
WHERE strftime('%Y', order_date) = '2020'
  AND strftime('%m', order_date) = '01';
```

Lição 9

Desafio 1

```
SELECT SUM(quantity) AS items_ordered
FROM OrderItems;
```

Desafio 2

```
SELECT SUM(quantity) AS items_ordered
FROM OrderItems
WHERE prod_id = 'BR01';
```

Desafio 3

```
SELECT MAX(prod_price) AS max_price
FROM Products
WHERE prod_price <= 10;
```

Lição 10

Desafio 1

```
SELECT order_num, COUNT(*) as order_lines
FROM OrderItems
GROUP BY order_num
ORDER BY order_lines;
```

Desafio 2

```
SELECT vend_id, MIN(prod_price) AS cheapest_item
FROM Products
GROUP BY vend_id
ORDER BY cheapest_item;
```

Desafio 3

```
SELECT order_num
FROM OrderItems
GROUP BY order_num
HAVING SUM(quantity) >= 100
ORDER BY order_num;
```

Desafio 4

```
SELECT order_num, SUM(item_price*quantity) AS total_price
FROM OrderItems
GROUP BY order_num
HAVING SUM(item_price*quantity) >= 1000
ORDER BY order_num;
```

Desafio 5

```
SELECT order_num, COUNT() AS items
FROM OrderItems
GROUP BY items
HAVING COUNT() >= 3
ORDER BY items, order_num;
```

O item `GROUP BY` está incorreto. `GROUP BY` deve ser uma coluna real, não aquela que está sendo usada para realizar os cálculos agregados. `GROUP BY order_num` seria permitido.

Lição 11

Desafio 1

```
SELECT cust_id
FROM Orders
WHERE order_num IN (SELECT order_num
                    FROM OrderItems
                    WHERE item_price >= 10);
```

Desafio 2

```
SELECT cust_id, order_date
FROM orders
WHERE order_num IN (SELECT order_num
                    FROM OrderItems
                    WHERE prod_id = 'BR01')
ORDER BY order_date;
```

Desafio 3

```
SELECT cust_email FROM Customers
WHERE cust_id IN (SELECT cust_id
```

```
                FROM Orders
                WHERE order_num IN (SELECT order_num
                                    FROM OrderItems
                                    WHERE prod_id = 'BR01'));
```

Desafio 4

```
SELECT cust_id,
       (SELECT SUM(item_price*quantity)
        FROM OrderItems
        WHERE Orders.order_num = OrderItems.order_num) AS total_ordered
FROM Orders
ORDER BY total_ordered DESC;
```

Desafio 5

```
SELECT prod_name,
       (SELECT Sum(quantity)
        FROM OrderItems
        WHERE Products.prod_id=OrderItems.prod_id) AS quant_sold
FROM Products;
```

Lição 12

Desafio 1

```
-- Sintaxe Equijoin
SELECT cust_name, order_num
FROM Customers, Orders
WHERE Customers.cust_id = Orders.cust_id
ORDER BY cust_name, order_num;

-- Sintaxe ANSI INNER JOIN
SELECT cust_name, order_num
FROM Customers INNER JOIN Orders
 ON Customers.cust_id = Orders.cust_id
ORDER BY cust_name, order_num;
```

Desafio 2

```
-- Solução usando subconsultas
SELECT cust_name,
       order_num,
       (SELECT Sum(item_price*quantity)
       FROM OrderItems
       WHERE Orders.order_num=OrderItems.order_num) AS OrderTotal
FROM Customers, Orders
WHERE Customers.cust_id = Orders.cust_id
ORDER BY cust_name, order_num;
```

```
-- Solução usando joins
SELECT cust_name,
       Orders.order_num,
       Sum(item_price*quantity) AS OrderTotal
FROM Customers, Orders, OrderItems
WHERE Customers.cust_id = Orders.cust_id
 AND Orders.order_num = OrderItems.order_num
GROUP BY cust_name, Orders.order_num
ORDER BY cust_name, order_num;
```

Desafio 3

```
SELECT cust_id, order_date
FROM Orders, OrderItems
WHERE Orders.order_num = OrderItems.order_num
 AND prod_id = 'BR01'
ORDER BY order_date;
```

Desafio 4

