

Aborda as versões 11g, 10g, 9i e 8i

# Oracle Database 11g SQL

Domine SQL e PL/SQL no banco de dados Oracle

**Jason Price**

Desenvolvedor de Aplicações e Administrador  
de Banco de Dados Oracle Certified Professional



### O Autor

**Jason Price** é consultor freelancer e ex-gerente de produto da Oracle Corporation. Ele colaborou no desenvolvimento de diversos produtos Oracle, incluindo o banco de dados, o servidor de aplicações e inúmeros aplicativos CRM. Price é Administrador de Banco de Dados Oracle Certified Professional e Desenvolvedor de Aplicações, e tem mais de 15 anos de experiência na indústria de software. Bacharel em Física pela University of Bristol, Inglaterra, Price escreveu livros sobre Oracle, Java e .NET.

### O Editor Técnico

**Scott Mikolaitis** é arquiteto de aplicativos na Oracle Corporation e trabalha na Oracle há dez anos. Ele desenvolve protótipos e padrões para a tecnologia SOA na equipe Oracle Fusion. Mikolaitis gosta de trabalhar com web services em Java e com Jabber para padrões de interação entre sistemas e entre pessoas. Scott ocupa seu tempo livre com reformas na casa e carros de controle remoto.



P945o Price, Jason.

Oracle Database 11g SQL [recurso eletrônico] / Jason Price; tradução João Eduardo Nóbrega Tortello. – Dados eletrônicos – Porto Alegre : Bookman, 2009.

Editado também como livro impresso em 2009.

ISBN 978-85-7780-437-5

1. Base de dados. 2. Oracle. 3. Linguagem-padrão de consultas (SQL). I. Título.

CDU 004.655.3

**Jason Price**

# **Oracle Database 11g SQL**

**Tradução:**

João Eduardo Nóbrega Tortello

**Consultoria, supervisão e revisão técnica desta edição:**

Denis Dias de Souza Abrantes

Bacharel em Ciências da Computação pela UNISANTA – SP

Profissional Certificado em Oracle Application Server 10g

Consultor de Vendas da Oracle no Brasil

Versão impressa  
desta obra: 2009



2009

Obra originalmente publicada sob o título  
*Oracle Database 11g SQL*

ISBN 978-0-07-149850-0

Copyright © 2008 by the McGraw-Hill Companies, Inc.

Capa: *Gustavo Demarchi*

Leitura final: *Vinícius Selbach*

Supervisão editorial: *Elisa Viali*

Editoração eletrônica: *Techbooks*

Oracle é marca registrada da Oracle Corporation e/ou suas afiliadas. Todas as outras marcas registradas são propriedade de seus donos.

As capturas de tela de softwares registrados da Oracle foram reproduzidas neste livro com permissão da Oracle Corporation e/ou de suas afiliadas.

Reservados todos os direitos de publicação, em língua portuguesa, à  
ARTMED® EDITORA S.A.  
(BOOKMAN® COMPANHIA EDITORA é uma divisão da ARTMED® EDITORA S.A.)  
Av. Jerônimo de Ornelas, 670 - Santana  
90040-340 Porto Alegre RS  
Fone (51) 3027-7000 Fax (51) 3027-7070

É proibida a duplicação ou reprodução deste volume, no todo ou em parte,  
sob quaisquer formas ou por quaisquer meios (eletrônico, mecânico, gravação,  
fotocópia, distribuição na Web e outros), sem permissão expressa da Editora.

SÃO PAULO  
Av. Angélica, 1.091 - Higienópolis  
01227-100 São Paulo SP  
Fone (11) 3665-1100 Fax (11) 3667-1333

SAC 0800 703-3444

IMPRESSO NO BRASIL  
*PRINTED IN BRAZIL*

Este livro é dedicado à minha família.  
Mesmo longe, vocês estão no meu coração.

Obrigado à maravilhosa equipe da McGraw-Hill, incluindo  
Lisa McClain, Mandy Canales, Carl Wikander e Laura Stone.  
Obrigado também a Scott Mikolaitis pela sua detalhada  
revisão técnica.



# Prefácio

Os sistemas de gerenciamento de banco de dados atuais são acessados por meio de uma linguagem padrão conhecida como *Structured Query Language* ou SQL. A linguagem SQL permite recuperar, adicionar, atualizar e excluir informações em um banco de dados. Neste livro, você vai aprender a dominar a linguagem SQL com muitos exemplos práticos. É possível obter online todos os scripts e programas apresentados neste livro (consulte a última seção da introdução, “Para obter os exemplos”, para mais detalhes).

Com este livro, você vai:

- Dominar a linguagem SQL padrão, assim como as extensões desenvolvidas pela Oracle Corporation para uso com os recursos específicos do banco de dados Oracle.
- Explorar o PL/SQL (Procedural Language/SQL), que é baseado na linguagem SQL e permite escrever programas contendo instruções SQL.
- Usar o SQL\*Plus para executar instruções SQL, scripts e relatórios. O SQL\*Plus é uma ferramenta que permite interagir com o banco de dados.
- Executar consultas, inserções, atualizações e exclusões em um banco de dados.
- Criar tabelas de banco de dados, seqüências, índices, visões e usuários.
- Realizar transações contendo várias instruções SQL.
- Definir tipos de objeto de banco de dados e criar tabelas de objeto para manipular dados avançados.
- Usar LOBs (Large Objects) para manipular arquivos multimídia contendo imagens, músicas e filmes.
- Efetuar cálculos complexos usando funções analíticas.

- Usar todos os recursos mais recentes do Oracle Database 11g, como PIVOT e UNPIVOT, arquivos de flashback e muito mais.
- Implementar técnicas de ajuste de alto desempenho para aumentar a performance das suas instruções SQL.
- Escrever programas em Java para acessar um banco de dados Oracle usando JDBC.
- Explorar os recursos XML do banco de dados Oracle.

Este livro contém 17 capítulos e um apêndice.

## **Capítulo 1: Introdução**

Neste capítulo, você irá aprender sobre bancos de dados relacionais, conhecer a linguagem SQL, ver algumas consultas simples, usar o SQL\*Plus e o SQL Developer para executar consultas e ver brevemente o PL/SQL.

## **Capítulo 2: Recuperando informações de tabelas de banco de dados**

Você irá explorar o modo de recuperar informações de uma ou mais tabelas de banco de dados usando instruções `SELECT`, usar expressões aritméticas para efetuar cálculos, filtrar linhas usando uma cláusula `WHERE` e classificar as linhas recuperadas de uma tabela.

## **Capítulo 3: Usando o SQL\*Plus**

Neste capítulo, você usará o SQL\*Plus para ver a estrutura de uma tabela, editar uma instrução SQL, salvar e executar scripts, formatar saída de coluna, definir e usar variáveis e criar relatórios.

## **Capítulo 4: Usando funções simples**

Neste capítulo, você irá aprender a respeito de algumas das funções internas do banco de dados Oracle. Uma função pode aceitar parâmetros de entrada e retornar um parâmetro de saída. As funções permitem executar tarefas como calcular médias e raízes quadradas de números.

## **Capítulo 5: Armazenando e processando datas e horas**

Você irá aprender como o banco de dados Oracle processa e armazena datas e horas, coletivamente conhecidas como data/horários (datetimes). Você também irá aprender sobre timestamps, que permitem armazenar uma data e hora específica, e sobre intervalos de tempo, que permitem armazenar um período de tempo.

## **Capítulo 6: Subconsultas**

Você irá aprender a colocar uma instrução `SELECT` dentro de uma instrução SQL externa. A instrução `SELECT` interna é conhecida como subconsulta. Você irá conhecer os diferentes tipos de subconsultas e ver como eles permitem construir instruções muito complexas a partir de componentes simples.

## **Capítulo 7: Consultas avançadas**

Neste capítulo, você irá aprender a executar consultas contendo funções e operadores avançados, como: operadores de conjunto que combinam linhas retornadas por várias consultas, a função `TRANSLATE()` para converter caracteres de uma string nos caracteres de outra, a função `DECODE()` para procurar determinado valor em um conjunto de valores, a expressão `CASE` para executar



lógica if-then-else e as cláusulas ROLLUP e CUBE para retornar linhas contendo subtotais. Você irá conhecer as funções analíticas que permitem efetuar cálculos complexos, como encontrar o tipo de produto mais vendido para cada mês, os vendedores que se destacam etc. Você irá aprender a executar consultas em dados organizados em uma hierarquia. Você também irá explorar a cláusula MODEL, que efetua cálculos dentro de uma linha. Por fim, você irá estudar as novas cláusulas PIVOT e UNPIVOT do Oracle Database 11g, que são úteis para ver tendências globais em grandes volumes de dados.

## **Capítulo 8: Alterando o conteúdo da tabela**

Você irá aprender a adicionar, modificar e remover linhas usando as instruções INSERT, UPDATE e DELETE e a tornar os resultados de suas transações permanentes usando a instrução COMMIT ou desfazer seus resultados inteiramente usando a instrução ROLLBACK. Você também saberá como um banco de dados Oracle pode processar várias transações ao mesmo tempo.

## **Capítulo 9: Usuários, privilégios e atribuições**

Neste capítulo, você irá aprender sobre usuários de banco de dados e ver como privilégios e atribuições são usados para permitir que eles executem tarefas específicas no banco de dados.

## **Capítulo 10: Criando tabelas, seqüências, índices e visões**

Você irá aprender sobre tabelas e seqüências, as quais geram uma série de números, e índices que atuam como um índice de um livro permitindo acessar linhas rapidamente. Você também irá aprender sobre as visões, que são consultas pré-definidas sobre uma ou mais tabelas; dentre outras vantagens, as visões permitem ocultar a complexidade do usuário e implementam uma camada adicional de segurança, permitindo que uma visão acesse somente um conjunto limitado de dados das tabelas. Você também irá examinar os arquivos de dados de flashback, uma novidade do Oracle Database 11g. Um arquivo de dados de flashback armazena as alterações feitas em uma tabela durante um período de tempo.

## **Capítulo 11: Introdução à programação em PL/SQL**

Neste capítulo, você irá explorar o PL/SQL, que é baseado na linguagem SQL e permite escrever programas armazenados no banco de dados contendo instruções SQL. O PL/SQL contém construções de programação padrão.

## **Capítulo 12: Objetos de banco de dados**

Você irá aprender a criar tipos de objetos de banco de dados, que podem conter atributos e métodos. Você usará tipos de objetos para definir objetos de coluna e tabelas de objetos e verá como manipular objetos usando SQL e PL/SQL.

## **Capítulo 13: Coleções**

Neste capítulo, você irá aprender a criar tipos de coleção, que podem conter vários elementos. Você irá usar tipos de coleções para definir colunas em tabelas além de entender como manipular coleções usando SQL e PL/SQL.

## **Capítulo 14: Large Objects (LOBs)**

Você irá aprender sobre large objects, que podem ser usados para armazenar até 128 terabytes de caracteres e dados binários ou apontar para um arquivo externo. Você também irá aprender sobre o tipo LONG, que ainda é suportado no Oracle Database 11g para compatibilidade com versões anteriores.

## Capítulo 15: Executando SQL usando Java

Neste capítulo, você irá aprender os fundamentos da execução de SQL usando Java por meio da interface de programação de aplicativos JDBC (Java Database Connectivity), que é a “cola” que permite a um programa Java acessar um banco de dados.

## Capítulo 16: Ajuste de SQL

Você irá receber dicas de ajuste de SQL para reduzir o tempo de execução de suas consultas. Você também irá aprender sobre o otimizador Oracle e vai ver como passar dicas para ele.

## Capítulo 17: XML e o banco de dados Oracle

A XML (Extensible Markup Language) é uma linguagem de marcação de propósito geral. Ela permite compartilhar dados estruturados pela Internet e pode ser usada para codificar dados e outros documentos. Neste capítulo, você irá aprender a gerar código XML a partir de dados relacionais e salvar dados XML no banco de dados.

## Apêndice: Tipos de dados do Oracle

Este apêndice mostra os tipos de dados disponíveis na linguagem SQL e no PL/SQL do Oracle.

## PÚBLICO-ALVO

Este livro é adequado para os seguintes leitores:

- Desenvolvedores que precisam escrever código em SQL e PL/SQL.
- Administradores de banco de dados que precisam de conhecimento aprofundado de SQL.
- Usuários de empresas que precisam escrever consultas em SQL para obter informações do banco de dados de suas organizações.
- Gerentes ou consultores técnicos que precisam de uma introdução ao SQL e PL/SQL.

Não é necessário conhecimento prévio de banco de dados Oracle, SQL ou PL/SQL; neste livro, você vai encontrar tudo o que precisa saber para se tornar um mestre.

## PARA OBTER OS EXEMPLOS

Todos os scripts SQL, programas e outros arquivos usados neste livro podem ser baixados do site da Oracle Press, no endereço [www.OraclePressBooks.com](http://www.OraclePressBooks.com). Eles estão contidos em um arquivo Zip. Depois de baixar esse arquivo, você precisará extrair seu conteúdo. Isso criará um diretório chamado `sql_book` contendo os seguintes subdiretórios:

- **Java**    Contém os programas Java usados no Capítulo 15.
- **sample\_files**    Contém os arquivos de exemplo usados no Capítulo 14.
- **SQL**    Contém os scripts SQL usados em todo o livro, incluindo os scripts para criar e preencher as tabelas de banco de dados de exemplo.
- **xml\_files**    Contém o código XML usado no Capítulo 17.

Esperamos que você goste deste livro!

# Sumário Resumido

1	Introdução .....	29
2	Recuperando informações de tabelas de banco de dados.....	55
3	Usando o SQL*Plus.....	91
4	Usando funções simples .....	117
5	Armazenando e processando datas e horas .....	157
6	Subconsultas.....	195
7	Consultas avançadas.....	211
8	Alterando o conteúdo de tabelas .....	279
9	Usuários, privilégios e atribuições .....	303
10	Criando tabelas, seqüências, índices e visões.....	327
11	Introdução à programação PL/SQL .....	367
12	Objetos de banco de dados .....	407
13	Coleções.....	455
14	Large objects (objetos grandes) .....	503
15	Executando SQL usando Java .....	559
16	Ajuste de SQL.....	607
17	XML e o banco de dados Oracle .....	631
	Apêndice: Tipos de dados Oracle.....	663
	Índice .....	667

# Sumário

<b>1 Introdução .....</b>	<b>29</b>
O que é um banco de dados relacional? .....	30
Apresentando a linguagem SQL (Structured Query Language).....	31
Usando o SQL*Plus .....	32
Iniciando o SQL*Plus .....	32
Iniciando o SQL*Plus a partir da linha de comando.....	34
Executando uma instrução SELECT usando o SQL*Plus .....	34
SQL Developer .....	35
Criando o esquema da loja .....	38
Executando o script SQL*Plus para criar o esquema da loja.....	38
Instruções DDL (Data Definition Language) usadas para criar o esquema da loja.....	39
Adicionando, modificando e removendo linhas.....	48
Adicionando uma linha em uma tabela .....	48
Modificando uma linha existente em uma tabela.....	50
Removendo uma linha de uma tabela.....	50
Os tipos BINARY_FLOAT e BINARY_DOUBLE.....	51
Vantagens de BINARY_FLOAT e BINARY_DOUBLE .....	51
Usando BINARY_FLOAT e BINARY_DOUBLE em uma tabela .....	52
Valores especiais .....	52
Saindo do SQL*Plus.....	53
Introdução ao PL/SQL da Oracle.....	53
Resumo .....	54

<b>2 Recuperando informações de tabelas de banco de dados.....</b>	<b>55</b>
Executando instruções SELECT em uma única tabela .....	56
Recuperando todas as colunas de uma tabela .....	57
Especificando as linhas a serem recuperadas usando a cláusula WHERE.....	57
Identificadores de linha .....	58
Números de linha .....	58
Efetuando cálculos aritméticos.....	59
Efetuando aritmética de data .....	59
Usando colunas na aritmética .....	60
Usando apelidos de coluna.....	62
Combinando saída de coluna usando concatenação.....	63
Valores nulos .....	63
Exibindo linhas distintas .....	65
Comparando valores.....	65
Usando os operadores SQL.....	67
Usando o operador LIKE .....	68
Usando o operador IN.....	69
Usando o operador BETWEEN.....	70
Usando os operadores lógicos .....	71
Precedência de operadores .....	72
Classificando linhas usando a cláusula ORDER BY .....	72
Executando instruções SELECT que usam duas tabelas .....	73
Usando apelidos de tabela.....	75
Produtos cartesianos .....	76
Executando instruções SELECT que usam mais de duas tabelas .....	77
Condições de join e tipos de join .....	78
Não-equijoins .....	78
Joins externas .....	79
Autojoins.....	83
Realizando joins usando a sintaxe SQL/92.....	84
Realizando joins internas em duas tabelas usando SQL/92 .....	84
Simplificando joins com a palavra-chave USING.....	85
Realizando joins internas em mais de duas tabelas usando SQL/92 .....	86
Realizando joins internas em várias colunas usando SQL/92 .....	86
Realizando joins externas usando SQL/92 .....	87
Realizando autojoin usando SQL/92.....	88
Realizando join cruzada usando SQL/92 .....	89
Resumo .....	89
<b>3 Usando o SQL*Plus.....</b>	<b>91</b>
Exibindo a estrutura de uma tabela .....	92
Editando instruções SQL .....	93
Salvando, recuperando e executando arquivos .....	94
Formatando colunas .....	98
Definindo o tamanho da página.....	100

Definindo o tamanho da linha .....	101
Limpando formatação de coluna.....	101
Usando variáveis .....	102
Variáveis temporárias .....	102
Variáveis definidas.....	105
Criando relatórios simples .....	107
Usando variáveis temporárias em um script.....	108
Usando variáveis definidas em um script .....	108
Passando um valor para uma variável em um script .....	109
Adicionando um cabeçalho e um rodapé .....	110
Calculando subtotais .....	111
Obtendo ajuda do SQL*Plus .....	113
Gerando instruções SQL automaticamente .....	114
Desconectando-se do banco de dados e saindo do SQL*Plus .....	114
Resumo .....	115
<b>4 Usando funções simples .....</b>	<b>117</b>
Usando funções de uma única linha .....	118
Funções de caractere.....	118
Funções numéricas.....	126
Funções de conversão .....	130
Funções de expressão regular .....	137
Usando funções agregadas .....	145
Agrupando linhas .....	148
Usando a cláusula GROUP BY para agrupar linhas .....	149
Utilização incorreta de chamadas de funções agregadas .....	152
Usando a cláusula HAVING para filtrar grupos de linhas.....	153
Usando as cláusulas WHERE e GROUP BY juntas .....	154
Usando as cláusulas WHERE, GROUP BY e HAVING juntas .....	154
Resumo .....	155
<b>5 Armazenando e processando datas e horas.....</b>	<b>157</b>
Exemplos simples de armazenamento e recuperação de datas .....	158
Convertendo data/horários com TO_CHAR() e TO_DATE() .....	159
Usando TO_CHAR() para converter uma data/horário em uma string.....	160
Usando TO_DATE() para converter uma string em uma data/horário .....	164
Configurando o formato de data padrão.....	167
Como o Oracle interpreta anos de dois dígitos.....	168
Usando o formato YY.....	168
Usando o formato RR .....	168
Usando funções de data/horário .....	170
ADD_MONTHS() .....	170
LAST_DAY().....	172
MONTHS_BETWEEN() .....	172
NEXT_DAY().....	172

ROUND() .....	173
SYSDATE .....	173
TRUNC() .....	174
Usando fusos horários .....	174
Funções de fuso horário .....	175
O fuso horário do banco de dados e o fuso horário da sessão.....	175
Obtendo diferenças de fuso horário.....	177
Obtendo nomes de fuso horário .....	177
Convertendo uma data/horário de um fuso horário para outro .....	178
Usando timestamp .....	178
Usando os tipos de timestamp .....	178
Funções de timestamp .....	182
Usando intervalos de tempo .....	187
Usando o tipo INTERVAL YEAR TO MONTH .....	188
Usando o tipo INTERVAL DAY TO SECOND.....	190
Funções de intervalo de tempo .....	192
Resumo .....	194
<b>6 Subconsultas.....</b>	<b>195</b>
Tipos de subconsultas .....	196
Escrevendo subconsultas de uma única linha.....	196
Subconsultas em uma cláusula WHERE .....	196
Usando outros operadores de uma única linha .....	197
Subconsultas em uma cláusula HAVING .....	198
Subconsultas em uma cláusula FROM (visões inline).....	199
Erros que você pode encontrar .....	200
Escrevendo subconsultas de várias linhas.....	201
Usando IN em uma subconsulta de várias linhas .....	201
Usando ANY em uma subconsulta de várias linhas.....	202
Usando ALL em uma subconsulta de várias linhas .....	203
Escrevendo subconsultas de várias colunas.....	203
Escrevendo subconsultas correlacionadas .....	203
Exemplo de subconsulta correlacionada.....	204
Usando EXISTS e NOT EXISTS em uma subconsulta correlacionada .....	204
Escrevendo subconsultas aninhadas.....	207
Escrevendo instruções UPDATE e DELETE contendo subconsultas .....	208
Escrevendo uma instrução UPDATE contendo uma subconsulta .....	208
Escrevendo uma instrução DELETE contendo uma subconsulta .....	209
Resumo .....	209
<b>7 Consultas avançadas.....</b>	<b>211</b>
Usando os operadores de conjunto.....	212
As tabelas de exemplo.....	213
Usando o operador UNION ALL .....	214
Usando o operador UNION .....	215
Usando o operador INTERSECT .....	216

Usando o operador MINUS .....	216
Combinando operadores de conjunto .....	216
Usando a função TRANSLATE() .....	218
Usando a função DECODE() .....	219
Usando a expressão CASE .....	221
Usando expressões CASE simples .....	221
Usando expressões CASE pesquisadas .....	222
Consultas hierárquicas .....	224
Os dados de exemplo .....	224
Usando as cláusulas CONNECT BY e START WITH .....	226
Usando a pseudocoluna LEVEL .....	226
Formatando os resultados de uma consulta hierárquica .....	227
Começando em um nó que não é o raiz .....	228
Usando uma subconsulta em uma cláusula START WITH .....	228
Percorrendo a árvore para cima .....	229
Eliminando nós e ramos de uma consulta hierárquica .....	229
Incluindo outras condições em uma consulta hierárquica .....	230
Usando as cláusulas GROUP BY estendidas .....	231
As tabelas de exemplo .....	231
Usando a cláusula ROLLUP .....	233
Usando a cláusula CUBE .....	235
Usando a função GROUPING() .....	237
Usando a cláusula GROUPING SETS .....	239
Usando a função GROUPING_ID() .....	240
Usando uma coluna várias vezes em uma cláusula GROUP BY .....	242
Usando a função GROUP_ID() .....	243
Usando as funções analíticas .....	244
A tabela de exemplo .....	244
Usando as funções de classificação .....	245
Usando as funções de percentil inversas .....	252
Usando as funções de janela .....	253
Usando as funções de relatório .....	258
Usando as funções LAG() e LEAD() .....	260
Usando as funções FIRST e LAST .....	261
Usando as funções de regressão linear .....	261
Usando as funções de classificação hipotética e distribuição .....	263
Usando a cláusula MODEL .....	264
Um exemplo da cláusula MODEL .....	264
Usando notação posicional e simbólica para acessar células .....	265
Acessando um intervalo de células com BETWEEN e AND .....	266
Acessando todas as células com ANY e IS ANY .....	266
Obtendo o valor atual de uma dimensão com CURRENTV() .....	267
Acessando células com um loop FOR .....	268
Tratando de valores nulos e ausentes .....	269
Atualizando células existentes .....	271



Usando as cláusulas PIVOT e UNPIVOT .....	272
Um exemplo simples da cláusula PIVOT .....	272
Usando pivô em várias colunas .....	274
Usando várias funções agregadas em um pivô.....	275
Usando a cláusula UNPIVOT .....	276
Resumo .....	277
<b>8 Alterando o conteúdo de tabelas .....</b>	<b>279</b>
Adicionando linhas com a instrução INSERT .....	280
Omitindo a lista de colunas .....	281
Especificando um valor nulo para uma coluna .....	281
Incluindo apóstrofes e aspas em um valor de coluna .....	282
Copiando linhas de uma tabela para outra.....	282
Modificando linhas com a instrução UPDATE.....	282
A cláusula RETURNING .....	283
Removendo linhas com a instrução DELETE.....	284
Integridade do banco de dados.....	284
Aplicação das restrições de chave primária.....	285
Aplicação das restrições de chave estrangeira.....	285
Usando valores padrão .....	286
Mesclando linhas com MERGE .....	287
Transações de banco de dados.....	290
Confirmando e revertendo uma transação .....	290
Iniciando e terminando uma transação .....	291
Savepoints (pontos de salvamento) .....	292
Propriedades de transação ACID .....	293
Transações concorrentes.....	294
Bloqueio de transação .....	294
Níveis de isolamento de transação.....	295
Exemplo de transação SERIALIZABLE .....	297
Consultas Flashback .....	298
Concedendo o privilégio de usar flashbacks .....	298
Consultas flashback de tempo .....	298
Consultas flashback com número de alteração de sistema .....	300
Resumo .....	301
<b>9 Usuários, privilégios e atribuições .....</b>	<b>303</b>
Usuários .....	304
Criando um usuário.....	304
Alterando a senha de um usuário.....	305
Excluindo um usuário.....	306
Privilégios de sistema.....	306
Concedendo privilégio de sistema a um usuário.....	306
Verificando os privilégios de sistema concedidos a um usuário.....	308
Utilizando privilégios de sistema .....	309
Revogando privilégios de sistema de um usuário .....	309

Privilégios de objeto .....	309
Concedendo privilégios de objeto a um usuário .....	310
Verificando os privilégios de objeto concedidos .....	311
Verificando os privilégios de objeto recebidos .....	312
Utilizando privilégios de objeto.....	314
Sinônimos .....	315
Sinônimos públicos .....	315
Revogando privilégios de objeto.....	316
Atribuições (Roles).....	317
Criando atribuições .....	317
Concedendo privilégios a atribuições .....	318
Concedendo atribuições a um usuário.....	318
Verificando as atribuições concedidas a um usuário .....	318
Verificando os privilégios de sistema concedidos a uma atribuição.....	319
Verificando os privilégios de objeto concedidos a uma atribuição .....	320
Utilizando os privilégios concedidos a uma atribuição .....	321
Atribuições padrão .....	322
Revogando uma atribuição.....	322
Revogando privilégios de uma atribuição .....	322
Excluindo uma atribuição.....	323
Auditoria .....	323
Privilégios necessários para fazer auditoria .....	323
Exemplos de auditoria .....	323
Visões de trilha de auditoria .....	325
Resumo .....	325
<b>10 Criando tabelas, seqüências, índices e visões.....</b>	<b>327</b>
Tabelas .....	328
Criando uma tabela .....	328
Obtendo informações sobre tabelas.....	330
Obtendo informações sobre colunas nas tabelas .....	331
Alterando uma tabela .....	331
Mudando o nome de uma tabela .....	341
Adicionando um comentário em uma tabela .....	341
Truncando uma tabela.....	342
Excluindo uma tabela .....	342
Seqüências .....	342
Criando uma seqüência.....	342
Recuperando informações sobre seqüências.....	344
Usando uma seqüência .....	345
Preenchendo uma chave primária usando uma seqüência .....	347
Modificando uma seqüência .....	348
Excluindo uma seqüência.....	348
Índices.....	348
Criando um índice de árvore B .....	349

Criando um índice baseado em função .....	350
Recuperando informações sobre índices.....	351
Recuperando informações sobre índices em uma coluna.....	351
Modificando um índice .....	352
Excluindo um índice.....	352
Criando um índice de bitmap .....	352
Visões .....	353
Criando e usando uma visão .....	354
Modificando uma visão.....	361
Excluindo uma visão .....	362
Arquivos de Dados de Flashback .....	362
Resumo .....	365
<b>11 Introdução à programação PL/SQL .....</b>	<b>367</b>
Estrutura de bloco.....	368
Variáveis e tipos.....	370
Lógica condicional .....	370
Loops.....	371
Loops simples.....	371
Loops WHILE .....	372
Loops FOR .....	372
Cursors .....	373
Passo 1: Declarar as variáveis para armazenar os valores de coluna.....	374
Passo 2: Declarar o cursor .....	374
Passo 3: Abrir o cursor .....	374
Passo 4: Buscar as linhas do cursor.....	375
Passo 5: Fechar o cursor.....	375
Exemplo completo: product_cursor.sql .....	376
Cursors e loops FOR.....	377
Instrução OPEN-FOR .....	378
Cursors irrestritos.....	380
Exceções.....	381
Exceção ZERO_DIVIDE.....	383
Exceção DUP_VAL_ON_INDEX.....	384
Exceção INVALID_NUMBER.....	384
Exceção OTHERS .....	385
Procedures.....	386
Criando uma procedure.....	386
Chamando uma procedure.....	388
Obtendo informações sobre procedures .....	389
Excluindo uma procedure.....	390
Vendo erros em uma procedure.....	390
Funções.....	391
Criando uma função.....	391
Chamando uma função .....	392

Obtendo informações sobre funções .....	393
Excluindo uma função.....	393
Pacotes (Packages).....	393
Criando uma especificação de pacote .....	393
Criando o corpo de um pacote .....	394
Chamando funções e procedures em um pacote.....	395
Obtendo informações sobre funções e procedures em um pacote .....	396
Excluindo um pacote.....	396
Triggers.....	397
Quando um trigger é disparado .....	397
Configuração do trigger de exemplo .....	397
Criando um trigger .....	397
Disparando um trigger.....	399
Obtendo informações sobre triggers .....	400
Desativando e ativando um trigger.....	402
Excluindo um trigger .....	402
Novos recursos PL/SQL no Oracle Database 11g .....	402
Tipo SIMPLE_INTEGER.....	403
Seqüências em PL/SQL.....	403
Geração de código de máquina nativo PL/SQL .....	405
Resumo .....	405
<b>12 Objetos de banco de dados .....</b>	<b>407</b>
Introdução aos objetos.....	408
Criando tipos de objeto.....	409
Usando DESCRIBE para obter informações sobre tipos de objeto .....	410
Usando tipos de objeto em tabelas de banco de dados .....	411
Objetos de coluna.....	411
Tabelas de objeto .....	414
Identificadores de objeto e referências de objeto .....	418
Comparando valores de objeto .....	420
Usando objetos em PL/SQL.....	422
A função get_products().....	423
A procedure display_product().....	424
A procedure insert_product() .....	425
A procedure update_product_price() .....	426
A função get_product() .....	426
A procedure update_product().....	427
A função get_product_ref() .....	428
A procedure delete_product() .....	428
A procedure product_lifecycle().....	429
A procedure product_lifecycle2().....	430
Herança de tipo .....	431
Usando um objeto de subtipo no lugar de um objeto de supertipo.....	433
Exemplos em SQL .....	433

Exemplos em PL/SQL.....	434
Objetos NOT SUBSTITUTABLE .....	435
Outras funções de objeto úteis.....	436
IS OF() .....	436
TREAT() .....	440
SYS_TYPEID() .....	444
Tipos de objeto NOT INSTANTIABLE.....	444
Construtores definidos pelo usuário .....	446
Sobrescrevendo métodos .....	450
Invocação generalizada .....	451
Resumo .....	453
<b>13 Coleções.....</b>	<b>455</b>
Introdução às coleções .....	456
Criando tipos de coleção .....	457
Criando um tipo de varray .....	457
Criando um tipo de tabela aninhada.....	457
Usando um tipo de coleção para definir uma coluna em uma tabela .....	458
Usando um tipo de varray para definir uma coluna em uma tabela.....	458
Usando um tipo de tabela aninhada para definir uma coluna em uma tabela .....	458
Obtendo informações sobre coleções .....	459
Obtendo informações sobre um varray .....	459
Obtendo informações sobre uma tabela aninhada .....	460
Preenchendo uma coleção com elementos .....	462
Preenchendo um varray com elementos .....	462
Preenchendo uma tabela aninhada com elementos .....	462
Recuperando elementos de coleções .....	463
Recuperando elementos de um varray .....	463
Recuperando elementos de uma tabela aninhada .....	464
Usando TABLE() para tratar uma coleção como uma série de linhas .....	464
Usando TABLE() com um varray.....	465
Usando TABLE() com uma tabela aninhada.....	466
Modificando elementos de coleções .....	466
Modificando elementos de um varray .....	466
Modificando elementos de uma tabela aninhada.....	467
Usando um método de mapeamento para comparar o conteúdo de tabelas aninhadas .....	468
Usando CAST() para converter coleções de um tipo para outro .....	471
Usando CAST() para converter um varray em uma tabela aninhada .....	471
Usando CAST() para converter uma tabela aninhada em um varray .....	471
Usando coleções em PL/SQL .....	472
Manipulando um varray .....	472
Manipulando uma tabela aninhada .....	474
Métodos de coleção PL/SQL.....	476
Coleções de múltiplos níveis.....	486
Aprimoramentos feitos nas coleções pelo Oracle Database 10g.....	489

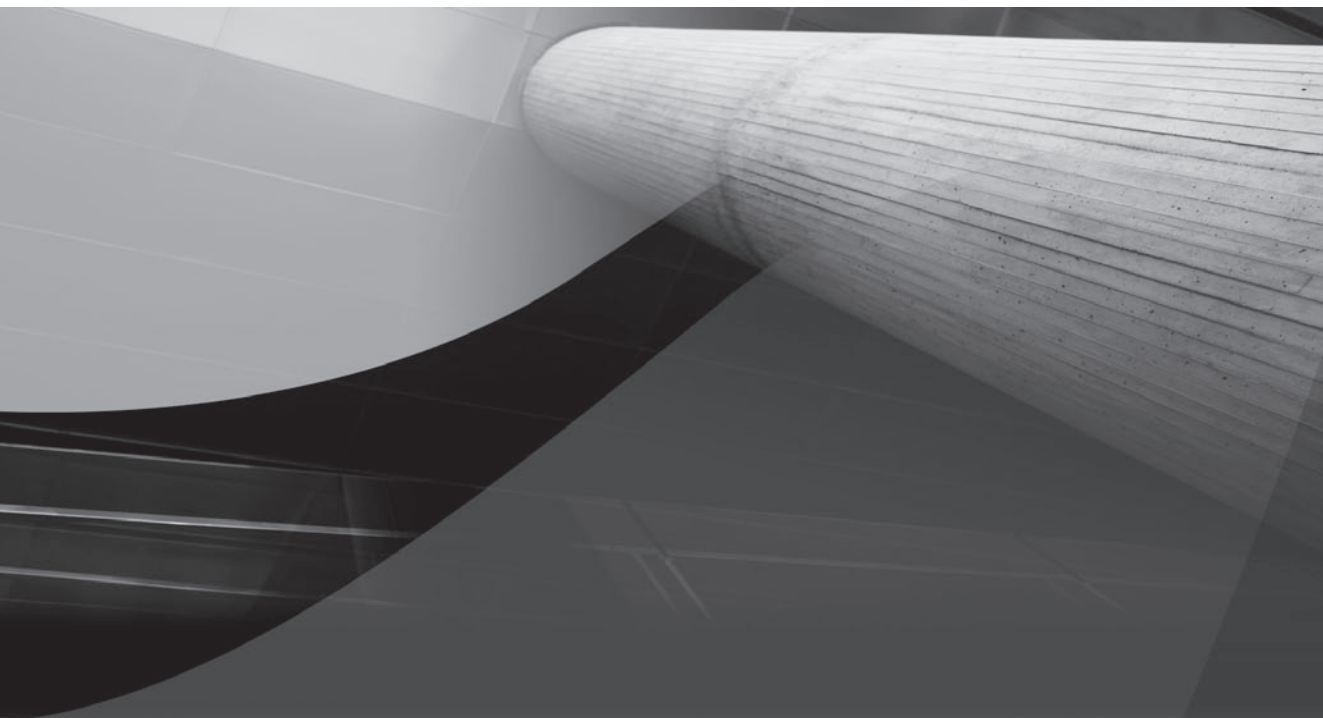
Arrays associativos.....	490
Alterando o tamanho de um tipo de elemento .....	491
Aumentando o número de elementos em um varray .....	491
Usando varrays em tabelas temporárias .....	491
Usando um tablespace diferente para a tabela de armazenamento de uma tabela aninhada.....	491
Suporte ANSI para tabelas aninhadas.....	492
Resumo .....	501
<b>14 Large objects (objetos grandes) .....</b>	<b>503</b>
Introdução aos large objects (LOBs) .....	504
Os arquivos de exemplo .....	504
Tipos de large object.....	505
Criando tabelas contendo large objects.....	506
Usando large objects em SQL .....	506
Usando CLOBs e BLOBs .....	506
Usando BFILEs .....	509
Usando large objects em PL/SQL .....	510
APPEND().....	513
CLOSE().....	513
COMPARE().....	514
COPY().....	515
CREATETEMPORARY() .....	516
ERASE() .....	516
FILECLOSE() .....	517
FILECLOSEALL() .....	517
FILEEXISTS() .....	517
FILEGETNAME() .....	518
FILEISOPEN().....	518
FILEOPEN() .....	519
FREETEMPORARY() .....	520
GETCHUNKSIZE() .....	520
GET_STORAGE_LIMIT().....	520
GETLENGTH().....	521
INSTR().....	521
ISOPEN() .....	522
ISTEMPORARY().....	523
LOADFROMFILE() .....	524
LOADBLOBFROMFILE().....	525
LOADCLOBFROMFILE().....	525
OPEN() .....	526
READ() .....	527
SUBSTR().....	528
TRIM() .....	529
WRITE().....	530

WRITEAPPEND() .....	531
Exemplos de procedures em PL/SQL .....	531
Tipos LONG e LONG RAW .....	549
As tabelas de exemplo .....	549
Adicionando dados em colunas LONG e LONG RAW .....	549
Convertendo colunas LONG e LONG RAW em LOBs .....	550
Aprimoramentos feitos pelo Oracle Database 10g nos large objects .....	551
Conversão implícita entre objetos CLOB e NCLOB .....	551
Uso do atributo :new ao utilizar LOBs em um trigger .....	552
Aprimoramentos feitos pelo Oracle Database 11g nos large objects .....	553
Criptografia de dados de LOB .....	553
Compactando dados de LOB .....	557
Removendo dados de LOB duplicados .....	557
Resumo .....	557
<b>15 Executando SQL usando Java .....</b>	<b>559</b>
Começando .....	560
Configurando seu computador .....	561
Configurando a variável de ambiente ORACLE_HOME .....	561
Configurando a variável de ambiente JAVA_HOME .....	562
Configurando a variável de ambiente PATH .....	562
Configurando a variável de ambiente CLASSPATH .....	562
Configurando a variável de ambiente LD_LIBRARY_PATH .....	563
Os drivers JDBC da Oracle .....	563
O driver Thin .....	563
O driver OCI .....	564
O driver interno server-side .....	564
O driver Thin server-side .....	564
Importando pacotes JDBC .....	564
Registrando os drivers JDBC da Oracle .....	565
Abrindo uma conexão de banco de dados .....	565
Conectando-se no banco de dados com getConnection() .....	565
A URL do banco de dados .....	566
Conectando-se com o banco de dados usando uma origem de dados Oracle .....	567
Criando um objeto JDBC Statement .....	570
Recuperando linhas do banco de dados .....	571
Passo 1: Criar e preencher um objeto ResultSet .....	571
Passo 2: Ler os valores de coluna do objeto ResultSet .....	572
Passo 3: Fechar o objeto ResultSet .....	574
Adicionando linhas no banco de dados .....	575
Modificando linhas no banco de dados .....	576
Excluindo linhas do banco de dados .....	576
Manipulando números .....	577
Manipulando valores nulos no banco de dados .....	578
Controlando transações de banco de dados .....	580

Executando instruções Data Definition Language.....	581
Tratamento de exceções.....	581
Fechando seus objetos JDBC.....	582
Exemplo de programa: BasicExample1.java .....	584
Compilando BasicExample1 .....	588
Executando BasicExample1 .....	589
SQL Prepared Statements.....	590
Exemplo de programa: BasicExample2.java .....	593
As extensões da Oracle para JDBC.....	595
O pacote oracle.sql .....	596
O pacote oracle.jdbc.....	599
Exemplo de programa: BasicExample3.java .....	603
Resumo .....	606
<b>16 Ajuste de SQL .....</b>	<b>607</b>
Introdução ao ajuste de SQL .....	608
Use uma cláusula WHERE para filtrar linhas .....	608
Use joins de tabela em vez de várias consultas .....	609
Use referências de coluna totalmente qualificadas ao fazer joins .....	610
Use expressões CASE em vez de várias consultas .....	611
Adicione índices nas tabelas.....	612
Use WHERE em vez de HAVING .....	612
Use UNION ALL em vez de UNION .....	613
Use EXISTS em vez de IN.....	614
Use EXISTS em vez de DISTINCT.....	615
Use GROUPING SETS em vez de CUBE .....	616
Use variáveis de bind.....	616
Instruções SQL não idênticas.....	616
Instruções SQL idênticas que usam variáveis de bind.....	616
Listando e imprimindo variáveis de bind .....	618
Usando uma variável de bind para armazenar um valor retornado por uma função PL/SQL .....	618
Usando uma variável de bind para armazenar linhas de um REFCURSOR.....	618
Comparando o custo da execução de consultas.....	619
Examinando planos de execução.....	619
Comparando planos de execução.....	625
Passando dicas para o otimizador .....	626
Ferramentas de ajuste adicionais.....	628
Oracle Enterprise Manager Diagnostics Pack .....	628
Automatic Database Diagnostic Monitor .....	628
Resumo .....	629
<b>17 XML e o banco de dados Oracle .....</b>	<b>631</b>
Introdução à XML .....	632
Gerando código XML a partir de dados relacionais.....	632
XMLELEMENT().....	633



XMLATTRIBUTES()	636
XMLFOREST()	636
XMLAGG()	637
XMLCOLATTVAL()	640
XMLCONCAT()	641
XMLPARSE()	641
XMLPI()	642
XMLCOMMENT()	642
XMLSEQUENCE()	643
XMLSERIALIZE()	644
Um exemplo em PL/SQL que grava os dados XML em um arquivo	644
XMLQUERY()	646
Salvando XML no banco de dados	650
O arquivo de exemplo XML	651
Criando o esquema de exemplo XML	651
Recuperando informações do esquema XML de exemplo	653
Atualizando informações no esquema de exemplo XML	658
Resumo	661
<b>Apêndice: Tipos de dados Oracle</b>	<b>663</b>
Tipos SQL do Oracle	664
Tipos PL/SQL do Oracle	666
<b>Índice</b>	<b>667</b>



# CAPÍTULO 1

## Introdução

Neste capítulo, você vai aprender sobre:

- Banco de dados relacionais
- A linguagem SQL (Structured Query Language), usada para acessar um banco de dados
- SQL \*Plus, a ferramenta interativa da Oracle para executar instruções SQL
- SQL Developer, uma ferramenta gráfica para desenvolvimento de banco de dados
- PL/SQL, a linguagem de programação procedural da Oracle. O PL/SQL permite desenvolver programas que são armazenados no banco de dados

Vamos começar entendendo o que é um banco de dados relacional.

## O QUE É UM BANCO DE DADOS RELACIONAL?

O conceito de banco de dados relacional foi originalmente desenvolvido em 1970 pelo Dr. E. F. Codd. Ele esboçou a teoria dos bancos de dados relacionais em seu artigo intitulado “A Relational Model of Data for Large Shared Data Banks” (Um modelo de dados relacional para grandes bancos de dados compartilhados), publicado na *Communications of the ACM* (Association for Computing Machinery), vol.13, nº6, junho de 1970.

Os conceitos básicos de um banco de dados relacional são muito fáceis de entender. Um *banco de dados relacional* é uma coleção de informações relacionadas, organizadas em *tabelas*. Cada tabela armazena dados em *linhas*; os dados são organizados em *colunas*. As tabelas são armazenadas em *esquemas* de banco de dados, que são áreas onde os usuários podem armazenar suas próprias tabelas. Um usuário pode conceder *permissões* a outros usuários para que eles possam acessar suas tabelas.

A maioria de nós conhece o armazenamento de dados em tabelas — preços de ações e horários de trem às vezes são organizados em tabelas. A tabela de exemplo usada neste livro registra informações de clientes para uma loja imaginária; a tabela armazena os nomes, sobrenomes, datas de nascimento (dob) e números de telefone dos clientes:

nome	sobrenome	dob	telefone
John	Brown	01-JAN-1965	800-555-1211
Cynthia	Green	05-FEV-1968	800-555-1212
Steve	White	16-MAR-1971	800-555-1213
Gail	Black		800-555-1214
Doreen	Blue	20-MAI-1970	

Essa tabela poderia ser armazenada de várias formas:

- Um cartão em uma caixa
- Uma página HTML na Web
- Uma tabela em um banco de dados

Um ponto importante a ser lembrado é que as informações que compõem um banco de dados são diferentes do sistema usado para acessar essas informações. O software usado para acessar um banco de dados é conhecido como *sistema de gerenciamento de banco de dados*. O banco de dados Oracle é um desses softwares; outros exemplos incluem o SQL Server, o DB2 e o MySQL.

Evidentemente, todo banco de dados precisa ter algum modo de inserir e extrair dados, de preferência usando uma linguagem comum, entendida por todos os bancos de dados. Os sistemas de gerenciamento de banco de dados implementam uma linguagem padrão conhecida como *Structured Query Language* ou SQL. Dentre outras coisas, a linguagem SQL permite recuperar, adicionar, modificar e excluir informações em um banco de dados.

## APRESENTANDO A LINGUAGEM SQL (STRUCTURED QUERY LANGUAGE)

A linguagem SQL (Structured Query Language) é a linguagem padrão projetada para acessar banco de dados relacionais. Pronuncia-se SQL soletrando as letras “S-Q-L”.



### NOTA

“S-Q-L” é a maneira correta de pronunciar SQL, de acordo com o American National Standards Institute. Contudo, em inglês, a palavra “sequel” é usada com frequência.

A linguagem SQL é baseada no trabalho pioneiro do Dr. E.F. Codd. Sua primeira implementação foi desenvolvida pela IBM em meados dos anos 1970, dentro de um projeto de pesquisa conhecido como System R. Posteriormente, em 1979, uma empresa então chamada Relational Software Inc. (hoje Oracle Corporation) lançou a primeira implementação comercial da linguagem SQL. Atualmente, a linguagem SQL está totalmente padronizada e é reconhecida pelo American National Standards Institute.

A linguagem SQL usa uma sintaxe simples, fácil de aprender e utilizar. Você vai ver alguns exemplos de sua utilização neste capítulo. Existem cinco tipos de instruções SQL, descritas a seguir:

- **Instruções de consulta** recuperam linhas armazenadas nas tabelas do banco de dados. Você escreve uma consulta usando a instrução SQL `SELECT`.
- **Instruções DML (Data Manipulation Language)** modificam o conteúdo das tabelas. Existem três instruções DML:
  - **INSERT** adiciona linhas em uma tabela.
  - **UPDATE** altera linhas.
  - **DELETE** remove linhas.
- **Instruções DDL (Data Definition Language)** definem as estruturas de dados, como as tabelas, que compõem um banco de dados. Existem cinco tipos básicos de instruções DDL:
  - **CREATE** cria uma estrutura de banco de dados. Por exemplo, `CREATE TABLE` é usada para criar uma tabela; outro exemplo é `CREATE USER`, usada para criar um usuário do banco de dados.
  - **ALTER** modifica uma estrutura de banco de dados. Por exemplo, `ALTER TABLE` é usada para modificar uma tabela.
  - **DROP** remove uma estrutura de banco de dados. Por exemplo, `DROP TABLE` é usada para remover uma tabela.
  - **RENAME** muda o nome de uma tabela.
  - **TRUNCATE** exclui todas as linhas de uma tabela.

- **Instruções TC (Transaction Control)** registram permanentemente as alterações feitas em linhas ou desfazem essas alterações. Existem três instruções TC:
  - **COMMIT** registra permanentemente as alterações feitas em linhas.
  - **ROLLBACK** desfaz as alterações feitas em linhas.
  - **SAVEPOINT** define um “ponto de salvamento” no qual você pode reverter alterações.
- **Instruções DCL (Data Control Language)** alteram as permissões nas estruturas de banco de dados. Existem duas instruções DCL:
  - **GRANT** concede a outro usuário acesso às estruturas de seu banco de dados.
  - **REVOKE** impede que outro usuário acesse as estruturas de seu banco de dados.