```
SELECT cust_email
FROM Customers
 INNER JOIN Orders ON Customers.cust_id = Orders.cust_id
 INNER JOIN OrderItems ON Orders.order_num = OrderItems.order_num
WHERE prod_id = 'BR01';
```

Desafio 5

```
-- Sintaxe Equijoin
SELECT cust_name, SUM(item_price*quantity) AS total_price
FROM Customers, Orders, OrderItems
WHERE Customers.cust_id = Orders.cust_id
 AND Orders.order_num = OrderItems.order_num
GROUP BY cust_name HAVING SUM(item_price*quantity) >= 1000
ORDER BY cust_name;

-- Sintaxe ANSI INNER JOIN
SELECT cust_name, SUM(item_price*quantity) AS total_price
FROM Customers
 INNER JOIN Orders ON Customers.cust_id = Orders.cust_id
 INNER JOIN OrderItems ON Orders.order_num = OrderItems.order_num
GROUP BY cust_name
HAVING SUM(item_price*quantity) >= 1000
ORDER BY cust_name;
```

Lição 13

Desafio 1

```
SELECT cust_name, order_num
FROM Customers
 JOIN Orders ON Customers.cust_id = Orders.cust_id
ORDER BY cust_name;
```

Desafio 2

```
SELECT cust_name, order_num
FROM Customers
 LEFT OUTER JOIN Orders ON Customers.cust_id = Orders.cust_id
ORDER BY cust_name;
```

Desafio 3

```
SELECT prod_name, order_num
FROM Products LEFT OUTER JOIN OrderItems
 ON Products.prod_id = OrderItems.prod_id
ORDER BY prod_name;
```

Desafio 4

```
SELECT prod_name, COUNT(order_num) AS orders
FROM Products LEFT OUTER JOIN OrderItems
 ON Products.prod_id = OrderItems.prod_id
GROUP BY prod_name
ORDER BY prod_name;
```

Desafio 5

```
SELECT Vendors.vend_id, COUNT(prod_id)
FROM Vendors
 LEFT OUTER JOIN Products ON Vendors.vend_id = Products.vend_id
GROUP BY Vendors.vend_id;
```

Lição 14

Desafio 1

```
SELECT prod_id, quantity FROM OrderItems
WHERE quantity = 100
UNION
SELECT prod_id, quantity FROM OrderItems
WHERE prod_id LIKE 'BNBG%'
ORDER BY prod_id;
```

Desafio 2

```
SELECT prod_id, quantity FROM OrderItems
WHERE quantity = 100 OR prod_id LIKE 'BNBG%'
ORDER BY prod_id;
```

Desafio 3

```
SELECT prod_name
FROM Products
UNION
SELECT cust_name
FROM Customers
ORDER BY prod_name;
```

Desafio 4

```
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_state  = 'MI'
ORDER BY cust_name;
UNION
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_state = 'IL'
ORDER BY cust_name;
```

O caractere ";" depois que a primeira instrução `SELECT` não deveria ser especificado, ele está encerrando a instrução. Além disso, ao classificar instruções `SELECT` combinadas com `UNION`, apenas um `ORDER BY` pode ser usado e deve vir após o último `SELECT`.

Lição 15

Desafio 1

```
-- Obviamente substitua os detalhes com seus próprios dados
INSERT INTO Customers(cust_id,
                      cust_name,
                      cust_address,
                      cust_city,
                      cust_state,
                      cust_zip,
                      cust_country,
                      cust_email)
VALUES(1000000042,
       'Ben''s Toys',
       '123 Main Street',
       'Oak Park',
       'MI',
       '48237',
       'USA',
```

```
       'ben@forta.com');
```

Desafio 2

```
-- MySQL, MariaDB, Oracle, PostgreSQL, SQLite
CREATE TABLE OrdersBackup AS SELECT * FROM Orders;
CREATE TABLE OrderItemsBackup AS SELECT * FROM OrderItems;

-- SQL Server
SELECT * INTO OrdersBackup FROM Orders;
SELECT * INTO OrderItemsBackup FROM OrderItems;
```

Lição 16

Desafio 1

```
UPDATE Vendors
SET vend_state = UPPER(vend_state)
WHERE vend_country = 'USA';
UPDATE Customers
SET cust_state = UPPER(cust_state)
WHERE cust_country = 'USA';
```

Desafio 2

```
-- Primeiro teste o WHERE para selecionar somente o que deseja excluir
SELECT * FROM Customers
WHERE cust_id = 1000000042;
-- Then do it!
DELETE Customers
WHERE cust_id = 1000000042;
```

Lição 17

Desafio 1

```
ALTER TABLE Vendors
ADD vend_web CHAR(100);
```

Desafio 2

```
UPDATE Vendors
SET vend_web = 'https://google.com/'
WHERE vend_id = 'DLL01';
```

Lição 18

Desafio 1

```
CREATE VIEW CustomersWithOrders AS
SELECT Customers.cust_id,
```

```
       Customers.cust_name,
       Customers.cust_address,
       Customers.cust_city,
       Customers.cust_state,
       Customers.cust_zip,
       Customers.cust_country,
       Customers.cust_contact,
       Customers.cust_email
FROM Customers
JOIN Orders ON Customers.cust_id = Orders.cust_id;

SELECT * FROM CustomersWithOrders;
```

Desafio 2

```
CREATE VIEW OrderItemsExpanded AS
SELECT order_num,
       prod_id,
       quantity,
       item_price,
       quantity*item_price AS expanded_price
FROM OrderItems
ORDER BY order_num;
```

`ORDER BY` não é permitido em visualizações (views). As visualizações são usadas como tabelas, se você precisar de dados classificados, use `ORDER BY` no `SELECT` que retorna os dados da visualização.

Para… Veja…

Instruções SQL usadas com frequência

ALTER TABLE

`ALTER TABLE` é usada para atualizar o esquema de uma tabela existente.

Para criar uma nova tabela, use `CREATE TABLE`.

Consulte a *Lição 17 – Criando e manipulando tabelas*.

COMMIT

`COMMIT` é usada para gravar uma transação no banco de dados.

Consulte a *Lição 20 – Gerenciando o processamento de transações*.

CREATE INDEX

`CREATE INDEX` é usado para criar um índice em uma ou mais colunas.

Consulte a *Lição 22 – Compreendendo os recursos SQL avançados*.

CREATE TABLE

`CREATE TABLE` é usado para criar novas tabelas de banco de dados.

Para atualizar o esquema de uma tabela existente, use `ALTER TABLE`.

Consulte a *Lição 17 – Criando e manipulando tabelas*.

CREATE VIEW

`CREATE VIEW` é usado para criar uma nova visualização de uma ou mais tabelas.

Consulte a *Lição 18 – Usando visualizações (views)*.

DELETE

`DELETE` exclui uma ou mais linhas de uma tabela.

Consulte a *Lição 16 – Atualizando e excluindo dados*.

DROP

`DROP` remove objetos de banco de dados permanentemente (tabelas,

visualizações, índices e assim por diante).

Consulte a *Lição 17 – Criando e manipulando tabelas*, e a *Lição 18 – Usando visualizações (views)*.

INSERT

INSERT adiciona uma única linha a uma tabela.

Consulte a *Lição 15 – Inserindo dados*.

INSERT SELECT

INSERT SELECT insere os resultados de uma instrução SELECT em uma tabela.

Consulte a *Lição 15 – Inserindo dados*.

ROLLBACK

ROLLBACK é usado para desfazer um bloco de transação.

Consulte a *Lição 20 – Gerenciando o processamento de transações*.

SELECT

SELECT é usado para obter dados de uma ou mais tabelas (ou visualizações).

Consulte a *Lição 2 – Obtendo dados*, a *Lição 3 – Classificando dados obtidos*, e a *Lição 4 – Filtrando dados*. (As lições de 2 a 14 cobrem vários aspectos da instrução SELECT).

UPDATE

UPDATE atualiza uma ou mais linhas em uma tabela.

Consulte a *Lição 16 – Atualizando e excluindo dados*.