Existem muitas maneiras de executar instruções SQL e obter resultados do banco de dados, algumas das quais incluem programas escritos usando o Oracle Forms e Reports. As instruções SQL também podem ser incorporadas em programas escritos em outras linguagens, como Pro\*C++ da Oracle, que permite adicionar instruções SQL em um programa C++. Você também pode adicionar instruções SQL em um programa Java usando JDBC; para obter mais detalhes, consulte o livro *Oracle9i JDBC Programming* (Oracle Press, 2002).

A Oracle também tem uma ferramenta chamada SQL\*Plus, que permite inserir instruções SQL usando o teclado ou executar um script contendo instruções SQL. O SQL\*Plus possibilita “conversar” com o banco de dados; você digita instruções SQL e vê os resultados retornados pelo banco de dados. Apresentamos o SQL\*Plus a seguir.

## USANDO O SQL\*PLUS

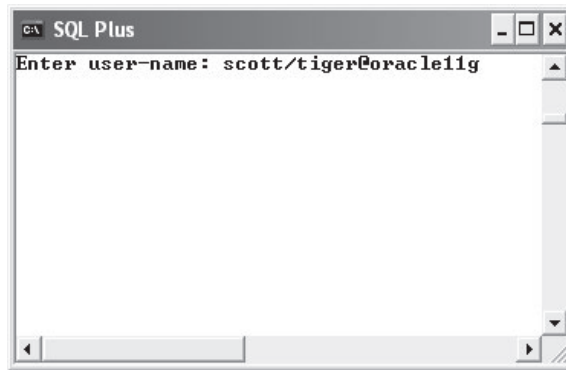
Se você conhece o banco de dados Oracle, é possível que já esteja familiarizado com o SQL\*Plus. Se não estiver, não se preocupe; neste livro você vai aprender a usá-lo.

Nas seções a seguir, você vai aprender a iniciar o SQL\*Plus e a executar uma consulta.

### Iniciando o SQL\*Plus

Se você está usando Windows XP Professional Edition e Oracle Database 11g, pode iniciar o SQL\*Plus clicando no botão Iniciar e selecionando Programas | Oracle | Application development | SQLPlus.

A Figura 1-1 mostra o SQL\*Plus em execução no Windows XP. O SQL\*Plus pede um nome de usuário. A Figura 1-1 mostra o usuário `scott` conectando-se no banco de dados (`scott` é um exemplo de usuário contido em muitas versões do banco de dados Oracle; sua senha padrão é `tiger`). A string de host após o caractere `@` informa ao SQL\*Plus onde o banco de dados está sendo executado. Se você estiver executando o banco de dados em seu próprio computador, normalmente omitirá a string de host (isto é, você digitará `scott/tiger`) — isso faz com que o SQL\*Plus tente conectar a um banco de dados na mesma máquina em que está sendo executado. Se o banco de dados não estiver sendo executado em sua máquina, fale com o administrador do banco de dados (DBA) para obter a string de host. Se o usuário `scott` não existe ou está bloqueado, peça ao DBA um usuário e uma senha alternativos (para os exemplos da primeira parte deste capítulo, você pode utilizar qualquer usuário; não é necessário utilizar o usuário `scott`).



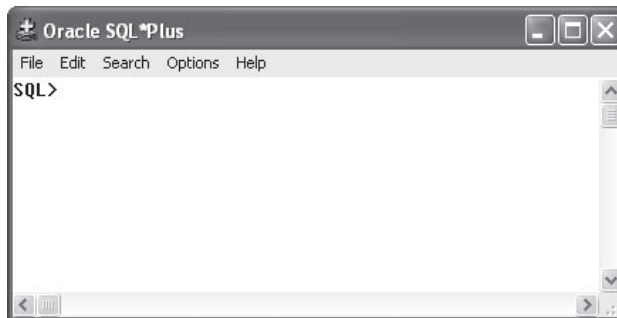
**Figura 1-1** SQL\*Plus do Oracle Database 11g em execução no Windows XP.

Se você está usando Windows XP e Oracle Database 10g ou anterior, pode executar uma versão especial do SQL\*Plus para Windows. Você inicia essa versão clicando em Iniciar e selecionando Programas | Oracle | Application Development | SQL Plus. A versão para Windows do SQL\*Plus foi descontinuada no Oracle Database 11g (isto é, ela não vem com o 11g), mas ela ainda se conectará com um banco de dados 11g. A Figura 1-2 mostra a versão para Windows do SQL\*Plus do Oracle Database 10g em execução no Windows XP.



#### NOTA

A versão do SQL\*Plus do Oracle Database 11g é ligeiramente mais refinada do que a versão para Windows. Na versão 11g, você pode percorrer os comandos executados anteriormente pressionando as teclas de seta para cima e seta para baixo no teclado.



**Figura 1-2** SQL\*Plus do Oracle Database 10g em execução no Windows XP.

## Iniciando o SQL\*Plus a partir da linha de comando

Você também pode iniciar o SQL\*Plus a partir da linha de comando. Para tanto, use o comando `sqlplus`, cuja sintaxe completa é

```
sqlplus [nome_usuario[/senha[@string_host]]]
```

onde

- *nome\_usuario* é o nome do usuário do banco de dados
- *senha* é a senha do usuário do banco de dados
- *string\_host* é o banco de dados em que você deseja se conectar

Os exemplos a seguir mostram comandos `sqlplus`:

```
sqlplus scott/tiger
sqlplus scott/tiger@orcl
```

Se você estiver usando o SQL\*Plus com um sistema operacional Windows, o instalador Oracle adicionará automaticamente o diretório do SQL\*Plus na sua variável de ambiente PATH. Se você estiver usando outro sistema operacional que não seja Windows (por exemplo, Unix ou Linux), deverá estar no mesmo diretório que o programa SQL\*Plus para executá-lo ou, melhor ainda, deverá adicionar o diretório em sua variável de ambiente PATH. Se precisar de ajuda para fazer isso, fale com seu administrador de sistema.

Por segurança, você pode ocultar a senha quando se conectar ao banco de dados. Por exemplo, você pode digitar:

```
sqlplus scott@orcl
```

Neste momento, o SQL\*Plus pedirá para que você digite a senha, que fica oculta durante a digitação. Isso também funciona ao iniciar o SQL\*Plus no Windows.

Você também pode digitar apenas

```
sqlplus
```

O SQL\*Plus pedirá o nome de usuário e a senha. Você pode especificar a string de host adicionando-a no nome de usuário (por exemplo, `scott@orcl`).

## Executando uma instrução SELECT usando o SQL\*Plus

Quando você estiver conectado ao banco de dados usando o SQL\*Plus, execute a seguinte instrução `SELECT` (ela retorna a data atual):

```
SELECT SYSDATE FROM dual;
```

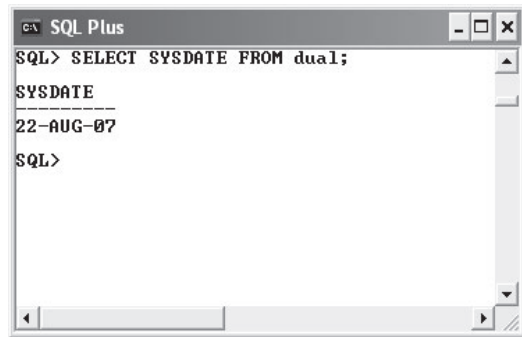
`SYSDATE` é uma função de banco de dados interna que retorna a data atual e `dual` é uma tabela que contém uma única linha. A tabela `dual` é útil quando você precisa que o banco de dados avalie uma expressão (por exemplo, `2 * 15/5`) ou quando quer obter a data atual.

### NOTA

As instruções SQL digitadas diretamente no SQL\*Plus são terminadas com um caractere de ponto-e-vírgula (;).

Esta tela mostra o resultado dessa instrução `SELECT` no SQL\*Plus em execução no Windows. Como você pode ver, a consulta exibe a data atual do banco de dados.

Você pode editar sua última instrução SQL no SQL\*Plus, digitando `EDIT`. Isso é útil quando você comete um erro ou deseja fazer uma alteração em sua instrução SQL. No Windows, quando digita `EDIT`, o aplicativo Bloco de Notas abre; você o usa para editar sua instrução SQL. Quando fecha o Bloco de Notas e salva sua instrução, a nova instrução é passada ao SQL\*Plus, onde pode ser novamente executada pela digitação de uma barra normal (`/`). No Linux ou no Unix, normalmente o editor padrão é definido como o `vi` ou `emacs`.

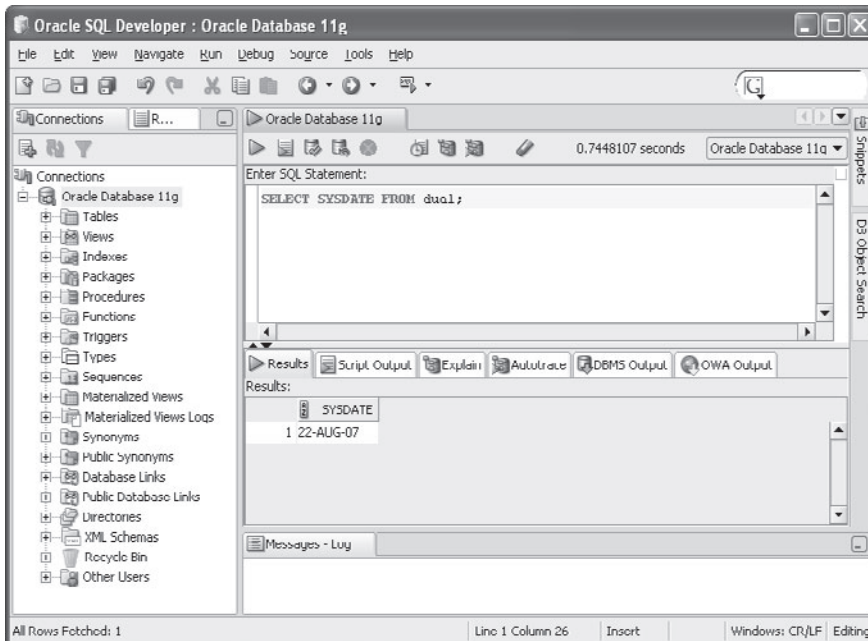


#### NOTA

Você vai aprender mais sobre a edição de instruções SQL com SQL\*Plus no Capítulo 3.

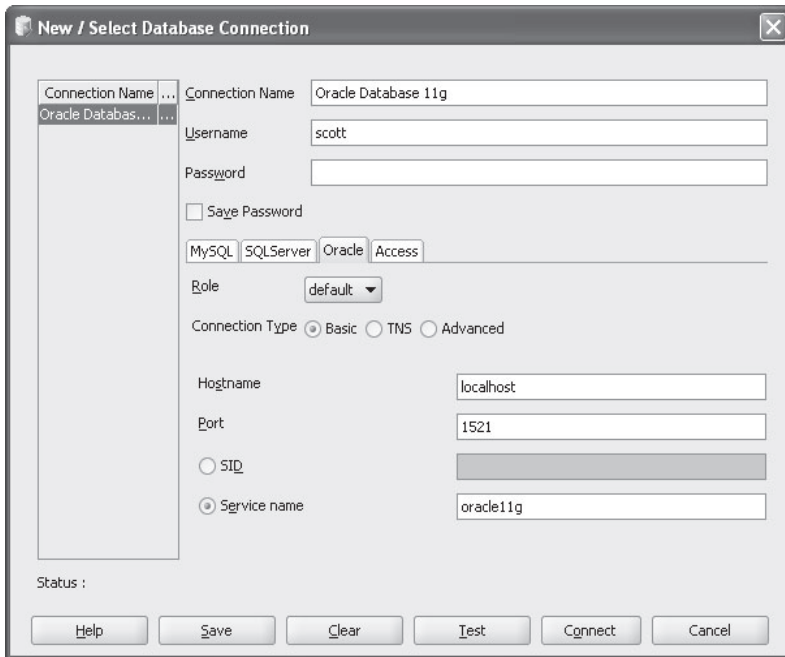
## SQL DEVELOPER

Você também pode inserir instruções SQL usando o SQL Developer. O SQL Developer usa uma interface gráfica muito interessante, por meio da qual você pode inserir instruções SQL, examinar tabelas de banco de dados, executar scripts, editar e depurar código PL/SQL e muito mais. Ele pode conectar-se a qualquer banco de dados Oracle (versão 9.2.0.1 e superiores) e é executado no Windows, Linux e Mac OSX. A ilustração a seguir mostra o SQL Developer em execução.





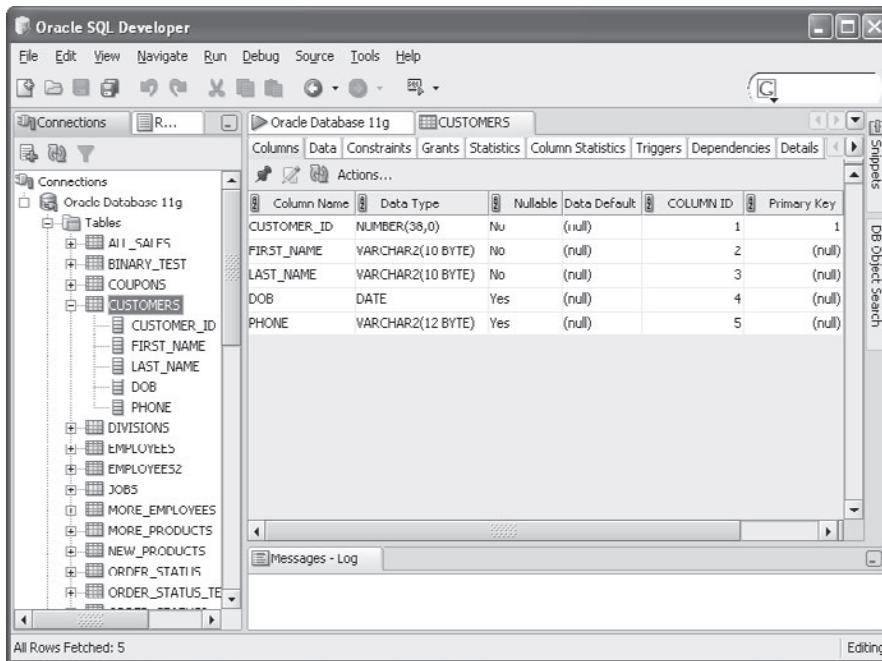
Antes de poder executar o SQL Developer, você precisa ter Java instalado em seu computador. Se estiver usando Windows XP Professional Edition e Oracle Database 11g, inicie o SQL Developer clicando em Iniciar e selecionando Programas | Oracle | Application Development | SQL Developer. O SQL Developer pedirá para que você selecione o executável Java. Navegue até o local onde o instalou e selecione o executável(\*). Em seguida, crie uma conexão, clicando o botão direito do mouse em Connections e selecionando New Connection, como mostrado a seguir.



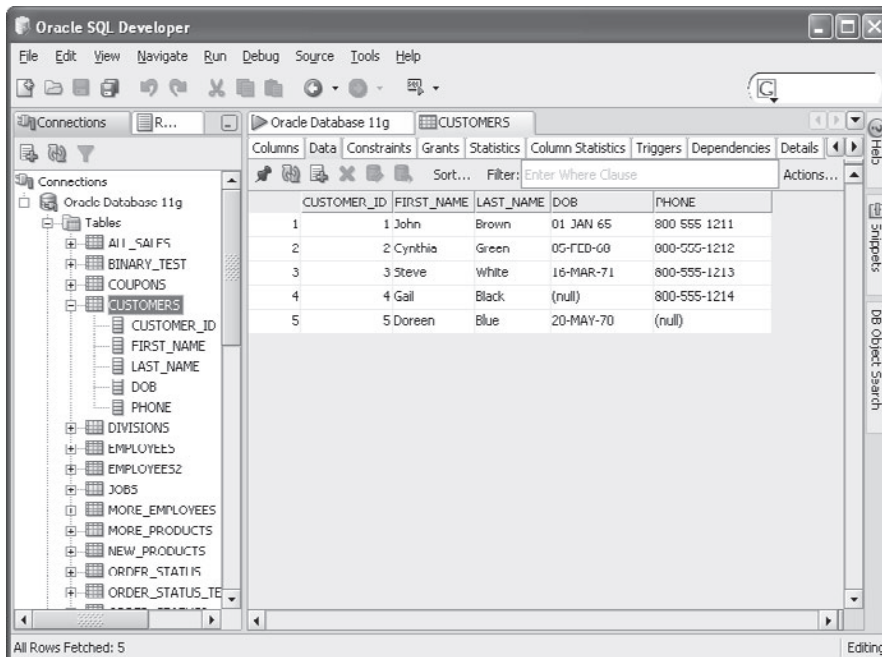
Quando tiver criado e testado uma conexão, você poderá usá-la para conectar-se ao banco de dados e executar consultas, examinar tabelas etc. A imagem a seguir mostra os detalhes de uma tabela de banco de dados chamada customers.

---

\* N. de R.T.: Se você não tem o software Java instalado em sua máquina, pode fazer a instalação pelo site <http://java.sun.com>.



Você também pode exibir os dados armazenados em uma tabela, como mostrado abaixo.



Os detalhes completos sobre o uso do SQL Developer podem ser vistos selecionando Help | Table of Contents na barra de menus do programa.

Na próxima seção, você irá aprender a criar o esquema da loja imaginária usada neste livro.

## CRIANDO O ESQUEMA DA LOJA

A loja imaginária vende artigos como livros, vídeos, DVDs e CDs. O banco de dados da loja conterá informações sobre clientes, funcionários, produtos e vendas. O script SQL\*Plus para criar o banco de dados é chamado `store_schema.sql` e está localizado no diretório `SQL` onde você extraiu o arquivo Zip deste livro. O script `store_schema.sql` contém as instruções DDL e DML usadas para criar o esquema `store`. Agora você irá aprender a executar o script `store_schema.sql`.

### Executando o script SQL\*Plus para criar o esquema da loja

Para criar o esquema `store`, execute os passos a seguir:

1. Inicie o SQL\*Plus.
2. Conecte-se no banco de dados como um usuário com privilégios para criar novos usuários, tabelas e packages PL/SQL. Estes scripts podem ser executados utilizando o usuário `system`; esse usuário tem todos os privilégios necessários. Talvez você precise falar com o administrador de seu banco de dados para configurar um usuário com os privilégios necessários (ele também poderá executar o script `store_schema.sql` para você).
3. Execute o script `store_schema.sql` dentro do SQL\*Plus, usando o comando `@`.

O comando `@` tem a seguinte sintaxe:

```
@ diretório\store_schema.sql
```

onde `diretório` é o diretório no qual seu script `store_schema.sql` está localizado. Por exemplo, se o script está armazenado em `E:\sql_book\SQL`, digite

```
@ E:\sql_book\SQL\store_schema.sql
```

Se você tiver colocado o script `store_schema.sql` em um diretório que contém espaços, deve colocar o diretório e o script entre aspas após o comando `@`. Por exemplo:

```
@ "E:\Oracle SQL book\sql_book\SQL\store_schema.sql"
```

Se você estiver usando Unix ou Linux e salvou o script em um diretório chamado `SQL` no sistema de arquivos `tmp`, digite

```
@ /tmp/SQL/store_schema.sql
```

#### NOTA

*O Windows usa caracteres de barra invertida (\) em caminhos de diretórios, já o Unix e o Linux usam caracteres de barra normal (/).*

A primeira linha executável no script `store_schema.sql` tenta eliminar o usuário `store`, gerando um erro, porque o usuário ainda não existe. Não se preocupe: a linha está lá para que você não precise eliminar o usuário `store` manualmente, quando recriar o esquema mais adiante no livro.

Quando o script `store_schema.sql` tiver terminado de executar, você estará conectado como o usuário `store`. Se desejar, abra o script `store_schema.sql` usando um editor de textos como o Bloco de Notas do Windows e examine as instruções nele contidas. Não se preocupe com os detalhes das instruções contidas no script — você irá conhecê-los à medida que avançar neste livro.



#### NOTA

*Para finalizar o SQL\*Plus, digite EXIT. Para conectar-se novamente no esquema store no SQL\*Plus, digite store como nome de usuário, com a senha store\_password. Enquanto você está conectado no banco de dados, o SQL\*Plus mantém uma sessão aberta. Quando se desconecta do banco de dados, sua sessão é finalizada. Você pode desconectar-se do banco de dados e manter o SQL\*Plus em execução digitando DISCONNECT. Você pode conectar-se novamente digitando CONNECT.*

## Instruções DDL (Data Definition Language) usadas para criar o esquema da loja

Conforme mencionado anteriormente, as instruções DDL (Data Definition Language) são usadas para criar usuários e tabelas, além de muitos outros tipos de estruturas no banco de dados. Nesta seção, você vai ver as instruções DDL usadas para criar o usuário `store` e algumas das tabelas.



#### NOTA

*As instruções SQL que você verá no restante deste capítulo são as mesmas contidas no script `store_schema.sql`. Você não precisa digitar as instruções, basta executar o script `store_schema.sql`.*

As próximas seções descrevem:

- Como criar um usuário de banco de dados
- Os tipos de dados comumente usados em um banco de dados Oracle
- Algumas das tabelas da loja imaginária

### ***Criando um usuário de banco de dados***

Para criar um usuário no banco de dados, use a instrução `CREATE USER`. A sintaxe simplificada da instrução `CREATE USER` é:

```
CREATE USER nome_usuario IDENTIFIED BY senha;
```

onde

- `nome_usuario` é o nome do usuário
- `senha` é a senha do usuário

Por exemplo, a instrução `CREATE USER` a seguir cria o usuário `store` com a senha `store_password`:

```
CREATE USER store IDENTIFIED BY store_password;
```

Se você quiser que o usuário possa trabalhar no banco de dados, ele deverá receber as *permissões* necessárias para realizar esse trabalho. No caso de `store`, esse usuário deve ser capaz de conectar-se no banco de dados (o que exige a permissão `connect`) e criar itens como tabelas de banco de dados (o que exige a permissão `resource`). As permissões são concedidas por um usuário privilegiado (por exemplo, o usuário `system`) usando a instrução `GRANT`.

O exemplo a seguir concede as permissões `connect` e `resource` para `store`:

```
GRANT connect, resource TO store;
```

Uma vez criado o usuário, as tabelas e outros objetos de banco de dados podem ser criados para esse usuário no esquema associado. Muitos exemplos deste livro usam o esquema `store`. Antes de entrarmos nos detalhes das tabelas da loja, você precisa saber a respeito dos tipos comumente usados do banco de dados Oracle.

### Os tipos comuns do banco de dados Oracle

Existem muitos tipos que podem ser usados para manipular dados em um banco de dados Oracle. Alguns dos mais usados são mostrados na Tabela 1-1.

O Apêndice apresenta uma relação completa dos tipos de dados. A tabela a seguir ilustra alguns exemplos de como números de tipo `NUMBER` são armazenados no banco de dados.

Formato	Número fornecido	Número armazenado
<code>NUMBER</code>	1234.567	1234.567
<code>NUMBER (6, 2)</code>	123.4567	123.46
<code>NUMBER (6, 2)</code>	12345.67	O número excede a precisão especificada, portanto, é rejeitado pelo banco de dados.

### Examinando as tabelas da loja

Nesta seção, você vai aprender como as tabelas do esquema `store` são criadas. Algumas das informações mantidas no esquema `store` incluem:

- Detalhes do cliente
- Tipos de produtos vendidos
- Detalhes do produto
- Um histórico dos produtos adquiridos pelos clientes
- Funcionários da loja
- Nível salarial

As tabelas a seguir são usadas para conter as informações:

- `customers` contém os detalhes dos clientes
- `product_types` contém os tipos de produtos vendidos pela loja

- **products** contém os detalhes dos produtos
- **purchases** mostra quais produtos foram adquiridos por quais clientes
- **employees** contém os detalhes dos funcionários
- **salary\_grades** contém os detalhes dos níveis salariais

**Tabela 1-1** *Tipos de dados comumente usados do Oracle*

Tipo do oracle	Significado
CHAR ( <i>comprimento</i> )	Armazena strings de comprimento fixo. O parâmetro <i>comprimento</i> especifica o comprimento da string. Se uma string de comprimento menor for armazenada, ela será preenchida com espaços no final. Por exemplo, CHAR (2) pode ser usado para armazenar uma string de comprimento fixo de dois caracteres; se "C" for armazenado em CHAR (2), um espaço será adicionado no final; "CA" é armazenado como está, sem preenchimento.
VARCHAR2 ( <i>comprimento</i> )	Armazena strings de comprimento variável. O parâmetro <i>comprimento</i> especifica o comprimento máximo da string. Por exemplo, VARCHAR2 (20) pode ser usado para armazenar uma string de até 20 caracteres de comprimento. Nenhum preenchimento é usado no final de uma string menor.
DATE	Armazena data e horas. O tipo DATE armazena o século, todos os quatro dígitos de um ano, o mês, o dia, a hora (no formato de 24 horas), o minuto e o segundo. Ele pode ser usado para armazenar datas e horas entre 1º de janeiro de 4712 a.C. e 31 de dezembro de 4712 d.C.
INTEGER	Armazena valores inteiros. Um valor inteiro não contém um ponto flutuante: trata-se de um número inteiro, como 1, 10 e 115.
NUMBER ( <i>precisão, escala</i> )	Armazena números de ponto flutuante, mas também pode ser usado para armazenar valores inteiros. A <i>precisão</i> é o número máximo de dígitos (à esquerda e à direita de um ponto decimal, se for usado) que podem ser usados para o número. A precisão máxima suportada pelo banco de dados Oracle é 38. A <i>escala</i> é o número máximo de dígitos à direita de um ponto decimal (se for usado). Se nem a <i>precisão</i> nem a <i>escala</i> forem especificadas, qualquer número poderá ser armazenado, até uma precisão de 38 dígitos. Qualquer tentativa de armazenar um número que ultrapasse a <i>precisão</i> será rejeitada pelo banco de dados.
BINARY_FLOAT	Lançado no Oracle Database 10g, armazena um número de ponto flutuante de 32 bits e precisão simples. Você vai aprender mais sobre BINARY_FLOAT posteriormente, na seção "Os tipos BINARY_FLOAT e BINARY_DOUBLE".
BINARY_DOUBLE	Introduzindo no Oracle Database 10g, armazena um número de ponto flutuante de 64 bits e precisão dupla. Você vai aprender mais sobre BINARY_DOUBLE posteriormente, na seção "Os tipos BINARY_FLOAT e BINARY_DOUBLE".

**NOTA**

*O script `store_schema.sql` cria outras tabelas e itens de banco de dados não mencionados na lista anterior. Você vai aprender sobre esses itens em capítulos posteriores.*

Nas seções a seguir, você vai ver os detalhes de algumas das tabelas e as instruções `CREATE TABLE` incluídas no script `store_schema.sql` que as criam.

**A tabela `customers`** A tabela `customers` contém os detalhes dos clientes. Nessa tabela estão contidos os seguintes itens:

- Nome
- Sobrenome
- Data de nascimento (`dob`)
- Número do telefone

Cada um destes itens exige uma coluna na tabela `customers`. A tabela `customers` é criada pelo script `store_schema.sql` usando a seguinte instrução `CREATE TABLE`:

```
CREATE TABLE customers (
    customer_id INTEGER CONSTRAINT customers_pk PRIMARY KEY,
    first_name VARCHAR2(10) NOT NULL,
    last_name VARCHAR2(10) NOT NULL,
    dob DATE,
    phone VARCHAR2(12)
);
```

Como você pode ver, a tabela `customers` contém cinco colunas, uma para cada item da lista anterior, e uma coluna adicional chamada `customer_id`. As colunas são:

- **`customer_id`** Contém um valor numérico único para cada linha da tabela. Cada tabela deve ter uma ou mais colunas que identificam cada linha exclusivamente; esta coluna (ou colunas) é conhecida como *chave primária*. A cláusula `CONSTRAINT` indica que a coluna `customer_id` é a chave primária. Uma cláusula `CONSTRAINT` restringe os valores armazenados em uma coluna. Para a coluna `customer_id`, as palavras-chaves `PRIMARY KEY` indicam que essa coluna deve conter um valor único para cada linha. Você também pode anexar um nome opcional em uma constraint, o qual deve vir imediatamente após a palavra-chave `CONSTRAINT` — por exemplo, `customers_pk`. Você sempre deve nomear suas constraints de chave primária para que, quando ocorrer um erro de restrição, seja fácil identificar onde ele aconteceu.
- **`first_name`** Contém o nome do cliente. A constraint `NOT NULL` é usada nessa coluna — isso significa que você deve fornecer um valor para `first_name` ao adicionar ou modificar uma linha. Se a constraint `NOT NULL` for omitida, não é preciso fornecer um valor e a coluna poderá permanecer vazia.

- **last\_name** Contém o sobrenome do cliente. Essa coluna é NOT NULL e, portanto, um valor deve ser fornecido ao se adicionar ou modificar uma linha.
- **dob** Contém a data nascimento do cliente. Não há uma constraint NOT NULL especificada para essa coluna; portanto, é pressuposto o valor padrão NULL e o valor é opcional ao se adicionar ou modificar uma linha.
- **phone** Contém o número do telefone do cliente. Este é um valor opcional.

O script `store_schema.sql` preenche a tabela `customers` com as seguintes linhas:

customer_id	first_name	last_name	dob	phone
1	John	Brown	01-JAN-65	800-555-1211
2	Cynthia	Green	05-FEB-68	800-555-1212
3	Steve	White	16-MAR-71	800-555-1213
4	Gail	Black		800-555-1214
5	Doreen	Blue	20-MAY-70	

Observe que a data de nascimento do cliente nº 4 é nula, assim como o número do telefone do cliente nº 5. Você pode ver as linhas da tabela `customers` executando a instrução `SELECT` a seguir com o `SQL*Plus`:

```
SELECT * FROM customers;
```

O asterisco (\*) indica que você deseja recuperar todas as colunas das tabelas `customers`.

#### NOTA

Neste livro, as instruções SQL mostradas em **negrito** são as que você deve digitar e executar se quiser acompanhar os exemplos. As instruções que não estão em **negrito** são as que você não precisa digitar.

**A tabela `product_types`** A tabela `product_types` contém os nomes dos tipos de produtos vendidos pela loja. Essa tabela é criada pelo script `store_schema.sql` com a seguinte instrução `CREATE TABLE`:

```
CREATE TABLE product_types (
    product_type_id INTEGER CONSTRAINT product_types_pk PRIMARY KEY,
    name VARCHAR2(10) NOT NULL
);
```

A tabela `product_types` contém as duas colunas a seguir:

- **product\_type\_id** identifica exclusivamente cada linha da tabela; a coluna `product_type_id` é a chave primária dessa tabela. Cada linha da tabela `product_types` deve ter um valor inteiro único para a coluna `product_type_id`.
- **name** contém o nome do tipo de produto. Essa é uma coluna NOT NULL e, portanto, um valor deve ser fornecido ao se adicionar ou modificar uma linha.



O script `store_schema.sql` preenche a tabela `product_types` com as seguintes linhas:

```
product_type_id name
-----
1 Book
2 Video
3 DVD
4 CD
5 Magazine
```

A tabela `product_types` contém os tipos de produto da loja. Cada produto vendido pela loja deve ser de um desses tipos. Você pode ver as linhas da tabela `product_types` executando a instrução `SELECT` a seguir com o SQL\* Plus:

```
SELECT * FROM product_types;
```

**A tabela `products`** A tabela `products` contém os produtos vendidos pela loja. Para cada produto são mantidas as seguintes informações:

- Tipo de produto
- Nome
- Descrição
- Preço

O script `store_schema.sql` cria a tabela `products` usando a seguinte instrução `CREATE TABLE`:

```
CREATE TABLE products (
    product_id INTEGER CONSTRAINT products_pk PRIMARY KEY,
    product_type_id INTEGER
        CONSTRAINT products_fk_product_types
        REFERENCES product_types(product_type_id),
    name VARCHAR2(30) NOT NULL,
    description VARCHAR2(50),
    price NUMBER(5, 2)
);
```

As colunas dessa tabela são:

- **product\_id** identifica exclusivamente cada linha da tabela. Essa coluna é a chave primária da tabela.
- **product\_type\_id** associa cada produto a um tipo de produto. Essa coluna é uma referência à coluna `product_type_id` da tabela `product_types`; isso é conhecido como *chave estrangeira*, pois faz referência a uma coluna em outra tabela. A tabela que contém a chave estrangeira (a tabela `products`) é conhecida como tabela *detalhe* ou *filha* e a que é referenciada (a tabela `product_types`) é conhecida como tabela *mestre* ou *pai*. Esse tipo de relação é conhecido como *mestre-detanha* ou *pai-filho*. Quando adiciona um novo

produto, você o associa a um tipo fornecendo um valor de `product_types.product_type_id` correspondente na coluna `products.product_type_id` (você verá um exemplo posteriormente).

- **name** contém o nome do produto, que deve ser especificado, pois a coluna `name` é NOT NULL.
- **description** contém uma descrição opcional do produto.
- **price** contém um preço opcional para um produto. Essa coluna é definida como `NUMBER (5, 2)` — a precisão é 5 e, portanto, no máximo cinco dígitos podem ser fornecidos para esse número. A escala é de 2, portanto, dois dígitos deste máximo de cinco podem estar à direita do ponto decimal.

A seguir está um subconjunto das linhas armazenadas na tabela `products`:

product_id	product_type_id	name	description	price
1	1	Modern Science	A description of modern science	19.95
2	1	Chemistry	Introduction to Chemistry	30
3	2	Supernova	A star explodes	25.99
4	2	Tank War	Action movie about a future war	13.95

A primeira linha da tabela `products` tem o valor `product_type_id` igual a 1, o que significa que o produto é um livro (esse valor de `product_type_id` corresponde ao tipo de produto “book” na tabela `product_types`). O segundo produto também é um livro, mas o terceiro e o quarto produtos são vídeos (seu `product_type_id` é 2, o que corresponde ao tipo de produto “video” na tabela `product_types`). Você pode ver todas as linhas da tabela `products` executando a instrução `SELECT` a seguir com o SQL\*Plus.

```
SELECT * FROM products;
```

**A tabela purchases** A tabela `purchases` contém as compras feitas por um cliente. Para cada compra feita por um cliente, as seguintes informações são mantidas:

- Identificação do produto
- Identificação do cliente
- Número de unidades do produto adquiridas pelo cliente

O script `store_schema.sql` usa a seguinte instrução `CREATE TABLE` para criar a tabela `purchases`:

```
CREATE TABLE purchases (
  product_id INTEGER
    CONSTRAINT purchases_fk_products
    REFERENCES products(product_id),
  customer_id INTEGER
    CONSTRAINT purchases_fk_customers
    REFERENCES customers(customer_id),
  quantity INTEGER NOT NULL,
  CONSTRAINT purchases_pk PRIMARY KEY (product_id, customer_id)
);
```

As colunas dessa tabela são:

- **product\_id** contém a identificação do produto que foi adquirido. Isso deve corresponder a um valor na coluna `product_id` da tabela `products`.
- **customer\_id** contém a identificação do cliente que fez a compra. Isso deve corresponder a um valor na coluna `customer_id` da tabela `customers`.
- **quantity** contém o número de unidades do produto que foram adquiridas pelo cliente.

A tabela `purchases` tem uma restrição de chave primária chamada `purchases_pk` que abrange duas colunas: `product_id` e `customer_id`. A combinação dos dois valores de coluna deve ser única para cada linha. Quando uma chave primária consiste em várias colunas, ela é conhecida como chave primária *composta*.

A seguir está um subconjunto das linhas armazenadas na tabela `purchases`:

product_id	customer_id	quantity
1	1	1
2	1	3
1	4	1
2	2	1
1	3	1

Como você pode ver, a combinação dos valores das colunas `product_id` e `customer_id` é única para cada linha. Você pode ver todas as linhas da tabela `purchases` executando a instrução `SELECT` a seguir com o SQL\*Plus:

```
SELECT * FROM purchases;
```

**A tabela `employees`** A tabela `employees` contém os detalhes dos funcionários. As seguintes informações são mantidas na tabela:

- Identificação do funcionário

- Identificação do gerente do funcionário (se aplicável)
- Nome
- Sobrenome
- Cargo
- Salário

O script `store_schema.sql` usa a seguinte instrução `CREATE TABLE` para criar a tabela `employees`:

```
CREATE TABLE employees (
    employee_id INTEGER CONSTRAINT employees_pk PRIMARY KEY,
    manager_id INTEGER,
    first_name VARCHAR2(10) NOT NULL,
    last_name VARCHAR2(10) NOT NULL,
    title VARCHAR2(20),
    salary NUMBER(6, 0)
);
```

O script `store_schema.sql` preenche a tabela `employees` com as linhas a seguir:

employee_id	manager_id	first_name	last_name	title	salary
1		James	Smith	CEO	800000
2	1	Ron	Johnson	Sales Manager	600000
3	2	Fred	Hobbs	Salesperson	150000
4	2	Susan	Jones	Salesperson	500000

Como você pode ver, James Smith não tem gerente, pois ele é o diretor executivo da loja.

**A tabela `salary_grades`** A tabela `salary_grades` contém os diferentes níveis salariais disponíveis para os funcionários. São mantidas as seguintes informações:

- Identificação do nível salarial
- Limite salarial inferior para o nível
- Limite salarial superior para o nível

O script `store_schema.sql` usa a seguinte instrução `CREATE TABLE` para criar a tabela `salary_grades`:

```
CREATE TABLE salary_grades (
    salary_grade_id INTEGER CONSTRAINT salary_grade_pk PRIMARY KEY,
    low_salary NUMBER(6, 0),
    high_salary NUMBER(6, 0)
);
```

O script `store_schema.sql` preenche a tabela `salary_grades` com as seguintes linhas:

salary_grade_id	low_salary	high_salary
1	1	250000
2	250001	500000
3	500001	750000
4	750001	999999

## ADICIONANDO, MODIFICANDO E REMOVENDO LINHAS

Nesta seção, você vai aprender a adicionar, modificar e remover linhas em tabelas de banco de dados usando as instruções `INSERT`, `UPDATE` e `DELETE`. Você pode tornar suas alterações permanentes no banco de dados usando a instrução `COMMIT` ou desfazê-las com a instrução `ROLLBACK`. Esta seção não aborda todos os detalhes do uso dessas instruções; você vai aprender mais sobre elas no Capítulo 8.

### Adicionando uma linha em uma tabela

A instrução `INSERT` é usada para adicionar novas linhas em uma tabela. Em uma instrução `INSERT`, você especifica as seguintes informações:

- A tabela na qual a linha vai ser inserida
- Uma lista de colunas para as quais você quer especificar valores
- Uma lista de valores a serem armazenados nas colunas especificadas

Ao inserir uma linha, você precisa fornecer um valor para a chave primária e para todas as outras colunas que são definidas como `NOT NULL`. Não é necessário especificar valores para as outras colunas, caso você não queira; essas colunas serão configuradas automaticamente como nulas se os seus valores forem omitidos. Você pode identificar quais colunas são definidas como `NOT NULL` usando o comando `DESCRIBE` no `SQL*Plus`. O exemplo a seguir utiliza o comando `DESCRIBE` na tabela `customers`:

SQL> DESCRIBE customers		
Name	Null?	Type
-----		
CUSTOMER_ID	NOT NULL	NUMBER(38)
FIRST_NAME	NOT NULL	VARCHAR2(10)
LAST_NAME	NOT NULL	VARCHAR2(10)
DOB		DATE
PHONE		VARCHAR2(12)

Como você pode ver, as colunas `customer_id`, `first_name`, e `last_name` são `NOT NULL`, significando que você deve fornecer um valor para elas. As colunas `dob` e `phone` não exigem um valor; se quiser, você pode omitir os valores e eles serão configurados automaticamente como nulos.

Execute a instrução `INSERT` a seguir, a qual adiciona uma linha na tabela `customers`; observe que a ordem dos valores na lista de `VALUES` corresponde à ordem na qual as colunas são especificadas na lista de colunas:

```
SQL> INSERT INTO customers (
      2   customer_id, first_name, last_name, dob, phone
```

```

3 ) VALUES (
4   6, 'Fred', 'Brown', '01-JAN-1970', '800-555-1215'
5 );

```

1 row created.

### NOTA

O SQL\*Plus enumera as linhas automaticamente depois que você pressiona ENTER no final de cada uma.

No exemplo anterior, o SQL\*Plus responde dizendo que uma linha foi criada após a instrução INSERT ser executada. Você pode verificar isso executando a seguinte instrução SELECT:

```
SELECT *
```

```
FROM customers;
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB	PHONE
1	John	Brown	01-JAN-65	800-555-1211
2	Cynthia	Green	05-FEB-68	800-555-1212
3	Steve	White	16-MAR-71	800-555-1213
4	Gail	Black		800-555-1214
5	Doreen	Blue	20-MAY-70	
6	Fred	Brown	01-JAN-70	800-555-1215

Observe a nova linha que foi adicionada no final da tabela.

Por padrão, o banco de dados Oracle exibe datas no formato DD-MMM-AA, onde DD é o número do dia, MMM são os três primeiros caracteres do mês (em maiúsculas) e AA são os dois últimos dígitos do ano. Na verdade, o banco de dados armazena todos os quatro dígitos do ano, mas, por padrão, ele exibe apenas os dois últimos.

Quando uma linha é adicionada na tabela customers, um valor exclusivo deve ser fornecido para a coluna customer\_id. O banco de dados Oracle não permite adicionar uma linha com um valor de chave primária que já exista na tabela; por exemplo, a instrução INSERT a seguir causa um erro, pois já existe uma linha com customer\_id igual a 1:

```

SQL> INSERT INTO customers (
2   customer_id, first_name, last_name, dob, phone
3 ) VALUES (
4   1, 'Lisa', 'Jones', '02-JAN-1971', '800-555-1225'
5 );

```

```
INSERT INTO customers (
```

```
*
```

```
ERROR at line 1:
```

```
ORA-00001: unique constraint (STORE.CUSTOMERS_PK) violated
```

Observe que o nome da constraint é mostrado no erro (CUSTOMERS\_PK). É por isso que você sempre deve dar nomes às suas constraints de chave primária; caso contrário, o banco de dados Oracle atribuirá à constraint um nome não amigável, gerado pelo sistema (por exemplo, SYS\_C0011277).

## Modificando uma linha existente em uma tabela

A instrução `UPDATE` é usada para alterar linhas em uma tabela. Normalmente, ao usar a instrução `UPDATE`, você especifica as seguintes informações:

- A tabela que contém as linhas a serem alteradas
- Uma cláusula `WHERE` especificando as linhas a serem alteradas
- Uma lista de nomes de colunas, junto com seus novos valores, especificados com a cláusula `SET`

Você pode alterar uma ou mais linhas usando a mesma instrução `UPDATE`. Se mais de uma linha for especificada, a mesma alteração será feita para todas as linhas. O exemplo a seguir atualiza o `last_name` do cliente nº 2 para Orange:

```
UPDATE customers
SET last_name = 'Orange'
WHERE customer_id = 2;
```

1 row updated.

O SQL\*Plus confirma que uma linha foi atualizada.

### CAUIDADO

*Se você se esquecer de adicionar uma cláusula `WHERE`, todas as linhas serão atualizadas.*

A consulta a seguir confirma que a atualização funcionou:

```
SELECT *
FROM customers
WHERE customer_id = 2;
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB	PHONE
2	Cynthia	Orange	05-FEB-68	800-555-1212

## Removendo uma linha de uma tabela

A instrução `DELETE` é usada para remover linhas de uma tabela. Normalmente, uma cláusula `WHERE` é utilizada para limitar as linhas que você deseja excluir; caso contrário, *todas* as linhas serão excluídas da tabela. A instrução `DELETE` a seguir remove o cliente nº 2:

```
DELETE FROM customers
WHERE customer_id = 2;
```

1 row deleted.

Para desfazer as alterações feitas nas linhas, use `ROLLBACK`:

```
ROLLBACK;
```

Rollback complete.

Execute a instrução `ROLLBACK` para desfazer todas as alterações que você fez até agora. Assim, seus resultados corresponderão àqueles mostrados nos próximos capítulos.

**NOTA**

*É possível tornar as alterações nas linhas permanentes usando `COMMIT`. Você vai aprender a fazer isso no Capítulo 8.*

## OS TIPOS `BINARY_FLOAT` E `BINARY_DOUBLE`

O Oracle Database 10g introduziu dois novos tipos de dados: `BINARY_FLOAT` e `BINARY_DOUBLE`. `BINARY_FLOAT` armazena um número de ponto flutuante de 32 bits e precisão simples; `BINARY_DOUBLE` armazena um número de ponto flutuante de 64 bits e precisão dupla. Esses novos tipos de dados são baseados no padrão IEEE (Institute of Electrical and Electronics Engineers) para aritmética binária de ponto flutuante.

### Vantagens de `BINARY_FLOAT` e `BINARY_DOUBLE`

`BINARY_FLOAT` e `BINARY_DOUBLE` são destinados a complementar o tipo `NUMBER` existente. Eles oferecem as seguintes vantagens em relação a `NUMBER`.

- **Necessidade de armazenamento menor** `BINARY_FLOAT` e `BINARY_DOUBLE` exigem 5 e 9 bytes de espaço de armazenamento, enquanto `NUMBER` pode usar até 22 bytes.
- **Maior intervalo de números representados** `BINARY_FLOAT` e `BINARY_DOUBLE` suportam números muito maiores e menores do que os que podem ser armazenados em `NUMBER`.
- **Execução mais rápida de operações** Normalmente, as operações envolvendo `BINARY_FLOAT` e `BINARY_DOUBLE` são executadas de forma mais rápida do que as operações com `NUMBER`. Isso porque geralmente as operações com `BINARY_FLOAT` e `BINARY_DOUBLE` são executadas no hardware, enquanto os valores `NUMBER` devem ser primeiro convertidos usando software, antes que as operações possam ser executadas.
- **Operações fechadas** As operações aritméticas envolvendo `BINARY_FLOAT` e `BINARY_DOUBLE` são fechadas, o que significa que é retornado um número ou um valor especial. Por exemplo, se você dividir um `BINARY_FLOAT` por outro `BINARY_FLOAT`, um `BINARY_FLOAT` será retornado.
- **Arredondamento transparente** `BINARY_FLOAT` e `BINARY_DOUBLE` usam o sistema binário (base 2) para representar um número, enquanto `NUMBER` usa o sistema decimal (base 10). A base usada para representar um número afeta o seu arredondamento. Por exemplo, um número decimal de ponto flutuante é arredondado para a casa decimal mais próxima, mas um número binário de ponto flutuante é arredondado para a casa binária mais próxima.

**DICA**

*Se você estiver desenvolvendo um sistema que envolva muitos cálculos numéricos, use `BINARY_FLOAT` e `BINARY_DOUBLE` para representar números. Você precisa usar Oracle Database 10g ou superior.*



## Usando BINARY\_FLOAT e BINARY\_DOUBLE em uma tabela

A instrução a seguir cria uma tabela chamada `binary_test` que contém uma coluna `BINARY_FLOAT` e uma coluna `BINARY_DOUBLE`:

```
CREATE TABLE binary_test (  
    bin_float BINARY_FLOAT,  
    bin_double BINARY_DOUBLE  
);
```



### NOTA

No diretório SQL do arquivo zip com os códigos de exemplo, você encontrará um script chamado `oracle_10g_examples.sql` que cria a tabela `binary_test` no esquema `store`. O script também executa as instruções `INSERT` apresentadas nesta seção. Você poderá executar esse script se estiver usando Oracle Database 10g ou superior.

O exemplo a seguir adiciona uma linha na tabela `binary_test`:

```
INSERT INTO binary_test (  
    bin_float, bin_double  
) VALUES (  
    39.5f, 15.7d  
);
```

Observe que `f` indica que um número é `BINARY_FLOAT` e que `d` indica que um número é `BINARY_DOUBLE`.

## Valores especiais

Com `BINARY_FLOAT` e `BINARY_DOUBLE`, você também pode usar os valores especiais mostrados na Tabela 1-2. O exemplo a seguir insere `BINARY_FLOAT_INFINITY` e `BINARY_DOUBLE_INFINITY` na tabela `binary_test`:

```
INSERT INTO binary_test (  
    bin_float, bin_double  
) VALUES (  
    BINARY_FLOAT_INFINITY, BINARY_DOUBLE_INFINITY  
);
```

Tabela 1-2 Valores especiais

Valor especial	Descrição
<code>BINARY_FLOAT_NAN</code>	Valor não-numérico (NaN--Not a number) para o tipo <code>BINARY_FLOAT</code>
<code>BINARY_FLOAT_INFINITY</code>	Infinito (INF--Infinity) para o tipo <code>BINARY_FLOAT</code>
<code>BINARY_DOUBLE_NAN</code>	Valor não-numérico (NaN--Not a number) para o tipo <code>BINARY_DOUBLE</code>
<code>BINARY_DOUBLE_INFINITY</code>	Infinito (INF--Infinity) para o tipo <code>BINARY_DOUBLE</code>

A consulta a seguir recupera as linhas de `binary_test`:

```
SELECT *
FROM binary_test;

BIN_FLOAT BIN_DOUBLE
-----
3.95E+001 1.57E+001
          Inf          Inf
```

## SAINDO DO SQL\*PLUS

Para sair do SQL\*Plus, use o comando `EXIT`, como mostra o exemplo a seguir:

```
EXIT
```

### NOTA

Quando você sai do SQL\*Plus usando este comando, ele automaticamente executa um `COMMIT`. Se o SQL\*Plus terminar de forma anômala — por exemplo, se o computador em que ele está sendo executado falha — uma instrução `ROLLBACK` é executada automaticamente. Você aprenderá mais sobre isso no Capítulo 8.

## INTRODUÇÃO AO PL/SQL DA ORACLE

PL/SQL é a linguagem procedural da Oracle que permite adicionar construções de programação em torno de instruções SQL. Os códigos PL/SQL são usados principalmente para criar procedures e funções em um banco de dados que contenha a lógica do negócio. O código PL/SQL contém construções de programação padrão, como:

- Declarações de variável
- Lógica condicional (if-then-else etc.)
- Loops
- Procedures e funções

A instrução `CREATE PROCEDURE` a seguir cria uma procedure chamada `update_product_price()`, que multiplica o preço de um produto por um fator — a identificação do produto e o fator de reajuste são passados como parâmetros para a procedure. Se o produto especificado não existe, a procedure nada faz; caso contrário, ela atualiza o preço do produto.

### NOTA

Não se preocupe com os detalhes do código PL/SQL mostrado na listagem a seguir — você vai aprender tudo sobre PL/SQL no Capítulo 11. Agora, o importante é que você tenha uma idéia do PL/SQL.

```
CREATE PROCEDURE update_product_price (
  p_product_id IN products.product_id%TYPE,
  p_factor      IN NUMBER
) AS
  product_count INTEGER;
```

```

BEGIN
  -- conta o número de produtos com
  -- product_id fornecido (será 1 se o produto existir)
  SELECT COUNT(*)
  INTO product_count
  FROM products
  WHERE product_id = p_product_id;

  -- se o produto existe (isto é, product_count = 1), então
  -- atualiza o preço desse produto
  IF product_count = 1 THEN
    UPDATE products
    SET price = price * p_factor
    WHERE product_id = p_product_id;
    COMMIT;
  END IF;
EXCEPTION
  WHEN OTHERS THEN
    ROLLBACK;
END update_product_price;
/

```

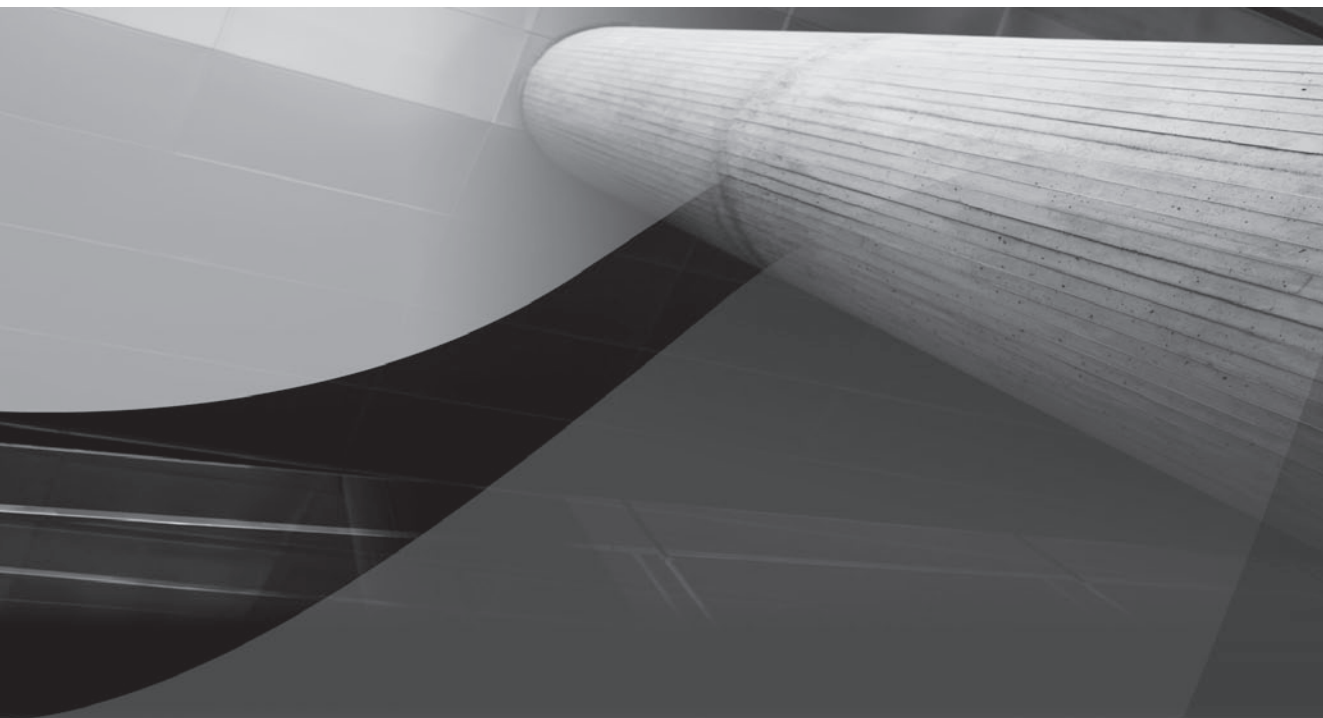
Os erros que ocorrem em códigos PL/SQL são tratados com exceções. No exemplo anterior, o bloco `EXCEPTION` executa uma instrução `ROLLBACK`, caso uma exceção seja lançada no código.

## RESUMO

Neste capítulo, você aprendeu que:

- Um banco de dados relacional é uma coleção de informações relacionadas que foram organizadas em estruturas conhecidas como tabelas. Cada tabela contém linhas organizadas em colunas. Essas tabelas são armazenadas no banco de dados, em estruturas conhecidas como esquemas, que são áreas onde os usuários do banco de dados podem armazenar seus objetos (como tabelas e procedures PL/SQL).
- SQL (Structured Query Language) é a linguagem padrão projetada para acessar bancos de dados relacionais.
- O SQL\*Plus permite executar instruções SQL e comandos SQL\*Plus.
- O SQL Developer é uma ferramenta gráfica para desenvolvimento de bancos de dados.
- Como executar instruções `SELECT`, `INSERT`, `UPDATE` e `DELETE`.
- PL/SQL é a linguagem procedural da Oracle que contém instruções de programação.

No próximo capítulo, você vai aprender mais sobre recuperação de informações de tabelas de banco de dados.



# CAPÍTULO 2

Recuperando informações  
de tabelas de banco  
de dados

Neste capítulo, você vai aprender a:

- Recuperar informações de uma ou mais tabelas do banco de dados usando instruções `SELECT`
- Usar expressões aritméticas para efetuar cálculos
- Limitar a recuperação de linhas apenas àquelas em que está interessado, usando uma cláusula `WHERE`
- Classificar as linhas recuperadas de uma tabela

Os exemplos desta seção utilizam o esquema `store`. Para acompanhar os exemplos, inicie a `SQL*Plus` e conecte-se como o usuário `store`.

## EXECUTANDO INSTRUÇÕES `SELECT` EM UMA ÚNICA TABELA

Você usa a instrução `SELECT` para recuperar informações de tabelas do banco de dados. Na forma mais simples da instrução, você especifica a tabela e as colunas das quais deseja recuperar dados. A instrução `SELECT` a seguir recupera as colunas `customer_id`, `first_name`, `last_name`, `dob` e `phone` da tabela `customers`:

```
SELECT customer_id, first_name, last_name, dob, phone
FROM customers;
```

Imediatamente após a palavra-chave `SELECT`, você fornece os nomes das colunas que deseja recuperar; depois da palavra-chave `FROM`, você fornece o nome da tabela. A instrução SQL é terminada com um ponto-e-vírgula (;). As instruções `SELECT` também são conhecidas como *consultas*.

Você não informa ao software de sistema de gerenciamento de banco de dados exatamente como acessar as informações desejadas, apenas diz quais informações deseja e deixa que o programa se preocupe com o modo de obtê-las. Os itens que aparecem imediatamente após a palavra-chave `SELECT` nem sempre precisam ser colunas de uma tabela: eles podem ser qualquer expressão válida. (Você vai ver exemplos de expressões posteriormente neste capítulo.) Depois que você pressiona `ENTER` no final da instrução SQL, ela é executada e os resultados são retornados à `SQL*Plus` para exibição na tela:

```
CUSTOMER_ID FIRST_NAME LAST_NAME DOB PHONE
-----
1 John Brown 01-JAN-65 800-555-1211
2 Cynthia Green 05-FEB-68 800-555-1212
3 Steve White 16-MAR-71 800-555-1213
4 Gail Black 800-555-1214
5 Doreen Blue 20-MAY-70
```

As linhas retornadas pelo banco de dados são conhecidas como *conjunto de resultados*. Como você pode ver a partir do exemplo, o banco de dados Oracle converte os nomes de coluna para seus equivalentes em maiúsculas. As colunas de caractere e data são justificadas à esquerda; as colunas numéricas são justificadas à direita. Por padrão, o banco de dados Oracle exibe datas no formato `DD-MMM-AA`, onde `DD` é o número do dia, `MMM` são os três primeiros caracteres do mês (em letras maiúsculas) e `AA` são os dois dígitos do ano. Na verdade, o banco de dados armazena todos os quatro dígitos do ano, mas ele só exibe os dois últimos.

**NOTA**

Um administrador de banco de dados pode alterar o formato de exibição padrão de datas, configurando um parâmetro do banco de dados Oracle chamado `NLS_DATE_FORMAT`. Você aprenderá mais sobre datas no Capítulo 5.

Embora você possa especificar nomes de coluna e nomes de tabela usando texto em maiúscula ou minúscula, é melhor usar apenas um estilo. Os exemplos deste livro utilizam letras maiúsculas para palavras-chave da linguagem SQL ou do Oracle e letras minúsculas para todo o resto.

## RECUPERANDO TODAS AS COLUNAS DE UMA TABELA

Se quiser recuperar todas as colunas de uma tabela, você pode usar o caractere de asterisco (\*) no lugar de uma lista de colunas. Na consulta a seguir, o asterisco é usado para recuperar todas as colunas da tabela `customers`:

```
SELECT *
FROM customers;
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB	PHONE
1	John	Brown	01-JAN-65	800-555-1211
2	Cynthia	Green	05-FEB-68	800-555-1212
3	Steve	White	16-MAR-71	800-555-1213
4	Gail	Black		800-555-1214
5	Doreen	Blue	20-MAY-70	

Como você pode ver, todas as colunas da tabela `customers` foram recuperadas.

## ESPECIFICANDO AS LINHAS A SEREM RECUPERADAS USANDO A CLÁUSULA WHERE

A cláusula `WHERE` é utilizada em uma consulta para especificar as linhas que você deseja recuperar. Isso é muito importante, pois o Oracle tem capacidade de armazenar uma grande quantidade de linhas em uma tabela\* e você pode estar interessado apenas em um subconjunto muito pequeno dessas linhas. Você coloca a cláusula `WHERE` após a cláusula `FROM`:

```
SELECT lista de itens
FROM lista de tabelas
WHERE lista de condições;
```

Na consulta a seguir, a cláusula `WHERE` é usada para recuperar a linha da tabela `customers` onde a coluna `customer_id` é igual a 2:

```
SELECT *
FROM customers
WHERE customer_id = 2;
```

---

\* N. de R.T.: Para que você tenha uma idéia da capacidade de armazenamento do banco Oracle, não existe limite de linhas por tabela; você pode ter milhões, bilhões de linhas em uma mesma tabela e mesmo assim manter o desempenho particionando as tabelas. O limite físico do banco de dados Oracle para armazenamento de dados é até 65.536 tablespaces, cada uma com um datafile de até 128 TB, o que permite ao Oracle armazenar até 8 hexabytes de informação em uma única instância. Um hexabyte corresponde a 1,000,000,000,000,000,000 de bytes ou 1 bilhão de gigabytes.

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB	PHONE
2	Cynthia	Green	05-FEB-68	800-555-1212

IDENTIFICADORES DE LINHA

Cada linha em um banco de dados Oracle tem um identificador exclusivo (*rowid*), usado internamente para armazenar a localização física da linha. Um rowid é um número de 18 dígitos representado na base 64. Você pode ver o rowid das linhas de uma tabela recuperando a coluna ROWID em uma consulta. Por exemplo, a consulta a seguir recupera as colunas ROWID e customer\_id da tabela customers; observe o número na base 64 na saída:

```
SELECT ROWID, customer_id
FROM customers;
```

ROWID	CUSTOMER_ID
AAAF4yAABAAAHAKAAA	1
AAAF4yAABAAAHAKAAB	2
AAAF4yAABAAAHAKAAC	3
AAAF4yAABAAAHAKAAD	4
AAAF4yAABAAAHAKAAE	5

Quando você descreve uma tabela usando o comando DESCRIBE do SQL\*Plus, a coluna ROWID não aparece na saída, pois é usada apenas internamente pelo banco de dados. A coluna ROWID é conhecida como uma *pseudocoluna*. O exemplo a seguir descreve a tabela customers; observe que a coluna ROWID não aparece na saída:

```
DESCRIBE customers
```

Name	Null?	Type
CUSTOMER_ID	NOT NULL	NUMBER(38)
FIRST_NAME	NOT NULL	VARCHAR2(10)
LAST_NAME	NOT NULL	VARCHAR2(10)
DOB		DATE
PHONE		VARCHAR2(12)

NÚMEROS DE LINHA

Outra pseudocoluna é ROWNUM, que retorna o número da linha em um conjunto de resultados. A primeira linha retornada por uma consulta tem o número 1, a segunda tem o número 2 e assim por diante. Por exemplo, a consulta a seguir inclui ROWNUM ao recuperar as linhas da tabela customers:

```
SELECT ROWNUM, customer_id, first_name, last_name
FROM customers;
```

ROWNUM	CUSTOMER_ID	FIRST_NAME	LAST_NAME
1	1	John	Brown
2	2	Cynthia	Green
3	3	Steve	White

4	4 Gail	Black
5	5 Doreen	Blue

Aqui está outro exemplo:

```
SELECT ROWNUM, customer_id, first_name, last_name
FROM customers
WHERE customer_id = 3;
```

ROWNUM	CUSTOMER_ID	FIRST_NAME	LAST_NAME
1	3	Steve	White

## EFETUANDO CÁLCULOS ARITMÉTICOS

O Oracle permite efetuar cálculos em instruções SQL usando expressões aritméticas, consistindo em adição, subtração, multiplicação e divisão. As expressões aritméticas consistem em dois *operandos* — números ou datas — e um operador aritmético. Os quatro operadores aritméticos estão mostrados na tabela a seguir:

Operador	Descrição
+	Adição
-	Subtração
*	Multiplicação
/	Divisão

A consulta a seguir mostra como se usa o operador de multiplicação (\*) para calcular 2 multiplicado por 6 (os números 2 e 6 são os operandos):

```
SELECT 2*6
FROM dual;
```

2*6
-----
12

Como você pode ver a partir dessa consulta, o resultado correto (12) é exibido. O uso de 2\*6 na consulta é um exemplo de *expressão*. Uma expressão pode conter uma combinação de colunas, valores literais e operadores.

## Efetuando aritmética de data

Você pode usar os operadores de adição e subtração com datas. Você pode somar um número — representando um número de dias — a uma data. O exemplo a seguir soma dois dias a 25 de julho de 2007 e exibe a data resultante:

```
SELECT TO_DATE('25-JUL-2007') + 2
FROM dual;
```

TO_DATE (
-----
27-JUL-07



### A tabela dual

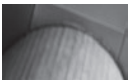
Você deve ter notado o uso da tabela `dual` no exemplo anterior. No capítulo anterior, mencionamos que a tabela `dual` contém uma única linha. A saída do comando `DESCRIBE` a seguir mostra a estrutura da tabela `dual`, junto com uma consulta que recupera a linha dessa tabela:

```
DESCRIBE dual
Name                               Null?    Type
-----
DUMMY                                        VARCHAR2(1)

SELECT *
FROM dual;

D
-
X
```

Observe que a tabela `dual` tem uma coluna `VARCHAR2` chamada `dummy` e uma única linha com o valor `x`.



### NOTA

`TO_DATE()` é uma função que converte uma string em uma data. Você irá aprender mais sobre `TO_DATE()` no Capítulo 5.

O exemplo a seguir subtrai três dias de 2 de agosto de 2007:

```
SELECT TO_DATE('02-AUG-2007') - 3
FROM dual;

TO_DATE('
-----
30-JUL-07
```

Você também pode subtrair uma data de outra, produzindo o número de dias entre as duas datas. O exemplo a seguir subtrai 25 de julho de 2007, de 2 de agosto de 2007:

```
SELECT TO_DATE('02-AUG-2007') - TO_DATE('25-JUL-2007')
FROM dual;

TO_DATE('02-AUG-2007') - TO_DATE('25-JUL-2007')
-----
8
```

## Usando colunas na aritmética

Os operandos não precisam ser números literais ou datas; eles também podem ser colunas de uma tabela. Na consulta a seguir, as colunas `name` e `price` são recuperadas da tabela `products`; note que 2 é somado ao valor na coluna `price` usando o operador de adição (+) para formar a expressão `price + 2`:

```
SELECT name, price + 2
FROM products;
```

NAME	PRICE+2
Modern Science	21.95
Chemistry	32
Supernova	27.99
Tank War	15.95
Z Files	51.99
2412: The Return	16.95
Space Force 9	15.49
From Another Planet	14.99
Classical Music	12.99
Pop 3	17.99
Creative Yell	16.99
My Front Line	15.49

Você também pode combinar mais de um operador em uma expressão. Na consulta a seguir, a coluna price é multiplicada por 3 e, então, 1 é somado ao valor resultante:

```
SELECT name, price * 3 + 1
FROM products;
```

NAME	PRICE*3+1
Modern Science	60.85
Chemistry	91
Supernova	78.97
Tank War	42.85
Z Files	150.97
2412: The Return	45.85
Space Force 9	41.47
From Another Planet	39.97
Classical Music	33.97
Pop 3	48.97
Creative Yell	45.97
My Front Line	41.47

As regras normais de precedência de operador aritmético se aplicam na linguagem SQL: multiplicação e divisão são efetuadas primeiro, seguidas pela adição e subtração. Se forem usados operadores de mesma precedência, eles serão efetuados da esquerda para a direita. Por exemplo, na expressão  $10 * 12 / 3 - 1$ , o primeiro cálculo seria 10 multiplicado por 12, gerando o resultado 120; então, 120 seria dividido por 3, resultando em 40; finalmente, 1 seria subtraído de 40, gerando 39:

```
SELECT 10 * 12 / 3 - 1
FROM dual;
```

```
10*12/3-1
-----
39
```

Também é possível usar parênteses () para especificar a ordem de execução dos operadores:

```
SELECT 10 * (12 / 3 - 1)
FROM dual;

10*(12/3-1)
-----
30
```

Nesse exemplo, os parênteses são usados para efetuar primeiro o cálculo de 12/3-1, cujo resultado é então multiplicado por 10, gerando 30 como resposta final.

## USANDO APELIDOS DE COLUNA

Como foi visto, quando você seleciona uma coluna de uma tabela, o Oracle usa a versão em letras maiúsculas do nome da coluna como cabeçalho da coluna na saída. Por exemplo, quando a coluna `price` é selecionada, o cabeçalho na saída resultante é `PRICE`. Quando você usa uma expressão, o Oracle remove os espaços e utiliza a expressão como cabeçalho. Você não está limitado a usar o cabeçalho gerado pelo Oracle; é possível fornecer o seu próprio cabeçalho, usando um *apelido*. Na consulta a seguir, a expressão `price * 2` recebe o apelido `DOUBLE_PRICE`:

```
SELECT price * 2 DOUBLE_PRICE
FROM products;

DOUBLE_PRICE
-----
39.9
60
51.98
27.9
99.98
29.9
26.98
25.98
21.98
31.98
29.98
26.98
```

Se quiser usar espaços e preservar a caixa do texto de seu nome alternativo, você deve colocar o texto entre aspas (""),

```
SELECT price * 2 "Double Price"
FROM products;

Double Price
-----
39.9
...
```

Também é possível usar a palavra-chave opcional `AS` antes do nome alternativo, como mostrado na consulta a seguir:

```
SELECT 10 * (12 / 3 - 1) AS "Computation"
FROM dual;
```

```
Computation
-----
30
```

## COMBINANDO SAÍDA DE COLUNA USANDO CONCATENAÇÃO

Você pode combinar os valores de coluna recuperados por uma consulta usando concatenação, o que permite criar uma saída mais amigável e significativa. Por exemplo, na tabela `customers`, as colunas `first_name` e `last_name` contêm o nome do cliente e, nas consultas anteriores, os valores de coluna foram exibidos de forma independente. Mas não seria ótimo combinar as colunas `first_name` e `last_name`? Você pode fazer isso usando o operador de concatenação (`||`), como mostrado na consulta a seguir; note que um caractere de espaço é adicionado depois da coluna `first_name` e, então, a coluna `last_name` é adicionada:

```
SELECT first_name || ' ' || last_name AS "Customer Name"
FROM customers;
```

```
Customer Name
-----
John Brown
Cynthia Green
Steve White
Gail Black
Doreen Blue
```

Os valores das colunas `first_name` e `last_name` são combinados na saída, sob o nome alternativo "Customer Name".

## VALORES NULOS

Como um banco de dados representa um valor desconhecido? Ele usa um valor especial chamado *valor nulo*. Um valor nulo não é uma string em branco — é um valor único e significa que o valor da coluna é desconhecido.

Quando você recupera uma coluna que contém um valor nulo, nada vê na saída dessa coluna. Você viu isso (ou melhor, não viu!) nos exemplos anteriores que recuperavam linhas da tabela `customers`: o cliente nº 4 tem um valor nulo na coluna `dob` e o cliente nº 5 tem um valor nulo na coluna `phone`. No caso de você não ter notado isso, aqui está a consulta novamente:

```
SELECT *
FROM customers;
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB	PHONE
1	John	Brown	01-JAN-65	800-555-1211
2	Cynthia	Green	05-FEB-68	800-555-1212
3	Steve	White	16-MAR-71	800-555-1213
4	Gail	Black		800-555-1214
5	Doreen	Blue	20-MAY-70	

Você também pode verificar a existência de valores nulos usando `IS NULL` em uma consulta. No exemplo a seguir, o cliente nº 4 é recuperado, pois seu valor de `dob` é nulo:

```
SELECT customer_id, first_name, last_name, dob
FROM customers
WHERE dob IS NULL;
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB
4	Gail	Black	

No próximo exemplo, o cliente nº 5 é recuperado, pois seu valor de `phone` é nulo:

```
SELECT customer_id, first_name, last_name, phone
FROM customers
WHERE phone IS NULL;
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	PHONE
5	Doreen	Blue	

Se os valores nulos nada exibem, como você sabe a diferença entre um valor nulo e uma string em branco? A resposta é: usando a função interna `NVL()` do Oracle. A função `NVL()` retorna outro valor no lugar de um nulo. Ela aceita dois parâmetros: uma coluna (ou, de forma geral, qualquer expressão que resulte em um valor) e o valor a ser retornado caso o primeiro parâmetro seja nulo. Na consulta a seguir, `NVL()` retorna a string 'Unknown phone number' (número de telefone desconhecido) quando a coluna `phone` contém um valor nulo:

```
SELECT customer_id, first_name, last_name,
NVL(phone, 'Unknown phone number') AS PHONE_NUMBER
FROM customers;
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	PHONE_NUMBER
1	John	Brown	800-555-1211
2	Cynthia	Green	800-555-1212
3	Steve	White	800-555-1213
4	Gail	Black	800-555-1214
5	Doreen	Blue	Unknown phone number

Você também pode usar `NVL()` para converter datas e números nulos. Na consulta a seguir, `NVL()` retorna a data 01-JAN-2000 quando a coluna `dob` contém um valor nulo:

```
SELECT customer_id, first_name, last_name, NVL(dob, '01-JAN-2000') AS DOB
FROM customers;
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB
1	John	Brown	01-JAN-65
2	Cynthia	Green	05-FEB-68

3	Steve	White	16-MAR-71
4	Gail	Black	01-JAN-00
5	Doreen	Blue	20-MAY-70

Note que o valor de `dob` do cliente nº 4 agora é exibido como `01-JAN-00`.

## EXIBINDO LINHAS DISTINTAS

Suponha que você quisesse obter a lista de clientes que compraram produtos de nossa loja imaginária. É possível obter essa lista usando a consulta a seguir, que recupera a coluna `customer_id` da tabela `purchases`:

```
SELECT customer_id
FROM purchases;
```

CUSTOMER\_ID

-----

1  
2  
3  
4  
1  
2  
3  
4  
3

A coluna `customer_id` contém as identificações (IDs) dos clientes que compraram um produto. Como você pode ver na saída retornada pela consulta, alguns clientes fizeram mais de uma compra e, portanto, aparecem duas vezes. Não seria ótimo se você pudesse eliminar as linhas duplicadas que contêm a mesma identificação de cliente? É possível fazer isso usando a palavra-chave `DISTINCT`. Na consulta a seguir, `DISTINCT` é usada para suprimir as linhas duplicadas:

```
SELECT DISTINCT customer_id
FROM purchases;
```

CUSTOMER\_ID

-----

1  
2  
4  
3

A partir dessa lista, é fácil ver que os clientes nº 1, 2, 3 e 4 fizeram compras; as linhas duplicadas foram suprimidas.

## COMPARANDO VALORES

A tabela a seguir lista os operadores que você pode usar para comparar valores:

Operador	Descrição
=	Igual
<> ou !=	Diferente
<	Menor que
>	Maior que
<=	Menor ou igual a
>=	Maior ou igual a
ANY	Compara um valor com qualquer valor em uma lista
SOME	Idêntico ao operador ANY; você deve usar ANY, em vez de SOME, pois ANY é mais utilizado e legível.
ALL	Compara um valor com todos os valores em uma lista

A consulta a seguir usa o operador diferente (<>) na cláusula WHERE para recuperar as linhas da tabela customers cujo valor na coluna customer\_id é diferente de 2:

```
SELECT *
FROM customers
WHERE customer_id <> 2;
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB	PHONE
1	John	Brown	01-JAN-65	800-555-1211
3	Steve	White	16-MAR-71	800-555-1213
4	Gail	Black		800-555-1214
5	Doreen	Blue	20-MAY-70	

A próxima consulta usa o operador > para recuperar as colunas product\_id e name da tabela products, onde a coluna product\_id é maior do que 8:

```
SELECT product_id, name
FROM products
WHERE product_id > 8;
```

PRODUCT_ID	NAME
9	Classical Music
10	Pop 3
11	Creative Yell
12	My Front Line

A consulta a seguir usa a pseudocoluna ROWNUM e o operador <= para recuperar as três primeiras linhas da tabela products:

```
SELECT ROWNUM, product_id, name
FROM products
WHERE ROWNUM <= 3;
```

ROWNUM	PRODUCT_ID	NAME
1	1	Modern Science
2	2	Chemistry
3	3	Supernova

Você usa o operador `ANY` em uma cláusula `WHERE` para comparar um valor com *qualquer* um dos valores de uma lista. Você deve colocar um operador `=`, `<>`, `<`, `>`, `<=` ou `>=` antes de `ANY`. A consulta a seguir usa `ANY` para recuperar as linhas da tabela `customers` onde o valor na coluna `customer_id` é maior do que qualquer um dos valores 2, 3 ou 4:

```
SELECT *
FROM customers
WHERE customer_id > ANY (2, 3, 4);
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB	PHONE
3	Steve	White	16-MAR-71	800-555-1213
4	Gail	Black		800-555-1214
5	Doreen	Blue	20-MAY-70	

Você usa o operador `ALL` em uma cláusula `WHERE` para comparar um valor com *todos* os valores de uma lista. Você deve colocar um operador `=`, `<>`, `<`, `>`, `<=` ou `>=` antes de `ALL`. A consulta a seguir usa `ALL` para recuperar as linhas da tabela `customers` onde o valor na coluna `customer_id` é maior do que os valores 2, 3 e 4:

```
SELECT *
FROM customers
WHERE customer_id > ALL (2, 3, 4);
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB	PHONE
5	Doreen	Blue	20-MAY-70	

Somente o cliente nº 5 é retornado, pois 5 é maior do que 2, 3 e 4.

## USANDO OS OPERADORES SQL

Os operadores SQL permitem limitar as linhas com base na correspondência de padrão de strings, listas de valores, intervalos de valores e valores nulos. Os operadores SQL estão listados na tabela a seguir:

Operador	Descrição
LIKE	Corresponde a padrões em strings
IN	Corresponde a listas de valores
BETWEEN	Corresponde a um intervalo de valores
IS NULL	Corresponde a valores nulos
IS NAN	Corresponde ao valor especial <code>NAN</code> , que significa “not a number” (não é número) (a partir do Oracle Database 10g)
IS INFINITE	Corresponde a valores <code>BINARY_FLOAT</code> e <code>BINARY_DOUBLE</code> infinitos (a partir do Oracle Database 10g)



Você também pode usar `NOT` para inverter o significado de um operador:

- `NOT LIKE`
- `NOT IN`
- `NOT BETWEEN`
- `IS NOT NULL`
- `IS NOT NAN`
- `IS NOT INFINITE`

Você vai aprender sobre os operadores `LIKE`, `IN` e `BETWEEN` nas seções a seguir.

## Usando o operador `LIKE`

Você usa o operador `LIKE` em uma cláusula `WHERE` para procurar um padrão em uma string. Os padrões são especificados usando uma combinação de caracteres normais e os dois caracteres curinga a seguir:

- **Sublinhado (`_`)** Corresponde a um caractere em uma posição específica
- **Porcentagem (`%`)** Corresponde a qualquer número de caracteres a partir da posição especificada

Por exemplo, considere o seguinte padrão:

```
'_o%'
```

O sublinhado (`_`) corresponde a qualquer caractere na primeira posição, `"o"` corresponde a um caractere `o` na segunda posição e a porcentagem (`%`) corresponde a todos os caracteres após o caractere `o`. A consulta a seguir usa o operador `LIKE` para procurar o padrão `'_o%'` na coluna `first_name` da tabela `customers`:

```
SELECT *
FROM customers
WHERE first_name LIKE '_o%';
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB	PHONE
1	John	Brown	01-JAN-65	800-555-1211
5	Doreen	Blue	20-MAY-70	

Como você pode ver a partir dos resultados, são retornadas duas linhas, pois as strings `John` e `Doreen` têm `"o"` como segundo caractere. A próxima consulta usa `NOT LIKE` para obter as linhas não recuperadas pela consulta anterior:

```
SELECT *
FROM customers
WHERE first_name NOT LIKE '_o%';
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB	PHONE
2	Cynthia	Green	05-FEB-68	800-555-1212
3	Steve	White	16-MAR-71	800-555-1213
4	Gail	Black		800-555-1214

Se precisar procurar os caracteres de sublinhado ou porcentagem reais em uma string, use a opção `ESCAPE` para identificá-los. Por exemplo, considere o padrão a seguir:

```
'%\\%%' ESCAPE '\\'
```

O caractere após `ESCAPE` diz ao banco de dados como diferenciar os caracteres a serem pesquisados dos curingas, sendo que no exemplo é usado o caractere de barra invertida (`\`). O primeiro `%` é tratado como curinga e corresponde a qualquer número de caracteres; o segundo `%` é tratado como um caractere real a ser procurado; o terceiro `%` é tratado como curinga e corresponde a qualquer número de caracteres. A consulta a seguir usa a tabela `promotions`, que contém os detalhes dos produtos com descontos na loja (você vai aprender mais sobre essa tabela posteriormente neste livro). A consulta usa o operador `LIKE` para procurar o padrão `'%\\%%' ESCAPE '\\'` na coluna `name` da tabela `promotions`:

```
SELECT name
FROM promotions
WHERE name LIKE '%\\%%' ESCAPE '\\';
```

```
NAME
-----
10% off Z Files
20% off Pop 3
30% off Modern Science
20% off Tank War
10% off Chemistry
20% off Creative Yell
15% off My Front Line
```

Como você pode ver a partir dos resultados, a consulta retorna as linhas cujos nomes contêm um caractere de porcentagem.

## Usando o operador IN

Você pode usar o operador `IN` em uma cláusula `WHERE` para recuperar as linhas cujo valor de coluna está em uma lista. A consulta a seguir usa `IN` para recuperar as linhas da tabela `customers` onde `customer_id` é 2, 3 ou 5:

```
SELECT *
FROM customers
WHERE customer_id IN (2, 3, 5);
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB	PHONE
2	Cynthia	Green	05-FEB-68	800-555-1212
3	Steve	White	16-MAR-71	800-555-1213
5	Doreen	Blue	20-MAY-70	

NOT IN recupera as linhas não recuperadas por IN:

```
SELECT *
FROM customers
WHERE customer_id NOT IN (2, 3, 5);
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB	PHONE
1	John	Brown	01-JAN-65	800-555-1211
4	Gail	Black		800-555-1214

Um ponto importante a saber é que NOT IN retorna falso se o valor que está na lista é nulo. Isso é ilustrado pela consulta a seguir, que não retorna uma linha, pois o valor nulo está incluído na lista:

```
SELECT *
FROM customers
WHERE customer_id NOT IN (2, 3, 5, NULL);
```

no rows selected

### CUIDADO

NOT IN retorna falso se um valor na lista é nulo. Isso é importante, pois, como você pode usar qualquer expressão na lista e não apenas valores literais, talvez seja difícil identificar quando um valor nulo ocorre. Considere o uso de NVL () nas expressões que possam retornar um valor nulo.

## Usando o operador BETWEEN

É possível usar o operador BETWEEN em uma cláusula WHERE para recuperar as linhas cujo valor de coluna está em um intervalo especificado. O intervalo inclui os valores das duas extremidades do intervalo. A consulta a seguir usa BETWEEN para recuperar as linhas da tabela customers onde customer\_id está entre 1 e 3:

```
SELECT *
FROM customers
WHERE customer_id BETWEEN 1 AND 3;
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB	PHONE
1	John	Brown	01-JAN-65	800-555-1211
2	Cynthia	Green	05-FEB-68	800-555-1212
3	Steve	White	16-MAR-71	800-555-1213

NOT BETWEEN recupera as linhas não recuperadas por BETWEEN:

```
SELECT *
FROM customers
WHERE customer_id NOT BETWEEN 1 AND 3;
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB	PHONE
4	Gail	Black		800-555-1214
5	Doreen	Blue	20-MAY-70	

## USANDO OS OPERADORES LÓGICOS

Os operadores lógicos permitem limitar as linhas com base em condições lógicas. Eles estão listados na tabela a seguir:

Operador	Descrição
$x$ AND $y$	Retorna verdadeiro quando $x$ e $y$ são verdadeiros
$x$ OR $y$	Retorna verdadeiro quando $x$ ou $y$ são verdadeiros
NOT $x$	Retorna verdadeiro se $x$ for falso e retorna falso se $x$ for verdadeiro

A consulta a seguir ilustra o uso do operador AND para recuperar as linhas da tabela `customers` onde *as duas* condições a seguir são verdadeiras:

- A coluna `dob` é maior do que 1º de janeiro de 1970.
- A coluna `customer_id` é maior do que 3.

```
SELECT *
FROM customers
WHERE dob > '01-JAN-1970'
AND customer_id > 3;
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB	PHONE
5	Doreen	Blue	20-MAY-70	

A próxima consulta ilustra o uso do operador OR para recuperar as linhas da tabela `customers` onde *uma das duas* condições a seguir é verdadeira:

- A coluna `dob` é maior do que 1º de janeiro de 1970.
- A coluna `customer_id` é maior do que 3.

```
SELECT *
FROM customers
WHERE dob > '01-JAN-1970'
OR customer_id > 3;
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB	PHONE
3	Steve	White	16-MAR-71	800-555-1213
4	Gail	Black		800-555-1214
5	Doreen	Blue	20-MAY-70	

Você também pode usar AND e OR para combinar expressões em uma cláusula WHERE, conforme mostra a próxima seção.

## PRECEDÊNCIA DE OPERADORES

Se você combinar AND e OR na mesma expressão, o operador AND terá precedência sobre o operador OR (“ter precedência sobre” significa que ele será executado primeiro). Os operadores de comparação têm precedência sobre AND. Evidentemente, você pode anular a precedência padrão usando parênteses para indicar a ordem em que deseja executar as expressões.

O exemplo a seguir recupera as linhas da tabela `customers` onde *uma das duas* condições a seguir é verdadeira:

- A coluna `dob` é maior do que 1º de janeiro de 1970.
- A coluna `customer_id` é menor do que 2 e a coluna `phone` tem 1211 no final.

```
SELECT *
FROM customers
WHERE dob > '01-JAN-1970'
OR customer_id < 2
AND phone LIKE '%1211';
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB	PHONE
1	John	Brown	01-JAN-65	800-555-1211
3	Steve	White	16-MAR-71	800-555-1213
5	Doreen	Blue	20-MAY-70	

Conforme mencionado anteriormente, AND tem precedência sobre OR; portanto, você pode pensar na cláusula WHERE da consulta anterior como segue:

```
dob > '01-JAN-1970' OR (customer_id < 2 AND phone LIKE '%1211')
```

Portanto, os clientes nº 1, 3 e 5 são retornados pela consulta.

## CLASSIFICANDO LINHAS USANDO A CLÁUSULA ORDER BY

A cláusula ORDER BY é usada para classificar as linhas recuperadas por uma consulta. A cláusula ORDER BY pode especificar uma ou mais colunas nas quais os dados serão classificados; além disso, a cláusula ORDER BY deve vir após a cláusula FROM ou a cláusula WHERE (se uma cláusula WHERE for fornecida). A consulta a seguir usa ORDER BY para classificar por `last_name` as linhas recuperadas da tabela `customers`:

```
SELECT *
FROM customers
ORDER BY last_name;
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB	PHONE
4	Gail	Black		800-555-1214
5	Doreen	Blue	20-MAY-70	
1	John	Brown	01-JAN-65	800-555-1211
2	Cynthia	Green	05-FEB-68	800-555-1212
3	Steve	White	16-MAR-71	800-555-1213

Por padrão, `ORDER BY` classifica as colunas em ordem crescente (os valores menores aparecem primeiro). É possível usar a palavra-chave `DESC` para classificar as colunas em ordem decrescente (os valores maiores aparecem primeiro). Você também pode usar a palavra-chave `ASC` para especificar explicitamente uma classificação crescente — como mencionamos, a ordem crescente é o padrão, mas você ainda pode especificá-la se quiser tornar claro qual é a ordem da classificação. A próxima consulta usa `ORDER BY` para classificar as linhas recuperadas da tabela `customers` por `first_name` em ordem crescente e `last_name` em ordem decrescente:

```
SELECT *
FROM customers
ORDER BY first_name ASC, last_name DESC;
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB	PHONE
2	Cynthia	Green	05-FEB-68	800-555-1212
5	Doreen	Blue	20-MAY-70	
4	Gail	Black		800-555-1214
1	John	Brown	01-JAN-65	800-555-1211
3	Steve	White	16-MAR-71	800-555-1213

Um número de posição de coluna na cláusula `ORDER BY` pode ser usado para indicar qual coluna deve ser classificada: use 1 para classificar pela primeira coluna selecionada, 2 para classificar pela segunda coluna selecionada e assim por diante. Na consulta a seguir, a coluna 1 (a coluna `customer_id`) é usada para classificar as linhas:

```
SELECT customer_id, first_name, last_name
FROM customers
ORDER BY 1;
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME
1	John	Brown
2	Cynthia	Green
3	Steve	White
4	Gail	Black
5	Doreen	Blue

Como a coluna `customer_id` está na posição 1 após a palavra-chave `SELECT`, ela é a coluna usada na classificação.

## EXECUTANDO INSTRUÇÕES SELECT QUE USAM DUAS TABELAS

Os esquemas de banco de dados têm mais de uma tabela, que armazenam dados diferentes. Por exemplo, o esquema `store` tem tabelas que armazenam informações sobre clientes, produtos, funcionários etc. Até agora, todas as consultas mostradas neste livro recuperam linhas de apenas uma tabela. No mundo real, freqüentemente você precisará obter informações de várias tabelas; por exemplo, talvez precise obter o nome de um produto e o nome de seu tipo de produto. Nesta seção, você irá aprender a executar consultas que usam duas tabelas; posteriormente, verá consultas que utilizam mais de duas tabelas.

Vamos voltar ao exemplo onde você queria obter o nome do produto nº 3 e seu tipo de produto. O nome do produto é armazenado na coluna `name` da tabela `products` e o nome do

tipo de produto é armazenado na coluna `name` da tabela `product_types`. As tabelas `products` e `product_types` são relacionadas entre si por meio da coluna de chave estrangeira `product_type_id`; a coluna `product_type_id` (a chave estrangeira) da tabela `products` aponta para a coluna `product_type_id` (a chave primária) da tabela `product_types`. A consulta a seguir recupera as colunas `name` e `product_type_id` da tabela `products` para o produto nº 3:

```
SELECT name, product_type_id
FROM products
WHERE product_id = 3;
```

NAME	PRODUCT_TYPE_ID
Supernova	2

A próxima consulta recupera a coluna `name` da tabela `product_types` para `product_type_id` 2:

```
SELECT name
FROM product_types
WHERE product_type_id = 2;
```

NAME
Video

A partir disso, é possível saber que o produto nº 3 é um vídeo. Nada de novo até aqui, mas o que você quer realmente é recuperar o nome do produto e o nome de seu tipo de produto usando apenas uma consulta. Para tanto, use uma *join de tabela* na consulta. Para unir duas tabelas em uma consulta, inclua as duas na cláusula `FROM` da consulta e inclua as colunas relacionadas de cada tabela na cláusula `WHERE`. No nosso exemplo de consulta, a cláusula `FROM` se torna:

```
FROM products, product_types
```

E a cláusula `WHERE` é:

```
WHERE products.product_type_id = product_types.product_type_id
AND products.product_id = 3;
```

A *join* é a primeira condição na cláusula `WHERE` (`products.product_type_id = product_types.product_type_id`); normalmente, as colunas usadas na *join* são uma chave primária de uma tabela e uma chave estrangeira da outra tabela. Como o operador de igualdade (=) é usado na condição da *join*, esta *join* é conhecida como *equijoin*. A segunda condição na cláusula `WHERE` (`products.product_id = 3`) obtém o produto nº 3, o produto que estamos interessados em ver.

Você vai notar que as tabelas, assim como suas colunas, são incluídas na cláusula `WHERE`. Isso porque existe uma coluna `product_type_id` na tabela `products` e outra na tabela `product_types`, e é preciso informar ao banco de dados em qual tabela está a coluna que se deseja usar. Se as colunas tivessem nomes diferentes, você poderia omitir os nomes de tabela, mas eles sempre devem ser incluídos para tornar claro de onde vêm as colunas.

A cláusula `SELECT` da consulta é:

```
SELECT products.name, product_types.name
```

Observe que as tabelas e suas colunas são especificadas novamente.

Reunindo tudo, a consulta completa é:

```
SELECT products.name, product_types.name
FROM products, product_types
WHERE products.product_type_id = product_types.product_type_id
AND products.product_id = 3;
```

NAME	NAME
Supernova	Video

Perfeito! É exatamente isso que queríamos: o nome do produto e o nome do tipo de produto. A próxima consulta obtém todos os produtos e os ordena pela coluna `products.name`:

```
SELECT products.name, product_types.name
FROM products, product_types
WHERE products.product_type_id = product_types.product_type_id
ORDER BY products.name;
```

NAME	NAME
2412: The Return	Video
Chemistry	Book
Classical Music	CD
Creative Yell	CD
From Another Planet	DVD
Modern Science	Book
Pop 3	CD
Space Force 9	DVD
Supernova	Video
Tank War	Video
Z Files	Video

Observe que o produto com o nome “My Front Line” está ausente dos resultados. O valor de `product_type_id` para essa linha de produto é nulo e a condição da join não retorna a linha. Você irá ver como incluir essa linha posteriormente, na seção “Joins externas”.

A sintaxe de join que você viu nesta seção utiliza a sintaxe do Oracle para joins, que é baseada no padrão SQL/86 do ANSI (American National Standards Institute). Com o lançamento do Oracle Database 9i, o banco de dados também implementa a sintaxe do padrão SQL/92 do ANSI para joins; você verá essa nova sintaxe posteriormente, na seção “Realizando joins usando a sintaxe SQL/92”. Você deve usar o padrão SQL/92 em suas consultas quando trabalhar com o Oracle Database 9i e superiores e deve usar consultas SQL/86 somente quando estiver utilizando o Oracle Database 8i e inferiores.

## USANDO APELIDOS DE TABELA

Na seção anterior, você viu a consulta a seguir:

```
SELECT products.name, product_types.name
FROM products, product_types
WHERE products.product_type_id = product_types.product_type_id
ORDER BY products.name;
```



Note que os nomes de tabela `products` e `product_types` são usados nas cláusulas `SELECT` e `WHERE`. Repetir nomes de tabela é digitação redundante. Uma maneira melhor é definir apelidos de tabela na cláusula `FROM` e, então, usar esses apelidos ao referenciar as tabelas em outro lugar na consulta. Por exemplo, a consulta a seguir usa o apelido `p` para a tabela `products` e `pt` para a tabela `product_types`; observe que os apelidos de tabela são especificados na cláusula `FROM` e colocados antes das colunas no restante da consulta:

```
SELECT p.name, pt.name
FROM products p, product_types pt
WHERE p.product_type_id = pt.product_type_id
ORDER BY p.name;
```

### DICA

*Os apelidos de tabela também tornam suas consultas mais legíveis, especialmente quando você começa a escrever consultas mais longas, que usam muitas tabelas.*

## PRODUTOS CARTESIANOS

Se estiver faltando uma condição de join, você acabará unindo todas as linhas de uma tabela com todas as linhas da outra; esse conjunto de resultados é conhecido como *produto cartesiano*. Quando isso ocorrer, muitas linhas serão retornadas pela consulta. Por exemplo, suponha que você tivesse uma tabela contendo 50 linhas e uma segunda tabela contendo 100 linhas. Se você selecionasse colunas dessas duas tabelas sem uma join, obteria 5.000 linhas de retorno. Esse resultado aconteceria porque cada linha da tabela 1 seria juntada a cada linha da tabela 2, o que geraria um total de 50 linhas multiplicadas por 100 linhas ou 5.000 linhas. O exemplo a seguir mostra um subconjunto das linhas de um produto cartesiano entre as tabelas `product_types` e `products`:

```
SELECT pt.product_type_id, p.product_id
FROM product_types pt, products p;
```

PRODUCT_TYPE_ID	PRODUCT_ID
1	1
1	2
1	3
1	4
1	5
...	
5	8
5	9
5	10
5	11
5	12

60 rows selected.

No total, 60 linhas são selecionadas, pois as tabelas `product_types` e `products` contêm 5 e 12 linhas, respectivamente, e  $5 \times 12 = 60$ .

## EXECUTANDO INSTRUÇÕES SELECT QUE USAM MAIS DE DUAS TABELAS

Joins podem ser usadas para conectar qualquer número de tabelas. Você usa a seguinte fórmula para calcular o número de joins que precisará em sua cláusula `WHERE`:

*Número de joins = número de tabelas usadas na consulta – 1*

Por exemplo, a consulta a seguir usa duas tabelas e uma join:

```
SELECT p.name, pt.name
FROM products p, product_types pt
WHERE p.product_type_id = pt.product_type_id
ORDER BY p.name;
```

Vamos considerar um exemplo mais complicado, que usa quatro tabelas — e, portanto, exige três joins. Digamos que você queira ver as seguintes informações:

- As compras feitas por cada cliente (vêm da tabela `purchases`)
- O nome e o sobrenome do cliente (vêm da tabela `customers`)
- O nome do produto que eles compraram (vem da tabela `products`)
- O nome do tipo de produto (vem da tabela `product_types`)

Para ver essas informações, você precisa consultar as tabelas `customers`, `purchases`, `products` e `product_types` e as joins usarão os relacionamentos de chave estrangeira entre essas tabelas. A lista a seguir mostra as joins necessárias:

1. Para obter o cliente que fez a compra, una as tabelas `customers` e `purchases` usando as colunas `customer_id` (`customers.customer_id = purchases.customer_id`).
2. Para obter o produto adquirido, una as tabelas `products` e `purchases` usando as colunas `product_id` (`products.product_id = purchases.product_id`).
3. Para obter o nome do tipo de produto, una as tabelas `products` e `product_types` usando as colunas `product_type_id` (`products.product_type_id = product_types.product_type_id`).

A consulta a seguir usa estas joins; observe que usamos apelidos de tabela e mudamos o nome dos cabeçalhos do nome do produto para `PRODUCT` e o nome do tipo de produto para `TYPE`:

```
SELECT c.first_name, c.last_name, p.name AS PRODUCT, pt.name AS TYPE
FROM customers c, purchases pr, products p, product_types pt
WHERE c.customer_id = pr.customer_id
AND p.product_id = pr.product_id
AND p.product_type_id = pt.product_type_id
ORDER BY p.name;
```

FIRST_NAME	LAST_NAME	PRODUCT	TYPE
John	Brown	Chemistry	Book
Cynthia	Green	Chemistry	Book
Steve	White	Chemistry	Book
Gail	Black	Chemistry	Book
John	Brown	Modern Science	Book
Cynthia	Green	Modern Science	Book
Steve	White	Modern Science	Book
Gail	Black	Modern Science	Book
Steve	White	Supernova	Video

As consultas com várias tabelas mostradas até aqui utilizam o operador de igualdade (=) nas condições de join; esses tipos de join são conhecidas como *equijoins*. Conforme você verá na próxima seção, a equijoin não é a única que pode ser usada.

## CONDIÇÕES DE JOIN E TIPOS DE JOIN

Nesta seção, você vai explorar as condições de join e os tipos de join que permitem criar consultas mais avançadas. Existem dois tipos de condições de join, os quais são baseados no operador utilizado na join:

- As **equijoins** utilizam o operador de igualdade (=). Você já viu exemplos de equijoins.
- As **não-equijoins** utilizam um operador que não é o de igualdade, como <, >, BETWEEN etc. Você vai ver um exemplo de não-equijoin em breve.

Existem ainda três tipos diferentes de joins:

- As **joins internas** retornam uma linha *somente* quando as colunas da join contêm valores que satisfazem essa condição. Isso significa que, se uma linha tem um valor nulo em uma das colunas na condição de join, ela não é retornada. Os exemplos que você viu até aqui eram join internas.
- As **joins externas** retornam uma linha *mesmo quando* uma das colunas na condição de join contém um valor nulo.
- As **autojoins** retornam linhas unidas na mesma tabela.

Você vai aprender sobre não-equijoins, joins externas e autojoins a seguir.

### Não-equijoins

Uma não-equijoin utiliza um operador que não é o de igualdade (=) na join. Esses operadores são: diferente (<>), menor que (<), maior que (>), menor ou igual a (<=), maior ou igual a (>=), LIKE, IN e BETWEEN. Situações em que você precisa utilizar uma não-equijoin são raras, mas já precisei utilizar uma; nessas ocasiões, tive de usar o operador BETWEEN.

Por exemplo, digamos que você queira obter os níveis salariais dos funcionários. Primeiro, a consulta a seguir recupera os níveis salariais da tabela `salary_grades`:

```
SELECT *
FROM salary_grades;
```

SALARY_GRADE_ID	LOW_SALARY	HIGH_SALARY
1	1	250000
2	250001	500000
3	500001	750000
4	750001	999999

A próxima consulta usa uma não-equijoin para recuperar o salário e os níveis salariais dos funcionários; o nível salarial é determinado com o operador BETWEEN:

```
SELECT e.first_name, e.last_name, e.title, e.salary, sg.salary_grade_id
FROM employees e, salary_grades sg
WHERE e.salary BETWEEN sg.low_salary AND sg.high_salary
ORDER BY salary_grade_id;
```

FIRST_NAME	LAST_NAME	TITLE	SALARY	SALARY_GRADE_ID
Fred	Hobbs	Salesperson	150000	1
Susan	Jones	Salesperson	500000	2
Ron	Johnson	Sales Manager	600000	3
James	Smith	CEO	800000	4

Nessa consulta, o operador BETWEEN retorna verdadeiro se o salário do funcionário está entre o salário mais baixo e o salário mais alto do nível salarial; quando verdadeiro é retornado, o nível salarial encontrado é o do funcionário. Por exemplo, o salário de Fred Hobbs é de US\$150.000; isso está entre o salário mais baixo que é US\$1 e o salário mais alto que é US\$250.000 na tabela salary\_grades para salary\_grade\_id igual a 1; portanto, o nível salarial de Fred Hobbs é 1. Da mesma forma, o salário de Susan Jones é US\$500.000; isso está entre o salário mais baixo que é US\$250.001 e o salário mais alto que é US\$500.000 para a ID de nível salarial 2; portanto, o nível salarial de Susan Jones é 2. Ron Johnson e James Smith têm níveis salariais 3 e 4, respectivamente.

## Joins externas

Uma join externa recupera uma linha mesmo quando uma de suas colunas contém um valor nulo. Você realiza uma join externa fornecendo o operador desta na condição de join; o operador de join externa proprietário do Oracle é um sinal de adição entre parênteses (+).

Vamos ver um exemplo. Lembra-se da consulta anterior que não mostrava o produto “My Front Line” porque seu valor de product\_type\_id era nulo? Você pode usar uma join externa para obter essa linha. Na consulta a seguir, observe que o operador de join externa (+) do Oracle está no lado oposto da coluna product\_type\_id na tabela product (essa é a coluna que contém o valor nulo):

```
SELECT p.name, pt.name
FROM products p, product_types pt
WHERE p.product_type_id = pt.product_type_id (+)
ORDER BY p.name;
```

NAME	NAME
-----	-----
2412: The Return	Video
Chemistry	Book
Classical Music	CD
Creative Yell	CD
From Another Planet	DVD
Modern Science	Book
My Front Line	
Pop 3	CD
Space Force 9	DVD
Supernova	Video
Tank War	Video
Z Files	Video

Note que “My Front Line” — o produto com `product_type_id` nulo — agora é recuperado, mesmo sendo nulo o valor de `product_type_id`.



#### NOTA

Você pode colocar o operador de *join* externa em qualquer lado do operador de *join*, mas sempre o coloque no lado oposto da coluna que contém o valor nulo.

A consulta a seguir retorna os mesmos resultados da anterior, mas observe que a coluna com o valor nulo (`pt.product_type_id`) e o operador de *join* externa da Oracle estão à esquerda do operador de igualdade:

```
SELECT p.name, pt.name
FROM products p, product_types pt
WHERE pt.product_type_id (+) = p.product_type_id
ORDER BY p.name;
```

### Joins externas esquerda e direita

As joins externas podem ser divididas em dois tipos:

- Join externa esquerda
- Join externa direita

Para entender a diferença entre a *join* externa esquerda e direita, considere a sintaxe a seguir:

```
SELECT...
FROM tabela1, tabela2
...
```

Suponha que as tabelas serão unidas em `tabela1.coluna1` e `tabela2.coluna2`. Além disso, suponha que `tabela1` contenha uma linha com um valor nulo na `coluna1`. Para realizar uma *join* externa esquerda, a cláusula `WHERE` fica:

```
WHERE tabela1.coluna1 = tabela2.coluna2 (+);
```

**NOTA**

Em uma *join* externa esquerda, o operador de *join* externa fica, na verdade, à direita do operador de igualdade.

Em seguida, suponha que a *tabela2* contenha uma linha com um valor nulo na *coluna2*. Para realizar uma *join* externa direita, você troca a posição do operador de *join* externa do Oracle para a *esquerda* do operador de igualdade e a cláusula *WHERE* se torna:

```
WHERE tabela1.coluna1 (+) = tabela2.coluna2;
```

**NOTA**

Conforme será visto, se tanto a *tabela1* como a *tabela2* contêm linhas com valores nulos, você obtém resultados diferentes, dependendo de usar uma *join* externa esquerda ou direita.

Vamos ver alguns exemplos concretos para tornar as *joins* externas esquerda e direita mais claras.

**Um exemplo de *join* externa esquerda** A consulta a seguir usa uma *join* externa esquerda; observe que o operador de *join* externa do Oracle aparece à *direita* do operador de igualdade:

```
SELECT p.name, pt.name
FROM products p, product_types pt
WHERE p.product_type_id = pt.product_type_id (+)
ORDER BY p.name;
```

NAME	NAME
2412: The Return	Video
Chemistry	Book
Classical Music	CD
Creative Yell	CD
From Another Planet	DVD
Modern Science	Book
My Front Line	
Pop 3	CD
Space Force 9	DVD
Supernova	Video
Tank War	Video
Z Files	Video

Observe que todas as linhas da tabela *products* são recuperadas, incluindo a linha “My Front Line”, que tem um valor nulo na coluna *product\_type\_id*.

**Um exemplo de *join* externa direita** A tabela *product\_types* contém um tipo de produto não referenciado na tabela *products* (não existem produtos do tipo *magazine* na tabela *products*); o tipo de produto *magazine* aparece no final do exemplo a seguir:

```
SELECT *
FROM product_types;
```

```

PRODUCT_TYPE_ID NAME
-----
1 Book
2 Video
3 DVD
4 CD
5 Magazine

```

Você pode recuperar a revista em uma união das tabelas `products` e `product_types` usando uma join externa direita, como mostrado na consulta a seguir; observe que o operador de join externa do Oracle aparece à esquerda do operador de igualdade:

```

SELECT p.name, pt.name
FROM products p, product_types pt
WHERE p.product_type_id (+) = pt.product_type_id
ORDER BY p.name;

```

NAME	NAME
2412: The Return	Video
Chemistry	Book
Classical Music	CD
Creative Yell	CD
From Another Planet	DVD
Modern Science	Book
Pop 3	CD
Space Force 9	DVD
Supernova	Video
Tank War	Video
Z Files	Video
	Magazine

### Limitações das joins externas

Existem limitações ao se usar joins externas e você vai aprender sobre algumas delas nesta seção.

Você só pode colocar o operador de join externa em um lado da join (não em ambos). Se você tentar colocar o operador de join externa da Oracle nos dois lados, obterá um erro, como mostrado:

```

SQL> SELECT p.name, pt.name
2 FROM products p, product_types pt
3 WHERE p.product_type_id (+) = pt.product_type_id (+);
WHERE p.product_type_id (+) = pt.product_type_id (+)
*
ERROR at line 3:
ORA-01468: a predicate may reference only one outer-joined table

```

Você não pode usar uma condição de join externa com o operador IN:

```

SQL> SELECT p.name, pt.name
2 FROM products p, product_types pt
3 WHERE p.product_type_id (+) IN (1, 2, 3, 4);
WHERE p.product_type_id (+) IN (1, 2, 3, 4)
*
ERROR at line 3:
ORA-01719: outer join operator (+) not allowed in operand of OR or IN

```

Você não pode usar uma condição de join externa com outra join que esteja usando o operador OR:

```
SQL> SELECT p.name, pt.name
       2 FROM products p, product_types pt
       3 WHERE p.product_type_id (+) = pt.product_type_id
       4 OR p.product_type_id = 1;
WHERE p.product_type_id (+) = pt.product_type_id
                        *
```

ERROR at line 3:  
ORA-01719: outer join operator (+) not allowed in operand of OR or IN

### NOTA

Essas são as limitações normalmente encontradas ao se usar o operador de join externa. Para ver todas as limitações, leia o manual *Oracle Database SQL Reference* da Oracle Corporation.

## Autojoins

Uma autojoin é uma join feita na mesma tabela. Para realizar uma autojoin, você deve usar um apelido de tabela diferente para identificar cada referência à tabela na consulta. Vamos considerar um exemplo. A tabela `employees` tem uma coluna `manager_id` que contém o `employee_id` do gerente de cada funcionário; se o funcionário não tem gerente, então o `manager_id` é nulo. A tabela `employees` contém as seguintes linhas:

EMPLOYEE_ID	MANAGER_ID	FIRST_NAME	LAST_NAME	TITLE	SALARY
1		James	Smith	CEO	800000
2	1	Ron	Johnson	SalesManager	600000
3	2	Fred	Hobbs	Salesperson	150000
4	2	Susan	Jones	Salesperson	500000

James Smith — o diretor executivo (CEO) — tem um valor nulo para `manager_id`, significando que ele não tem gerente. Susan Jones e Fred Hobbs são gerenciados por Ron Johnson e este é gerenciado por James Smith.

Você pode usar uma autojoin para exibir os nomes de cada funcionário e seu gerente. Na consulta a seguir, a tabela `employees` é referenciada duas vezes, usando dois apelidos: `w` e `m`. O apelido `w` é usado para obter o nome do funcionário e o apelido `m` é usado para obter o nome do gerente. A autojoin é feita entre `w.manager_id` e `m.employee_id`:

```
SELECT w.first_name || ' ' || w.last_name || ' works for ' ||
       m.first_name || ' ' || m.last_name
FROM employees w, employees m
WHERE w.manager_id = m.employee_id
ORDER BY w.first_name;
```

```
W.FIRST_NAME||' '||W.LAST_NAME||'WORKSFOR'||M.FIRST_NA
-----
Fred Hobbs works for Ron Johnson
Ron Johnson works for James Smith
Susan Jones works for Ron Johnson
```

Como `manager_id` de James Smith é nulo, a condição de join não retorna a linha.



Você pode realizar joins externas combinadas com autojoins. Na consulta a seguir, uma join externa é usada com a autojoin mostrada no exemplo anterior para recuperar a linha de James Smith. A função NVL() é usada para fornecer uma string indicando que James Smith trabalha para os acionistas (ele é o diretor executivo, portanto, se reporta aos acionistas da loja):

```
SELECT w.last_name || ' works for ' ||
       NVL(m.last_name, 'the shareholders')
FROM employees w, employees m
WHERE w.manager_id = m.employee_id (+)
ORDER BY w.last_name;
```

```
W.LAST_NAME || 'WORKSFOR' || NVL(M.LAST_N
-----
Hobbs works for Johnson
Johnson works for Smith
Jones works for Johnson
Smith works for the shareholders
```

## REALIZANDO JOINS USANDO A SINTAXE SQL/92

As joins vistas até aqui usam a sintaxe da Oracle, que é baseada no padrão SQL/86 do ANSI. Com o lançamento do Oracle Database 9i, o banco de dados implementa a sintaxe do padrão SQL/92 do ANSI para joins e você deve usar SQL/92 em suas consultas. Você verá como utilizar o SQL/92 nesta seção, incluindo seu uso para evitar produtos cartesianos indesejados.

### NOTA

*Você pode visitar o site do ANSI no endereço [www.ansi.org](http://www.ansi.org).*

## Realizando joins internas em duas tabelas usando SQL/92

Anteriormente, você viu esta consulta, que usa o padrão SQL/86 para realizar uma join interna:

```
SELECT p.name, pt.name
FROM products p, product_types pt
WHERE p.product_type_id = pt.product_type_id
ORDER BY p.name;
```

O padrão SQL/92 introduz as cláusulas INNER JOIN e ON para realizar uma join interna. O exemplo a seguir reescreve a consulta anterior usando as cláusulas INNER JOIN e ON:

```
SELECT p.name, pt.name
FROM products p INNER JOIN product_types pt
ON p.product_type_id = pt.product_type_id
ORDER BY p.name;
```

Você também pode usar operadores de não-equijoin com a cláusula ON. Anteriormente, você viu a consulta a seguir, que usa o padrão SQL/86 para realizar uma não-equijoin:

```
SELECT e.first_name, e.last_name, e.title, e.salary, sg.salary_grade_id
FROM employees e, salary_grades sg
WHERE e.salary BETWEEN sg.low_salary AND sg.high_salary
ORDER BY salary_grade_id;
```

O exemplo a seguir reescreve essa consulta para usar o padrão SQL/92:

```
SELECT e.first_name, e.last_name, e.title, e.salary, sg.salary_grade_id
FROM employees e INNER JOIN salary_grades sg
ON e.salary BETWEEN sg.low_salary AND sg.high_salary
ORDER BY salary_grade_id;
```

## Simplificando joins com a palavra-chave USING

O padrão SQL/92 permite simplificar ainda mais a condição de join com a cláusula `USING`, mas com as seguintes limitações:

- A consulta deve usar uma equijoin
- As colunas na equijoin devem ter o mesmo nome

A maioria das joins que você vai realizar será constituída de equijoins e, se você sempre usar o mesmo nome da chave primária para suas chaves estrangeiras, satisfará essas limitações. A consulta a seguir usa a cláusula `USING`, em vez de `ON`:

```
SELECT p.name, pt.name
FROM products p INNER JOIN product_types pt
USING (product_type_id);
```

Se você quisesse recuperar `product_type_id`, deveria fornecer apenas esse nome de coluna sozinho, sem um nome ou apelido de tabela na cláusula `SELECT`, como:

```
SELECT p.name, pt.name, product_type_id
FROM products p INNER JOIN product_types pt
USING (product_type_id);
```

Se você tentar fornecer um apelido de tabela com a coluna, como `p.product_type_id`, por exemplo, obterá um erro:

```
SQL> SELECT p.name, pt.name, p.product_type_id
      2 FROM products p INNER JOIN product_types pt
      3 USING (product_type_id);
SELECT p.name, pt.name, p.product_type_id
                        *
ERROR at line 1:
ORA-25154: column part of USING clause cannot have qualifier
```

Além disso, você só usa o nome de coluna sozinho dentro da cláusula `USING`. Por exemplo, se você especificar `USING (p.product_type_id)` na consulta anterior, em vez de `USING (product_type_id)`, obterá um erro:

```
SQL> SELECT p.name, pt.name, p.product_type_id
      2 FROM products p INNER JOIN product_types pt
      3 USING (p.product_type_id);
USING (p.product_type_id)
                        *
ERROR at line 3:
ORA-01748: only simple column names allowed here
```

**CUIDADO**

*Não use um nome ou apelido de tabela ao referenciar colunas usadas em uma cláusula USING. Se fizer isso, você receberá um erro.*

## Realizando joins internas em mais de duas tabelas usando SQL/92

Anteriormente, você viu a seguinte consulta SQL/86, que recupera linhas das tabelas customers, purchases, products e product\_types:

```
SELECT c.first_name, c.last_name, p.name AS PRODUCT, pt.name AS TYPE
FROM customers c, purchases pr, products p, product_types pt
WHERE c.customer_id = pr.customer_id
AND p.product_id = pr.product_id
AND p.product_type_id = pt.product_type_id
ORDER BY p.name;
```

O exemplo a seguir reescreve essa consulta usando SQL/92; observe como os relacionamentos de chave estrangeira são percorridos usando-se várias cláusulas INNER JOIN e USING:

```
SELECT c.first_name, c.last_name, p.name AS PRODUCT, pt.name AS TYPE
FROM customers c INNER JOIN purchases pr
USING (customer_id)
INNER JOIN products p
USING (product_id)
INNER JOIN product_types pt
USING (product_type_id)
ORDER BY p.name;
```

## Realizando joins internas em várias colunas usando SQL/92

Se sua join utiliza mais de uma coluna das duas tabelas, forneça essas colunas em sua cláusula ON e use o operador AND. Por exemplo, digamos que você tenha duas tabelas chamadas tabela1 e tabela2 e queira juntá-las usando colunas chamadas coluna1 e coluna2 nas duas tabelas. Sua consulta seria:

```
SELECT...
FROM tabela1 INNER JOIN tabela2
ON tabela1.coluna1 = tabela2.coluna1
AND tabela1.coluna2 = tabela2.coluna2;
```

Você pode simplificar ainda mais sua consulta com a cláusula USING, mas somente se estiver realizando uma equijoin e os nomes de coluna forem idênticos. Por exemplo, a consulta a seguir reescreve o exemplo anterior com a cláusula USING:

```
SELECT...
FROM tabela1 INNER JOIN tabela2
USING (coluna1, coluna2);
```

## Realizando joins externas usando SQL/92

Anteriormente, você viu como realizar joins externas usando o operador de join externa (+), que é uma sintaxe proprietária do Oracle. O padrão SQL/92 utiliza uma sintaxe diferente para realizar joins externas. Em vez de usar (+), você especifica o tipo de join na cláusula FROM, usando esta sintaxe:

```
FROM tabela1 { LEFT | RIGHT | FULL } OUTER JOIN tabela2
```

onde

- *tabela1* e *tabela2* são as tabelas que você deseja juntar.
- **LEFT** significa que você quer realizar uma join externa esquerda.
- **RIGHT** significa que você quer realizar uma join externa direita.
- **FULL** significa que você quer realizar uma join externa integral. Uma join externa integral utiliza todas as linhas da *tabela1* e da *tabela2*, incluindo aquelas que têm valores nulos nas colunas usadas na junção. Usando o operador (+), você não pode realizar uma join externa integral diretamente.

Você verá como realizar joins externas esquerda, direita e integrais usando a sintaxe SQL/92 nas seções a seguir.

### Realizando join externa esquerda usando SQL/92

Anteriormente, você viu a consulta a seguir, que realizava uma join externa esquerda usando o operador (+), proprietário do Oracle:

```
SELECT p.name, pt.name  
FROM products p, product_types pt  
WHERE p.product_type_id = pt.product_type_id (+)  
ORDER BY p.name;
```

Este exemplo reescreve essa consulta usando as palavras-chave **LEFT OUTER JOIN** da SQL/92:

```
SELECT p.name, pt.name  
FROM products p LEFT OUTER JOIN product_types pt  
USING (product_type_id)  
ORDER BY p.name;
```

### Realizando join externa direita usando SQL/92

Anteriormente, você viu a consulta a seguir, que realizava uma join externa direita usando o operador (+), proprietário do Oracle:

```
SELECT p.name, pt.name  
FROM products p, product_types pt  
WHERE p.product_type_id (+) = pt.product_type_id  
ORDER BY p.name;
```

O exemplo a seguir reescreve essa consulta usando as palavras-chave `RIGHT OUTER JOIN` da SQL/92:

```
SELECT p.name, pt.name
FROM products p RIGHT OUTER JOIN product_types pt
USING (product_type_id)
ORDER BY p.name;
```

### **Realizando join externa integral usando SQL/92**

Uma join externa integral utiliza todas as linhas das tabelas unidas, incluindo aquelas que têm valores nulos em uma ou outra das colunas usadas na join. O exemplo a seguir mostra uma consulta que utiliza as palavras-chave `FULL OUTER JOIN` da SQL/92:

```
SELECT p.name, pt.name
FROM products p FULL OUTER JOIN product_types pt
USING (product_type_id)
ORDER BY p.name;
```

NAME	NAME
-----	-----
2412: The Return	Video
Chemistry	Book
Classical Music	CD
Creative Yell	CD
From Another Planet	DVD
Modern Science	Book
My Front Line	
Pop 3	CD
Space Force 9	DVD
Supernova	Video
Tank War	Video
Z Files	Video
	Magazine

Note que tanto “My Front Line” da tabela `products` como “Magazine” da tabela `product_types` são retornados.

### **Realizando autojoin usando SQL/92**

O exemplo a seguir usa SQL/86 para realizar uma autojoin na tabela `employees`:

```
SELECT w.last_name || ' works for ' || m.last_name
FROM employees w, employees m
WHERE w.manager_id = m.employee_id;
```

O exemplo a seguir reescreve essa consulta para usar as palavras-chave `INNER JOIN` e `ON` da SQL/92:

```
SELECT w.last_name || ' works for ' || m.last_name
FROM employees w INNER JOIN employees m
ON w.manager_id = m.employee_id;
```

## Realizando join cruzada usando SQL/92

Você aprendeu que omitir uma condição de join entre duas tabelas leva a um produto cartesiano. Usando a sintaxe de join do padrão SQL/92, você evita a produção acidental de um produto cartesiano, pois sempre precisa fornecer uma cláusula `ON` ou `USING` para juntar as tabelas — isso é bom, pois um produto cartesiano, em geral, não é desejável.

Se você realmente quiser um produto cartesiano, o padrão SQL/92 exige que indique isso explicitamente em sua consulta, usando as palavras-chave `CROSS JOIN`. Na consulta a seguir, um produto cartesiano entre as tabelas `product_types` e `products` é gerado usando as palavras-chave `CROSS JOIN`:

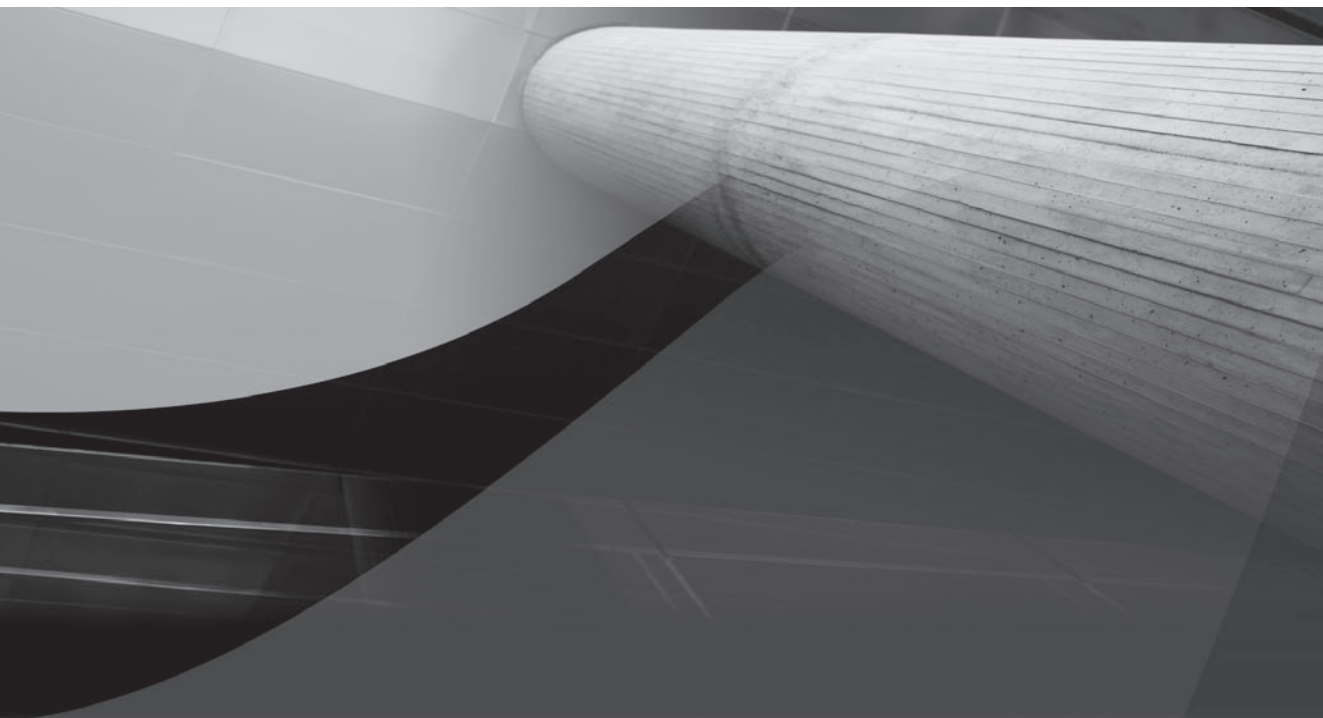
```
SELECT *  
FROM product_types CROSS JOIN products;
```

## RESUMO

Neste capítulo, você aprendeu a:

- Fazer consultas em uma e em várias tabelas
- Selecionar todas as colunas de uma tabela usando um asterisco (\*) em uma consulta
- Fazer com que um identificador de linha (rowid) seja usado internamente pelo banco de dados Oracle para armazenar a localização de uma linha
- Efetuar cálculos aritméticos em SQL
- Usar os operadores de adição e subtração com datas
- Referenciar tabelas e colunas usando apelidos
- Mesclar saída de coluna usando o operador de concatenação (||)
- Usar valores nulos para representar valores desconhecidos
- Exibir linhas distintas usando o operador `DISTINCT`
- Limitar a recuperação de linhas usando a cláusula `WHERE`
- Classificar linhas usando a cláusula `ORDER BY`
- Realizar joins internas, externas e autojoins usando a sintaxe SQL/86 e SQL/92

No próximo capítulo, você vai aprender sobre o SQL\*Plus.



# CAPÍTULO 3

Usando o SQL\*Plus

Neste capítulo, você vai aprender a:

- Exibir a estrutura de uma tabela
- Editar uma instrução SQL
- Salvar e executar scripts contendo instruções SQL e comandos SQL\*Plus
- Formatar os resultados retornados pelo SQL\*Plus
- Usar variáveis no SQL\*Plus
- Criar relatórios simples
- Obter ajuda do SQL\*Plus
- Gerar instruções SQL automaticamente
- Desconectar-se de um banco de dados e sair do SQL\*Plus

Primeiro vamos examinar como exibir a estrutura de uma tabela.

## EXIBINDO A ESTRUTURA DE UMA TABELA

Conhecer a estrutura de uma tabela é útil, pois você pode usar essa informação para formular uma instrução SQL. Por exemplo, você pode descobrir as colunas que deseja recuperar em uma consulta. Para exibir a estrutura de uma tabela, use o comando `DESCRIBE`.

O exemplo a seguir usa `DESCRIBE` para exibir a estrutura da tabela `customers`; observe que o caractere de ponto-e-vírgula (;) foi omitido do final do comando:

```
SQL> DESCRIBE customers
Name                Null?    Type
-----
CUSTOMER_ID         NOT NULL NUMBER(38)
FIRST_NAME          NOT NULL VARCHAR2(10)
LAST_NAME           NOT NULL VARCHAR2(10)
DOB                 DATE
PHONE               VARCHAR2(12)
```

A saída do comando `DESCRIBE` tem três colunas que mostram a estrutura da tabela:

- **Name** lista os nomes das colunas contidas na tabela. No exemplo, você pode ver que a tabela `customers` tem cinco colunas: `customer_id`, `first_name`, `last_name`, `dob` e `phone`.
- **Null?** indica se a coluna pode armazenar valores nulos. Se for definida como `NOT NULL`, a coluna não pode armazenar um valor nulo; se estiver em branco, pode armazenar um valor nulo. No exemplo anterior, as colunas `customer_id`, `first_name` e `last_name` não podem armazenar valores nulos, mas as colunas `dob` e `phone` podem.



- **Type** indica o tipo da coluna. No exemplo anterior, o tipo da coluna `customer_id` é `NUMBER(38)` e que o tipo da coluna `first_name` é `VARCHAR2(10)`.

Você pode economizar digitação abreviando o comando `DESCRIBE` para `DESC` (`DESCRIBE`). O comando a seguir usa `DESC` para exibir a estrutura da tabela `products`:

```
SQL> DESC products
```

Name	Null?	Type
-----	-----	-----
PRODUCT_ID	NOT NULL	NUMBER(38)
PRODUCT_TYPE_ID		NUMBER(38)
NAME	NOT NULL	VARCHAR2(30)
DESCRIPTION		VARCHAR2(50)
PRICE		NUMBER(5,2)

## EDITANDO INSTRUÇÕES SQL

Pode ser maçante digitar várias vezes instruções SQL semelhantes no SQL\*Plus. Pois fique contente em saber que o SQL\*Plus armazena sua instrução SQL anterior em um buffer. Assim, você pode editar as linhas que compõem a instrução SQL armazenada no buffer.

Alguns dos comandos de edição estão listados na tabela a seguir; observe a parte opcional de cada comando entre colchetes (por exemplo, você pode abreviar o comando `APPEND` para `A`).

Comando	Descrição
<code>A[PPEND] texto</code>	Anexa texto na linha atual.
<code>C[HANGE] /antigo/novo</code>	Altera o texto especificado de antigo para novo na linha atual.
<code>CL[EAR] BUFF[ER]</code>	Apaga todas as linhas do buffer.
<code>DEL</code>	Exclui a linha atual.
<code>DEL x</code>	Exclui a linha especificada pelo número de linha <code>x</code> (os números de linha começam com 1).
<code>L[IST]</code>	Lista todas as linhas presentes no buffer.
<code>L[IST] x</code>	Lista o número de linha <code>x</code> .
<code>R[UN] ou / x</code>	Executa a instrução armazenada no buffer. Você também pode usar <code>/</code> para executar a instrução.
	Torna corrente a linha especificada pelo número de linha <code>x</code> .

Vejamos alguns exemplos do uso dos comandos de edição do SQL\*Plus. O exemplo a seguir mostra uma consulta em SQL\*Plus:

```
SQL> SELECT customer_id, first_name, last_name
2 FROM customers
3 WHERE customer_id = 1;
```

O SQL\*Plus incrementa o número de linha automaticamente quando você pressiona ENTER. Para tornar a linha 1 corrente, digite 1 no prompt:

```
SQL> 1
1* SELECT customer_id, first_name, last_name
```

Note que o SQL\*Plus exibe a linha atual e o número de linha. O exemplo a seguir usa APPEND para adicionar ", dob" no final da linha:

```
SQL> APPEND, dob
1* SELECT customer_id, first_name, last_name, dob
```

O exemplo a seguir usa LIST para mostrar todas as linhas que estão no buffer:

```
SQL> LIST
1 SELECT customer_id, first_name, last_name, dob
2 FROM customers
3* WHERE customer_id = 1
```

Note que a linha atual foi alterada para a última linha, conforme indicado pelo caractere de asterisco (\*). O exemplo a seguir usa CHANGE para substituir "customer\_id = 1" por "customer\_id = 2" nessa última linha:

```
SQL> CHANGE /customer_id = 1/customer_id = 2
3* WHERE customer_id = 2
```

O exemplo a seguir usa RUN para executar a consulta:

```
SQL> RUN
1 SELECT customer_id, first_name, last_name, dob
2 FROM customers
3* WHERE customer_id = 2
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB
2	Cynthia	Green	05-FEB-68

Você também pode usar um caractere de barra normal (/) para executar a instrução SQL; por exemplo:

```
SQL> /

CUSTOMER_ID FIRST_NAME LAST_NAME DOB
-----
2 Cynthia    Green    05-FEB-68
```

## SALVANDO, RECUPERANDO E EXECUTANDO ARQUIVOS

O SQL\*Plus permite salvar, recuperar e executar scripts contendo comandos SQL\*Plus e instruções SQL. Você já viu um exemplo de execução de um script SQL\*Plus: o arquivo de script store\_schema.sql, que foi executado no Capítulo 1.

Alguns dos comandos de arquivo estão listados na tabela a seguir.

**Comando**

```
SAV[E] nome_de_arquivo
[{ REPLACE | APPEND }]
```

```
GET nome_de_arquivo
```

```
STA[RT] nome_de_arquivo
```

```
@ nome_de_arquivo
```

```
ED[IT]
```

```
ED[IT] nome_de_arquivo
```

```
SPO[OL] nome_de_arquivo
```

```
SPO[OL] OFF
```

**Descrição**

Salva o conteúdo do buffer do SQL\*Plus em um arquivo especificado por *nome\_de\_arquivo*. Você anexa o conteúdo do buffer em um arquivo existente usando a opção APPEND. Você sobrescreve um arquivo existente usando a opção REPLACE.

Recupera o conteúdo do arquivo especificado por *nome\_de\_arquivo* no buffer do SQL\*Plus.

Recupera o conteúdo do arquivo especificado por *nome\_de\_arquivo* para o buffer do SQL\*Plus e, então, tenta executar o conteúdo do buffer.

Igual ao comando START.

Copia o conteúdo do buffer do SQL\*Plus em um arquivo chamado *afiedt.buf* e, então, inicia o editor de texto padrão do sistema operacional. Quando você sai do editor, o conteúdo do arquivo editado é copiado no buffer do SQL\*Plus.

Igual ao comando EDIT, mas você pode especificar um arquivo para começar a editar. Você especifica o arquivo a editar usando o parâmetro *nome\_de\_arquivo*.

Copia a saída do SQL\*Plus para o arquivo especificado por *nome\_de\_arquivo*.

Interrompe a cópia da saída do SQL\*Plus no arquivo e, então, fecha esse arquivo.

Vejamos alguns exemplos de uso desses comandos SQL\*Plus. Se quiser acompanhar os exemplos, digite a consulta a seguir no SQL\*Plus:

```
SQL> SELECT customer_id, first_name, last_name
2 FROM customers
3 WHERE customer_id = 1;
```

O exemplo a seguir usa SAVE para salvar o conteúdo do buffer do SQL\*Plus em um arquivo chamado *cust\_query.sql*:

```
SQL> SAVE cust_query.sql
Created file cust_query.sql
```

**NOTA**

*Em meu computador, o arquivo cust\_query.sql é salvo no diretório E:\oracle\_11g\product\11.1.0\db\_1\BIN.*

O exemplo a seguir usa GET para recuperar o conteúdo do arquivo *cust\_query.sql*:

```
SQL> GET cust_query.sql
1 SELECT customer_id, first_name, last_name
2 FROM customers
3* WHERE customer_id = 1
```

O exemplo a seguir executa a consulta usando /:

```
SQL> /

CUSTOMER_ID FIRST_NAME LAST_NAME
-----
1 John      Brown
```

O exemplo a seguir usa `START` para carregar e executar o conteúdo do arquivo `cust_query.sql` em um só passo:

```
SQL> START cust_query.sql

CUSTOMER_ID FIRST_NAME LAST_NAME
-----
1 John      Brown
```

Você pode editar o conteúdo do buffer do SQL\*Plus usando o comando `EDIT`:

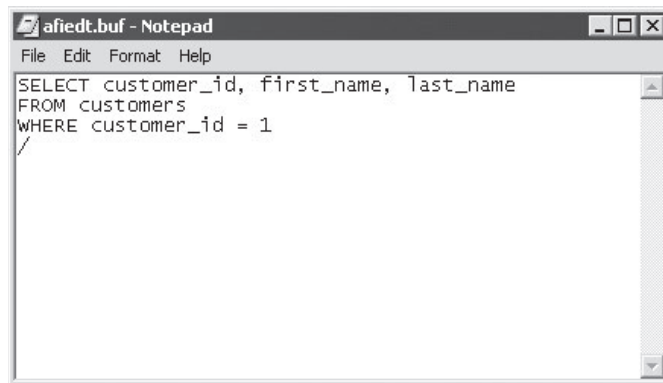
```
SQL> EDIT
```

O comando `EDIT` inicia o editor de texto padrão de seu sistema operacional. No Windows, o editor padrão é o Bloco de Notas. No Unix e no Linux, os editores padrão são o `vi` ou o `emacs` respectivamente.

A Figura 3-1 mostra o conteúdo do buffer do SQL\*Plus no Bloco de Notas. Observe que a instrução SQL é terminada com um caractere de barra normal (/), em vez de um ponto-e-vírgula.

Em seu editor, altere a cláusula `WHERE` para `WHERE customer_id = 2` e, em seguida, salve e saia do editor (no Bloco de Notas, selecione Arquivo | Sair e clique em Sim para salvar a consulta). O SQL\*Plus exibe a seguinte saída, contendo sua consulta modificada; observe que a cláusula `WHERE` foi alterada:

```
1 SELECT customer_id, first_name, last_name
2 FROM customers
3* WHERE customer_id = 2
```



**Figura 3-1** Editando o conteúdo do buffer do SQL\*Plus com o Bloco de Notas.

### Alterando o editor padrão

Você pode alterar o editor padrão usando o comando `DEFINE` do SQL\*Plus:

```
DEFINE _EDITOR = 'editor'
```

onde *editor* é o nome de seu editor preferido. Por exemplo, o comando a seguir define o editor padrão como o `vi`:

```
DEFINE _EDITOR = 'vi'
```

Você também pode alterar o editor padrão utilizado pelo SQL\*Plus adicionando a linha `DEFINE _EDITOR = 'editor'` em um novo arquivo chamado `login.sql`, onde *editor* é o nome de seu editor preferido. Você pode adicionar os comandos SQL\*Plus que desejar nesse arquivo. O SQL\*Plus procurará um arquivo `login.sql` no diretório atual e o executará quando for iniciado. Se não houver um arquivo `login.sql` no diretório atual, o SQL\*Plus procurará um arquivo `login.sql` em todos os diretórios (e em seus subdiretórios) na variável de ambiente `SQLPATH`. No Windows, `SQLPATH` é definida como uma entrada de registro em `HKEY_LOCAL_MACHINE\SOFTWARE\ORACLE\oracle_home_key` (onde *oracle\_home\_key* é a chave da instalação associada do banco de dados Oracle). Em um computador Windows XP executando Oracle Database 11g, `SQLPATH` está definida como `E:\oracle_11g\product\11.1.0\db_1\dfs`. No Unix ou no Linux, não há uma variável `SQLPATH` definida e você precisará adicioná-la como uma variável de ambiente. Para obter mais detalhes sobre a configuração de um arquivo `login.sql`, consulte *SQL\*Plus User's Guide and Reference*, publicado pela Oracle Corporation.

Você executa sua consulta modificada usando o caractere de barra normal (/):

```
SQL> /
```

```
CUSTOMER_ID FIRST_NAME LAST_NAME
-----
          2 Cynthia      Green
```

### DICA

*Na versão de SQL\*Plus do Oracle Database 11g, você também pode rolar pelas instruções executadas anteriormente usando as teclas de seta para cima e seta para baixo do teclado. Uma vez selecionada uma instrução, você pode usar as teclas de seta para esquerda e seta para direita a fim de mover o cursor para um ponto específico na instrução.*

Você usa o comando `SPOOL` para copiar a saída do SQL\*Plus em um arquivo. O exemplo a seguir transfere a saída para um arquivo chamado `cust_results.txt`, executa a consulta novamente e, então, interrompe a transferência executando `SPOOL OFF`:

```
SQL> SPOOL cust_results.txt
SQL> /
```

```
CUSTOMER_ID FIRST_NAME LAST_NAME
-----
          2 Cynthia      Green
```

```
SQL> SPOOL OFF
```

Examine o arquivo `cust_results.txt`; ele conterá a saída anterior entre a barra normal (/) e `SPOOL OFF`. Em um computador Windows XP, o arquivo é armazenado em `E:\oracle_11g\product\11.1.0\db_1\BIN`; o diretório usado é o diretório atual em que você está quando inicia a SQL\*Plus. Você também pode especificar o caminho de diretório completo onde deseja que o arquivo de spool seja gravado; por exemplo:

```
SPOOL C:\my_files\spools\cust_results.txt
```

## FORMATANDO COLUNAS

Você usa o comando `COLUMN` para formatar a exibição de cabeçalhos de coluna e dados de coluna. A sintaxe simplificada do comando `COLUMN` é:

```
COLUMN {coluna | apelido} [opções]
```

onde

- *coluna* é o nome da coluna.
- *apelido* é o apelido da coluna a ser formatada. No Capítulo 2, você viu que pode “renomear” uma coluna usando um apelido de coluna; é possível referenciar um apelido no comando `COLUMN`.
- *opções* são uma ou mais opções a serem usadas para formatar a coluna ou o apelido.

Existem várias opções que você pode usar com o comando `COLUMN`. A tabela a seguir mostra algumas delas.

Opção	Descrição
<code>FOR[MAT] <i>formato</i></code>	Define o formato de exibição da coluna ou apelido com a string <i>formato</i> .
<code>HEA[DING] <i>cabeçalho</i></code>	Define o cabeçalho da coluna ou apelido com a string <i>cabeçalho</i> .
<code>JUS[TIFY]</code> <code>[{ LEFT   CENTER   RIGHT }]</code>	Coloca a saída da coluna na esquerda, no centro ou na direita.
<code>WRA[PPED]</code>	Passa o final de uma string para a próxima linha de saída. Esta opção pode fazer com que palavras individuais sejam divididas em várias linhas.
<code>WOR[D_WRAPPED]</code>	Semelhante à opção <code>WRAPPED</code> , exceto que palavras individuais não são divididas em duas linhas.
<code>CLE[AR]</code>	Limpa toda formatação de colunas (isto é, configura a formatação de volta no padrão).

A string *formato* na tabela anterior pode receber diversos parâmetros de formatação. Os parâmetros especificados dependem dos dados armazenados na coluna:

- Se a sua coluna contém caracteres, use `Ax` para formatá-los, onde *x* especifica a largura dos caracteres. Por exemplo, `A12` define a largura como 12 caracteres.

- Se a sua coluna contém números, use uma variedade de formatos numéricos, os quais serão mostrados posteriormente na Tabela 4-4 do Capítulo 4. Por exemplo, \$99.99 define o formato como um cifrão, seguido de dois dígitos, o ponto decimal, mais outros dois dígitos.
- Se a sua coluna contém uma data, use um dos formatos de data mostrados posteriormente na Tabela 5-2 do Capítulo 5. Por exemplo, MM-DD-YYYY define o formato como um mês com dois dígitos (MM), um dia com dois dígitos (DD) e um ano com quatro dígitos (YYYY).

Por exemplo, veja como formatar a saída de uma consulta que recupera as colunas `product_id`, `name`, `description` e `price` da tabela `products`. Os requisitos de exibição, as strings de formato e os comandos `COLUMN` estão mostrados a seguir:

Coluna	Exibida como	Formato	Comando COLUMN
product_id	Dois dígitos	99	COLUMN product_id FORMAT 99
name	Strings de 13 caracteres com mudança automática de linha e o cabeçalho de coluna definido como PRODUCT_NAME	A13	COLUMN name HEADING PRODUCT_NAME FORMAT A13 WORD_WRAPPED
description	Strings de 13 caracteres com mudança automática de linha	A13	COLUMN description FORMAT A13 WORD_WRAPPED
price	Cifrão, com dois dígitos antes e depois do ponto decimal	\$99.99	COLUMN price FORMAT \$99.99

O exemplo a seguir mostra os comandos `COLUMN` em SQL\*Plus:

```
SQL> COLUMN product_id FORMAT 99
SQL> COLUMN name HEADING PRODUCT_NAME FORMAT A13 WORD_WRAPPED
SQL> COLUMN description FORMAT A13 WORD_WRAPPED
SQL> COLUMN price FORMAT $99.99
```

O exemplo a seguir executa uma consulta para recuperar algumas linhas da tabela `products`; observe a formatação das colunas na saída:

```
SQL> SELECT product_id, name, description, price
2 FROM products
3 WHERE product_id < 6;
```

PRODUCT_ID	PRODUCT_NAME	DESCRIPTION	PRICE
1	Modern Science	A description of modern science	\$19.95
2	Chemistry	Introduction to Chemistry	\$30.00

3	Supernova	A star explodes	\$25.99
4	Tank War	Action movie	\$13.95
PRODUCT_ID	PRODUCT_NAME	DESCRIPTION	PRICE
-----	-----	-----	-----
		about a future war	
5	Z Files	Series on mysterious activities	\$49.99

Essa saída é legível, mas não seria ótimo se você pudesse exibir os cabeçalhos apenas uma vez, no início? É possível fazer isso definindo o tamanho da página, conforme você verá a seguir.

## DEFININDO O TAMANHO DA PÁGINA

Você define o número de linhas em uma página usando o comando `SET PAGESIZE`. Esse comando define o número de linhas que o SQL\*Plus considera como uma “página” de saída, após a qual o SQL\*Plus exibirá os cabeçalhos novamente. O exemplo a seguir define o tamanho da página como 100 linhas usando o comando `SET PAGESIZE` e executa a consulta novamente usando `/`:

```
SQL> SET PAGESIZE 100
SQL> /
```

PRODUCT_ID	PRODUCT_NAME	DESCRIPTION	PRICE
-----	-----	-----	-----
1	Modern Science	A description of modern science	\$19.95
2	Chemistry	Introduction to Chemistry	\$30.00
3	Supernova	A star explodes	\$25.99
4	Tank War	Action movie about a future war	\$13.95
5	Z Files	Series on mysterious activities	\$49.99



Note que os cabeçalhos são mostrados apenas uma vez (no início) e a saída resultante aparece melhor.

#### NOTA

O número máximo para o tamanho da página é 50.000.

## DEFININDO O TAMANHO DA LINHA

Você define o número de caracteres em uma linha usando o comando `SET LINESIZE`. O exemplo a seguir define o tamanho da linha como 50 linhas e executa outra consulta:

```
SQL> SET LINESIZE 50
SQL> SELECT * FROM customers;
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB
1	John	Brown	01-JAN-65
2	Cynthia	Green	05-FEB-68
3	Steve	White	16-MAR-71
4	Gail	Black	
5	Doreen	Blue	20-MAY-70

As linhas não ultrapassam os 50 caracteres.

#### NOTA

O número máximo para o tamanho da linha é 32.767.

## LIMPANDO FORMATAÇÃO DE COLUNA

Você limpa a formatação de uma coluna usando a opção `CLEAR` do comando `COLUMN`. Por exemplo, o comando `COLUMN` a seguir limpa a formatação da coluna `product_id`:

```
SQL> COLUMN product_id CLEAR
```

Você pode limpar a formatação de todas as colunas usando `CLEAR COLUMNS`. Por exemplo:

```
SQL> CLEAR COLUMNS
```

Uma vez que você tenha limpado as colunas, a saída das consultas usará o formato padrão.

# USANDO VARIÁVEIS

Nesta seção, você vai ver como criar variáveis que podem ser usadas no lugar dos valores reais em instruções SQL. Essas variáveis são conhecidas como *variáveis de substituição*, pois são usadas como substitutas de valores. Quando executa uma instrução SQL, você insere valores para as variáveis; então, os valores são “substituídos” na instrução SQL. Existem dois tipos de variáveis de substituição:

- **Variáveis temporárias** Uma variável temporária é válida apenas para a instrução SQL em que é usada — ela não persiste.
- **Variáveis definidas** Uma variável definida persiste até que você a remova explicitamente, a redefina ou saia do SQL\*Plus.

Você vai aprender a usar esses tipos de variáveis nesta seção.

## Variáveis temporárias

Você define uma variável temporária usando o caractere de E comercial (&) em uma instrução SQL, seguido do nome que deseja dar à sua variável. Por exemplo, &v\_product\_id define uma variável chamada v\_product\_id.

Quando você executa a consulta a seguir, o SQL\*Plus solicita a inserção de um valor para v\_product\_id e depois usa esse valor na cláusula WHERE. Se você inserir o valor 2 para v\_product\_id, os detalhes do produto nº 2 serão exibidos.

```
SQL> SELECT product_id, name, price
      2 FROM products
      3 WHERE product_id = &v_product_id;
Enter value for v_product_id: 2
old   3: WHERE product_id = &v_product_id
new   3: WHERE product_id = 2

PRODUCT_ID NAME                                PRICE
-----
          2 Chemistry                                30
```

Note que o SQL\*Plus:

- Solicita a inserção de um valor para v\_product\_id.
- Substitui o valor inserido para v\_product\_id na cláusula WHERE.

O SQL\*Plus mostra a substituição nas linhas old e new da saída, junto com o número de linha na consulta onde foi realizada a substituição. No exemplo anterior, você pode ver que as linhas old e new indicam que v\_product\_id é configurado como 2 na cláusula WHERE.

### Por que as variáveis são úteis?

As variáveis são úteis porque elas permitem criar scripts que um usuário que não conhece SQL pode executar. Seu script pediria ao usuário para inserir o valor para uma variável e usaria esse valor em uma instrução SQL. Vejamos um exemplo.

Suponha que você quisesse criar um script para um usuário que não conhece SQL, mas que quer ver os detalhes de um único produto especificado da loja. Para fazer isso, você poderia codificar o valor de `product_id` na cláusula `WHERE` de uma consulta e colocá-la em um script SQL\*Plus. Por exemplo, a consulta a seguir recupera o produto nº 1:

```
SELECT product_id, name, price
FROM products
WHERE product_id = 1;
```

Essa consulta funciona, mas só recupera o produto nº 1. E se você quisesse alterar o valor de `product_id` para recuperar uma linha diferente? Você poderia modificar o script, mas isso seria maçante. Não seria ótimo se você pudesse fornecer uma variável para o valor de `product_id`? É possível fazer isso usando uma variável de substituição.

Se você executar a consulta novamente usando o caractere de barra normal (/), o SQL\*Plus solicitará a inserção de um novo valor para `v_product_id`. Por exemplo:

```
SQL> /
Enter value for v_product_id: 3
old 3: WHERE product_id = &v_product_id
new 3: WHERE product_id = 3

PRODUCT_ID NAME                                PRICE
-----
3 Supernova                                25.99
```

Mais uma vez, o SQL\*Plus exibe o valor antigo da instrução SQL (old 3: WHERE product\_id = &v\_product\_id), seguida do novo valor da variável que você inseriu (new 3: WHERE product\_id = 3).

### Controlando linhas de saída

Você pode controlar a saída das linhas old e new usando o comando `SET VERIFY`. Se você insere `SET VERIFY OFF`, as linhas old e new são suprimidas. Por exemplo:

```
SQL> SET VERIFY OFF
SQL> /
Enter value for v_product_id: 4

PRODUCT_ID NAME                                PRICE
-----
4 Tank War                                13.95
```

Para voltar a exibir essas linhas, você digita `SET VERIFY ON`. Por exemplo:

```
SQL> SET VERIFY ON
```

### ***Alterando o caractere de definição de variável***

Você pode usar o comando `SET DEFINE` para especificar um caractere que não seja o E comercial (&) a fim de definir uma variável. O exemplo a seguir mostra como se configura o caractere de variável como o caractere # e a execução de uma nova consulta:

```
SQL> SET DEFINE '#'
SQL> SELECT product_id, name, price
  2 FROM products
  3 WHERE product_id = #v_product_id;
Enter value for v_product_id: 5
old  3: WHERE product_id = #v_product_id
new  3: WHERE product_id = 5

PRODUCT_ID NAME                                PRICE
-----
          5 Z Files                             49.99
```

O exemplo a seguir usa `SET DEFINE` para alterar o caractere de volta para o E comercial:

```
SQL> SET DEFINE '&'
```

### ***Substituindo nomes de tabela e coluna usando variáveis***

Você também pode usar variáveis para substituir os nomes de tabelas e colunas. Por exemplo, a consulta a seguir define variáveis para um nome de coluna (`v_col`), um nome de tabela (`v_table`) e um valor de coluna (`v_val`):

```
SQL> SELECT name, &v_col
  2 FROM &v_table
  3 WHERE &v_col = &v_val;
Enter value for v_col: product_type_id
old  1: SELECT name, &v_col
new  1: SELECT name, product_type_id
Enter value for v_table: products
old  2: FROM &v_table
new  2: FROM products
Enter value for v_col: product_type_id
Enter value for v_val: 1
old  3: WHERE &v_col = &v_val
new  3: WHERE product_type_id = 1

NAME                                PRODUCT_TYPE_ID
-----
Modern Science                        1
Chemistry                           1
```

Você pode evitar a digitação repetida de uma variável usando `&&`. Por exemplo:

```

SQL> SELECT name, &&v_col
      2 FROM &v_table
      3 WHERE &&v_col = &v_val;
Enter value for v_col: product_type_id
old 1: SELECT name, &&v_col
new 1: SELECT name, product_type_id
Enter value for v_table: products
old 2: FROM &v_table
new 2: FROM products
Enter value for v_val: 1
old 3: WHERE &&v_col = &v_val
new 3: WHERE product_type_id = 1

```

NAME	PRODUCT_TYPE_ID
Modern Science	1
Chemistry	1

As variáveis oferecem muita flexibilidade na escrita de consultas que outro usuário pode executar. Você pode fornecer um script ao usuário e fazer com que ele digite os valores de variável.

## Variáveis definidas

Você pode definir uma variável antes de usá-la em uma instrução SQL. É possível usar essas variáveis várias vezes dentro de uma instrução SQL. Uma variável definida persiste até que você a remova explicitamente, a redefina ou saia do SQL\*Plus.

Você define uma variável usando o comando **DEFINE**. Quando tiver terminado de usar a variável, ela pode ser removida usando **UNDEFINE**. Você vai aprender sobre esses comandos nesta seção e também vai aprender sobre o comando **ACCEPT**, que permite definir uma variável e configurar seu tipo de dados.

Você também pode definir variáveis em um script SQL\*Plus e passar valores para elas quando o script for executado. Esse recurso permite que você escreva relatórios genéricos que qualquer usuário pode executar — mesmo que não esteja familiarizado com a linguagem SQL. Você vai aprender a criar relatórios simples, na seção “Criando relatórios simples”.

### Definindo e listando variáveis com o comando **DEFINE**

Você usa o comando **DEFINE** para definir uma nova variável e para listar as variáveis correntemente definidas. O exemplo a seguir define uma variável chamada `v_product_id` e configura seu valor como 7:

```

SQL> DEFINE v_product_id = 7

```

Você pode ver a definição de uma variável usando o comando **DEFINE** seguido do nome da variável. O exemplo a seguir exibe a definição de `v_product_id`:

```

SQL> DEFINE v_product_id
DEFINE V_PRODUCT_ID          = "7" (CHAR)

```

Note que `v_product_id` é definida como uma variável **CHAR**.

Você pode ver todas as suas variáveis de sessão digitando apenas DEFINE. Por exemplo:

```
SQL> DEFINE
DEFINE _DATE = "12-AUG-07" (CHAR)
DEFINE _CONNECT_IDENTIFIER = "Oracle11g" (CHAR)
DEFINE _USER = "STORE" (CHAR)
DEFINE _PRIVILEGE = "" (CHAR)
DEFINE _SQLPLUS_RELEASE = "1101000400" (CHAR)
DEFINE _EDITOR = "Notepad" (CHAR)
DEFINE _O_VERSION = "Oracle Database 11g..." (CHAR)
DEFINE _O_RELEASE = "1101000500" (CHAR)
DEFINE _RC = "0" (CHAR)
DEFINE V_PRODUCT_ID = "7" (CHAR)
```

Você pode usar uma variável definida para especificar um elemento, como um valor de coluna, em uma instrução SQL. Por exemplo, a consulta a seguir usa referências v\_product\_id na cláusula WHERE:

```
SQL> SELECT product_id, name, price
2 FROM products
3 WHERE product_id = &v_product_id;
old 3: WHERE product_id = &v_product_id
new 3: WHERE product_id = 7
```

PRODUCT_ID	NAME	PRICE
7	Space Force 9	13.49

O valor de v\_product\_id não é solicitado; isso porque a variável v\_product\_id foi configurada como 7 quando foi definida anteriormente.

### ***Definindo e configurando variáveis com o comando ACCEPT***

O comando ACCEPT espera que o usuário digite um valor para uma variável. Você pode usar o comando ACCEPT para configurar uma variável existente com um novo valor ou para definir uma nova variável e inicializá-la com um valor. O comando ACCEPT também permite especificar o tipo de dados da variável. A sintaxe simplificada do comando ACCEPT é:

```
ACCEPT nome_variável [tipo] [FORMAT formato] [PROMPT prompt] [HIDE]
```

onde

- *nome\_variável* é o nome da variável.
- *tipo* é o tipo de dados da variável. Você pode usar os tipos CHAR, NUMBER e DATE. Por padrão, as variáveis são definidas com o tipo CHAR. As variáveis DATE são, na verdade, armazenadas como variáveis CHAR.
- *formato* é o formato usado para a variável. Alguns exemplos são: A15 (15 caracteres), 9999 (um número de quatro dígitos) e DD-MOM-YYYY (uma data). Você pode ver os formatos numéricos na Tabela 4-4 do Capítulo 4; você pode ver os formatos de data na Tabela 5-2 do Capítulo 5.
- *prompt* é o texto exibido pelo SQL\*Plus como prompt para o usuário digitar o valor da variável.

- HIDE significa ocultar o valor quando ele é digitado. Por exemplo, talvez você queira ocultar senhas ou outras informações confidenciais.

Vejamos alguns exemplos do comando ACCEPT. O exemplo a seguir define uma variável chamada `v_customer_id` como um número de dois dígitos:

```
SQL> ACCEPT v_customer_id NUMBER FORMAT 99 PROMPT 'Customer id: '
Customer id: 5
```

O exemplo a seguir define uma variável DATE chamada `v_date`; o formato é DD-MOM-YYYY:

```
SQL> ACCEPT v_date DATE FORMAT 'DD-MOM-YYYY' PROMPT 'Date: '
Date: 12-DEC-2006
```

O exemplo a seguir define uma variável CHAR chamada `v_password`; o valor digitado é ocultado usando HIDE:

```
SQL> ACCEPT v_password CHAR PROMPT 'Password: ' HIDE
Password:
```

No Oracle Database 9i e nas versões anteriores, o valor aparece como uma string de caracteres de asterisco (\*) para ocultar o valor enquanto você o digita. No Oracle Database 10g e nas versões posteriores, nada é exibido quando você digita o valor.

Você pode ver suas variáveis usando o comando DEFINE. Por exemplo:

```
SQL> DEFINE
...
DEFINE V_CUSTOMER_ID =          5 (NUMBER)
DEFINE V_DATE         = "12-DEC-2006" (CHAR)
DEFINE V_PASSWORD     = "1234567" (CHAR)
DEFINE V_PRODUCT_ID   = "7" (CHAR)
```

Note que `v_date` é armazenada como CHAR.

### ***Removendo variáveis com o comando UNDEFINE***

Você remove variáveis usando o comando UNDEFINE. O exemplo a seguir usa UNDEFINE para remover `v_customer_id`, `v_date`, `v_password` e `v_product_id`:

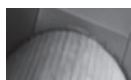
```
SQL> UNDEFINE v_customer_id
SQL> UNDEFINE v_date
SQL> UNDEFINE v_password
SQL> UNDEFINE v_product_id
```

### **NOTA**

*Todas as suas variáveis são removidas quando você sai do SQL\*Plus, mesmo que não as remova explicitamente usando o comando UNDEFINE.*

## **CRIANDO RELATÓRIOS SIMPLES**

Você pode usar variáveis em um script SQL\*Plus para criar relatórios que um usuário pode executar. Os scripts SQL\*Plus referenciados nesta seção podem ser encontrados no diretório SQL do arquivo zip que contém os códigos de exemplo (veja o Prefácio).

**DICA**

*O SQL\*Plus não foi projetado como uma ferramenta de geração de relatórios completa. Se precisar gerar relatórios complexos, use um software como o Oracle Reports.*

## Usando variáveis temporárias em um script

O script report1.sql a seguir usa uma variável temporária chamada v\_product\_id na cláusula WHERE de uma consulta:

```
-- suprime a exibição das instruções e mensagens de verificação
SET ECHO OFF
SET VERIFY OFF

SELECT product_id, name, price
FROM products
WHERE product_id = &v_product_id;
```

O comando SET ECHO OFF faz o SQL\*Plus parar de exibir as instruções SQL e os comandos do script. SET VERIFY OFF suprime a exibição das mensagens de verificação. Colocamos esses dois comandos para minimizar o número de linhas extras exibidas pelo SQL\*Plus quando você executa o script.

É possível executar report1.sql no SQL\*Plus usando o comando @. Por exemplo:

```
SQL> @ C:\sql_book\SQL\report1.sql
Enter value for v_product_id: 2
```

PRODUCT_ID	NAME	PRICE
2	Chemistry	30

Você precisará substituir o diretório que aparece no exemplo pelo diretório onde salvou os arquivos deste livro. Além disso, se tiver espaços no diretório, deve colocar entre aspas tudo que vem depois do comando @; por exemplo:

```
@ "C:\meu diretório\sql book\SQL\report1.sql"
```

## Usando variáveis definidas em um script

O script report2.sql a seguir usa o comando ACCEPT para definir uma variável chamada v\_product\_id:

```
SET ECHO OFF
SET VERIFY OFF

ACCEPT v_product_id NUMBER FORMAT 99 PROMPT 'Enter product id: '

SELECT product_id, name, price
FROM products
WHERE product_id = &v_product_id;
```



```
-- remove a variável
UNDEFINE v_product_id
```

Note que um prompt amigável é especificado para a entrada de `v_product_id` e que `v_product_id` é removida ao final do script — fazer isso torna o script mais limpo.

É possível executar o script `report2.sql` usando o SQL\*Plus:

```
SQL> @ C:\sql_book\SQL\report2.sql
Enter product id: 4
```

PRODUCT_ID	NAME	PRICE
4	Tank War	13.95

## Passando um valor para uma variável em um script

É possível passar um valor para uma variável ao executar seu script. Para tanto, você referencia a variável no script usando um número. O script `report3.sql` a seguir mostra um exemplo disso; observe que a variável é identificada usando `&1`:

```
SET ECHO OFF
SET VERIFY OFF

SELECT product_id, name, price
FROM products
WHERE product_id = &1;
```

Ao executar o `report3.sql`, o valor da variável é fornecido após o nome do script. O exemplo a seguir passa o valor 3 para `report3.sql`:

```
SQL> @ C:\sql_book\SQL\report3.sql 3
PRODUCT_ID NAME PRICE
-----
3 Supernova 25.99
```

Se você tiver espaços no diretório onde salvou os scripts, precisará colocar o diretório e o nome do script entre aspas, por exemplo:

```
@ "C:\meu diretório\sql_book\SQL\report3.sql" 3
```

Você pode passar qualquer número de parâmetros para um script, com cada valor correspondendo ao número adequado no script. O primeiro parâmetro corresponde a `&1`, o segundo a `&2` e assim por diante. O script `report4.sql` a seguir mostra um exemplo com dois parâmetros:

```
SET ECHO OFF
SET VERIFY OFF

SELECT product_id, product_type_id, name, price
FROM products
WHERE product_type_id = &1
AND price > &2;
```

O exemplo de execução de `report4.sql` a seguir mostra a adição de dois valores para `&1` e `&2`, os quais são configurados como 1 e 9.99 respectivamente:

```
SQL> @ C:\sql_book\SQL\report4.sql 1 9.99
```

PRODUCT_ID	PRODUCT_TYPE_ID	NAME	PRICE
1	1	Modern Science	19.95
2	1	Chemistry	30

Como `&1` é configurado como 1, a coluna `product_type_id` na cláusula `WHERE` é configurada como 1. Além disso, como `&2` é configurado como 9.99, a coluna `price` na cláusula `WHERE` é configurada como 9.99. Portanto, são exibidas as linhas com `product_type_id` igual a 1 e `price` maior do que 9.99.

## Adicionando um cabeçalho e um rodapé

Você adiciona um cabeçalho e um rodapé em seu relatório usando os comandos `TTITLE` e `BTITLE`. A seguir está um exemplo do comando `TTITLE`:

```
TTITLE LEFT 'Run date: ' _DATE CENTER 'Run by the ' SQL.USER ' user'
RIGHT 'Page: ' FORMAT 999 SQL.PNO SKIP 2
```

A lista a seguir explica o conteúdo desse comando:

- `_DATE` exibe a data atual
- `SQL.USER` exibe o usuário atual
- `SQL.PNO` exibe a página atual (`FORMAT` é usado para formatar o número)
- `LEFT`, `CENTER` e `RIGHT` justificam o texto
- `SKIP 2` pula duas linhas

Se o exemplo é executado em 12 de agosto de 2007 pelo usuário `store`, ele exibe:

```
Run date: 12-AUG-07      Run by the STORE user      Page:      1
```

O exemplo a seguir mostra um comando `BTITLE`:

```
BTITLE CENTER 'Thanks for running the report' RIGHT 'Page: '
FORMAT 999 SQL.PNO
```

Esse comando exibe:

```
Thanks for running the report      Page:      1
```

O script `report5.sql` a seguir contém os comandos `TTITLE` e `BTITLE`:

```
TTITLE LEFT 'Run date: ' _DATE CENTER 'Run by the ' SQL.USER ' user'
RIGHT 'Page: ' FORMAT 999 SQL.PNO SKIP 2
```

```
BTITLE CENTER 'Thanks for running the report' RIGHT 'Page: '
FORMAT 999 SQL.PNO
```

```
SET ECHO OFF
```

```

SET VERIFY OFF
SET PAGESIZE 30
SET LINESIZE 70
CLEAR COLUMNS
COLUMN product_id HEADING ID FORMAT 99
COLUMN name HEADING 'Product Name' FORMAT A20 WORD_WRAPPED
COLUMN description HEADING Description FORMAT A30 WORD_WRAPPED
COLUMN price HEADING Price FORMAT $99.99

SELECT product_id, name, description, price
FROM products;

CLEAR COLUMNS
TTITLE OFF
BTITLE OFF

```

As duas últimas linhas desativam o cabeçalho e o rodapé definidos pelos comandos TTITLE e BTITLE. O exemplo a seguir mostra uma execução de report5.sql:

```
SQL> @ C:\sql_book\SQL\report5.sql
```

```

Run date: 12-AUG-07      Run by the STORE user      Page:      1

  ID Product Name      Descrição      Price
-----
   1 Modern Science    A description of modern science    $19.95

   2 Chemistry         Introduction to Chemistry    $30.00
   3 Supernova         A star explodes    $25.99
   4 Tank War          Action movie about a future war    $13.95

   5 Z Files           Series on mysterious activities    $49.99

   6 2412: The Return   Aliens return    $14.95
   7 Space Force 9      Adventures of heroes    $13.49
   8 From Another Planet Alien from another planet lands on Earth    $12.99

   9 Classical Music    The best classical music    $10.99
  10 Pop 3              The best popular music    $15.99
  11 Creative Yell      Debut album    $14.99
  12 My Front Line      Their greatest hits    $13.49

```

```

Thanks for running the report      Page:      1

```

## Calculando subtotais

É possível adicionar um subtotal para uma coluna usando uma combinação dos comandos BREAK ON e COMPUTE. BREAK ON faz o SQL\*Plus dividir a saída com base na alteração de um valor da coluna e COMPUTE faz o SQL\*Plus calcular um valor para uma coluna.

O script `report6.sql` a seguir mostra como calcular um subtotal para produtos do mesmo tipo:

```
BREAK ON product_type_id
COMPUTE SUM OF price ON product_type_id

SET ECHO OFF
SET VERIFY OFF
SET PAGESIZE 50
SET LINESIZE 70

CLEAR COLUMNS
COLUMN price HEADING Price FORMAT $999.99

SELECT product_type_id, name, price
FROM products
ORDER BY product_type_id;

CLEAR COLUMNS
```

O exemplo a seguir mostra uma execução de `report6.sql`:

```
SQL> @ C:\sql_book\SQL\report6.sql
```

PRODUCT_TYPE_ID	NAME	Price
1	Modern Science	\$19.95
	Chemistry	\$30.00
*****		-----
sum		\$49.95
2	Supernova	\$25.99
	Tank War	\$13.95
	Z Files	\$49.99
	2412: The Return	\$14.95
*****		-----
sum		\$104.88
3	Space Force 9	\$13.49
	From Another Planet	\$12.99
*****		-----
sum		\$26.48
4	Classical Music	\$10.99
	Pop 3	\$15.99
	Creative Yell	\$14.99
*****		-----
sum		\$41.97
	My Front Line	\$13.49
*****		-----
sum		\$13.49

Note que, quando um novo valor de `product_type_id` é encontrado, o SQL\*Plus divide a saída e calcula a soma das colunas `price`, para as linhas com o mesmo `product_type_id`. O valor

de `product_type_id` é mostrado somente uma vez para linhas com o mesmo `product_type_id`. Por exemplo, tanto “Modern Science” como “Chemistry” são livros, têm um `product_type_id` igual a 1 e o valor 1 é mostrado apenas uma vez, para “Modern Science”. A soma dos preços desses dois livros é \$49.95. As outras seções do relatório contêm a soma dos preços dos produtos com valores de `product_type_id` diferentes.

## OBTENDO AJUDA DO SQL\*PLUS

Você pode obter ajuda do SQL\*Plus usando o comando `HELP`. O exemplo a seguir executa o comando `HELP`:

```
SQL> HELP
```

```
HELP
----
```

Accesses this command line help system. Enter `HELP INDEX` or? `INDEX` for a list of topics. In `iSQL*Plus`, click the Help button to display `iSQL*Plus` online help.

You can view SQL\*Plus resources at [http://otn.oracle.com/tech/sql\\_plus/](http://otn.oracle.com/tech/sql_plus/) and the Oracle Database Library at <http://otn.oracle.com/documentation/>

```
HELP|? [topic]
```

O exemplo a seguir executa o comando `HELP INDEX`:

```
SQL> HELP INDEX
```

```
Enter Help [topic] for help.
```

@	COPY	PAUSE	SHUTDOWN
@@	DEFINE	PRINT	SPOOL
/	DEL	PROMPT	SQLPLUS
ACCEPT	DESCRIBE	QUIT	START
APPEND	DISCONNECT	RECOVER	STARTUP
ARCHIVE LOG	EDIT	REMARK	STORE
ATTRIBUTE	EXECUTE	REPFOOTER	TIMING
BREAK	EXIT	REPHEADER	TTITLE
BTITLE	GET	RESERVED WORDS (SQL)	UNDEFINE
CHANGE	HELP	RESERVED WORDS (PL/SQL)	VARIABLE
CLEAR	HOST	RUN	WHENEVER OSERROR
COLUMN	INPUT	SAVE	WHENEVER SQLEERROR
COMPUTE	LIST	SET	
CONNECT	PASSWORD	SHOW	

O exemplo a seguir executa o comando `HELP EDIT`:

```
SQL> HELP EDIT
```

```
EDIT
----
```

Invokes an operating system text editor on the contents of the specified file or on the contents of the SQL buffer. The buffer has no command history list and does not record SQL\*Plus commands.

ED[IT] [file\_name[.ext]]

Not available in iSQL\*Plus

## GERANDO INSTRUÇÕES SQL AUTOMATICAMENTE

Nesta seção, vamos mostrar brevemente uma técnica de escrita de instruções SQL que produz outras instruções SQL. Esse recurso é muito útil e pode economizar digitação ao se escrever instruções SQL semelhantes. Um exemplo simples é uma instrução SQL que produz instruções `DROP TABLE`, as quais removem tabelas de um banco de dados. A consulta a seguir produz uma série de instruções `DROP TABLE` que removem as tabelas do esquema `store`:

```
SELECT 'DROP TABLE ' || table_name || ';'
FROM user_tables;
```

```
'DROPTABLE' || TABLE_NAME || ';'
-----
```

```
DROP TABLE COUPONS;
DROP TABLE CUSTOMERS;
DROP TABLE EMPLOYEES;
DROP TABLE PRODUCTS;
DROP TABLE PRODUCT_TYPES;
DROP TABLE PROMOTIONS;
DROP TABLE PURCHASES;
DROP TABLE PURCHASES_TIMESTAMP_WITH_TZ;
DROP TABLE PURCHASES_WITH_LOCAL_TZ;
DROP TABLE PURCHASES_WITH_TIMESTAMP;
DROP TABLE SALARY_GRADES;
```

### NOTA

*user\_tables contém os detalhes das tabelas no esquema do usuário. A coluna table\_name contém os nomes das tabelas.*

Você pode transferir as instruções SQL geradas para um arquivo e executá-las posteriormente.

## DESCONECTANDO-SE DO BANCO DE DADOS E SAINDO DO SQL\*PLUS

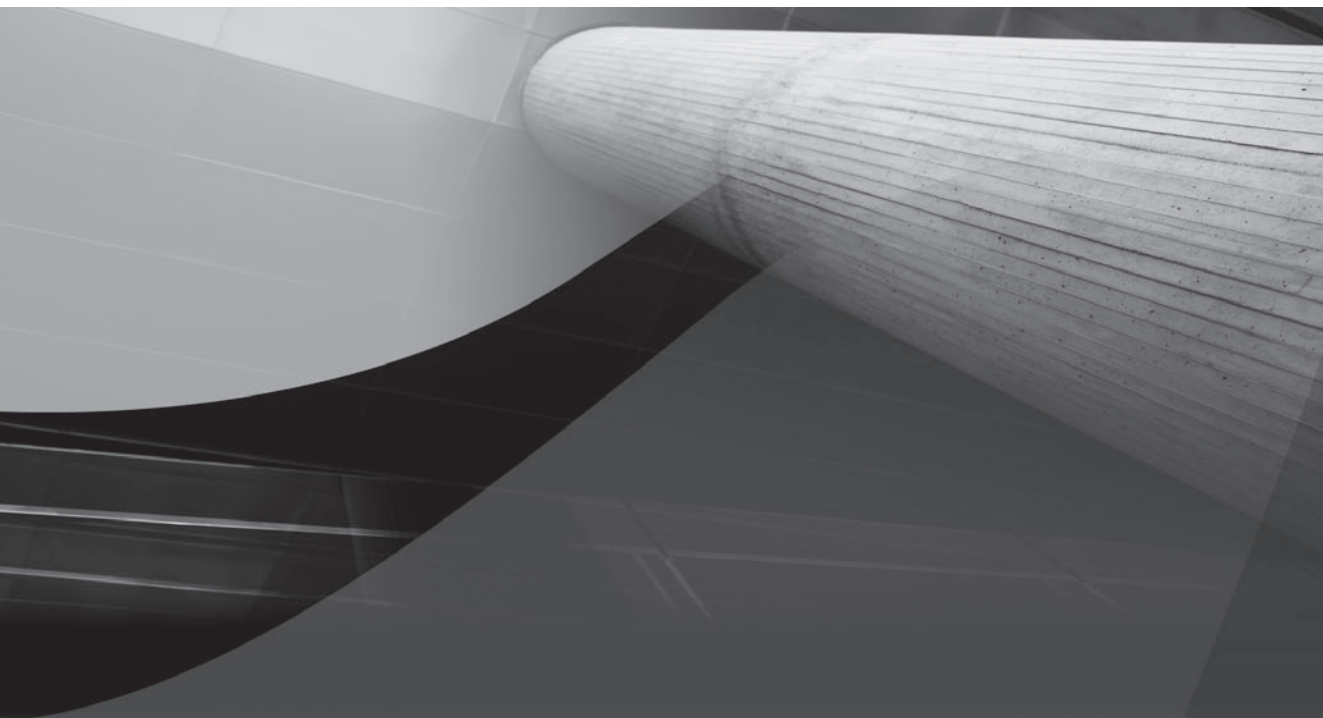
É possível desconectar-se do banco de dados e manter o SQL\*Plus em execução digitando `DISCONNECT` (o SQL\*Plus executará uma instrução `COMMIT` automaticamente). Enquanto você está conectado no banco de dados, o SQL\*Plus mantém uma sessão aberta. Ao se desconectar do banco de dados, sua sessão é finalizada. Você pode reconectar-se em um banco de dados digitando `CONNECT`. Para finalizar o SQL\*Plus, digite `EXIT` (o SQL\*Plus também executa uma instrução `COMMIT` de forma automática).

## RESUMO

Neste capítulo, você aprendeu a:

- Exibir a estrutura de uma tabela
- Editar uma instrução SQL
- Salvar, recuperar e executar arquivos contendo comandos SQL e SQL\*Plus
- Formatar os resultados retornados pelo SQL\*Plus
- Definir o tamanho da página e da linha da saída do SQL\*Plus
- Usar variáveis no SQL\*Plus
- Criar relatórios simples
- Obter ajuda do SQL\*Plus
- Escrever instruções SQL que geram outras instruções SQL
- Desconectar-se do banco de dados e sair do SQL\*Plus

Para obter mais detalhes sobre o SQL\*Plus, consulte o *SQL\*Plus User's Guide and Reference*, publicado pela Oracle Corporation. No próximo capítulo, você irá aprender a usar as funções.



# CAPÍTULO 4

Usando funções simples



Neste capítulo, você vai aprender sobre algumas das funções internas do banco de dados Oracle. Uma função aceita zero ou mais parâmetros de entrada e retorna um parâmetro de saída. Existem dois tipos principais de funções que você pode usar em um banco de dados Oracle:

- As **funções de uma única linha** operam sobre uma linha por vez e retornam uma linha de saída para cada linha de entrada. Um exemplo de função de uma única linha é `CONCAT(x, y)`, que anexa `y` a `x` e retorna a string resultante.
- As **funções agregadas** operam sobre várias linhas por vez e retornam uma linha de saída. Um exemplo de função agregada é `AVG(x)`, que retorna a média de `x`, onde `x` pode ser uma coluna ou, de modo geral, qualquer expressão.

Você irá aprender primeiro sobre as funções de uma única linha, depois sobre funções agregadas. As funções avançadas serão abordadas mais adiante neste livro.

## USANDO FUNÇÕES DE UMA ÚNICA LINHA

Uma função de uma única linha opera sobre apenas uma linha por vez e retorna uma linha de saída para cada linha. Existem cinco tipos principais de funções de uma única linha:

- As **funções de caractere** manipulam strings de caracteres.
- As **funções numéricas** efetuam cálculos.
- As **funções de conversão** convertem um valor de um tipo de banco de dados para outro.
- As **funções de data** processam datas e horas.
- As **funções de expressão regular** utilizam expressões regulares para procurar dados. Essas funções foram introduzidas no Oracle Database 10g e foram ampliadas no 11g.

Você irá aprender primeiro sobre as funções de caractere, seguidas das funções numéricas, das funções de conversão e das funções de expressão regular. As funções de data serão abordadas no próximo capítulo.

### Funções de caractere

As funções de caractere aceitam entrada de caracteres, que podem vir de uma coluna em uma tabela ou, de modo geral, de qualquer expressão. Essa entrada é processada e um resultado é retornado. Um exemplo de função de caractere é `UPPER()`, que converte as letras de uma string de entrada para maiúsculas e retorna a nova string. Outro exemplo é `NVL()`, que converte um valor nulo para outro valor. Na Tabela 4-1, que mostra algumas das funções de caractere, e em todas as definições de sintaxe que se seguem, `x` e `y` podem representar colunas de uma tabela ou, de modo geral, qualquer expressão válida. Nas seções a seguir você irá aprender mais sobre algumas das funções mostradas na Tabela 4-1.

Tabela 4-1 Funções de caractere

Função	Descrição
ASCII(x)	Retorna o valor ASCII do caractere x.
CHR(x)	Retorna o caractere com o valor ASCII x.
CONCAT(x, y)	Anexa y a x e depois retorna a nova string.
INITCAP(x)	Converte a letra inicial de cada palavra da string x em maiúsculas e retorna a nova string.
INSTR(x, localizar_string [, início] [, ocorrência])	Procura <i>localizar_string</i> em x e retorna a posição em que <i>localizar_string</i> ocorre. Você pode fornecer uma posição <i>início</i> opcional para iniciar a busca. Você também pode fornecer uma <i>ocorrência</i> opcional, que indica qual ocorrência de <i>localizar_string</i> deve ser retornada.
LENGTH(x)	Retorna o número de caracteres em x.
LOWER(x)	Converte as letras de x para minúsculas e retorna a nova string.
LPAD(x, largura [, string_preenchimento])	Preenche x com espaços à esquerda para que o comprimento total da string tenha até caracteres <i>largura</i> . Você pode fornecer uma <i>string_preenchimento</i> opcional, que especifica uma string a ser repetida à esquerda de x para ocupar o espaço preenchido e, então, a string preenchida resultante é retornada.
LTRIM(x [, string_corte])	Corta caracteres à esquerda de x. Você pode fornecer uma <i>string_corte</i> opcional, que especifica os caracteres a serem cortados; se nenhuma <i>string_corte</i> for fornecida, então espaços serão cortados por padrão.
NANVL(x, valor)	Retorna <i>valor</i> , caso x corresponda ao valor especial NAN (not a number); caso contrário, x será retornado. (Esta função foi introduzida no Oracle Database 10g.)
NVL(x, valor)	Retorna <i>valor</i> , caso x seja nulo; caso contrário, x será retornado.
NVL2(x, valor1, valor2)	Retorna <i>valor1</i> se x não é nulo; caso contrário, <i>valor2</i> é retornado.
REPLACE(x, string_busca, string_substituta)	Procura <i>string_busca</i> em x e substitui por <i>string_substituta</i> .
RPAD(x, largura [, string_preenchimento])	Igual a LPAD(), mas x é preenchido à direita.
RTRIM(x [, string_corte])	Igual a LTRIM(), mas x é cortado à direita.
SOUNDEX(x)	Retorna uma string contendo a representação fonética de x. Isso permite que você compare palavras homófonas, mas não homógrafas, em inglês.
SUBSTR(x, início [, comprimento])	Retorna uma substring de x que começa na posição especificada por <i>início</i> . Você pode fornecer um <i>comprimento</i> opcional para a substring.
TRIM([car_corte FROM] x)	Corta caracteres à esquerda e à direita de x. Você pode fornecer um <i>car_corte</i> opcional, o qual especifica os caracteres a serem cortados; se nenhum <i>car_corte</i> for fornecido, espaços serão cortados por padrão.
UPPER(x)	Converte as letras de x em maiúsculas e retorna a nova string.

### ASCII() e CHR()

Você usa ASCII(*x*) para obter o valor ASCII do caractere *x*. Você usa CHR(*x*) para obter o caractere com o valor ASCII *x*.

A consulta a seguir obtém o valor ASCII de a, A, z, Z, 0 e 9 usando ASCII():

```
SELECT ASCII('a'), ASCII('A'), ASCII('z'), ASCII('Z'), ASCII(0), ASCII(9)
FROM dual;
```

ASCII('a')	ASCII('A')	ASCII('z')	ASCII('Z')	ASCII(0)	ASCII(9)
97	65	122	90	48	57

#### NOTA

A tabela dual é usada nessa consulta. Como você viu no Capítulo 2, a tabela dual contém uma única linha com a qual podem ser feitas consultas que não afetam uma tabela específica.

A consulta a seguir obtém os caracteres com os valores ASCII 97, 65, 122, 90, 48 e 57 usando CHR():

```
SELECT CHR(97), CHR(65), CHR(122), CHR(90), CHR(48), CHR(57)
FROM dual;
```

a	A	z	Z	0	9
---	---	---	---	---	---

Observe que os caracteres retornados por CHR() nessa consulta são os mesmos passados para ASCII() na consulta anterior. Isso mostra que CHR() e ASCII() têm o efeito oposto.

### CONCAT()

Você usa CONCAT(*x*, *y*) para anexar *y* em *x* e depois retornar a nova string. A consulta a seguir anexa last\_name a first\_name usando CONCAT():

```
SELECT CONCAT(first_name, last_name)
FROM customers;
```

JohnBrown
CynthiaGreen
SteveWhite
GailBlack
DoreenBlue

#### NOTA

CONCAT() é igual ao operador || que você viu no Capítulo 2.

### INITCAP()

Você usa INITCAP(*x*) para converter a letra inicial de cada palavra de *x* em maiúsculas. A consulta a seguir recupera as colunas product\_id e description da tabela products e, então, usa INITCAP() para converter a primeira letra de cada palavra de description em maiúscula:

```
SELECT product_id, INITCAP(description)
FROM products
WHERE product_id < 4;
```

```
PRODUCT_ID INITCAP(DESCRIPTION)
-----
1 A Description Of Modern Science
2 Introduction To Chemistry
3 A Star Explodes
```

## INSTR()

Você usa `INSTR(x, localizar_string [, início] [, ocorrência])` para procurar `localizar_string` em `x`. `INSTR()` retorna a posição em que `localizar_string` ocorre. Você pode fornecer uma posição `início` opcional para iniciar a busca e também pode fornecer uma `ocorrência` opcional que indica qual ocorrência de `localizar_string` deve ser retornada. A consulta a seguir obtém a posição onde a string `Science` ocorre na coluna `name` para o produto nº 1:

```
SELECT name, INSTR(name, 'Science')
FROM products
WHERE product_id = 1;
```

NAME	INSTR(NAME, 'SCIENCE')
Modern Science	8

A próxima consulta exibe a posição onde a segunda ocorrência do caractere `e` ocorre, começando no início do nome do produto:

```
SELECT name, INSTR(name, 'e', 1, 2)
FROM products
WHERE product_id = 1;
```

NAME	INSTR(NAME, 'E', 1, 2)
Modern Science	11

Note que o segundo `e` em `Modern Science` é o undécimo caractere.

Você também pode usar datas com funções de caractere. A consulta a seguir obtém a posição onde a string `JAN` ocorre na coluna `dob` para o cliente nº 1:

```
SELECT customer_id, dob, INSTR(dob, 'JAN')
FROM customers
WHERE customer_id = 1;
```

CUSTOMER_ID	DOB	INSTR(DOB, 'JAN')
1	01-JAN-65	4

### LENGTH()

Você usa `LENGTH(x)` para obter o número de caracteres em `x`. A consulta a seguir obtém o comprimento das strings na coluna `name` da tabela `products` usando `LENGTH()`:

```
SELECT name, LENGTH(name)
FROM products;
```

NAME	LENGTH (NAME)
-----	-----
Modern Science	14
Chemistry	9
Supernova	9
Tank War	8
Z Files	7
2412: The Return	16
Space Force 9	13
From Another Planet	19
Classical Music	15
Pop 3	5
Creative Yell	13
My Front Line	13

A próxima consulta obtém o número total de caracteres que compõem o preço (`price`) do produto; observe que o ponto decimal (.) é contado no número de caracteres de `price`:

```
SELECT price, LENGTH(price)
FROM products
WHERE product_id < 3;
```

PRICE	LENGTH (PRICE)
-----	-----
19.95	5
30	2

### LOWER() e UPPER()

`LOWER(x)` é usado para converter as letras de `x` para minúsculas. Da mesma forma, `UPPER(x)` é utilizado para converter as letras de `x` para maiúsculas. A consulta a seguir converte as strings da coluna `first_name` para maiúsculas usando a função `UPPER()` e as strings da coluna `last_name` para minúsculas usando a função `LOWER()`:

```
SELECT UPPER(first_name), LOWER(last_name)
FROM customers;
```

UPPER (FIRST	LOWER (LAST
-----	-----
JOHN	brown
CYNTHIA	green

```
STEVE      white
GAIL       black
DOREEN     blue
```

### LPAD() e RPAD()

Você usa `LPAD(x, largura [, string_preenchimento])` para preencher `x` com espaços à esquerda, a fim de que o comprimento total da string seja de até caracteres `largura`. Também é possível fornecer uma `string_preenchimento` opcional, a qual especifica uma string a ser repetida à esquerda de `x` para ocupar o espaço preenchido. Então, a string preenchida resultante é retornada. Do mesmo modo, você usa `RPAD(x, largura [, string_preenchimento])` para preencher `x` com strings à direita.

A consulta a seguir recupera as colunas `name` e `price` da tabela `products`. A coluna `name` é preenchida à direita, usando `RPAD()`, com um comprimento de 30 caracteres, com pontos ocupando o espaço preenchido. A coluna `price` é preenchida à esquerda, usando `LPAD()`, com um comprimento igual a 8, com a string `++` ocupando o espaço preenchido.

```
SELECT RPAD(name, 30, '.'), LPAD(price, 8, '++')
FROM products
WHERE product_id < 4;
```

```
RPAD (NAME,30, '.')          LPAD (PRI
-----
Modern Science..... ++*19.95
Chemistry..... ++*++*30
Supernova..... ++*25.99
```

### NOTA

Esse exemplo mostra que as funções de caractere podem usar números. Especificamente, a coluna `price` do exemplo contém um número que foi preenchido à esquerda por `LPAD()`.

### LTRIM(), RTRIM() e TRIM()

Você usa `LTRIM(x [, string_corte])` para cortar caracteres à esquerda de `x`. Você pode fornecer uma `string_corte` opcional, a qual especifica os caracteres a serem cortados; se nenhuma `string_corte` for fornecida, os espaços serão cortados por padrão. Da mesma forma, você usa `RTRIM()` para cortar caracteres à direita de `x`; já `TRIM()` é usado para cortar caracteres à esquerda e à direita de `x`. A consulta a seguir usa essas três funções:

```
SELECT
  LTRIM(' Hello Gail Seymour!'),
  RTRIM('Hi Doreen Oakley!abcabc', 'abc'),
  TRIM('0' FROM '000Hey Steve Button!00000')
FROM dual;

LTRIM('HELLOGAILSEY RTRIM('HIDOREENOA TRIM('0'FROM'000H
-----
Hello Gail Seymour! Hi Doreen Oakley! Hey Steve Button!
```

**NVL()**

Você usa `NVL()` para converter um valor nulo em outro valor. `NVL(x, valor)` retorna *valor* caso *x* seja nulo; caso contrário, *x* será retornado. A consulta a seguir recupera as colunas `customer_id` e `phone` da tabela `customers`. Os valores nulos da coluna `phone` são convertidos na string 'Unknown Phone Number' por `NVL()`:

```
SELECT customer_id, NVL(phone, 'Unknown Phone Number')
FROM customers;
```

```
CUSTOMER_ID NVL(PHONE, 'UNKNOWNPH
-----
1 800-555-1211
2 800-555-1212
3 800-555-1213
4 800-555-1214
5 Unknown Phone Number
```

A coluna `phone` do cliente nº 5 é convertida em 'Unknown Phone Number', pois é nula para essa linha.

**NVL2()**

`NVL2(x, valor1, valor2)` retorna *valor1* se *x* não é nulo; caso contrário, *valor2* é retornado. A consulta a seguir recupera as colunas `customer_id` e `phone` da tabela `customers`. Os valores não nulos da coluna `phone` são convertidos na string 'Known' e os valores nulos são convertidos em 'Unknown':

```
SELECT customer_id, NVL2(phone, 'Known', 'Unknown')
FROM customers;
```

```
CUSTOMER_ID NVL2(PH
-----
1 Known
2 Known
3 Known
4 Known
5 Unknown
```

Note que os valores da coluna `phone` são convertidos em `Known` para os clientes nº 1 a 4, pois para essas linhas eles não são nulos. Para o cliente nº 5, o valor da coluna `phone` é convertido em `Unknown`, pois ele é nulo para essa linha.

**REPLACE()**

Você usa `REPLACE(x, string_busca, string_substituta)` para procurar a *string\_busca* em *x* e substituí-la por *string\_substituta*. O exemplo a seguir recupera a coluna `name` da tabela `products` para o produto nº 1 (onde `name` é `Modern Science`) e substitui a string `Science` por `Physics` usando `REPLACE()`:

```
SELECT REPLACE(name, 'Science', 'Physics')
FROM products
WHERE product_id = 1;

REPLACE(NAME, 'SCIENCE', 'PHYSICS')
-----
Modern Physics
```

**NOTA**

*REPLACE() não modifica a linha real no banco de dados; somente a linha retornada pela função é modificada.*

**SOUNDEX()**

Você usa `SOUNDEX(x)` para obter uma string contendo a representação fonética de `x`. Isso permite que você compare palavras homófonas, mas não homógrafas, em inglês. A consulta a seguir recupera a coluna `last_name` da tabela `customers`, onde `last_name` tem o som de “whyte” (em inglês):

```
SELECT last_name
FROM customers
WHERE SOUNDEX(last_name) = SOUNDEX('whyte');

LAST_NAME
-----
White
```

A próxima consulta obtém os sobrenomes que têm o som de “bloo” (em inglês):

```
SELECT last_name
FROM customers
WHERE SOUNDEX(last_name) = SOUNDEX('bloo');

LAST_NAME
-----
Blue
```

**SUBSTR()**

Você usa `SUBSTR(x, início [, comprimento])` para retornar uma substring de `x` que começa na posição especificada por `início`. Você também pode fornecer um `comprimento` opcional para a substring. A consulta a seguir usa `SUBSTR()` para obter a substring de 7 caracteres a partir da posição 2 da coluna `name` da tabela `products`:

```
SELECT SUBSTR(name, 2, 7)
FROM products
WHERE product_id < 4;

SUBSTR(
-----
odern S
hemistr
upernov
```



### Usando expressões com funções

Você não está limitado a usar colunas em funções: é possível fornecer qualquer expressão válida que seja avaliada como uma string. A consulta a seguir usa a função `SUBSTR()` para obter a substring 'little' da string 'Mary had a little lamb':

```
SELECT SUBSTR('Mary had a little lamb', 12, 6)
FROM dual;

SUBSTR
-----
little
```

### Combinando funções

Você pode usar qualquer combinação válida de funções em uma instrução SQL. A consulta a seguir combina as funções `UPPER()` e `SUBSTR()`; observe que a saída de `SUBSTR()` é passada para `UPPER()`:

```
SELECT name, UPPER(SUBSTR(name, 2, 8))
FROM products
WHERE product_id < 4;

NAME                                UPPER(SU
-----
Modern Science                      ODERN SC
Chemistry                          HEMISTRY
Supernova                          UPERNOVA
```

### NOTA

Essa capacidade de combinar funções não está limitada às funções de caractere. Qualquer combinação válida de funções funcionará.

## Funções numéricas

As funções numéricas são usadas para efetuar cálculos. Essas funções aceitam um número de entrada, que pode vir de uma coluna numérica, ou qualquer expressão que seja avaliada como um número. Então, um cálculo é feito e um número é retornado. Um exemplo de função numérica é `SQRT(x)`, que retorna a raiz quadrada de  $x$ . A Tabela 4-2 mostra algumas das funções numéricas. Você vai aprender mais sobre algumas das funções mostradas na Tabela 4-2 nas seções a seguir.

**Tabela 4-2** Funções numéricas

Função	Descrição	Exemplos
<code>ABS(x)</code>	Retorna o valor absoluto de $x$ .	<code>ABS(10) = 10</code> <code>ABS(-10) = 10</code>
<code>ACOS(x)</code>	Retorna o arco co-seno de $x$ .	<code>ACOS(1) = 0</code> <code>ACOS(-1) = 3.14159265</code>
<code>ASIN(x)</code>	Retorna o arco seno de $x$ .	<code>ASIN(1) = 1.57079633</code> <code>ASIN(-1) = -1.5707963</code>
<code>ATAN(x)</code>	Retorna o arco tangente de $x$ .	<code>ATAN(1) = .785398163</code> <code>ATAN(-1) = -.78539816</code>

(continua)

**Tabela 4-2** Funções numéricas (continuação)

Função	Descrição	Exemplos
ATAN2( <i>x</i> , <i>y</i> )	Retorna o arco tangente de <i>x</i> e <i>y</i> .	ATAN2(1, -1) = 2.35619449
BITAND( <i>x</i> , <i>y</i> )	Retorna o resultado da execução da função bitwise AND em <i>x</i> e <i>y</i> .	BITAND(0, 0) = 0 BITAND(0, 1) = 0 BITAND(1, 0) = 0 BITAND(1, 1) = 1 BITAND(1010, 1100) = 64
COS( <i>x</i> )	Retorna o co-seno de <i>x</i> , onde <i>x</i> é um ângulo em radianos.	COS(90 * 3.1415926) = 1 COS(45 * 3.1415926) = -1
COSH( <i>x</i> )	Retorna o co-seno hiperbólico de <i>x</i> .	COSH(3.1415926) = 11.5919527
CEIL( <i>x</i> )	Retorna o menor inteiro maior ou igual a <i>x</i> .	CEIL(5.8) = 6 CEIL(-5.2) = -5
EXP( <i>x</i> )	Retorna o resultado do número <i>e</i> elevado à potência <i>x</i> , onde <i>e</i> é aproximadamente 2,71828183.	EXP(1) = 2.71828183 EXP(2) = 7.3890561
FLOOR( <i>x</i> )	Retorna o maior inteiro menor ou igual a <i>x</i> .	FLOOR(5.8) = 5 FLOOR(-5.2) = -6
LOG( <i>x</i> , <i>y</i> )	Retorna o logaritmo, base <i>x</i> , de <i>y</i> .	LOG(2, 4) = 2 LOG(2, 5) = 2.32192809
LN( <i>x</i> )	Retorna o logaritmo natural de <i>x</i> .	LN(2.71828183) = 1
MOD( <i>x</i> , <i>y</i> )	Retorna o resto, quando <i>x</i> é dividido por <i>y</i> .	MOD(8, 3) = 2 MOD(8, 4) = 0
POWER( <i>x</i> , <i>y</i> )	Retorna o resultado de <i>x</i> elevado à potência <i>y</i> .	POWER(2, 1) = 2 POWER(2, 3) = 8
ROUND( <i>x</i> [, <i>y</i> ])	Retorna o resultado do arredondamento de <i>x</i> com <i>y</i> casas decimais opcionais. Se <i>y</i> for omitido, <i>x</i> será arredondado para zero casa decimal. Se <i>y</i> for negativo, <i>x</i> será arredondado à esquerda do ponto decimal.	ROUND(5.75) = 6 ROUND(5.75, 1) = 5.8 ROUND(5.75, -1) = 10
SIGN( <i>x</i> )	Retorna -1 se <i>x</i> é negativo, 1 se <i>x</i> é positivo ou 0 se <i>x</i> é zero.	SIGN(-5) = -1 SIGN(5) = 1 SIGN(0) = 0
SIN( <i>x</i> )	Retorna o seno de <i>x</i> .	SIN(0) = 0
SINH( <i>x</i> )	Retorna o seno hiperbólico de <i>x</i> .	SINH(1) = 1.17520119
SQRT( <i>x</i> )	Retorna a raiz quadrada de <i>x</i> .	SQRT(25) = 5 SQRT(5) = 2.23606798
TAN( <i>x</i> )	Retorna a tangente de <i>x</i> .	TAN(0) = 0
TANH( <i>x</i> )	Retorna a tangente hiperbólica de <i>x</i> .	TANH(1) = .761594156
TRUNC( <i>x</i> [, <i>y</i> ])	Retorna o resultado do truncamento de <i>x</i> com <i>y</i> casas decimais opcionais. Se <i>y</i> for omitido, <i>x</i> será truncado em zero casa decimal. Se <i>y</i> for negativo, <i>x</i> será truncado à esquerda do ponto decimal.	TRUNC(5.75) = 5 TRUNC(5.75, 1) = 5.7 TRUNC(5.75, -1) = 0

### **ABS()**

Você usa `ABS(x)` para obter o valor absoluto de `x`. O valor absoluto de um número é esse número sem qualquer sinal, positivo ou negativo. A consulta a seguir obtém o valor absoluto de 10 e -10:

```
SELECT ABS(10), ABS(-10)
FROM dual;
```

ABS(10)	ABS(-10)
10	10

O valor absoluto de 10 é 10. O valor absoluto de -10 é 10.

Evidentemente, os parâmetros inseridos em qualquer uma das funções numéricas não precisam ser números literais. A entrada também pode ser uma coluna numérica de uma tabela ou, de forma geral, qualquer expressão válida. A consulta a seguir obtém o valor absoluto da subtração de 30 da coluna `price` da tabela `products` para os três primeiros produtos:

```
SELECT product_id, price, price - 30, ABS(price - 30)
FROM products
WHERE product_id < 4;
```

PRODUCT_ID	PRICE	PRICE-30	ABS(PRICE-30)
1	19.95	-10.05	10.05
2	30	0	0
3	25.99	-4.01	4.01

### **CEIL()**

Você usa `CEIL(x)` para obter o menor inteiro maior ou igual a `x`. A consulta a seguir usa `CEIL()` para obter os valores absolutos de 5,8 e -5,2:

```
SELECT CEIL(5.8), CEIL(-5.2)
FROM dual;
```

CEIL(5.8)	CEIL(-5.2)
6	-5

O teto de 5,8 é 6, porque 6 é o menor inteiro maior do que 5,8. O teto de -5,2 é -5, porque -5,2 é negativo e o menor inteiro maior do que este é -5.

### **FLOOR()**

Você usa `FLOOR(x)` para obter o maior inteiro menor ou igual a `x`. A consulta a seguir usa `FLOOR()` para obter o valor absoluto de 5,8 e -5,2:

```
SELECT FLOOR(5.8), FLOOR(-5.2)
FROM dual;
```

FLOOR(5.8)	FLOOR(-5.2)
5	-6

O piso de 5,8 é 5; porque 5 é o maior inteiro menor do que 5,8. O piso de -5,2 é -6, porque -5,2 é negativo e o maior inteiro menor do que este é -6.

### MOD()

Você usa MOD(*x*, *y*) para obter o resto da divisão de *x* por *y*. A consulta a seguir usa MOD() para obter o resto, quando 8 é dividido por 3 e por 4:

```
SELECT MOD(8, 3), MOD(8, 4)
FROM dual;
```

MOD(8,3)	MOD(8,4)
2	0

O resto, quando 8 é dividido por 3, é 2: 3 cabe duas vezes em 8, deixando uma sobra de 2 — o resto. O resto, quando 8 é dividido por 4, é 0: 4 cabe duas vezes em 8, não deixando uma sobra.

### POWER()

Você usa POWER(*x*, *y*) para obter o resultado de *x* elevado à potência *y*. A consulta a seguir usa POWER() para obter 2 elevado às potências 1 e 3:

```
SELECT POWER(2, 1), POWER(2, 3)
FROM dual;
```

POWER(2,1)	POWER(2,3)
2	8

Quando 2 é elevado à potência 1, que é equivalente a  $2^1$ , o resultado é 2; 2 elevado à potência 3 é equivalente a  $2^3$ , cujo resultado é 8.

### ROUND()

Você usa ROUND(*x*, [*y*]) para obter o resultado do arredondamento de *x* com *y* casas decimais opcionais. Se *y* for omitido, *x* será arredondado com zero casa decimal. Se *y* for negativo, *x* será arredondado à esquerda do ponto decimal. A consulta a seguir usa ROUND() para obter o resultado do arredondamento de 5,75 com zero, 1 e -1 casas decimais:

```
SELECT ROUND(5.75), ROUND(5.75, 1), ROUND(5.75, -1)
FROM dual;
```

ROUND(5.75)	ROUND(5.75,1)	ROUND(5.75,-1)
6	5.8	10

5,75 arredondado com zero casa decimal dá 6; 5,75 arredondado com uma casa decimal (à direita do ponto decimal) dá 5,8; e 5,75 arredondado com uma casa decimal à esquerda do ponto decimal (conforme indicado pelo uso de um sinal negativo) dá 10.

### **SIGN()**

Você usa `SIGN(x)` para obter o sinal de `x`. `SIGN()` retorna `-1` se `x` é negativo, `1` se `x` é positivo ou `0` se `x` é zero. A consulta a seguir obtém o sinal de `-5`, `5` e `0`:

```
SELECT SIGN(-5), SIGN(5), SIGN(0)
FROM dual;
```

SIGN(-5)	SIGN(5)	SIGN(0)
-1	1	0

O sinal de `-5` é `-1`; o sinal de `5` é `1`; o sinal de `0` é `0`.

### **SQRT()**

Você usa `SQRT(x)` para obter a raiz quadrada de `x`. A consulta a seguir obtém a raiz quadrada de `25` e `5`:

```
SELECT SQRT(25), SQRT(5)
FROM dual;
```

SQRT(25)	SQRT(5)
5	2.23606798

A raiz quadrada de `25` é `5`; a raiz quadrada de `5` é aproximadamente `2,236`.

### **TRUNC()**

Você usa `TRUNC(x, [y])` para obter o resultado do truncamento do número `x` com `y` casas decimais opcionais. Se `y` for omitido, `x` será truncado com zero casa decimal. Se `y` for negativo, `x` será truncado à esquerda do ponto decimal. A consulta a seguir trunca `5,75` com zero, `1` e `-1` casas decimais:

```
SELECT TRUNC(5.75), TRUNC(5.75, 1), TRUNC(5.75, -1)
FROM dual;
```

TRUNC(5.75)	TRUNC(5.75,1)	TRUNC(5.75,-1)
5	5.7	0

Na consulta acima, `5,75` truncado com zero casa decimal dá `5`; `5,75` truncado com uma casa decimal (à direita do ponto decimal) dá `5,7`; e `5,75` truncado com uma casa decimal à esquerda do ponto decimal (conforme indicado pelo sinal negativo) dá `0`.

## **Funções de conversão**

Às vezes, é preciso converter um valor de um tipo de dados para outro. Por exemplo, talvez você queira formatar o preço de um produto que está armazenado como um número (por exemplo, `1346,95`), em uma string contendo cifrão e pontos de milhar (por exemplo, `$1.346,95`). Para tanto, você usa uma função de conversão a fim de converter um valor de um tipo de dados para outro. A Tabela 4-3 mostra algumas das funções de conversão.

Você aprenderá mais sobre as funções `TO_CHAR()` e `TO_NUMBER()` nas seções a seguir. Algumas das outras funções da Tabela 4-3 serão abordadas ao longo deste livro. Para saber mais sobre conjuntos de caracteres de idioma nacional e Unicode, consulte o *Oracle Database Globalization Support Guide* da Oracle Corporation.

**Tabela 4-3** Funções de conversão

Função	Descrição
ASCIISTR( <i>x</i> )	Converte <i>x</i> em uma string ASCII, onde <i>x</i> pode ser uma string de qualquer conjunto de caracteres.
BIN_TO_NUM( <i>x</i> )	Converte um número binário <i>x</i> em um valor NUMBER.
CAST( <i>x</i> AS <i>tipo</i> )	Converte <i>x</i> em um tipo de banco de dados compatível, especificado em <i>tipo</i> .
CHARTOROWID( <i>x</i> )	Converte <i>x</i> em um valor ROWID.
COMPOSE( <i>x</i> )	Converte <i>x</i> em uma string Unicode na forma totalmente normalizada, no mesmo conjunto de caracteres que <i>x</i> . Unicode usa um conjunto de caracteres de 2 bytes e pode representar mais de 65.000 caracteres; ele também pode ser usado para representar caracteres de outros idiomas que não o inglês.
CONVERT( <i>x</i> , <i>conjunto_car_origem</i> , <i>conjunto_car_destino</i> )	Converte <i>x</i> do <i>conjunto_car_origem</i> para o <i>conjunto_car_destino</i> .
DECODE( <i>x</i> , <i>busca</i> , <i>resultado</i> , <i>padrão</i> )	Compara <i>x</i> com o valor presente em <i>busca</i> ; se forem iguais, DECODE() retornará o valor em <i>resultado</i> ; caso contrário, o valor presente em <i>padrão</i> será retornado.
DECOMPOSE( <i>x</i> )	Converte <i>x</i> em uma string Unicode, após a decomposição da string, no mesmo conjunto de caracteres que <i>x</i> .
HEXTORAW( <i>x</i> )	Converte o caractere <i>x</i> contendo dígitos hexadecimais (base 16) em um número binário (RAW). Então, esta função retorna o número RAW.
NUMTODSINTERVAL( <i>x</i> )	Converte o número <i>x</i> em um valor INTERVAL DAY TO SECOND. Você irá aprender sobre funções relacionadas a intervalo de data e hora no próximo capítulo.
NUMTOYMINTERVAL( <i>x</i> )	Converte o número <i>x</i> em um valor INTERVAL YEAR TO MONTH.
RAWTOHEX( <i>x</i> )	Converte o número binário (RAW) <i>x</i> em uma string VARCHAR2 contendo o número hexadecimal equivalente.
RAWTONHEX( <i>x</i> )	Converte o número binário (RAW) <i>x</i> em uma string NVARCHAR2 contendo o número hexadecimal equivalente. (NVARCHAR2 armazena uma string usando o conjunto de caracteres nacional.)
ROWIDTOCHAR( <i>x</i> )	Converte o ROWID <i>x</i> em uma string VARCHAR2.
ROWIDTONCHAR( <i>x</i> )	Converte o ROWID <i>x</i> em uma string NVARCHAR2.
TO_BINARY_DOUBLE( <i>x</i> )	Converte <i>x</i> em um valor BINARY_DOUBLE. (Esta função foi introduzida no Oracle Database 10g.)
TO_BINARY_FLOAT( <i>x</i> )	Converte <i>x</i> em um valor BINARY_FLOAT. (Esta função foi introduzida no Oracle Database 10g.)
TO_BLOB( <i>x</i> )	Converte <i>x</i> em um Binary Large Object (BLOB, objeto binário grande). Um BLOB é usado para armazenar grandes volumes de dados binários. Você irá aprender sobre BLOBs no Capítulo 14.

(continua)

**Tabela 4-3** Funções de conversão (continuação)

Função	Descrição
TO_CHAR( <i>x</i> [, <i>formato</i> ])	Converte <i>x</i> em uma string VARCHAR2. Você pode fornecer um <i>formato</i> opcional indicando o formato de <i>x</i> .
TO_CLOB( <i>x</i> )	Converte <i>x</i> em um Character Large Object (CLOB, objeto de caracteres grande). Um CLOB é usado para armazenar grandes volumes de dados de caractere.
TO_DATE( <i>x</i> [, <i>formato</i> ])	Converte <i>x</i> em um valor DATE.
TO_DSINTERVAL( <i>x</i> )	Converte a string <i>x</i> em um valor INTERVAL DAY TO SECOND.
TO_MULTI_BYTE( <i>x</i> )	Converte os caracteres single-byte presentes em <i>x</i> em seus caracteres multi-byte correspondentes. O tipo de retorno é igual ao tipo de <i>x</i> .
TO_NCHAR( <i>x</i> )	Converte <i>x</i> do conjunto de caracteres do banco de dados em uma string NVARCHAR2.
TO_NCLOB( <i>x</i> )	Converte <i>x</i> em um NCLOB. Um NCLOB é usado para armazenar grandes volumes de dados de caractere de idioma nacional.
TO_NUMBER( <i>x</i> [, <i>formato</i> ])	Converte <i>x</i> em um valor NUMBER.
TO_SINGLE_BYTE( <i>x</i> )	Converte os caracteres multi-byte presentes em <i>x</i> para seus caracteres single-byte correspondentes. O tipo de retorno é igual ao tipo de <i>x</i> .
TO_TIMESTAMP( <i>x</i> )	Converte a string <i>x</i> em um valor TIMESTAMP.
TO_TIMESTAMP_TZ( <i>x</i> )	Converte a string <i>x</i> em um valor TIMESTAMP WITH TIME ZONE.
TO_YMINTERVAL( <i>x</i> )	Converte a string <i>x</i> em um valor INTERVAL YEAR TO MONTH.
TRANSLATE( <i>x</i> , <i>da_string</i> , <i>para_string</i> )	Converte todas as ocorrências de <i>da_string</i> de <i>x</i> em <i>para_string</i> .
UNISTR( <i>x</i> )	Converte os caracteres de <i>x</i> em um caractere NCHAR. (NCHAR armazena um caractere usando o conjunto de caracteres de idioma nacional.)

**TO\_CHAR()**

Você usa TO\_CHAR(*x* [, *formato*]) para converter *x* em uma string. Também é possível fornecer um *formato* opcional indicando o formato de *x*. A estrutura de *formato* depende de *x* ser um número ou uma data. Você irá aprender a usar TO\_CHAR() para converter um número em uma string nesta seção e verá como converter uma data em uma string no próximo capítulo.

Vejamos duas consultas simples que utilizam TO\_CHAR() para converter um número em uma string. A consulta a seguir converte 12345,67 em uma string:

```
SELECT TO_CHAR(12345.67)
FROM dual;

TO_CHAR(1
-----
12345.67
```

A próxima consulta usa `TO_CHAR()` para converter 12345678,90 em uma string e especifica que esse número deve ser convertido usando o formato 99,999.99. Isso resulta na string retornada por `TO_CHAR()` tendo uma vírgula para delimitar os milhares:

```
SELECT TO_CHAR(12345.67, '99,999.99')
FROM dual;

TO_CHAR(12
-----
12,345.67
```

A string *formato* opcional que pode ser passada para `TO_CHAR()` tem vários parâmetros que afetam a string retornada por essa função. Alguns desses parâmetros estão listados na Tabela 4-4.

Tabela 4-4 Parâmetros de formatação numérica

Parâmetro	Exemplos de formato	Descrição
9	999	Retorna dígitos nas posições especificadas, com um sinal negativo à esquerda se o número é negativo.
0	0999 9990	0999: Retorna um número com zeros à esquerda. 9990: Retorna um número com zeros à direita.
.	999.99	Retorna um ponto decimal na posição especificada.
,	9,999	Retorna uma vírgula na posição especificada.
\$	\$999	Retorna um símbolo de dólar à esquerda.
B	B9.99	Se a parte inteira de um número de ponto fixo é zero, retorna espaços para os zeros.
C	C999	Retorna o símbolo de moeda ISO na posição especificada. O símbolo vem do parâmetro de banco de dados <code>NLS_ISO_CURRENCY</code> definido pelo DBA.
D	9D99	Retorna o símbolo de ponto decimal na posição especificada. O símbolo vem do parâmetro de banco de dados <code>NLS_NUMERIC_CHARACTER</code> (o padrão é um caractere de ponto-final).
EEEE	9.99EEEE	Retorna o número usando a notação científica.
FM	FM90.9	Remove os espaços à esquerda e à direita do número.
G	9G999	Retorna o símbolo de separador de grupo na posição especificada. O símbolo vem do parâmetro de banco de dados <code>NLS_NUMERIC_CHARACTER</code> .
L	L999	Retorna o símbolo de moeda local na posição especificada. O símbolo vem do parâmetro de banco de dados <code>NLS_CURRENCY</code> .

(continua)



**Tabela 4-4** *Parâmetros de formatação numérica (continuação)*

Parâmetro	Exemplos de formato	Descrição
MI	999MI	Retorna um número negativo com um sinal de menos à direita. Retorna um número positivo com um espaço à direita.
PR	999PR	Retorna um número negativo entre sinais de menor e maior (< >). Retorna um número positivo com espaços à esquerda e à direita.
RN rn	RN rn	Retorna o número como algarismos romanos. RN retorna numerais maiúsculos; rn retorna numerais minúsculos. O número deve ser um valor inteiro entre 1 e 3999.
S	S999 999S	S999: Retorna um número negativo com um sinal de negativo à esquerda; retorna um número positivo com um sinal de positivo à esquerda. 999S: Retorna um número negativo com um sinal de negativo à direita; retorna um número positivo com um sinal de positivo à direita.
TM	TM	Retorna o número usando a quantidade mínima de caracteres. O padrão é TM9, que retorna o número usando notação fixa, a não ser que o número de caracteres seja maior do que 64. Se for maior do que 64, o número será retornado usando notação científica.
U	U999	Retorna o símbolo de moeda duplo (o Euro, por exemplo) na posição especificada. O símbolo vem do parâmetro de banco de dados NLS_DUAL_CURRENCY.
V	99V99	Retorna o número multiplicado por 10 <sup>x</sup> , onde x é o número de caracteres 9 após a letra v. Se necessário, o número é arredondado.
X	XXXX	Retorna o número em hexadecimal. Se o número não é um valor inteiro, ele é arredondado para um inteiro.

Vejamos mais alguns exemplos que convertem números em strings usando `TO_CHAR()`. A tabela a seguir mostra exemplos de chamada de `TO_CHAR()`, junto com a saída retornada.

Chamada da função <code>TO_CHAR()</code>	Saída
<code>TO_CHAR(12345.67, '99999.99')</code>	12345.67
<code>TO_CHAR(12345.67, '99,999.99')</code>	12,345.67
<code>TO_CHAR(-12345.67, '99,999.99')</code>	-12,345.67
<code>TO_CHAR(12345.67, '099,999.99')</code>	012,345.67
<code>TO_CHAR(12345.67, '99,999.9900')</code>	12,345.6700
<code>TO_CHAR(12345.67, '\$99,999.99')</code>	\$12,345.67
<code>TO_CHAR(0.67, 'B9.99')</code>	.67

**Chamada da função TO\_CHAR()****Saída**

TO_CHAR(12345.67, 'C99,999.99')	USD12,345.67
TO_CHAR(12345.67, '99999D99')	12345.67
TO_CHAR(12345.67, '99999.99EEEE')	1.23E+04
TO_CHAR(0012345.6700, 'FM99999.99')	12345.67
TO_CHAR(12345.67, '99999G99')	123,46
TO_CHAR(12345.67, 'L99,999.99')	\$12,345.67
TO_CHAR(-12345.67, '99,999.99MI')	12,345.67
TO_CHAR(-12345.67, '99,999.99PR')	12,345.67
TO_CHAR(2007, 'RN')	MMVII
TO_CHAR(12345.67, 'TM')	12345.67
TO_CHAR(12345.67, 'U99,999.99')	\$12,345.67
TO_CHAR(12345.67, '99999V99')	1234567

TO\_CHAR() retornará uma string de caracteres # se você tentar formatar um número que contenha dígitos demais para o formato. Por exemplo:

```
SELECT TO_CHAR(12345678.90, '99,999.99')
FROM dual;

TO_CHAR(12
-----
#####
```

Os caracteres # são retornados por TO\_CHAR() porque o número 12345678.90 tem mais dígitos do que os permitidos no formato 99,999.99.

Você também pode usar TO\_CHAR() para converter colunas contendo números em strings. Por exemplo, a consulta a seguir usa TO\_CHAR() para converter a coluna price da tabela products em uma string:

```
SELECT product_id, 'The price of the product is' || TO_CHAR(price,
'$99.99')
FROM products
WHERE product_id < 5;

PRODUCT_ID 'THEPRICEOFTHEPRODUCTIS' || TO_CHAR(
-----
1 The price of the product is $19.95
2 The price of the product is $30.00
3 The price of the product is $25.99
4 The price of the product is $13.95
```

**TO\_NUMBER()**

Você usa TO\_NUMBER(*x* [, *formato*]) para converter *x* em um número. É possível fornecer uma string de *formato* opcional para indicar o formato de *x*. Sua string de *formato* pode usar os mesmos parâmetros listados anteriormente na Tabela 4-4.

A consulta a seguir converte a string 970.13 em um número usando TO\_NUMBER():

```
SELECT TO_NUMBER('970.13')
FROM dual;

TO_NUMBER('970.13')
-----
          970.13
```

A próxima consulta converte a string 970.13 em um número usando TO\_NUMBER() e depois soma 25.5 a esse número:

```
SELECT TO_NUMBER('970.13') + 25.5
FROM dual;

TO_NUMBER('970.13')+25.5
-----
          995.63
```

A próxima consulta converte a string -\$12,345.67 em um número, passando a string de formato \$99,999.99 para TO\_NUMBER():

```
SELECT TO_NUMBER('- $12,345.67', '$99,999.99')
FROM dual;

TO_NUMBER('- $12,345.67', '$99,999.99')
-----
        -12345.67
```

**CAST()**

Você usa CAST(*x* AS *tipo*) para converter *x* em um tipo de banco de dados compatível especificado por *tipo*. A tabela a seguir mostra as conversões de tipo válidas (as conversões válidas estão marcadas com um X):

De	Para						
	BINARY_FLOAT BINARY_DOUBLE	CHAR VARCHAR2	NUMBER	DATE TIMESTAMP INTERVAL	RAW	ROWID UROWID	NCHAR NVARCHAR2
BINARY_FLOAT BINARY_DOUBLE	X	X	X				X
CHAR VARCHAR2	X	X	X	X	X	X	
NUMBER	X	X	X				X
DATE TIMESTAMP INTERVAL		X		X			
RAW		X			X		
ROWID UROWID		X				X	
NCHAR NVARCHAR2	X		X	X	X	X	X

A consulta a seguir mostra o uso de `CAST()` para converter valores literais em tipos específicos:

```
SELECT
  CAST(12345.67 AS VARCHAR2(10)),
  CAST('9A4F' AS RAW(2)),
  CAST('05-JUL-07' AS DATE),
  CAST(12345.678 AS NUMBER(10,2))
FROM dual;

CAST(12345 CAST CAST('05- CAST(12345.678ASNUMBER(10,2))
-----
12345.67    9A4F 05-JUL-07                                12345.68
```

Você também pode converter valores de coluna de um tipo para outro, como mostrado na consulta a seguir:

```
SELECT
  CAST(price AS VARCHAR2(10)),
  CAST(price + 2 AS NUMBER(7,2)),
  CAST(price AS BINARY_DOUBLE)
FROM products
WHERE product_id = 1;

CAST(PRICE CAST(PRICE+2ASNUMBER(7,2)) CAST(PRICEASBINARY_DOUBLE)
-----
19.95                                21.95                                1.995E+001
```

No Capítulo 5, você verá mais exemplos de como usar `CAST()` para converter datas, horas e intervalos. Além disso, o Capítulo 13 ensina a usar `CAST()` para converter coleções.

## Funções de expressão regular

Nesta seção, você aprenderá sobre as expressões regulares e suas funções de banco de dados Oracle associadas. Essas funções permitem procurar um padrão de caracteres em uma string. Por exemplo, digamos que você tenha a lista de anos a seguir:

```
1965
1968
1971
1970
```

e queira obter os anos de 1965 a 1968. Você pode fazer isso usando a seguinte expressão regular:

```
^196[5-8]$
```

A expressão regular contém alguns *metacaracteres*. Nesse exemplo, `^`, `[5-8]` e `$` são os metacaracteres; `^` corresponde à posição inicial de uma string; `[5-8]` corresponde aos caracteres entre 5 e 8; `$` corresponde à posição final de uma string. Portanto, `^196` corresponde a uma string que começa com 196 e `[5-8]$` corresponde a uma string que termina com 5, 6, 7 ou 8. Portanto, `^196[5-8]$` corresponde a 1965, 1966, 1967 e 1968, que são os anos que você queria obter da lista.

O próximo exemplo usa a string a seguir, que contém uma citação (em inglês) de *Romeu e Julieta* de Shakespeare:

```
But, soft! What light through yonder window breaks?
```

Digamos que você queira obter a substring `light`. Você faz isso usando a seguinte expressão regular:

```
1[[:alpha:]]{4}
```

Nessa expressão regular, `[[:alpha:]]` e `{4}` são os metacaracteres. `[[:alpha:]]` corresponde a um caractere alfanumérico de A-Z e de a-z; `{4}` repete quatro vezes a correspondência anterior. Quando `1`, `[[:alpha:]]` e `{4}` são combinados, eles correspondem a uma sequência de cinco letras, começando com `1`. Portanto, a expressão regular `1[[:alpha:]]{4}` corresponde a `light` na string. A Tabela 4-5 lista alguns dos metacaracteres que você pode usar em uma expressão regular, junto com seus significados e um exemplo de seu uso.

**Tabela 4-5** Metacaracteres de expressão regular

Metacaracteres	Significado	Exemplos
\	Corresponde a um caractere especial ou a uma literal ou realiza uma referência retroativa. (Uma referência retroativa repete a correspondência anterior.)	\n corresponde ao caractere de nova linha \\ corresponde a \ \( corresponde a ( \) corresponde a )
^	Corresponde à posição no início da string.	^A corresponde a A, se A é o primeiro caractere na string.
\$	Corresponde à posição no final da string.	\$B corresponde a B, se B é o último caractere na string.
*	Corresponde ao caractere anterior, zero ou mais vezes.	ba*rk corresponde a brk, bark, baark etc.
+	Corresponde ao caractere anterior, uma ou mais vezes.	ba+rk corresponde a bark, baark etc., mas não a brk.
?	Corresponde ao caractere anterior zero ou uma vez.	ba?rk corresponde somente a brk e a bark.
{n}	Corresponde a um caractere exatamente n vezes, onde n é um valor inteiro.	hob{2}it corresponde a hobbit.
{n,m}	Corresponde a um caractere pelo menos n vezes e no máximo m vezes, onde n e m são ambos valores inteiros.	hob{2,3}it corresponde somente a hobbit e hobbit.
.	Corresponde a qualquer caractere único, exceto um valor nulo.	hob.it corresponde a hobait, hobbit etc.

(continua)

**Tabela 4-5** Metacaracteres de expressão regular (continuação)

Metacaracteres	Significado	Exemplos
(padrão)	Uma subexpressão que corresponde ao padrão especificado. Você usa subexpressões para construir expressões regulares complexas. Você pode acessar as correspondências individuais, conhecidas como capturas, a partir desse tipo de sub-expressão.	<code>anatom(y ies)</code> corresponde a <code>anatomy</code> e a <code>anatomies</code> .
<code>x y</code>	Corresponde a <code>x</code> ou a <code>y</code> , onde <code>x</code> e <code>y</code> são um ou mais caracteres.	<code>war peace</code> corresponde a <code>war</code> ou a <code>peace</code> .
<code>[abc]</code>	Corresponde a qualquer um dos caracteres incluídos.	<code>[ab]bc</code> corresponde a <code>abc</code> e a <code>bbc</code> .
<code>[a-z]</code>	Corresponde a qualquer caractere no intervalo especificado.	<code>[a-c]bc</code> corresponde a <code>abc</code> , <code>bbc</code> e a <code>cbc</code> .
<code>[ : ]</code>	Especifica uma classe de caracteres e corresponde a qualquer caractere nessa classe.	<p><code>[ :alphanum: ]</code> corresponde aos caracteres alfanuméricos 0–9, A–Z e a–z.</p> <p><code>[ :alpha: ]</code> corresponde aos caracteres alfabéticos A–Z e a–z.</p> <p><code>[ :blank: ]</code> corresponde a espaço ou tabulação.</p> <p><code>[ :digit: ]</code> corresponde aos dígitos 0–9.</p> <p><code>[ :graph: ]</code> corresponde a caracteres não em branco.</p> <p><code>[ :lower: ]</code> corresponde aos caracteres alfabéticos minúsculos a–z.</p> <p><code>[ :print: ]</code> é semelhante a <code>[ :graph: ]</code>, exceto que inclui o caractere de espaço.</p> <p><code>[ :punct: ]</code> corresponde aos caracteres de pontuação. “,” etc.</p> <p><code>[ :espaço: ]</code> corresponde a todos os caracteres de espaço em branco.</p> <p><code>[ :upper: ]</code> corresponde a todos os caracteres alfabéticos maiúsculos A–Z.</p> <p><code>[ :xdigit: ]</code> corresponde aos caracteres permitidos em um número hexadecimal 0–9, A–F e a–f.</p>
<code>[ . ]</code>	Corresponde a um elemento de comparação, como um elemento de multacaracteres.	Sem exemplos.
<code>[ == ]</code>	Especifica classes de equivalência.	Sem exemplos.

(continua)

**Tabela 4-5** Metacaracteres de expressão regular (continuação)

Metacaracteres	Significado	Exemplos
\n	Esta é uma referência retroativa para uma captura anterior, onde <i>n</i> é um valor inteiro positivo.	(.)\1 corresponde a dois caracteres consecutivos idênticos. O (.) captura qualquer caractere único, exceto um valor nulo, e \1 repete a captura, correspondendo novamente ao mesmo caractere, portanto, correspondendo a dois caracteres consecutivos idênticos.

O Oracle Database 10g release 2 introduziu vários metacaracteres influenciados pela linguagem Perl, os quais estão mostrados na Tabela 4-6. A Tabela 4-7 mostra as funções de expressão regular. As funções de expressão regular foram introduzidas no Oracle Database 10g e mais itens foram adicionados no 11g, como mostrado na tabela.

**Tabela 4-6** Metacaracteres influenciados pela linguagem Perl

Metacaracteres	Significado
\d	Caractere de dígito
\D	Caractere não dígito
\w	Caractere de palavra
\W	Caractere não palavra
\s	Caractere de espaço em branco
\S	Caractere não espaço em branco
\A	Corresponde somente ao início de uma string ou antes de um caractere de nova linha no final de uma string
\Z	Corresponde somente ao final de uma string
*?	Corresponde ao elemento do padrão precedente, 0 ou mais vezes
+?	Corresponde ao elemento do padrão precedente, uma ou mais vezes
??	Corresponde ao elemento do padrão precedente, 0 ou uma vez
{n}	Corresponde ao elemento do padrão precedente exatamente <i>n</i> vezes
{n, }	Corresponde ao elemento do padrão precedente pelo menos <i>n</i> vezes
{n, m}	Corresponde ao elemento do padrão precedente pelo menos <i>n</i> , mas não mais do que <i>m</i> vezes

Tabela 4-7 Funções de expressão regular

Função	Descrição
<pre>REGEXP_LIKE(x, padrão [, opção_correspondência])</pre>	<p>Procura em <i>x</i> a expressão regular definida no parâmetro <i>padrão</i>. Você também pode fornecer uma <i>opção_correspondência</i> opcional, a qual pode ser definida como um dos seguintes caracteres:</p> <ul style="list-style-type: none"> <li>■ 'c', que especifica correspondência com diferenciação de maiúsculas e minúsculas (este é o padrão)</li> <li>■ 'I', que especifica correspondência sem diferenciação de maiúsculas e minúsculas</li> <li>■ 'n', que permite usar o operador de correspondência com qualquer caractere</li> <li>■ 'm', que trata <i>x</i> como uma linha múltipla</li> </ul>
<pre>REGEXP_INSTR(x, padrão [, início [, ocorrência [, opção_retorno [, opção_correspondência [, opção_sub-exp]]]])</pre>	<p>Procura o <i>padrão</i> em <i>x</i> e retorna a posição na qual o <i>padrão</i> ocorre. Opcionalmente, você pode fornecer:</p> <ul style="list-style-type: none"> <li>■ a posição de <i>início</i> para começar a busca. O padrão é 1, que é o primeiro caractere em <i>x</i>.</li> <li>■ a <i>ocorrência</i>, que indica qual ocorrência do <i>padrão</i> deve ser retornada. O padrão é 1, que significa que a função retorna a posição da primeira ocorrência do <i>padrão</i> em <i>x</i>.</li> <li>■ a <i>opção_retorno</i>, que indica qual valor inteiro deve retornar. O especifica que o valor inteiro a ser retornado é a posição do primeiro caractere em <i>x</i>; 1 especifica que o valor inteiro a ser retornado é a posição do caractere em <i>x</i> após a ocorrência.</li> <li>■ a <i>opção_correspondência</i>, para alterar a correspondência padrão. Funciona da mesma maneira como especificado em REGEXP_LIKE().</li> <li>■ a <i>opção_sub-exp</i> (novidade do Oracle Database 11g) funciona da seguinte forma: para um padrão com subexpressões, <i>opção_sub-exp</i> é um valor inteiro não-negativo de 0 a 9, indicando qual subexpressão no <i>padrão</i> é o alvo da função. Por exemplo, considere a expressão a seguir: 0123 ((abc)(de)f)ghi) 45 (678) Se a <i>opção_sub-exp</i> é 0, a posição de <i>padrão</i> é retornada. Se o <i>padrão</i> não tem o número correto de subexpressões, então a função retorna 0. Um valor de <i>opção_sub-exp</i> nulo retorna nulo. O valor padrão de <i>opção_sub-exp</i> é 0.</li> </ul>

(continua)



**Tabela 4-7** Funções de expressão regular (continuação)

Função	Descrição
<code>REGEXP_REPLACE(x, padrão [, string_substituta [, início [, ocorrência [, opção_correspondência]]])</code>	Procura o <i>padrão</i> em <i>x</i> e o substitui por <i>string_substituta</i> . As outras opções têm o mesmo significado daquelas mostradas anteriormente.
<code>REGEXP_SUBSTR(x, padrão [, início [, ocorrência [, opção_correspondência [, opção_sub-exp]]])</code>	Retorna uma substring de <i>x</i> que corresponde ao <i>padrão</i> ; a busca começa na posição especificada por <i>início</i> . As outras opções têm o mesmo significado daquelas mostradas anteriormente. A <i>opção_sub-exp</i> (novidade do Oracle Database 11g) funciona da mesma maneira mostrada para <code>REGEXP_INSTR()</code> .
<code>REGEXP_COUNT(x, padrão [, início [, opção_correspondência]])</code>	Novidade do Oracle Database 11g. Procura o <i>padrão</i> em <i>x</i> e retorna o número de vezes que o <i>padrão</i> é encontrado em <i>x</i> . Opcionalmente, você pode fornecer: <ul style="list-style-type: none"><li>■ a posição de <i>início</i> para começar a busca. O <i>padrão</i> é 1, que é o primeiro caractere em <i>x</i>.</li><li>■ a <i>opção_correspondência</i> para alterar a correspondência padrão. Funciona da mesma maneira mostrada para <code>REGEXP_LIKE()</code>.</li></ul>

Você aprenderá mais sobre as funções de expressão regular nas seções a seguir.

**REGEXP\_LIKE()**

Você usa `REGEXP_LIKE(x, padrão [, opção_correspondência])` para procurar em *x* a expressão regular definida no parâmetro *padrão*. Também é possível fornecer uma *opção\_correspondência* opcional, que pode ser configurada com um dos seguintes caracteres:

- 'c', que especifica correspondência com diferenciação de maiúsculas e minúsculas (este é o padrão)
- 'I', que especifica correspondência sem diferenciação de maiúsculas e minúsculas
- 'n', que permite usar o operador de correspondência com qualquer caractere
- 'm', que trata *x* como uma linha múltipla

A consulta a seguir recupera os clientes cuja data de nascimento está entre 1965 e 1968 usando `REGEXP_LIKE()`:

```
SELECT customer_id, first_name, last_name, dob
FROM customers
WHERE REGEXP_LIKE(TO_CHAR(dob, 'YYYY'), '^196[5-8]$');

CUSTOMER_ID FIRST_NAME LAST_NAME DOB
-----
1 John Brown 01-JAN-65
2 Cynthia Green 05-FEB-68
```

A próxima consulta recupera os clientes cujo nome começa com J ou j. Observe que a expressão regular passada para `REGEXP_LIKE()` é `^j` e que a opção de correspondência é `i` (`i` indica correspondência sem diferenciação de maiúsculas e minúsculas e, portanto, neste exemplo, `^j` corresponde a J ou j).

```
SELECT customer_id, first_name, last_name, dob
FROM customers
WHERE REGEXP_LIKE(first_name, '^j', 'i');
```

```
CUSTOMER_ID FIRST_NAME LAST_NAME DOB
-----
1 John Brown 01-JAN-65
```

### REGEXP\_INSTR()

Você usa `REGEXP_INSTR(x, padrão [, início [, ocorrência [, opção_retorno [, opção_correspondência]]])` para procurar o *padrão* em *x*. Essa função retorna a posição em que o *padrão* ocorre (as posições começam no número 1). A consulta a seguir retorna a posição correspondente à expressão regular `1[[:alpha:]]{4}` usando `REGEXP_INSTR()`:

```
SELECT
  REGEXP_INSTR('But, soft! What light through yonder window breaks?',
    '1[[:alpha:]]{4}') AS result
FROM dual;
```

```
RESULT
-----
17
```

Note que 17 é retornado, que é a posição do 1 em *light*. A próxima consulta retorna a posição da segunda ocorrência correspondente à expressão regular `s[[:alpha:]]{3}` a partir da posição 1:

```
SELECT
  REGEXP_INSTR('But, soft! What light through yonder window softly breaks?',
    's[[:alpha:]]{3}', 1, 2) AS result
FROM dual;
```

```
RESULT
-----
45
```

A próxima consulta retorna a posição da segunda ocorrência correspondente à letra *o*, iniciando a pesquisa na posição 10:

```
SELECT
  REGEXP_INSTR('But, soft! What light through yonder window breaks?',
    'o', 10, 2) AS result
FROM dual;
```

```
RESULT
-----
32
```

**REGEXP\_REPLACE()**

Você usa `REGEXP_REPLACE(x, padrão [, string_substituta [, início [, ocorrência [, opção_correspondência]]])` para procurar o *padrão* em *x* e substituí-lo pela *string\_substituta*. A consulta a seguir substitui a substring correspondente à expressão regular `1[[:alpha:]]{4}` pela string 'sound' usando `REGEXP_REPLACE()`:

```
SELECT
  REGEXP_REPLACE('But, soft! What light through yonder window breaks?',
    '1[[:alpha:]]{4}', 'sound') AS result
FROM dual;
```

```
RESULT
-----
But, soft! What sound through yonder window breaks?
```

Note que *light* foi substituída por *sound*.

**REGEXP\_SUBSTR()**

Você usa `REGEXP_SUBSTR(x, padrão [, início [, ocorrência [, opção_correspondência]]])` para obter uma substring de *x* correspondente ao *padrão*; a busca começa na posição especificada por *início*. A consulta a seguir retorna a substring correspondente à expressão regular `1[[:alpha:]]{4}` usando `REGEXP_SUBSTR()`:

```
SELECT
  REGEXP_SUBSTR('But, soft! What light through yonder window breaks?',
    '1[[:alpha:]]{4}') AS result
FROM dual;
```

```
RESUL
-----
light
```

**REGEXP\_COUNT()**

`REGEXP_COUNT()` é novidade do Oracle Database 11g. Você usa `REGEXP_COUNT(x, padrão [, início [, opção_correspondência]])` para procurar *padrão* em *x* e obter o número de vezes que *padrão* é encontrado em *x*. Você pode fornecer um número *início* opcional para indicar o caractere em *x* a fim de iniciar a busca de *padrão* e uma string *opção\_correspondência* opcional a fim de indicar a opção de correspondência. A consulta a seguir retorna o número de vezes que a expressão regular `s[[:alpha:]]{3}` ocorre em uma string usando `REGEXP_COUNT()`:

```
SELECT
  REGEXP_COUNT('But, soft! What light through yonder window softly breaks?',
    's[[:alpha:]]{3}') AS result
FROM dual;
```

```
RESULT
-----
2
```

Note que 2 é retornado, o que significa que a expressão regular tem duas correspondências na string fornecida.

## USANDO FUNÇÕES AGREGADAS

As funções mostradas até aqui operam em uma única linha por vez e retornam uma linha de saída para cada linha de entrada. Nesta seção, você irá aprender sobre as funções agregadas, que operam em um grupo de linhas e retornam uma linha de saída.



### NOTA

As funções agregadas também são conhecidas como funções de grupo, pois operam em grupos de linhas.

A Tabela 4-8 lista algumas das funções agregadas, todas as quais retornam um valor `NUMBER`. Aqui estão alguns pontos a serem lembrados ao se usar funções agregadas:

- Você pode usar as funções agregadas com qualquer expressão válida. Por exemplo, é possível usar as funções `COUNT()`, `MAX()` e `MIN()` com números, strings e data/horários.
- Os valores nulos são ignorados pelas funções agregadas, pois um valor nulo indica que o valor é desconhecido e, portanto, não pode ser usado no cálculo da função agregada.
- Você pode usar a palavra-chave `DISTINCT` em uma função agregada para excluir entradas duplicadas do cálculo da função agregada.

Você irá aprender mais sobre algumas das funções agregadas mostradas na Tabela 4-8 nas seções a seguir. Nos capítulos 7 e 8, você verá como utilizar essas funções em conjunto com as cláusulas `ROLLUP` e `RETURNING` da instrução `SELECT`. `ROLLUP` permite obter um subtotal de um grupo de linhas, onde o subtotal é calculado com uma das funções agregadas; `RETURNING` permite armazenar em uma variável o valor retornado por uma função agregada.

**Tabela 4-8** Funções agregadas

Função	Descrição
<code>AVG(x)</code>	Retorna o valor médio de $x$
<code>COUNT(x)</code>	Retorna o número de linhas retornadas por uma consulta envolvendo $x$
<code>MAX(x)</code>	Retorna o valor máximo de $x$
<code>MEDIAN(x)</code>	Retorna o valor da mediana de $x$
<code>MIN(x)</code>	Retorna o valor mínimo de $x$
<code>STDDEV(x)</code>	Retorna o desvio padrão de $x$
<code>SUM(x)</code>	Retorna a soma de $x$
<code>VARIANCE(x)</code>	Retorna a variância de $x$

**AVG()**

Você usa `AVG(x)` para obter o valor médio de `x`. A consulta a seguir obtém o preço médio dos produtos; observe que a coluna `price` da tabela `products` é passada para a função `AVG()`:

```
SELECT AVG(price)
FROM products;

AVG (PRICE)
-----
19.7308333
```

Você pode usar as funções agregadas com qualquer expressão válida. Por exemplo, a consulta a seguir passa a expressão `price + 2` para `AVG()`; isso soma 2 ao preço de cada linha e, então, retorna a média desses valores.

```
SELECT AVG(price + 2)
FROM products;

AVG (PRICE)
-----
21.7308333
```

Você pode usar a palavra-chave `DISTINCT` para excluir valores idênticos de um cálculo. Por exemplo, a consulta a seguir usa a palavra-chave `DISTINCT` para excluir valores idênticos na coluna `price` ao calcular a média usando `AVG()`:

```
SELECT AVG(DISTINCT price)
FROM products;

AVG (DISTINCTPRICE)
-----
20.2981818
```

Note que a média nesse exemplo é ligeiramente mais alta do que a média retornada pela primeira consulta desta seção. Isso porque o valor do produto nº 12 (13.49) na coluna `price` é igual ao valor do produto nº 7; ele é considerado uma duplicata e é excluído do cálculo efetuado por `AVG()`. Portanto, neste exemplo a média é ligeiramente mais alta.

**COUNT()**

Você usa `COUNT(x)` para obter o número de linhas retornadas por uma consulta. A consulta a seguir obtém o número de linhas na tabela `products` usando `COUNT()`:

```
SELECT COUNT(product_id)
FROM products;

COUNT (PRODUCT_ID)
-----
12
```

**DICA**

É aconselhável evitar o uso do asterisco (\*) com a função COUNT(), pois ele pode fazer COUNT() demorar mais para retornar o resultado. Em vez disso, você deve usar uma coluna da tabela ou a pseudocoluna ROWID. (Conforme vimos no Capítulo 2, a coluna ROWID contém a localização interna da linha no banco de dados Oracle.)

O exemplo a seguir passa ROWID para COUNT() e obtém o número de linhas na tabela products:

```
SELECT COUNT(ROWID)
FROM products;
```

```
COUNT(ROWID)
-----
          12
```

**MAX() e MIN()**

Você usa MAX(x) e MIN(x) para obter os valores máximo e mínimo de x. A consulta a seguir obtém os valores máximo e mínimo da coluna price da tabela products usando MAX() e MIN():

```
SELECT MAX(price), MIN(price)
FROM products;
```

```
MAX(PRICE) MIN(PRICE)
-----
      49.99      10.99
```

Você pode usar MAX() e MIN() com qualquer tipo, inclusive strings e datas. Quando você usa MAX() com strings, estas são classificadas em ordem alfabética, com a string “máxima” no final da lista e a string “mínima” no início. Por exemplo, a string Albert apareceria antes de Zeb nessa lista. O exemplo a seguir obtém as strings name máxima e mínima da tabela products usando MAX() e MIN():

```
SELECT MAX(name), MIN(name)
FROM products;
```

```
MAX(NAME)                                MIN(NAME)
-----
Z Files                                2412: The Return
```

No caso de datas, a data “máxima” ocorre no ponto mais recente no tempo e a data “mínima”, no ponto mais antigo. A consulta a seguir obtém o valor máximo e mínimo de dob da tabela customers usando MAX() e MIN():

```
SELECT MAX(dob), MIN(dob)
FROM customers;
```

```
MAX(DOB) MIN(DOB)
-----
16-MAR-71 01-JAN-65
```

**STDDEV()**

Você usa `STDDEV(x)` para obter o desvio padrão de  $x$ . O desvio padrão é uma função estatística, e é definido como a raiz quadrada da variância (você aprenderá sobre variância em breve). A consulta a seguir obtém o desvio padrão dos valores da coluna `price` da tabela `products` usando `STDDEV()`:

```
SELECT STDDEV(price)
FROM products;

STDDEV(PRICE)
-----
11.0896303
```

**SUM()**

`SUM(x)` soma todos os valores presentes em  $x$  e retorna o total. A consulta a seguir obtém a soma da coluna `price` da tabela `products` usando `SUM()`:

```
SELECT SUM(price)
FROM products;

SUM(PRICE)
-----
236.77
```

**VARIANCE()**

Você usa `VARIANCE(x)` para obter a variância de  $x$ . A variância é uma função estatística e é definida como a dispersão ou variação de um grupo de números em uma amostra. Ela é igual ao quadrado do desvio padrão. O exemplo a seguir obtém a variância dos valores da coluna `price` da tabela `products` usando `VARIANCE()`:

```
SELECT VARIANCE(price)
FROM products;

VARIANCE(PRICE)
-----
122.979899
```

**AGRUPANDO LINHAS**

Às vezes, você precisa agrupar blocos de linhas em uma tabela e obter alguma informação sobre esses grupos de linhas. Por exemplo, talvez queira obter o preço médio dos diferentes tipos de produtos da tabela `products`. Você primeiro aprenderá a fazer isso da maneira difícil e, depois, da maneira fácil, que envolve o uso da cláusula `GROUP BY` para agrupar linhas semelhantes.

Na maneira difícil, você limita as linhas passadas para a função `AVG()` usando uma cláusula `WHERE`. Por exemplo, a consulta a seguir obtém o preço médio dos livros da tabela `products` (os livros têm o valor de `product_type_id` igual a 1):

```
SELECT AVG(price)
FROM products
WHERE product_type_id = 1;

AVG(PRICE)
-----
24.975
```

Seria preciso realizar mais consultas com diferentes valores para `product_type_id` na cláusula `WHERE` para obter o preço médio dos outros tipos de produtos, mas isso é muito trabalhoso. Existe uma maneira mais fácil de fazer isso usando a cláusula `GROUP BY`.

## Usando a cláusula `GROUP BY` para agrupar linhas

Você usa a cláusula `GROUP BY` para agrupar linhas em blocos com um valor comum de coluna. Por exemplo, a consulta a seguir agrupa as linhas da tabela `products` em blocos com o mesmo valor de `product_type_id`:

```
SELECT product_type_id
FROM products
GROUP BY product_type_id;

PRODUCT_TYPE_ID
-----
1
2
3
4
```

Note que existe uma linha no conjunto de resultados para cada bloco de linhas com o mesmo valor de `product_type_id` e que existe uma lacuna entre 1 e 2 (você verá por que essa lacuna ocorre em breve). No conjunto de resultados, existe uma linha para produtos com um valor de `product_type_id` igual a 1, outra para produtos com um valor de `product_type_id` igual a 2 etc. Na verdade, existem duas linhas na tabela `products` com um valor de `product_type_id` igual a 1, quatro linhas com um valor de `product_type_id` igual a 2 e assim por diante para as outras linhas da tabela. Essas linhas são agrupadas em blocos separados pela cláusula `GROUP BY`, um bloco para cada `product_type_id`. O primeiro bloco contém duas linhas, o segundo, quatro linhas e assim por diante. A lacuna entre 1 e 2 é causada por uma linha cujo valor de `product_type_id` é nulo. Essa linha é mostrada no exemplo a seguir:

```
SELECT product_id, name, price
FROM products
WHERE product_type_id IS NULL;

PRODUCT_ID NAME PRICE
-----
12 My Front Line 13.49
```



Como o valor de `product_type_id` dessa linha é nulo, na consulta anterior a cláusula `GROUP BY` agrupa essa linha em um único bloco. No conjunto de resultados a linha aparece em branco, pois o valor de `product_type_id` é nulo para o bloco; portanto, existe uma lacuna entre 1 e 2.

### Usando várias colunas em um grupo

Você pode especificar várias colunas em uma cláusula `GROUP BY`. Por exemplo, a consulta a seguir inclui as colunas `product_id` e `customer_id` da tabela `purchases` em uma cláusula `GROUP BY`:

```
SELECT product_id, customer_id
FROM purchases
GROUP BY product_id, customer_id;
```

PRODUCT_ID	CUSTOMER_ID
1	1
1	2
1	3
1	4
2	1
2	2
2	3
2	4
3	3

### Usando grupos de linhas com funções agregadas

Você pode passar blocos de linhas para uma função agregada. A função agregada efetua seu cálculo no grupo de linhas em cada bloco e retorna um valor por bloco. Por exemplo, para obter o número de linhas com o mesmo valor de `product_type_id` da tabela `products`:

- Use a cláusula `GROUP BY` para agrupar as linhas em blocos com o mesmo valor de `product_type_id`.
- Use `COUNT (ROWID)` para obter o número de linhas em cada bloco.

A consulta a seguir mostra isso:

```
SELECT product_type_id, COUNT(ROWID)
FROM products
GROUP BY product_type_id
ORDER BY product_type_id;
```

PRODUCT_TYPE_ID	COUNT(ROWID)
1	2
2	4
3	2
4	3
	1

Note que existem cinco linhas no conjunto de resultados, com cada linha correspondendo a uma ou mais linhas da tabela `products` agrupadas com o mesmo valor de `product_type_id`. A partir do conjunto de resultados, você pode ver que existem duas linhas com um valor de `product_type_id` igual a 1, quatro linhas com um valor de `product_type_id` igual a 2 e assim por diante. A última linha no conjunto de resultados mostra que existe uma linha com um valor de `product_type_id` nulo (isso é causado pelo produto “My Front Line”, mencionado anteriormente). Por exemplo, para obter o preço médio dos diferentes tipos de produtos da tabela `products`:

- Use a cláusula `GROUP BY` para agrupar as linhas em blocos com o mesmo valor de `product_type_id`.
- Use `AVG(price)` para obter o preço médio de cada bloco de linhas.

A consulta a seguir mostra isso:

```
SELECT product_type_id, AVG(price)
FROM products
GROUP BY product_type_id
ORDER BY product_type_id;
```

PRODUCT_TYPE_ID	AVG(PRICE)
1	24.975
2	26.22
3	13.24
4	13.99
	13.49

Cada grupo de linhas com o mesmo valor de `product_type_id` é passado para a função `AVG()`. Então, a função `AVG()` calcula o preço médio de cada grupo. Como você pode ver a partir do conjunto de resultados, o preço médio do grupo de produtos com um valor de `product_type_id` igual a 1 é 24,975. Da mesma forma, o preço médio dos produtos com um valor de `product_type_id` igual a 2 é 26,22. Note que a última linha do conjunto de resultados mostra um preço médio de 13,49; esse é simplesmente o preço do produto “My Front Line”, a única linha com um valor de `product_type_id` nulo. Você pode usar qualquer uma das funções agregadas com a cláusula `GROUP BY`. Por exemplo, a consulta a seguir obtém a variância de preço dos produtos para cada valor de `product_type_id`:

```
SELECT product_type_id, VARIANCE(price)
FROM products
GROUP BY product_type_id
ORDER BY product_type_id;
```

PRODUCT_TYPE_ID	VARIANCE(PRICE)
1	50.50125
2	280.8772
3	.125
4	7
	0

Um ponto a ser lembrado é que você não precisa incluir as colunas usadas na cláusula `GROUP BY` na lista de colunas imediatamente após a instrução `SELECT`. Por exemplo, a consulta a seguir é igual à anterior, exceto que `product_type_id` é omitido da cláusula `SELECT`:

```
SELECT VARIANCE(price)
FROM products
GROUP BY product_type_id
ORDER BY product_type_id;

VARIANCE (PRICE)
-----
50.50125
280.8772
.125
7
0
```

Você também pode incluir uma chamada de função agregada na cláusula `ORDER BY`, como mostrado:

```
SELECT VARIANCE(price)
FROM products
GROUP BY product_type_id
ORDER BY VARIANCE(price);

VARIANCE (PRICE)
-----
0
.125
7
50.50125
280.8772
```

## Utilização incorreta de chamadas de funções agregadas

Quando sua consulta contém uma função agregada — e recupera colunas não colocadas dentro de uma função agregada —, essas colunas devem ser colocadas em uma cláusula `GROUP BY`. Se você se esquecer disso, obterá o seguinte erro: `ORA-00937: not a single-group group function`. Por exemplo, a consulta a seguir tenta recuperar a coluna `product_type_id` e `AVG(price)`, mas omite uma cláusula `GROUP BY` para `product_type_id`:

```
SQL> SELECT product_type_id, AVG(price)
2 FROM products;
SELECT product_type_id, AVG(price)
*
ERROR at line 1:
ORA-00937: not a single-group group function
```

O erro ocorre porque o banco de dados não sabe o que fazer com a coluna `product_type_id`. Pense sobre isso: a consulta tenta usar a função agregada `AVG()`, a qual opera em várias linhas,

mas também tenta obter os valores da coluna `product_type_id` para cada linha individual. Não é possível fazer as duas coisas ao mesmo tempo. Você deve fornecer uma cláusula `GROUP BY` para dizer ao banco de dados que agrupe várias linhas com o mesmo valor de `product_type_id`; então, o banco de dados passará esses grupos de linhas para a função `AVG()`.



### CUIDADO

*Quando uma consulta contém uma função agregada — e recupera colunas não colocadas dentro de uma função agregada —, essas colunas devem ser colocadas em uma cláusula `GROUP BY`.*

Além disso, você não pode usar uma função agregada para limitar linhas em uma cláusula `WHERE`. Se tentar fazer isso, obterá o seguinte erro: `ORA-00934: group function is not allowed here`. Por exemplo:

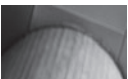
```
SQL> SELECT product_type_id, AVG(price)
       2 FROM products
       3 WHERE AVG(price) > 20
       4 GROUP BY product_type_id;
WHERE AVG(price) > 20
      *
ERROR at line 3:
ORA-00934: group function is not allowed here
```

O erro ocorre porque só é possível usar a cláusula `WHERE` para filtrar linhas *individuais* e não *grupos* de linhas. Para filtrar grupos de linhas, você deve usar a cláusula `HAVING`, abordada a seguir.

## Usando a cláusula `HAVING` para filtrar grupos de linhas

A cláusula `HAVING` é usada para filtrar grupos de linhas. Ela é colocada após a cláusula `GROUP BY`:

```
SELECT...
FROM...
WHERE
GROUP BY...
HAVING...
ORDER BY...;
```



### NOTA

*`GROUP BY` pode ser usada sem `HAVING`, mas `HAVING` deve ser usada em conjunto com `GROUP BY`.*

Digamos que você queira ver os tipos de produtos que têm um preço médio maior do que US\$20. Para tanto:

- Use a cláusula `GROUP BY` para agrupar as linhas em blocos com o mesmo valor de `product_type_id`.
- Use a cláusula `HAVING` para limitar os resultados retornados aos grupos que têm um preço médio maior do que US\$20.

A consulta a seguir mostra isso:

```
SELECT product_type_id, AVG(price)
FROM products
GROUP BY product_type_id
HAVING AVG(price) > 20;
```

PRODUCT_TYPE_ID	AVG(PRICE)
1	24.975
2	26.22

Somente os grupos de linhas que têm um preço médio maior do que US\$20 são exibidos.

## Usando as cláusulas WHERE e GROUP BY juntas

É possível usar as cláusulas WHERE e GROUP BY juntas na mesma consulta. Quando você faz isso, primeiro a cláusula WHERE filtra as linhas retornadas e, então, a cláusula GROUP BY agrupa as linhas restantes em blocos. Por exemplo, a consulta a seguir usa:

- Uma cláusula WHERE para filtrar as linhas da tabela products, a fim de selecionar aquelas cujo valor de price é menor do que US\$15.
- Uma cláusula GROUP BY para agrupar as linhas restantes pela coluna product\_type\_id.

```
SELECT product_type_id, AVG(price)
FROM products
WHERE price < 15
GROUP BY product_type_id
ORDER BY product_type_id;
```

PRODUCT_TYPE_ID	AVG(PRICE)
2	14.45
3	13.24
4	12.99
	13.49

## Usando as cláusulas WHERE, GROUP BY e HAVING juntas

É possível usar as cláusulas WHERE, GROUP BY e HAVING juntas na mesma consulta. Quando você faz isso, primeiro a cláusula WHERE filtra as linhas, então a cláusula GROUP BY agrupa as linhas restantes em blocos e, finalmente, a cláusula HAVING filtra os grupos de linhas. Por exemplo, a consulta a seguir usa:

- Uma cláusula WHERE para filtrar as linhas da tabela products, para selecionar aquelas cujo valor de price é menor do que US\$15.
- Uma cláusula GROUP BY para agrupar as linhas restantes pela coluna product\_type\_id.
- Uma cláusula HAVING para filtrar os grupos de linhas, para selecionar aqueles cujo preço médio é maior do que US\$13.

```

SELECT product_type_id, AVG(price)
FROM products
WHERE price < 15
GROUP BY product_type_id
HAVING AVG(price) > 13
ORDER BY product_type_id;

PRODUCT_TYPE_ID  AVG(PRICE)
-----
                2      14.45
                3      13.24
                3      13.49

```

Compare esses resultados com o exemplo anterior. Note que o grupo de linhas com o valor de `product_type_id` igual a 4 é filtrado. Isso porque o grupo de linhas tem um preço médio menor do que US\$13. A última consulta usa `ORDER BY AVG(price)` para reorganizar os resultados pelo preço médio:

```

SELECT product_type_id, AVG(price)
FROM products
WHERE price < 15
GROUP BY product_type_id
HAVING AVG(price) > 13
ORDER BY AVG(price);

PRODUCT_TYPE_ID  AVG(PRICE)
-----
                3      13.24
                3      13.49
                2      14.45

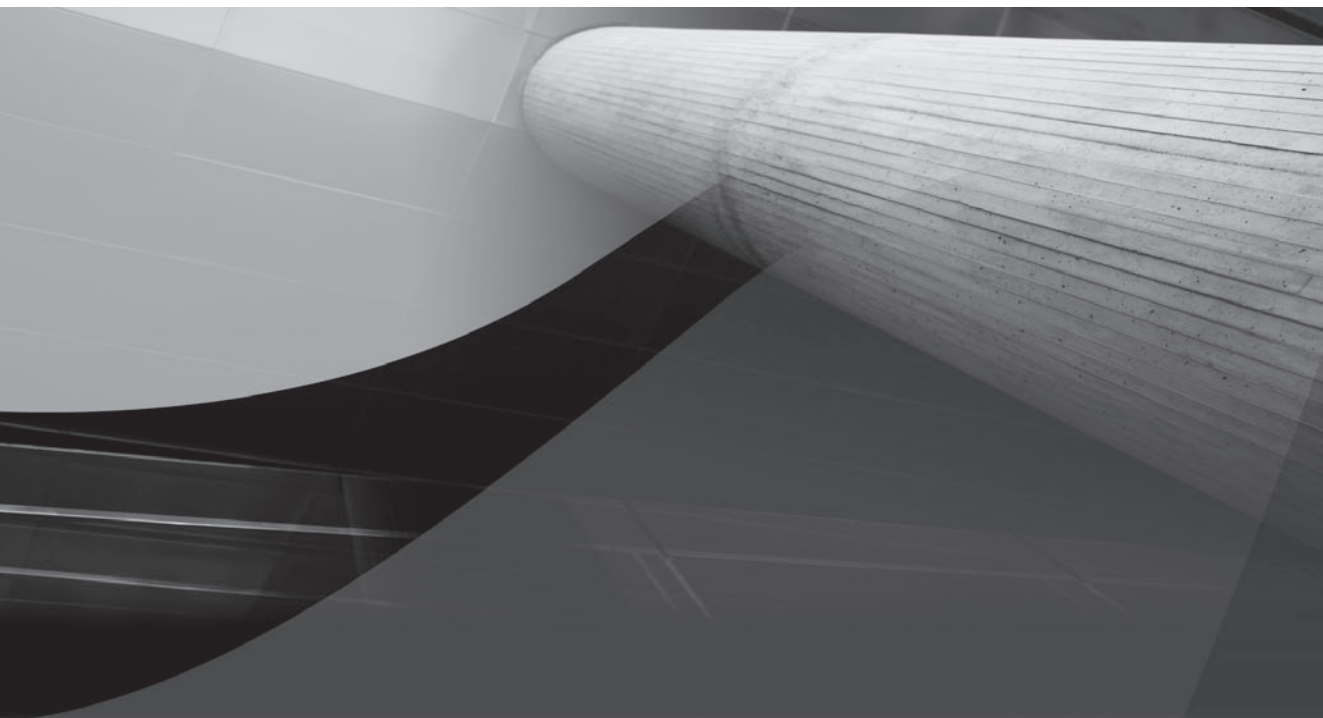
```

## RESUMO

Neste capítulo, você aprendeu que:

- O banco de dados Oracle tem dois grupos principais de funções: funções de uma única linha e funções agregadas.
- As funções de uma única linha operam em uma linha por vez e retornam uma linha de saída para cada linha de entrada. Existem cinco tipos principais de funções de uma única linha: funções de caractere, funções numéricas, funções de conversão, funções de data e funções de expressão regular.
- As funções agregadas operam em várias linhas e retornam uma linha de saída.
- Os blocos de linhas podem ser agrupados com a cláusula `GROUP BY`.
- Os grupos de linhas podem ser filtrados com a cláusula `HAVING`.

No próximo capítulo, você vai aprender sobre datas e horas.



# CAPÍTULO 5

Armazenando  
e processando  
datas e horas

Neste capítulo, você vai aprender a:

- Processar e armazenar uma data e hora específica, conhecida coletivamente como data/horário. Um exemplo de data/horário é 19:15:30 horas de 10 de outubro de 2007. Uma data/horário é armazenada usando o tipo `DATE`. O tipo `DATE` armazena o século, todos os quatro dígitos de um ano, o mês, o dia, a hora (no formato de 24 horas), o minuto e o segundo.
- Usar *timestamp* para armazenar uma data e hora específica. Um timestamp armazena o século, todos os quatro dígitos de um ano, o mês, o dia, a hora (no formato de 24 horas), o minuto e o segundo. As vantagens de um timestamp em relação a um tipo `DATE` são que um timestamp pode armazenar frações de segundo e um fuso horário.
- Usar *intervalos* de tempo para armazenar um período de tempo. Um exemplo de intervalo de tempo é 1 ano e 3 meses.

Vamos começar vendo alguns exemplos simples de armazenamento e recuperação de datas.

## EXEMPLOS SIMPLES DE ARMAZENAMENTO E RECUPERAÇÃO DE DATAS

Por padrão, o banco de dados usa o formato `DD-MON-YYYY` para representar uma data, onde:

- `DD` é um dia com dois dígitos; por exemplo, 05
- `MON` são as três primeiras letras do mês; por exemplo, FEB
- `YYYY` é um ano de quatro dígitos; por exemplo, 1968

Vejamos um exemplo de adição de uma linha na tabela `customers` que contém uma coluna `DATE` chamada `dob`. A instrução `INSERT` a seguir adiciona uma linha na tabela `customers`, configurando a coluna `dob` como 05-FEB-1968:

```
INSERT INTO customers (
    customer_id, first_name, last_name, dob, phone
) VALUES (
    6, 'Fred', 'Brown', '05-FEB-1968', '800-555-1215'
);
```

Você também pode usar a palavra-chave `DATE` para fornecer uma data literal ao banco de dados. A data deve usar o formato de data do padrão ANSI `YYYY-MM-DD`, onde:

- `YYYY` é um ano com quatro dígitos.
- `MM` é um mês com dois dígitos, de 1 a 12.
- `DD` é um dia com dois dígitos.

### DICA

Usar datas do padrão ANSI em instruções SQL apresenta a vantagem de que essas instruções podem ser executadas em bancos de dados que não sejam o Oracle.



Por exemplo, para especificar a data 25 de outubro de 1972, você usa `DATE '1972-10-25'`. A instrução `INSERT` a seguir adiciona uma linha na tabela `customers`, especificando `DATE '1972-10-25'` para a coluna `dob`:

```
INSERT INTO customers (
  customer_id, first_name, last_name, dob, phone
) VALUES (
  7, 'Steve', 'Purple', DATE '1972-10-25', '800-555-1215'
);
```

Por padrão, o banco de dados retorna datas no formato `DD-MON-YY`, onde `YY` são os dois últimos dígitos do ano. Por exemplo, o exemplo a seguir recupera linhas da tabela `customers` e depois executa uma instrução `ROLLBACK` para desfazer os resultados das duas instruções `INSERT` anteriores; observe os anos de dois dígitos na coluna `dob` retornados pela consulta:

```
SELECT *
FROM customers;
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB	PHONE
1	John	Brown	01-JAN-65	800-555-1211
2	Cynthia	Green	05-FEB-68	800-555-1212
3	Steve	White	16-MAR-71	800-555-1213
4	Gail	Black		800-555-1214
5	Doreen	Blue	20-MAY-70	
6	Fred	Brown	05-FEB-68	800-555-1215
7	Steve	Purple	25-OCT-72	800-555-1215

```
ROLLBACK;
```

O valor de `dob` do cliente nº 4 é nulo e, portanto, fica em branco no conjunto de resultados anterior.

#### NOTA

*Se você executou as duas instruções `INSERT`, desfça as alterações com a instrução `ROLLBACK`. Desse modo, você manterá o banco de dados em seu estado inicial e os resultados de suas consultas corresponderão aos deste capítulo.*

Nesta seção, você viu alguns exemplos simples do uso de datas que utilizam formatos padrão. Na seção a seguir, irá aprender a fornecer seus próprios formatos de data e verá como converter uma data/horário para outro tipo do banco de dados.

## CONVERTENDO DATA/HORÁRIOS COM `TO_CHAR()` E `TO_DATE()`

O banco de dados Oracle tem funções que permitem converter um valor de um tipo de dados para outro. Você viu algumas dessas funções no capítulo anterior. Nesta seção, aprenderá a utilizar as funções `TO_CHAR()` e `TO_DATE()` para converter uma data/horário em uma string e vice-versa. A Tabela 5-1 resume as funções `TO_CHAR()` e `TO_DATE()`.

**Tabela 5-1** Funções de conversão `TO_CHAR()` e `TO_DATE()`

Função	Descrição
<code>TO_CHAR(x [, formato])</code>	Converte <i>x</i> em uma string. Também é possível fornecer um <i>formato</i> opcional para <i>x</i> . Você viu como utilizar <code>TO_CHAR()</code> para converter um número em uma string no capítulo anterior. Neste capítulo, você verá como converter uma data/horário em uma string.
<code>TO_DATE(x [, formato])</code>	Converte a string <i>x</i> em um tipo <code>DATE</code> .

Vamos começar examinando como usar `TO_CHAR()` para converter uma data/horário em uma string. Posteriormente, você verá como utilizar `TO_DATE()` para converter uma string em um tipo `DATE`.

### Usando `TO_CHAR()` para converter uma data/horário em uma string

Você pode usar `TO_CHAR(x [, formato])` para converter a data/horário *x* em uma string. Você também pode fornecer um *formato* opcional para *x*. Um exemplo de formato é `MONTH DD, YYYY`, onde:

- `MONTH` é o nome completo do mês em maiúsculas, por exemplo, `JANUARY`
- `DD` é o dia com dois dígitos
- `YYYY` é o ano com quatro dígitos

A consulta a seguir usa `TO_CHAR()` para converter a coluna `dob` da tabela `customers` em uma string com o formato `MONTH DD, YYYY`:

```
SELECT customer_id, TO_CHAR(dob, 'MONTH DD, YYYY')
FROM customers;
```

```
CUSTOMER_ID TO_CHAR(DOB, 'MONTH
-----
1 JANUARY    01, 1965
2 FEBRUARY   05, 1968
3 MARCH      16, 1971
4
5 MAY        20, 1970
```

A próxima consulta obtém a data e hora atuais do banco de dados usando a função `SYSDATE` e, depois, converte a data e hora em uma string usando `TO_CHAR()` com o formato `MONTH DD, YYYY, HH24:MI:SS`. A parte referente à hora desse formato indica que as horas estão no formato de 24 horas e que os minutos e segundos também devem ser incluídos na string.

```
SELECT TO_CHAR(SYSDATE, 'MONTH DD, YYYY, HH24:MI:SS')
FROM dual;
```

```
TO_CHAR(SYSDATE, 'MONTHDD,YYY
-----
NOVEMBER 05, 2007, 12:34:36
```

Quando você usa `TO_CHAR()` para converter uma data/horário em uma string, o formato tem diversos parâmetros que afetam a string retornada. Alguns desses parâmetros estão listados na Tabela 5-2.

**Tabela 5-2** *Parâmetros de formatação de data/horário*

Aspecto	Parâmetro	Descrição	Exemplo
Século	CC	Século com dois dígitos.	21
	SCC	Século com dois dígitos, com um sinal negativo (-) para A.C.	-10
Trimestre	Q	Trimestre do ano com um dígito.	1
Ano	YYYY	Todos os quatro dígitos do ano.	2008
	IYYY	Todos os quatro dígitos do ano ISO.	2008
	RRRR	Todos os quatro dígitos do ano arredondado (governado pelo ano atual). Para ver os detalhes, consulte a seção “Como o Oracle interpreta anos de dois dígitos”, posteriormente neste capítulo.	2008
	SYYYY	Todos os quatro dígitos do ano com um sinal negativo (-) para A.C.	-1001
	Y,YYY	Todos os quatro dígitos do ano, com uma vírgula após o primeiro dígito.	2,008
	YYY	Últimos três dígitos do ano.	008
	IYY	Últimos três dígitos do ano ISO.	008
	YY	Últimos dois dígitos do ano.	08
	IY	Últimos dois dígitos do ano ISO.	06
	RR	Últimos dois dígitos do ano arredondado, que depende do ano atual. Para ver os detalhes, consulte a seção “Como o Oracle interpreta anos de dois dígitos”, posteriormente neste capítulo.	08
	Y	Último dígito do ano.	8
	I	Último dígito do ano ISO.	8
	YEAR	Nome do ano em maiúsculas.	TWO THOUSAND-EIGHT
	Year	Nome do ano com as primeiras letras maiúsculas.	Two Thousand-Eight
Month	MM	Mês do ano com dois dígitos.	01
	MONTH	Nome completo do mês em maiúsculas, preenchido com espaços à direita para um comprimento total de nove caracteres.	JANUARY
	Month	Nome completo do mês com a primeira letra em maiúscula, preenchido com espaços à direita para um comprimento total de nove caracteres.	January
	MON	Três primeiras letras do nome do mês em maiúsculas.	JAN
	Mon	Três primeiras letras do nome do mês com a primeira letra em maiúscula.	Jan

(continua)

**Tabela 5-2** *Parâmetros de formatação de data/horário (continuação)*

Aspecto	Parâmetro	Descrição	Exemplo
	RM	Mês em algarismos romanos.	O quarto mês em algarismos romanos (Abril) é IV.
Semana	WW	Semana do ano com dois dígitos.	02
	IW	Semana do ano ISO com dois dígitos.	02
	W	Semana do mês com um dígito.	2
Dia	DDD	Dia do ano com três dígitos.	103
	DD	Dia do mês com dois dígitos.	31
	D	Dia da semana com um dígito.	5
	DAY	Nome completo do dia em maiúsculas.	SATURDAY
	Day	Nome completo do dia com a primeira letra maiúscula.	Saturday
	DY	Três primeiras letras do nome do dia em maiúsculas.	SAT
	Dy	Três primeiras letras do nome do dia com a primeira letra maiúscula.	Sat
	J	Dia conforme o calendário Juliano — o número de dias decorridos desde 1º de janeiro de 4713 A.C.	2439892
Hora	HH24	Hora com dois dígitos no formato de 24 horas.	23
	HH	Hora com dois dígitos no formato de 12 horas.	11
Minuto	MI	Minuto com dois dígitos.	57
Segundo	SS	Segundo com dois dígitos.	45
	FF [1..9]	Segundos fracionários com um número de dígitos opcional à direita do ponto decimal. Só se aplica a timestamps, sobre os quais você aprenderá na seção "Usando timestamps", posteriormente neste capítulo.	Ao lidar com 0.123456789 segundos, FF3 arredondaria os segundos para 0.123.
	SSSSS	Número de segundos após as 12:00 horas.	46748
	MS	Milissegundo (milésimo de segundo).	100
	CS	Centissegundo (centésimo de segundo).	10
Separadores	- / , . ; : "texto"	Caracteres que permitem separar os aspectos de uma data e hora. Você pode fornecer texto de forma livre entre aspas como separador.	Para a data 13 de dezembro de 1969, DD-MM-YYYY produziria 13-12-1969 e DD/MM/YYYY produziria 13/12/1969.
Sufixos	AM ou PM	AM ou PM, conforme for apropriado.	AM
	A.M. ou P.M.	A.M. ou P.M., conforme for apropriado.	P.M.
	AD ou BC	DC ou AC, conforme for apropriado.	AD

(continua)

Tabela 5-2    Parâmetros de formatação de data/horário (continuação)

Aspecto	Parâmetro	Descrição	Exemplo
	A.D. ou B.C.	d.C. ou a.C., conforme for apropriado.	a.C.
	TH	Sufixo para um número. Você pode fazer o sufixo em maiúsculas especificando o formato numérico em maiúsculas e vice-versa para minúsculas.	Para um dia de número 28, <code>ddTH</code> produziria 28 <sup>th</sup> e <code>DDTH</code> produziria 28TH.
	SP	Número por extenso.	Para um dia de número 28, <code>DDSP</code> produziria TWENTY-EIGHT e <code>ddSP</code> produziria twenty-eight.
	SPTH	Combinação de TH e SP.	Para um dia de número 28, <code>DDSPTH</code> produzi-ria TWENTY-EIGHTH e <code>ddSPTH</code> produziria twenty-eighth.
Era	EE	Nome completo da era para os calendários impe-rial japonês, chinês oficial e budista tailandês.	Sem exemplos
	E	Nome da era abreviado.	Sem exemplos
Fusos horários	TZH	Hora do fuso horário. Você vai aprender sobre fu-sos horários na seção “Usando fusos horários”.	12
	TZM	Minuto do fuso horário.	30
	TZR	Região do fuso horário.	PST
	TZD	Fuso horário com informações de horário de verão.	Sem exemplos

A tabela a seguir mostra exemplos de strings para formatar a data 5 de fevereiro de 1968, junto com a string retornada de uma chamada de `TO_CHAR()`.

String de formato	String retornada
MONTH DD, YYYY	FEBRUARY 05, 1968
MM/DD/YYYY	02/05/1968
MM-DD-YYYY	02-05-1968
DD/MM/YYYY	05/02/1968
DAY MON, YY AD	MONDAY FEB, 68 AD
DDSPTH "of" MONTH, YEAR A.D.	FIFTH of FEBRUARY, NINETEEN SIXTY-EIGHT A.D.
CC, SCC	20, 20
Q	1

String de formato

YYYY, IYYY, RRRR, SYYYY, Y, YYY,  
YYY, IYY, YY, IY, RR, Y, I,  
YEAR,  
Year  
  
MM, MONTH, Month,  
MON, Mon, RM  
  
WW, IW, W  
  
DDD, DD, DAY,  
Day, DY, Dy, J  
  
ddTH, DDTH, ddSP, DDSP, DDSPTH

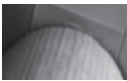
String retornada

1968, 1968, 1968, 1968, 1,968,  
968, 968, 68, 68, 68, 8, 8,  
NINETEEN SIXTY-EIGHT,  
Nineteen Sixty-Eight  
  
02, FEBRUARY, February,  
FEB, Feb, II  
  
06, 06, 1  
  
036, 05, MONDAY,  
Monday, MON, Mon, 2439892  
  
05th, 05TH, five, FIVE, FIFTH

Você pode ver os resultados mostrados nessa tabela chamando `TO_CHAR()` em uma consulta. Por exemplo, a consulta a seguir converte 5 de fevereiro de 1968 em uma string com o formato `MONTH DD, YYYY`:

```
SELECT TO_CHAR(TO_DATE('05-FEB-1968'), 'MONTH DD, YYYY')
FROM dual;

TO_CHAR(TO_DATE('0
-----
FEBRUARY 05, 1968
```



NOTA

A função `TO_DATE()` converte uma string em uma data/horário. Você vai aprender mais sobre a função `TO_DATE()` em breve.

A tabela a seguir mostra exemplos de strings para formatar a hora 19:32:36 (32 minutos e 36 segundos depois das 19:00 horas) — junto com a saída que seria retornada de uma chamada de `TO_CHAR()` com essa hora e essa string de formato.

String de formato	String retornada
HH24:MI:SS	19:32:36
HH.MI.SS AM	7.32.36 PM

Usando `TO_DATE()` para converter uma string em uma data/horário

Você usa `TO_DATE(x [, formato])` para converter a string `x` em uma data/horário. Você pode fornecer uma string de `formato` opcional para indicar o formato de `x`. Se você omitir o `formato`, a data deverá ser no formato padrão do banco de dados (normalmente, `DD-MOM-YYYY` ou `DD-MOM-YY`).



NOTA

O parâmetro de banco de dados `NLS_DATE_FORMAT` especifica o formato de data padrão do banco de dados. Na seção “Configurando o formato de data padrão”, você verá que é possível alterar a configuração de `NLS_DATE_FORMAT`.

A consulta a seguir usa `TO_DATE()` para converter as strings `04-JUL-2007` e `04-JUL-07` na data 4 de julho de 2007; observe que a data final é exibida no formato padrão `DD-MOM-YY`:

```
SELECT TO_DATE('04-JUL-2007'), TO_DATE('04-JUL-07')
FROM dual;

TO_DATE(' TO_DATE('
-----
04-JUL-07 04-JUL-07
```

### Especificando um formato de data/horário

Conforme mencionado anteriormente, você pode fornecer um formato opcional para uma data/horário na função `TO_DATE()`. São usados os mesmos parâmetros de formato definidos anteriormente na Tabela 5-2. A consulta a seguir usa `TO_DATE()` para converter a string `July 4, 2007` em uma data, passando a string de formato `MONTH DD, YYYY` para `TO_DATE()`:

```
SELECT TO_DATE('July 4, 2007', 'MONTH DD, YYYY')
FROM dual;

TO_DATE('
-----
04-JUL-07
```

A próxima consulta passa a string de formato `MM.DD.YY` para `TO_DATE()` e converte a string `7.4.07` na data `July 4, 2007`; novamente, a data final é exibida no formato padrão `DD-MOM-YY`:

```
SELECT TO_DATE('7.4.07', 'MM.DD.YY')
FROM dual;

TO_DATE('
-----
04-JUL-07
```

### Especificando horas

Também é possível especificar uma hora em uma data/horário. Se você não fornecer uma hora em uma data/horário, a parte referente à hora terá como padrão `12:00:00 A.M.` Você pode fornecer o formato para uma hora usando os diversos formatos mostrados anteriormente na Tabela 5-3. Um exemplo de formato de hora é `HH24:MI:SS`, onde:

- `HH24` é uma hora com dois dígitos no formato de 24 horas, de 00 a 23
- `MI` é um minuto com dois dígitos, de 00 a 59
- `SS` é um segundo com dois dígitos, de 00 a 59

Um exemplo de hora que usa o formato `HH24:MI:SS` é `19:32:36`. Um exemplo completo de data/horário que usa essa hora é:

```
05-FEB-1968 19:32:36
```

com o formato dessa data/horário sendo

```
DD-MON-YYYY HH24:MI:SS
```

A chamada de `TO_DATE()` a seguir mostra o uso desse formato e valor de data/horário:

```
TO_DATE('05-FEB-1968 19:32:36', 'DD-MOM-YYYY HH24:MI:SS')
```

A data/horário retornada por `TO_DATE()` no exemplo anterior é usada na instrução `INSERT` a seguir, que adiciona uma linha na tabela `customers`; observe que a coluna `dob` da nova linha é configurada com a data/horário retornada por `TO_DATE()`:

```
INSERT INTO customers (
    customer_id, first_name, last_name,
    dob,
    phone
) VALUES (
    6, 'Fred', 'Brown',
    TO_DATE('05-FEB-1968 19:32:36', 'DD-MON-YYYY HH24:MI:SS'),
    '800-555-1215'
);
```

Você usa `TO_CHAR()` para exibir a parte referente à hora de uma data/horário. Por exemplo, a consulta a seguir recupera as linhas da tabela `customers` e utiliza `TO_CHAR()` para converter os valores da coluna `dob`; observe que o cliente nº 6 tem a hora configurada anteriormente na instrução `INSERT`:

```
SELECT customer_id, TO_CHAR(dob, 'DD-MON-YYYY HH24:MI:SS')
FROM customers;
```

```
CUSTOMER_ID TO_CHAR(DOB, 'DD-MON-
-----
1 01-JAN-1965 00:00:00
2 05-FEB-1968 00:00:00
3 16-MAR-1971 00:00:00
4
5 20-MAY-1970 00:00:00
6 05-FEB-1968 19:32:36
```

Observe que a hora da coluna `dob` dos clientes nº 1, 2, 3 e 5 é configurada como 00:00:00 (12:00 horas). Essa é a hora padrão substituída pelo banco de dados quando você não fornece uma hora em uma data/horário. A instrução a seguir reverte a adição da nova linha:

```
ROLLBACK;
```

## NOTA

*Se você executou a instrução `INSERT` anterior, desfça a alteração usando `ROLLBACK`.*

## Combinando chamadas de `TO_CHAR()` e `TO_DATE()`

Você pode combinar chamadas de `TO_CHAR()` e `TO_DATE()`; isso permite utilizar data/-horários em diferentes formatos. Por exemplo, a consulta a seguir combina `TO_CHAR()` e `TO_DATE()` para exibir apenas a parte referente à hora de uma data/horário; observe que a saída de `TO_DATE()` é passada para `TO_CHAR()`:



```
SELECT TO_CHAR(TO_DATE('05-FEB-1968 19:32:36',
'DD-MON-YYYY HH24:MI:SS'), 'HH24:MI:SS')
FROM dual;

TO_CHAR(
-----
19:32:36
```

## CONFIGURANDO O FORMATO DE DATA PADRÃO

O formato de data padrão é especificado no parâmetro de banco de dados `NLS_DATE_FORMAT`. O administrador do banco de dados pode alterar a configuração de `NLS_DATE_FORMAT` definindo o valor desse parâmetro no arquivo `init.ora` ou `spfile.ora` do banco de dados, os quais são lidos quando o banco de dados é iniciado. Um administrador também pode configurar `NLS_DATE_FORMAT` usando o comando `ALTER SYSTEM`. Você também pode configurar o parâmetro `NLS_DATE_FORMAT` para sua própria sessão usando o SQL\*Plus, o que é feito com o comando `ALTER SESSION`. Por exemplo, a instrução `ALTER SESSION` a seguir configura `NLS_DATE_FORMAT` como `MONTH-DD-YYYY`:

```
ALTER SESSION SET NLS_DATE_FORMAT = 'MONTH-DD-YYYY';

Session altered
```

### NOTA

*Uma sessão é iniciada quando você se conecta a um banco de dados e é terminada quando você se desconecta.*

Veja o uso desse novo formato de data nos resultados da consulta a seguir, que recupera a coluna `dob` do cliente nº 1:

```
SELECT dob
FROM customers
WHERE customer_id = 1;

DOB
-----
JANUARY -01-1965
```

Você também pode usar o novo formato de data ao inserir uma linha no banco de dados. Por exemplo, a instrução `INSERT` a seguir adiciona uma nova linha na tabela `customers`; observe o uso do formato `MONTH-DD-YYYY` ao se fornecer o valor da coluna `dob`:

```
INSERT INTO customers (
    customer_id, first_name, last_name, dob, phone
) VALUES (
    6, 'Fred', 'Brown', 'MARCH-15-1970', '800-555-1215'
);
```

Desconecte-se do banco de dados e conecte-se novamente como o usuário `store`; o formato da data volta para o padrão. Isso porque as alterações feitas usando a instrução `ALTER SESSION` duraram somente para aquela sessão específica — quando você se desconectou, perdeu as alterações.



**NOTA**

Se você executou a instrução `INSERT` anterior, exclua a linha usando `DELETE FROM customers WHERE customer_id = 6`.

## COMO O ORACLE INTERPRETA ANOS DE DOIS DÍGITOS

O banco de dados Oracle armazena os quatro dígitos do ano, mas se você fornecer apenas dois dígitos, o banco interpretará o século de acordo com o formato `YY` ou `RR` usado.

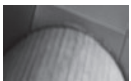


**DICA**

Você sempre deve especificar os quatro dígitos do ano para não se confundir mais tarde.

### Usando o formato `YY`

Se o seu formato de data usa `YY` para o ano e você fornece somente dois dígitos de um ano, será presumido que o século do ano é o atual, configurado no servidor de banco de dados. Portanto, os dois primeiros dígitos do ano fornecidos são configurados como os dois primeiros dígitos do ano atual. Por exemplo, se o ano fornecido for 15 e o ano atual é 2007, o ano fornecido será configurado como 2015; da mesma forma, 75 será configurado como 2075.



**NOTA**

Se você usar o formato `YYYY`, mas fornecer um ano com apenas dois dígitos, seu ano será interpretado usando o formato `YY`.

Vejamos uma consulta que usa o formato `YY` para interpretar os anos 15 e 75. Na consulta a seguir, observe que as datas de entrada 15 e 75 são passadas para `TO_DATE()`, cuja saída é passada para `TO_CHAR()`, que converte as datas em uma string com o formato `DD-MON-YYYY`. (O formato usado é `YYYY`; portanto, você pode ver que todos os quatro dígitos do ano são retornados por `TO_DATE()`.)

```
SELECT
    TO_CHAR(TO_DATE('04-JUL-15', 'DD-MON-YY'), 'DD-MON-YYYY'),
    TO_CHAR(TO_DATE('04-JUL-75', 'DD-MON-YY'), 'DD-MON-YYYY')
FROM dual;

TO_CHAR(TO_ TO_CHAR(TO_
-----
04-JUL-2015 04-JUL-2075
```

Conforme o esperado, os anos 15 e 75 são interpretados como 2015 e 2075.

### Usando o formato `RR`

Se seu formato de data é `RR` e você fornecer os dois últimos dígitos de um ano, os dois primeiros dígitos do ano são determinados usando o ano de dois dígitos fornecido (seu *ano fornecido*) e os

dois últimos dígitos da data atual no servidor de banco de dados (o *ano atual*). As regras usadas para determinar o século de seu ano fornecido são:

- **Regra 1**    Se seu ano fornecido está entre 00 e 49 e o ano atual está entre 00 e 49, o século é igual ao atual. Portanto, *os dois primeiros dígitos de seu ano fornecido são definidos como os dois primeiros dígitos do ano atual*. Por exemplo, se seu ano fornecido é 15 e o ano atual é 2007, seu ano fornecido é definido como 2015.
- **Regra 2**    Se seu ano fornecido está entre 50 e 99 e o ano atual está entre 00 e 49, o século é o atual menos 1. Portanto, *os dois primeiros dígitos de seu ano fornecido são definidos como os dois primeiros dígitos do ano atual menos 1*. Por exemplo, se seu ano fornecido é 75 e o ano atual é 2007, seu ano fornecido é definido como 1975.
- **Regra 3**    Se seu ano fornecido está entre 00 e 49 e o ano atual está entre 50 e 99, o século é o atual mais 1. Portanto, *os dois primeiros dígitos de seu ano fornecido são definidos como os dois primeiros dígitos do ano atual mais 1*. Por exemplo, se seu ano fornecido é 15 e o ano atual é 2075, seu ano fornecido é definido como 2115.
- **Regra 4**    Se seu ano fornecido está entre 50 e 99 e o ano atual está entre 50 e 99, o século é igual ao atual. Portanto, *os dois primeiros dígitos de seu ano fornecido são definidos como os dois primeiros dígitos do ano atual*. Por exemplo, se seu ano fornecido é 55 e o ano atual é 2075, seu ano fornecido é definido como 2055.

A Tabela 5-3 resume esses resultados.



**NOTA**  
*Se você usar o formato RRRR, mas fornecer um ano de apenas dois dígitos, o ano será interpretado pelo formato RR.*

**Tabela 5-3**    *Como os anos de dois dígitos são interpretados*

		Ano fornecido de dois dígitos	
		00–49	50–99
Dois últimos dígitos do ano atual	00–49	Regra 1: os dois primeiros dígitos do ano fornecido são definidos como os dois primeiros dígitos do ano atual.	Regra 2: os dois primeiros dígitos do ano fornecido são definidos como os dois primeiros dígitos do ano atual menos 1.
	50–99	Regra 3: os dois primeiros dígitos do ano fornecido são definidos como os dois primeiros dígitos do ano atual mais 1.	Regra 4: os dois primeiros dígitos do ano fornecido são definidos como os dois primeiros dígitos do ano atual.

Vejamos uma consulta que usa o formato RR para interpretar os anos 15 e 75. (Na consulta a seguir, você deve presumir que o ano atual é 2007.)

```
SELECT
    TO_CHAR(TO_DATE('04-JUL-15', 'DD-MON-RR'), 'DD-MON-YYYY'),
    TO_CHAR(TO_DATE('04-JUL-75', 'DD-MON-RR'), 'DD-MON-YYYY')
FROM dual;

TO_CHAR(TO_ TO_CHAR(TO_
-----
04-JUL-2015 04-JUL-1975
```

A partir das regras 1 e 2, os anos 15 e 75 são interpretados como 2015 e 1975. Na próxima consulta, você deve presumir que o ano atual é 2075.

```
SELECT
    TO_CHAR(TO_DATE('04-JUL-15', 'DD-MON-RR'), 'DD-MON-YYYY'),
    TO_CHAR(TO_DATE('04-JUL-55', 'DD-MON-RR'), 'DD-MON-YYYY')
FROM dual;

TO_CHAR(TO_ TO_CHAR(TO_
-----
04-JUL-2115 04-JUL-2055
```

A partir das regras 3 e 4, os anos 15 e 55 são interpretados como 2115 e 2055.

## USANDO FUNÇÕES DE DATA/HORÁRIO

As funções de data/horário são usadas para obter ou processar data/horários e timestamps (você vai aprender sobre timestamps posteriormente neste capítulo). A Tabela 5-4 mostra algumas das funções de data/horário. Nessa tabela, *x* representa uma data/horário ou um timestamp. Você aprenderá mais sobre as funções mostradas na Tabela 5-4 nas seções a seguir.

### ADD\_MONTHS()

`ADD_MONTHS(x, y)` retorna o resultado da adição de *y* meses a *x*. Se *y* é negativo, *y* meses são subtraídos de *x*. O exemplo a seguir soma 13 meses a 1º de janeiro de 2007:

```
SELECT ADD_MONTHS('01-JAN-2007', 13)
FROM dual;

ADD_MONTH
-----
01-FEB-08
```

O exemplo a seguir subtrai 13 meses de 1º de janeiro de 2008; observe que -13 meses são “somados” a essa data usando `ADD_MONTHS()`:

```
SELECT ADD_MONTHS('01-JAN-2008', -13)
FROM dual;

ADD_MONTH
-----
01-DEC-06
```

Tabela 5-4    Funções de data/horário

Função	Descrição
ADD_MONTHS (x, y)	Retorna o resultado da adição de y meses a x. Se y é negativo, y meses são subtraídos de x.
LAST_DAY (x)	Retorna o último dia da parte de x referente ao mês.
MONTHS_BETWEEN (x, y)	Retorna o número de meses entre x e y. Se x aparece antes de y no calendário, o número retornado é positivo; caso contrário, o número é negativo.
NEXT_DAY (x, dia)	Retorna a data/horário do próximo dia depois de x; dia é especificado como uma string literal (SATURDAY, por exemplo).
ROUND (x [, unidade])	Arredonda x. Por padrão, x é arredondado para o início do dia mais próximo. Você pode fornecer uma string de unidade opcional, que indica a unidade de arredondamento; por exemplo, YYYY arredonda x para o primeiro dia do ano mais próximo.
SYSDATE	Retorna a data/horário atual configurada no sistema operacional do servidor de banco de dados.
TRUNC (x [, unidade])	Trunca x. Por padrão, x é truncado no início do dia. Você pode fornecer uma string de unidade opcional, que indica a unidade de truncamento; por exemplo, MM trunca x no primeiro dia do mês.

Você pode fornecer uma hora e uma data para a função `ADD_MONTHS()`. Por exemplo, a consulta a seguir soma dois meses à data/horário 19:15:26 de 1º de janeiro de 2007:

```
SELECT ADD_MONTHS(TO_DATE('01-JAN-2007 19:15:26',
'DD-MON-YYYY HH24:MI:SS'), 2)
FROM dual;

ADD_MONTH
-----
01-MAR-07
```

A próxima consulta reescreve o exemplo anterior para converter a data/horário retornada de `ADD_MONTHS()` em uma string, usando `TO_CHAR()` com o formato `DD-MON-YYYY HH24:MI:SS`:

```
SELECT TO_CHAR(ADD_MONTHS(TO_DATE('01-JAN-2007 19:15:26',
'DD-MON-YYYY HH24:MI:SS'), 2), 'DD-MON-YYYY HH24:MI:SS')
FROM dual;

TO_CHAR(ADD_MONTHS(T
-----
01-MAR-2007 19:15:26
```

NOTA

É possível fornecer uma data e hora para qualquer uma das funções mostradas anteriormente na Tabela 5-4.

## LAST\_DAY()

LAST\_DAY(*x*) retorna a data do último dia da parte de *x* referente ao mês. O exemplo a seguir exibe a última data em janeiro de 2008:

```
SELECT LAST_DAY('01-JAN-2008')
FROM dual;

LAST_DAY(
-----
31-JAN-08
```

## MONTHS\_BETWEEN()

MONTHS\_BETWEEN(*x*, *y*) retorna o número de meses entre *x* e *y*. Se *x* ocorre antes de *y* no calendário, o número retornado por MONTHS\_BETWEEN() é negativo.

### NOTA

*A ordem das datas em sua chamada da função MONTHS\_BETWEEN() é importante: a data posterior deve aparecer primeiro, caso você queira o resultado como um número positivo.*

O exemplo a seguir exibe o número de meses entre 25 de maio de 2008 e 15 de janeiro de 2008. Note que, como a data posterior (25 de maio de 2008) aparece primeiro, o resultado retornado é um número positivo:

```
SELECT MONTHS_BETWEEN('25-MAY-2008', '15-JAN-2008')
FROM dual;

MONTHS_BETWEEN('25-MAY-2008', '15-JAN-2008')
-----
4.32258065
```

O exemplo a seguir inverte as mesmas datas na chamada da função MONTHS\_BETWEEN() e, portanto, o resultado retornado é um número negativo de meses:

```
SELECT MONTHS_BETWEEN('15-JAN-2008', '25-MAY-2008')
FROM dual;

MONTHS_BETWEEN('15-JAN-2008', '25-MAY-2008')
-----
-4.3225806
```

## NEXT\_DAY()

NEXT\_DAY(*x*, *dia*) retorna a data do próximo dia depois de *x*; você especifica *dia* como uma string literal (SATURDAY, por exemplo). O exemplo a seguir exibe a data do próximo sábado, após 1º de janeiro de 2008:

```
SELECT NEXT_DAY('01-JAN-2008', 'SATURDAY')
FROM dual;
```

```
NEXT_DAY (
-----
05-JAN-08
```

## ROUND()

`ROUND(x [, unidade])` arredonda *x*, por padrão, para o início do dia mais próximo. Se você fornecer uma string *unidade* opcional, *x* será arredondado para essa unidade; por exemplo, `YYYY` arredonda *x* para o primeiro dia do ano mais próximo. É possível usar muitos dos parâmetros mostrados anteriormente na Tabela 5-2 para arredondar uma data/horário.

O exemplo a seguir usa `ROUND()` para arredondar 25 de outubro de 2008 para o primeiro dia do ano mais próximo, que é 1º de janeiro de 2009. Note que a data é especificada como `25-OCT-2008` e está contida em uma chamada para a função `TO_DATE()`:

```
SELECT ROUND(TO_DATE('25-OCT-2008'), 'YYYY')
FROM dual;

ROUND(TO_
-----
01-JAN-09
```

O exemplo a seguir arredonda 25 de maio de 2008 para o primeiro dia do mês mais próximo, que é 1º de junho de 2008, pois 25 de maio está mais próximo do início de junho do que do início de maio:

```
SELECT ROUND(TO_DATE('25-MAY-2008'), 'MM')
FROM dual;

ROUND(TO_
-----
01-JUN-08
```

O exemplo a seguir arredonda 19:45:26 horas de 25 de maio de 2008 para a hora mais próxima, que é 20:00 horas:

```
SELECT TO_CHAR(ROUND(TO_DATE('25-MAY-2008 19:45:26',
'DD-MON-YYYY HH24:MI:SS'), 'HH24'), 'DD-MON-YYYY HH24:MI:SS')
FROM dual;

TO_CHAR(ROUND(TO_DAT
-----
25-MAY-2008 20:00:00
```

## SYSDATE

`SYSDATE` retorna a data/horário atual configurada no sistema operacional do servidor de banco de dados. O exemplo a seguir obtém a data atual:

```
SELECT SYSDATE
FROM dual;

SYSDATE
-----
05-NOV-07
```

## TRUNC()

`TRUNC(x [, unidade])` trunca *x*. Por padrão, *x* é truncado no início do dia. Se você fornecer uma string *unidade* opcional, *x* será truncado nessa unidade; por exemplo, MM trunca *x* no primeiro dia do mês. Você pode usar muitos dos parâmetros mostrados anteriormente na Tabela 5-2 para truncar uma data/horário. O exemplo a seguir usa `TRUNC()` para truncar 25 de maio de 2008 no primeiro dia do ano, que é 1º de janeiro de 2008:

```
SELECT TRUNC(TO_DATE('25-MAY-2008'), 'YYYY')
FROM dual;

TRUNC(TO_
-----
01-JAN-08
```

O exemplo a seguir trunca 25 de maio de 2008 no primeiro dia do mês, que é 1º de maio de 2008:

```
SELECT TRUNC(TO_DATE('25-MAY-2008'), 'MM')
FROM dual;

TRUNC(TO_
-----
01-MAY-08
```

O exemplo a seguir trunca a hora 19:45:26 de 25 de maio de 2008 na hora, que se torna 19:00 horas:

```
SELECT TO_CHAR(TRUNC(TO_DATE('25-MAY-2008 19:45:26',
'DD-MON-YYYY HH24:MI:SS'), 'HH24'), 'DD-MON-YYYY HH24:MI:SS')
FROM dual;

TO_CHAR(TRUNC(TO_DAT
-----
25-MAY-2008 19:00:00
```

## USANDO FUSOS HORÁRIOS

O Oracle Database 9i introduziu a capacidade de usar diferentes fusos horários. Um fuso horário é uma diferença em relação à hora de Greenwich, Inglaterra. A hora de Greenwich era conhecida como horário médio de Greenwich (GMT), mas agora é conhecida como Tempo Universal Coordenado (UTC, que vem das iniciais da expressão francesa).

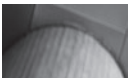
Você especifica um fuso horário usando uma diferença em relação ao UTC ou uma região geográfica (por exemplo, PST — hora padrão no Pacífico). Para especificar uma diferença, use HH:MI prefixado com um sinal de mais ou de menos:

```
+ | -HH:MI
```

onde

- + ou - indica um aumento ou uma diminuição da diferença em relação ao UTC
- HH:MI especifica a diferença em horas e minutos do fuso horário





**NOTA**

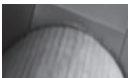
A hora e minuto do fuso horário usam os parâmetros de formato TZH e TZR, mostrados na Tabela 5-2.

O exemplo a seguir mostra diferenças de 8 horas antes de UTC e de 2 horas e 15 minutos depois de UTC:

-08:00

+02:15

Você também pode especificar um fuso horário usando a região geográfica. Por exemplo, PST indica Pacific Standard Time (hora padrão no Pacífico), que é 8 horas antes de UTC. EST indica Eastern Standard Time (hora da costa leste dos EUA), que é 5 horas antes de UTC.



**NOTA**

A região do fuso horário usa o parâmetro de formato TZR, mostrado na Tabela 5-2.

**Funções de fuso horário**

Existem várias funções relacionadas aos fusos horários; elas estão mostradas na Tabela 5-5. As seções a seguir abordam essas funções.

**O fuso horário do banco de dados e o fuso horário da sessão**

Se você trabalha em uma grande empresa mundial, o banco de dados que acessa pode estar localizado em um fuso horário diferente do seu. O fuso horário do banco de dados é conhecido como *fuso horário do banco de dados* e o fuso horário definido para sua sessão no banco de dados é conhecido como *fuso horário da sessão*.

**Tabela 5-5**    Funções de fuso horário

Função	Descrição
CURRENT_DATE	Retorna a data atual no fuso horário local definido para a sessão do banco de dados
DBTIMEZONE	Retorna o fuso horário do banco de dados
NEW_TIME( x, fuso_horário1, fuso_horário2 )	Converte x do <i>fuso_horário1</i> para o <i>fuso_horário2</i> e retorna a nova data/horário
SESSIONTIMEZONE	Retorna o fuso horário da sessão do banco de dados
TZ_OFFSET(fuso_horário)	Retorna a diferença do <i>fuso_horário</i> em horas e minutos

## O fuso horário do banco de dados

O fuso horário do banco de dados é controlado usando o parâmetro de banco de dados `TIME_ZONE`. O administrador do banco de dados pode alterar a configuração do parâmetro `TIME_ZONE` no arquivo `init.ora` ou `spfile.ora` do banco de dados ou usando `ALTER DATABASE SET TIME_ZONE = diferença | região` (por exemplo, `ALTER DATABASE SET TIME_ZONE = '-8:00'` ou `ALTER DATABASE SET TIME_ZONE = 'PST'`). Você pode obter o fuso horário do banco de dados usando a função `DBTIMEZONE`. Por exemplo, a consulta a seguir obtém o fuso horário do meu banco de dados:

```
SELECT DBTIMEZONE
FROM dual;
```

```
DBTIME
-----
+00:00
```

É retornado `+00:00`. Isso significa que meu banco de dados utiliza o fuso horário definido no sistema operacional, que está configurado como PST em meu computador.

### NOTA

*O sistema operacional Windows é normalmente configurado para ajustar o relógio para horários de verão. Na Califórnia, isso significa que, no verão, o relógio está apenas 7 horas antes de UTC, em vez de 8 horas. Quando escrevi este capítulo, configurei a data como 5 de novembro de 2007, o que significa que meu relógio estava 8 horas atrasado em relação a UTC (estou na Califórnia).*

## O fuso horário da sessão

O fuso horário da sessão é o fuso horário de uma sessão específica. Por padrão, o fuso horário da sessão é igual ao fuso horário do sistema operacional. Você pode alterar isso usando a instrução `ALTER SESSION` para configurar o parâmetro de sessão `TIME_ZONE` (por exemplo, `ALTER SESSION SET TIME_ZONE = 'PST'` configura o fuso horário local com a hora padrão no Pacífico). Você também pode configurar o parâmetro de sessão `TIME_ZONE` como `LOCAL`, o que define o fuso horário como aquele usado pelo sistema operacional do computador em que a instrução `ALTER SESSION` foi executada. Você também pode configurar o padrão de sessão `TIME_ZONE` como `DBTIMEZONE`, o que define o fuso horário como aquele usado pelo banco de dados.

É possível obter o fuso horário da sessão usando a função `SESSIONTIMEZONE`. Por exemplo, a consulta a seguir obtém o fuso horário de minha sessão:

```
SELECT SESSIONTIMEZONE
FROM dual;
```

```
SESSIONTIMEZONE
-----
-08:00
```

Meu fuso horário de sessão está 8 horas antes de UTC.

## Obtendo a data atual no fuso horário da sessão

A função `SYSDATE` obtém a data do banco de dados. Isso fornece a data no fuso horário do banco de dados. Você pode obter a data em seu fuso horário de sessão usando a função `CURRENT_DATE`. Por exemplo:

```
SELECT CURRENT_DATE
FROM dual;
```

```
CURRENT_D
-----
05-NOV-07
```

## Obtendo diferenças de fuso horário

Você pode obter as horas de diferença de fuso horário usando a função `TZ_OFFSET()`, passando o nome da região do fuso horário para `TZ_OFFSET()`. Por exemplo, a consulta a seguir usa `TZ_OFFSET()` para obter as horas de diferença de fuso horário para PST, que é 8 horas antes de UTC:

```
SELECT TZ_OFFSET('PST')
FROM dual;
```

```
TZ_OFFS
-----
-08:00
```

### NOTA

*No verão, isso seria -7:00 em um computador com Windows, que configura o relógio para ajustar horários de verão automaticamente.*

## Obtendo nomes de fuso horário

Você pode obter todos os nomes de fuso horário selecionando todas as linhas de `v$timezone_names`. Para consultar `v$timezone_names`, você primeiro deve conectar-se no banco de dados como o usuário `system`. A consulta a seguir mostra as primeiras cinco linhas de `v$timezone_names`:

```
SELECT *
FROM v$timezone_names
WHERE ROWNUM <= 5
ORDER BY tzabbrev;
```

TZNAME	TZABBREV
-----	-----
Africa/Algiers	CET
Africa/Algiers	LMT
Africa/Algiers	PMT
Africa/Algiers	WEST
Africa/Algiers	WET

Você pode usar qualquer um dos valores de `tzname` ou `tzabbrev` para sua configuração de fuso horário.

### NOTA

*A pseudocoluna `ROWNUM` contém o número da linha. Por exemplo, a primeira linha retornada por uma consulta tem o número de linha 1, a segunda tem o número de linha 2 e assim por diante. Portanto, a cláusula `WHERE` na consulta anterior faz a consulta retornar as cinco primeiras linhas.*

## Convertendo uma data/horário de um fuso horário para outro

Você usa a função `NEW_TIME()` para converter uma data/horário de um fuso horário para outro. Por exemplo, a consulta a seguir usa `NEW_TIME()` para converter 19:45 horas de 13 de maio de 2008, de PST para EST:

```
SELECT TO_CHAR(NEW_TIME(TO_DATE('25-MAY-2008 19:45',
                                'DD-MON-YYYY HH24:MI'), 'PST', 'EST'), 'DD-MON-YYYY HH24:MI')
FROM dual;

TO_CHAR(NEW_TIME(
-----
25-MAY-2008 22:45
```

EST está 3 horas na frente de PST; portanto, 3 horas são somadas a 19:45 horas para fornecer 22:45 horas no formato de 24 horas.

## USANDO TIMESTAMP

O Oracle Database 9i introduziu a capacidade de armazenar timestamps. Um timestamp armazena o século, todos os quatro dígitos de um ano, o mês, o dia, a hora (no formato de 24 horas), o minuto e o segundo. As vantagens de um timestamp em relação a um tipo `DATE` são:

- Um timestamp pode armazenar frações de segundo
- Um timestamp pode armazenar um fuso horário

Vamos examinar os tipos timestamp.

## Usando os tipos de timestamp

Existem três tipos de timestamp, que estão mostrados na Tabela 5-6.

**Tabela 5-6** *Tipos de timestamp*

Tipo	Descrição
<code>TIMESTAMP [</code> <code>(precisão_segundos)</code> <code>]</code>	Armazena o século, todos os quatro dígitos de um ano, o mês, o dia, a hora (no formato de 24 horas), o minuto e o segundo. Você pode especificar uma precisão opcional para os segundos, usando <i>precisão_segundos</i> , que pode ser um valor inteiro de 0 a 9; o padrão é 9, que significa que você pode armazenar até 9 dígitos à direita do ponto decimal do segundo. Se tentar adicionar uma linha com mais dígitos no segundo fracionário do que <code>TIMESTAMP</code> pode armazenar, então seu valor fracionário será arredondado.
<code>TIMESTAMP [</code> <code>(precisão_segundos)</code> <code>] WITH TIME ZONE</code>	Amplia <code>TIMESTAMP</code> para armazenar um fuso horário.
<code>TIMESTAMP [</code> <code>(precisão_segundos)</code> <code>] WITH LOCAL TIME ZONE</code>	Amplia <code>TIMESTAMP</code> para converter uma data/horário fornecida para o fuso horário local definido no banco de dados. O processo de conversão é conhecido como <i>normalização</i> da data/horário.

Você aprenderá a usar esses tipos de timestamp nas seções a seguir.

### Usando o tipo **TIMESTAMP**

Assim como nos outros tipos, você pode usar o tipo **TIMESTAMP** para definir uma coluna em uma tabela. A instrução a seguir cria uma tabela chamada `purchases_with_timestamp` que armazena as compras do cliente. Essa tabela contém uma coluna **TIMESTAMP** chamada `made_on` para registrar quando uma compra foi feita; observe que uma precisão 4 é definida para **TIMESTAMP** (isso significa que até quatro dígitos podem ser armazenados à direita do ponto decimal do segundo):

```
CREATE TABLE purchases_with_timestamp (  
    product_id INTEGER REFERENCES products(product_id),  
    customer_id INTEGER REFERENCES customers(customer_id),  
    made_on TIMESTAMP(4)  
);
```

#### NOTA

A tabela `purchases_with_timestamp` é criada e preenchida com linhas pelo script `store_schema.sql`. No restante deste capítulo, você verá outras tabelas que também são criadas pelo script; portanto, não é preciso digitar a instrução `CREATE TABLE` nem qualquer das instruções `INSERT` mostradas neste capítulo.

Para fornecer um valor literal de **TIMESTAMP** para o banco de dados, use a palavra-chave **TIMESTAMP** junto com uma data/horário, no seguinte formato:

```
TIMESTAMP 'YYYY-MM-DD HH24:MI:SS.SSSSSSSS'
```

Observe que existem nove caracteres `S` depois do ponto decimal, ou seja, você pode fornecer até nove dígitos para o segundo fracionário em sua string literal. O número de dígitos que podem realmente ser armazenados em uma coluna **TIMESTAMP** depende de quantos dígitos foram configurados para armazenamento de segundos fracionários quando a coluna foi definida. Por exemplo, até quatro dígitos podem ser armazenados na coluna `made_on` da tabela `purchases_with_timestamp`. Se você tentar adicionar uma linha com mais de quatro dígitos de segundo fracionário, o valor fracionário será arredondado. Por exemplo:

```
2005-05-13 07:15:31.123456789
```

seria arredondado para

```
2005-05-13 07:15:31.1235
```

A instrução `INSERT` a seguir adiciona uma linha na tabela `purchases_with_timestamp`; observe o uso da palavra-chave **TIMESTAMP** para fornecer uma literal de data/horário:

```
INSERT INTO purchases_with_timestamp (  
    product_id, customer_id, made_on  
) VALUES (  
    1, 1, TIMESTAMP '2005-05-13 07:15:31.1234'  
);
```

A consulta a seguir recupera a linha:

```
SELECT *
FROM purchases_with_timestamp;

PRODUCT_ID CUSTOMER_ID MADE_ON
-----
1 1 13-MAY-05 07.15.31.1234 AM
```

### Usando o tipo **TIMESTAMP WITH TIME ZONE**

O tipo **TIMESTAMP WITH TIME ZONE** amplia **TIMESTAMP** para armazenar um fuso horário. A instrução a seguir cria uma tabela chamada `purchases_timezone_with_tz` que armazena as compras do cliente; essa tabela contém uma coluna **TIMESTAMP WITH TIME ZONE** chamada `made_on` para registrar quando uma compra foi feita:

```
CREATE TABLE purchases_timezone_with_tz (
    product_id INTEGER REFERENCES products(product_id),
    customer_id INTEGER REFERENCES customers(customer_id),
    made_on TIMESTAMP(4) WITH TIME ZONE
);
```

Para fornecer uma literal de timestamp com um fuso horário para o banco de dados, basta adicionar o fuso horário na instrução **TIMESTAMP**. Por exemplo, a instrução **TIMESTAMP** a seguir inclui uma diferença de fuso horário de `-07:00`:

```
TIMESTAMP '2005-05-13 07:15:31.1234 -07:00'
```

Você também pode fornecer uma região de fuso horário, como mostrado no exemplo a seguir, que especifica `PST` como fuso horário:

```
TIMESTAMP '2005-05-13 07:15:31.1234 PST'
```

O exemplo a seguir adiciona duas linhas na tabela `purchases_timezone_with_tz`:

```
INSERT INTO purchases_timezone_with_tz (
    product_id, customer_id, made_on
) VALUES (
    1, 1, TIMESTAMP '2005-05-13 07:15:31.1234 -07:00'
);

INSERT INTO purchases_timezone_with_tz (
    product_id, customer_id, made_on
) VALUES (
    1, 2, TIMESTAMP '2005-05-13 07:15:31.1234 PST'
);
```

A consulta a seguir recupera as linhas:

```
SELECT *
FROM purchases_timezone_with_tz;
```

PRODUCT_ID	CUSTOMER_ID	MADE_ON
1	1	13-MAY-05 07.15.31.1234 AM -07:00
1	2	13-MAY-05 07.15.31.1234 AM PST

### Usando o tipo **TIMESTAMP WITH LOCAL TIME ZONE**

O tipo **TIMESTAMP WITH LOCAL TIME ZONE** amplia **TIMESTAMP** para armazenar um timestamp com o fuso horário local definido para seu banco de dados. Quando você fornece um timestamp para armazenamento em uma coluna **TIMESTAMP WITH LOCAL TIME ZONE**, ele é convertido — ou *normalizado* — no fuso horário definido para o banco de dados. Quando você recupera o timestamp, ele é normalizado para o fuso horário definido para sua sessão.



#### DICA

Você deve usar **TIMESTAMP WITH LOCAL TIME ZONE** para armazenar timestamps quando sua organização tiver um sistema global acessado em todo o mundo. Isso se deve ao fato de que o **TIMESTAMP WITH LOCAL TIME ZONE** armazena um timestamp usando a hora local onde o banco de dados está localizado, mas os usuários vêem o timestamp normalizado em seus próprios fusos horários.

Meu fuso horário do banco de dados é PST (PST é 8 horas antes de UTC) e quero armazenar o seguinte timestamp em meu banco de dados:

```
2005-05-13 07:15:30 EST
```

EST é 5 horas antes de UTC e a diferença entre PST e EST é de 3 horas ( $8 - 5 = 3$ ). Portanto, para normalizar o timestamp anterior para PST, 3 horas devem ser subtraídas para fornecer o seguinte timestamp normalizado:

```
2005-05-13 04:15:30
```

Esse é o timestamp que seria armazenado em uma coluna **TIMESTAMP WITH LOCAL TIME ZONE** em meu banco de dados.

A instrução a seguir cria uma tabela chamada `purchases_with_local_tz` que armazena as compras do cliente; essa tabela contém uma coluna **TIMESTAMP WITH LOCAL TIME ZONE** chamada `made_on` para registrar quando uma compra foi feita:

```
CREATE TABLE purchases_with_local_tz (
  product_id INTEGER REFERENCES products(product_id),
  customer_id INTEGER REFERENCES customers(customer_id),
  made_on TIMESTAMP(4) WITH LOCAL TIME ZONE
);
```

A instrução **INSERT** a seguir adiciona uma linha na tabela `purchases_with_local_tz` com a coluna `made_on` definida como `2005-05-13 07:15:30 EST`:

```
INSERT INTO purchases_with_local_tz (
  product_id, customer_id, made_on
) VALUES (
  1, 1, TIMESTAMP '2005-05-13 07:15:30 EST'
);
```

Embora o timestamp da coluna `made_on` seja configurado como 2005-05-13 07:15:30 EST, o timestamp realmente armazenado em meu banco de dados é 2005-05-13 04:15:30 (o timestamp normalizado para PST). A consulta a seguir recupera a linha:

```
SELECT *
FROM purchases_with_local_tz;

PRODUCT_ID CUSTOMER_ID MADE_ON
-----
1 1 13-MAY-05 04.15.30.0000 AM
```

Como meu fuso horário de banco de dados e meu fuso horário de sessão são ambos PST, o timestamp retornado pela consulta é para PST.

### CUIDADO

*O timestamp retornado pela consulta anterior é normalizado para PST. Se seu fuso horário de banco de dados ou fuso horário de sessão não for PST, o timestamp retornado quando você executar a consulta será diferente (será normalizado para seu fuso horário).*

Se eu configurar o fuso horário local para minha sessão como EST e repetir a consulta anterior, obterei o timestamp normalizado para EST:

```
ALTER SESSION SET TIME_ZONE = 'EST';

Session altered.

SELECT *
FROM purchases_with_local_tz;

PRODUCT_ID CUSTOMER_ID MADE_ON
-----
1 1 13-MAY-05 07.15.30.0000 AM
```

O timestamp retornado pela consulta é 13-MAY-05 07.15.30.0000 AM, que é o timestamp normalizado para o fuso horário de sessão de EST. Como EST está três horas adiante de PST, três horas devem ser adicionadas a 13-MAY-05 04:15:30 (o timestamp armazenado no banco de dados) para fornecer 13-MAY-05 07.15.30 AM (o timestamp retornado pela consulta). A instrução a seguir configura o fuso horário de sessão de volta para PST:

```
ALTER SESSION SET TIME_ZONE = 'PST';

Session altered.
```

## Funções de timestamp

Existem várias funções que permitem obter e processar timestamps. Elas estão mostradas na Tabela 5-7. Você vai aprender mais sobre as funções mostradas na Tabela 5-7 nas seções a seguir.



Tabela 5-7    Funções de timestamp

Função	Descrição
CURRENT_TIMESTAMP	Retorna um valor <code>TIMESTAMP WITH TIME ZONE</code> contendo a data e hora atuais da sessão, mais o fuso horário da sessão.
EXTRACT( { YEAR   MONTH   DAY   HOUR   MINUTE   SECOND }   { TIMEZONE_HOUR   TIMEZONE_MINUTE }   { TIMEZONE_REGION   } TIMEZONE_ABBR } FROM x)	Extraí e retorna o ano, mês, dia, hora, minuto, segundo ou fuso horário de <i>x</i> ; <i>x</i> pode ser um timestamp ou um tipo <code>DATE</code> .
FROM_TZ(x, fuso_horário)	Converte o <code>TIMESTAMP x</code> para o fuso horário especificado por <i>fuso_horário</i> e retorna um valor <code>TIMESTAMP WITH TIMEZONE</code> ; <i>fuso_horário</i> deve ser especificado como uma string da forma <code>+ - HH:MI</code> . Basicamente, a função mescla <i>x</i> e <i>fuso_horário</i> em um só valor.
LOCALTIMESTAMP	Retorna um valor <code>TIMESTAMP</code> contendo a data e hora atuais da sessão.
SYSTIMESTAMP	Retorna um valor <code>TIMESTAMP WITH TIME ZONE</code> contendo a data e hora atuais do banco de dados, mais o fuso horário do banco de dados.
SYS_EXTRACT_UTC(x)	Converte o <code>TIMESTAMP WITH TIMEZONE x</code> em um valor <code>TIME- STAMP</code> contendo a data e hora em UTC.
TO_TIMESTAMP(x, [formato])	Converte a string <i>x</i> em um valor <code>TIMESTAMP</code> . Você também pode especificar um <i>formato</i> opcional para <i>x</i> .
TO_TIMESTAMP_TZ(x, [formato])	Converte a string <i>x</i> em um valor <code>TIMESTAMP WITH TIMEZONE</code> . Você também pode especificar um <i>formato</i> opcional para <i>x</i> .

**CURRENT\_TIMESTAMP, LOCALTIMESTAMP e SYSTIMESTAMP**

A consulta a seguir chama as funções `CURRENT_TIMESTAMP`, `LOCALTIMESTAMP` e `SYSTIMESTAMP` (meu fuso horário de sessão e de banco de dados são ambos PST, que é 8 horas antes de UTC):

```
SELECT CURRENT_TIMESTAMP, LOCALTIMESTAMP, SYSTIMESTAMP
FROM dual;

CURRENT_TIMESTAMP
-----
LOCALTIMESTAMP
-----
SYSTIMESTAMP
-----
05-NOV-07 12.15.32.734000 PM PST
05-NOV-07 12.15.32.734000 PM
05-NOV-07 12.15.32.734000 PM -08:00
```

Se alterarmos o valor de `TIME_ZONE` de sessão para EST e repetirmos a consulta anterior, iremos obter os seguintes resultados:

```
ALTER SESSION SET TIME_ZONE = 'EST';
```

Session altered.

```
SELECT CURRENT_TIMESTAMP, LOCALTIMESTAMP, SYSTIMESTAMP
FROM dual;
```

CURRENT\_TIMESTAMP

LOCALTIMESTAMP

SYSTIMESTAMP

```
05-NOV-07 03.19.57.562000 PM EST
05-NOV-07 03.19.57.562000 PM
05-NOV-07 12.19.57.562000 PM -08:00
```

A instrução a seguir configura o fuso horário de sessão de volta para PST:

```
ALTER SESSION SET TIME_ZONE = 'PST';
```

Session altered.

### EXTRACT()

`EXTRACT()` extrai e retorna o ano, mês, dia, hora, minuto, segundo ou fuso horário de *x*; *x* pode ser um timestamp ou um tipo DATE. A consulta a seguir usa `EXTRACT()` para obter o ano, mês e dia de um tipo DATE retornado por `TO_DATE()`:

```
SELECT
  EXTRACT(YEAR FROM TO_DATE('01-JAN-2008 19:15:26',
    'DD-MON-YYYY HH24:MI:SS')) AS YEAR,
  EXTRACT(MONTH FROM TO_DATE('01-JAN-2008 19:15:26',
    'DD-MON-YYYY HH24:MI:SS')) AS MONTH,
  EXTRACT(DAY FROM TO_DATE('01-JAN-2008 19:15:26',
    'DD-MON-YYYY HH24:MI:SS')) AS DAY
FROM dual;
```

YEAR	MONTH	DAY
2008	1	1

A próxima consulta usa `EXTRACT()` para obter a hora, minuto e segundo de um valor `TIME-STAMP` retornado por `TO_TIMESTAMP()`:

```
SELECT
  EXTRACT(HOUR FROM TO_TIMESTAMP('01-JAN-2008 19:15:26',
    'DD-MON-YYYY HH24:MI:SS')) AS HOUR,
  EXTRACT(MINUTE FROM TO_TIMESTAMP('01-JAN-2008 19:15:26',
    'DD-MON-YYYY HH24:MI:SS')) AS MINUTE,
  EXTRACT(SECOND FROM TO_TIMESTAMP('01-JAN-2008 19:15:26',
    'DD-MON-YYYY HH24:MI:SS')) AS SECOND
```

```
FROM dual;
      HOUR      MINUTE      SECOND
-----
      19        15         26
```

A consulta a seguir usa `EXTRACT()` para obter a hora, minuto, segundo, região e abreviatura da região do fuso horário de um valor `TIMESTAMP WITH TIMEZONE` retornado por `TO_TIMESTAMP_TZ()`:

```
SELECT
  EXTRACT(TIMEZONE_HOUR FROM TO_TIMESTAMP_TZ(
    '01-JAN-2008 19:15:26 -7:15', 'DD-MON-YYYY HH24:MI:SS TZH:TZM'))
  AS TZH,
  EXTRACT(TIMEZONE_MINUTE FROM TO_TIMESTAMP_TZ(
    '01-JAN-2008 19:15:26 -7:15', 'DD-MON-YYYY HH24:MI:SS TZH:TZM'))
  AS TZM,
  EXTRACT(TIMEZONE_REGION FROM TO_TIMESTAMP_TZ(
    '01-JAN-2008 19:15:26 PST', 'DD-MON-YYYY HH24:MI:SS TZR'))
  AS TZR,
  EXTRACT(TIMEZONE_ABBR FROM TO_TIMESTAMP_TZ(
    '01-JAN-2008 19:15:26 PST', 'DD-MON-YYYY HH24:MI:SS TZR'))
  AS TZA
FROM dual;

      TZH      TZM      TZR      TZA
-----
      -7      -15      PST      PST
```

### FROM\_TZ()

`FROM_TZ(x, fuso_horário)` converte o `TIMESTAMP x` no fuso horário especificado por `fuso_horário` e retorna um valor `TIMESTAMP WITH TIMEZONE`; `fuso_horário` deve ser especificado como uma string da forma `+|- HH:MI`. Basicamente, a função mescla `x` e `fuso_horário` em um só valor. Por exemplo, a consulta a seguir mescla o timestamp `2008-05-13 07:15:31.1234` e a diferença de fuso horário de `-7:00` em relação a UTC:

```
SELECT FROM_TZ(TIMESTAMP '2008-05-13 07:15:31.1234', '-7:00')
FROM dual;

FROM_TZ(TIMESTAMP'2008-05-1307:15:31.1234','-7:00')
-----
13-MAY-08 07.15.31.1234000000 AM -07:00
```

### SYS\_EXTRACT\_UTC()

`SYS_EXTRACT_UTC(x)` converte o `TIMESTAMP WITH TIMEZONE x` em um valor `TIMESTAMP` contendo a data e hora em UTC. A consulta a seguir converte `2008-11-17 19:15:26 PST` para UTC:

```
SELECT SYS_EXTRACT_UTC(TIMESTAMP '2008-11-17 19:15:26 PST')
FROM dual;

SYS_EXTRACT_UTC(TIMESTAMP'2008-11-1719:15:26PST')
-----
18-NOV-08 03.15.26.0000000000 AM
```

Como PST é 8 horas antes de UTC no inverno, a consulta retorna um valor de `TIMESTAMP` de 8 horas adiante de 2008-11-17 19:15:26 PST, que é 18-NOV-08 03.15.26 AM. Para uma data no verão, o valor de `TIMESTAMP` retornado é de apenas 7 horas adiante de UTC:

```
SELECT SYS_EXTRACT_UTC(TIMESTAMP '2008-05-17 19:15:26 PST')
FROM dual;
```

```
SYS_EXTRACT_UTC(TIMESTAMP'2008-05-1719:15:26PST')
-----
18-MAY-08 02.15.26.000000000 AM
```

### ***TO\_TIMESTAMP()***

`TO_TIMESTAMP(x, formato)` converte a string *x* (que pode ser `CHAR`, `VARCHAR2`, `NCHAR` ou `NVARCHAR2`) em um valor `TIMESTAMP`. Você também pode especificar um *formato* opcional para *x*. A consulta a seguir converte a string 2005-05-13 07:15:31.1234 com o formato `YYYY-MM-DD HH24:MI:SS.FF` em um valor `TIMESTAMP`:

```
SELECT TO_TIMESTAMP('2008-05-13 07:15:31.1234', 'YYYY-MM-DD HH24:MI:SS.FF')
FROM dual;
```

```
TO_TIMESTAMP('2008-05-1307:15:31.1234', 'YYYY-MM-DDHH24:MI:SS.FF')
-----
13-MAY-08 07.15.31.123400000 AM
```

### ***TO\_TIMESTAMP\_TZ()***

`TO_TIMESTAMP_TZ(x, [formato])` converte *x* em um valor `TIMESTAMP WITH TIMEZONE`. Você pode especificar um *formato* opcional para *x*. A consulta a seguir passa o fuso horário PST (identificado com `TZR` na string de formato) para `TO_TIMESTAMP_TZ()`:

```
SELECT TO_TIMESTAMP_TZ('2008-05-13 07:15:31.1234 PST',
'YYYY-MM-DD HH24:MI:SS.FF TZR')
FROM dual;
```

```
TO_TIMESTAMP_TZ('2008-05-1307:15:31.1234PST', 'YYYY-MM-DDHH24:MI:SS.FFTZR')
-----
13-MAY-08 07.15.31.123400000 AM PST
```

A próxima consulta usa uma diferença de fuso horário de -7:00 em relação a UTC (-7:00 é identificado com `TZR` e `TZM` na string de formato):

```
SELECT TO_TIMESTAMP_TZ('2008-05-13 07:15:31.1234 -7:00',
'YYYY-MM-DD HH24:MI:SS.FF TZR:TZM')
FROM dual;
```

```
TO_TIMESTAMP_TZ('2008-05-1307:15:31.1234-7:00', 'YYYY-MM-DDHH24:MI:SS.FFTZH')
-----
13-MAY-08 07.15.31.123400000 AM -07:00
```

### Convertendo uma string em um valor **TIMESTAMP WITH LOCAL TIME ZONE**

É possível usar a função `CAST()` para converter uma string em um valor `TIMESTAMP WITH LOCAL TIME ZONE`. A função `CAST()` foi apresentada no capítulo anterior. Como lembrete, `CAST(x AS tipo)` converte `x` em um tipo de banco de dados compatível, especificado por `tipo`. A consulta a seguir usa `CAST()` para converter a string `13-JUN-08` em um valor `TIMESTAMP WITH LOCAL TIME ZONE`:

```
SELECT CAST('13-JUN-08' AS TIMESTAMP WITH LOCAL TIME ZONE)
FROM dual;
```

```
CAST('13-JUN-08' AS TIMESTAMP WITH LOCAL TIME ZONE)
-----
```

```
13-JUN-08 12.00.00.000000 AM
```

O timestamp retornado por essa consulta contém a data 13 de junho de 2008 e a hora 12:00. A próxima consulta usa `CAST()` para converter uma string mais complexa em um valor `TIMESTAMP WITH LOCAL TIME ZONE`:

```
SELECT CAST(TO_TIMESTAMP_TZ('2008-05-13 07:15:31.1234 PST',
'YYYY-MM-DD HH24:MI:SS.FF TZR') AS TIMESTAMP WITH LOCAL TIME ZONE)
FROM dual;
```

```
CAST(TO_TIMESTAMP_TZ('2008-05-13 07:15:31.1234 PST', 'YYYY-MM-DD HH24:MI:SS.FF
-----
```

```
13-MAY-08 06.15.31.123400 AM
```

O timestamp retornado por essa consulta contém a data 13 de maio de 2008 e a hora 6:15:31.1234 PST (PST é o fuso horário de meu banco de dados e de minha sessão). A consulta a seguir faz o mesmo que a anterior, exceto que, desta vez, o fuso horário é EST:

```
SELECT CAST(TO_TIMESTAMP_TZ('2008-05-13 07:15:31.1234 EST',
'YYYY-MM-DD HH24:MI:SS.FF TZR') AS TIMESTAMP WITH LOCAL TIME ZONE)
FROM dual;
```

```
CAST(TO_TIMESTAMP_TZ('2008-05-13 07:15:31.1234 EST', 'YYYY-MM-DD HH24:MI:SS.FF
-----
```

```
13-MAY-08 04.15.31.123400 AM
```

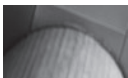
O timestamp retornado por essa consulta contém a data 13 de maio de 2008 e a hora 4:15:31.1234 PST (como PST é 3 horas antes de EST, a hora retornada no timestamp é 3 horas antes daquela que aparece na consulta).

## USANDO INTERVALOS DE TEMPO

O Oracle Database 9i introduziu tipos de dados que permitem armazenar intervalos de tempo. Exemplos de intervalos de tempo incluem:

- 1 ano e 3 meses
- 25 meses

- -3 dias, 5 horas e 16 minutos
- 1 dia e 7 horas
- -56 horas



**NOTA**

*Não confunda intervalos de tempo com data/horários ou timestamps. Um intervalo de tempo registra um período de tempo (por exemplo, 1 ano e 3 meses), enquanto uma data/horário ou timestamp registra uma data e hora específica (por exemplo, 19:32:16 de 28 de outubro de 2006).*

Em nossa loja online imaginária, talvez você queira oferecer descontos para os produtos por tempo limitado. Por exemplo, a loja poderia fornecer aos clientes um cupom válido por alguns meses ou fazer uma promoção especial por alguns dias. Você verá exemplos que retratam cupons e promoções posteriormente nesta seção.

A Tabela 5-8 mostra os tipos de intervalo. Você vai aprender a usar os tipos de intervalo de tempo nas seções a seguir.

**Usando o tipo INTERVAL YEAR TO MONTH**

INTERVAL YEAR TO MONTH armazena um intervalo de tempo medido em anos e meses. A instrução a seguir cria uma tabela chamada coupons que armazena informações de cupom. A tabela coupons contém uma coluna INTERVAL YEAR TO MONTH chamada duration para registrar o intervalo de tempo durante o qual o cupom é válido; observe que fornecemos uma precisão igual a 3 para a coluna duration, o que significa que até três dígitos podem ser armazenados para o ano:

**Tabela 5-8** Tipos de intervalo de tempo

Tipo	Descrição
INTERVAL YEAR [ (precisão_anos) ] TO MONTH	Armazena um intervalo de tempo medido em anos e meses. Você pode especificar uma precisão opcional para os anos, usando <i>precisão_anos</i> , que pode ser um valor inteiro de 0 a 9. A precisão padrão é 2, que significa que você pode armazenar dois dígitos para os anos no intervalo. Se tentar adicionar uma linha com mais dígitos de ano do que a coluna INTERVAL YEAR TO MONTH pode armazenar, obterá um erro. Você pode armazenar um intervalo de tempo positivo ou negativo.
INTERVAL DAY [ (precisão_dias) ] TO SECOND [ (precisão_segundos) ]	Armazena um intervalo de tempo medido em dias e segundos. Você pode especificar uma precisão opcional para os dias, usando <i>precisão_dias</i> (um valor inteiro de 0 a 9; o padrão é 2). Além disso, também pode especificar uma precisão opcional para as frações de segundos, usando <i>precisão_segundos</i> (um valor inteiro de 0 a 9; o padrão é 6). Você pode armazenar um intervalo de tempo positivo ou negativo.

```
CREATE TABLE coupons (
    coupon_id INTEGER CONSTRAINT coupons_pk PRIMARY KEY,
    name VARCHAR2(30) NOT NULL,
    duration INTERVAL YEAR(3) TO MONTH
);
```

Para fornecer um valor `INTERVAL YEAR TO MONTH` literal para o banco de dados, use a seguinte sintaxe simplificada:

```
INTERVAL '[+|-] [y] [-m]' [YEAR[(precisão_anos)]] [TO MONTH]
```

onde

- `+` ou `-` é um indicador opcional que especifica se o intervalo de tempo é positivo ou negativo (o padrão é positivo).
- `y` é o número de anos opcional para o intervalo.
- `m` é o número de meses opcional para o intervalo. Se você fornecer anos e meses, deverá incluir `TO MONTH` em sua literal.
- `precisão_anos` é a precisão opcional para os anos (o padrão é 2).

A tabela a seguir mostra alguns exemplos de literais de intervalo de ano para mês.

Literal	Descrição
<code>INTERVAL '1' YEAR</code>	Intervalo de 1 ano.
<code>INTERVAL '11' MONTH</code>	Intervalo de 11 meses.
<code>INTERVAL '14' MONTH</code>	Intervalo de 14 meses (equivalente a 1 ano e 2 meses).
<code>INTERVAL '1-3' YEAR TO MONTH</code>	Intervalo de 1 ano e 3 meses.
<code>INTERVAL '0-5' YEAR TO MONTH</code>	Intervalo de 0 anos e 5 meses.
<code>INTERVAL '123' YEAR(3) TO MONTH</code>	Intervalo de 123 anos com uma precisão de 3 dígitos.
<code>INTERVAL '-1-5' YEAR TO MONTH</code>	Um intervalo negativo de 1 ano e 5 meses.
<code>INTERVAL '1234' YEAR(3)</code>	Intervalo inválido: 1234 contém quatro dígitos e, portanto, contém um dígito a mais do que a precisão de 3 (máximo de três dígitos).

As instruções `INSERT` a seguir adicionam linhas na tabela `coupons`, com a coluna `duration` configurada com alguns dos intervalos mostrados na tabela anterior:

```
INSERT INTO coupons (coupon_id, name, duration)
VALUES (1, '$1 off Z Files', INTERVAL '1' YEAR);

INSERT INTO coupons (coupon_id, name, duration)
VALUES (2, '$2 off Pop 3', INTERVAL '11' MONTH);
```

```
INSERT INTO coupons (coupon_id, name, duration)
VALUES (3, '$3 off Modern Science', INTERVAL '14' MONTH);

INSERT INTO coupons (coupon_id, name, duration)
VALUES (4, '$2 off Tank War', INTERVAL '1-3' YEAR TO MONTH);

INSERT INTO coupons (coupon_id, name, duration)
VALUES (5, '$1 off Chemistry', INTERVAL '0-5' YEAR TO MONTH);

INSERT INTO coupons (coupon_id, name, duration)
VALUES (6, '$2 off Creative Yell', INTERVAL '123' YEAR(3));
```

Se você tentar adicionar uma linha com a coluna `duration` configurada com o intervalo inválido `INTERVAL '1234' YEAR(3)`, obterá um erro, pois a precisão da coluna `duration` é 3 e, portanto, pequena demais para acomodar o número 1234. A instrução `INSERT` a seguir mostra o erro:

```
SQL> INSERT INTO coupons (coupon_id, name, duration)
      2 VALUES (7, '$1 off Z Files', INTERVAL '1234' YEAR(3));
VALUES (7, '$1 off Z Files', INTERVAL '1234' YEAR(3))
*
ERROR at line 2:
ORA-01873: the leading precision of the interval is too small
```

A consulta a seguir recupera as linhas da tabela `coupons`; observe a formatação dos valores de `duration`:

```
SELECT *
FROM coupons;
```

COUPON_ID	NAME	DURATION
1	\$1 off Z Files	+001-00
2	\$2 off Pop 3	+000-11
3	\$3 off Modern Science	+001-02
4	\$2 off Tank War	+001-03
5	\$1 off Chemistry	+000-05
6	\$2 off Creative Yell	+123-00

## Usando o tipo INTERVAL DAY TO SECOND

`INTERVAL DAY TO SECOND` armazena intervalos de tempo medidos em dias e segundos. A instrução a seguir cria uma tabela chamada `promotions`, que armazena informações de promoção. A tabela `promotions` contém uma coluna `INTERVAL DAY TO SECOND` chamada `duration` para registrar o intervalo de tempo durante o qual a promoção é válida:

```
CREATE TABLE promotions (
  promotion_id INTEGER CONSTRAINT promotions_pk PRIMARY KEY,
  name VARCHAR2(30) NOT NULL,
  duration INTERVAL DAY(3) TO SECOND (4)
);
```

Observe que fornecemos uma precisão igual a 3 para o dia e igual a 4 para os segundos fracionários da coluna `duration`. Isso significa que até três dígitos podem ser armazenados para o dia do intervalo e até quatro dígitos à direita do ponto decimal para os segundos fracionários.



Para fornecer um valor `INTERVAL DAY TO SECOND` literal para o banco de dados, use a seguinte sintaxe simplificada:

```
INTERVAL '[+|-] [d] [h[:m[:s]]]' [DAY[(precisão_dias)]]
[TO HOUR | MINUTE | SECOND[(precisão_segundos)]]
```

onde

- `+` ou `-` é um indicador opcional que especifica se o intervalo de tempo é positivo ou negativo (o padrão é positivo).
- `d` é o número de dias do intervalo.
- `h` é o número de horas opcional do intervalo; se você fornecer dias e horas, deverá incluir `TO HOUR` em sua literal.
- `m` é o número de minutos opcional do intervalo; se você fornecer dias e minutos, deverá incluir `TO MINUTES` em sua literal.
- `s` é o número de segundos opcional do intervalo; se você fornecer dias e segundos, deverá incluir `TO SECOND` em sua literal.
- `precisão_dias` é a precisão opcional dos dias (o padrão é 2).
- `precisão_segundos` é a precisão opcional dos segundos fracionários (o padrão é 6).

A tabela a seguir mostra alguns exemplos de literais de intervalo de dias até segundos.

Literal	Descrição
INTERVAL '3' DAY	Intervalo de 3 dias.
INTERVAL '2' HOUR	Intervalo de 2 horas.
INTERVAL '25' MINUTE	Intervalo de 25 minutos.
INTERVAL '45' SECOND	Intervalo de 45 segundos.
INTERVAL '3 2' DAY TO HOUR	Intervalo de 3 dias e 2 horas.
INTERVAL '3 2:25' DAY TO MINUTE	Intervalo de 3 dias, 2 horas e 25 minutos.
INTERVAL '3 2:25:45' DAY TO SECOND	Intervalo de 3 dias, 2 horas, 25 minutos e 45 segundos.
INTERVAL '123 2:25:45.12' DAY(3) TO SECOND(2)	Intervalo de 123 dias, 2 horas, 25 minutos e 45,12 segundos; a precisão dos dias é de 3 dígitos e a precisão dos segundos fracionários é de 2 dígitos.
INTERVAL '3 2:00:45' DAY TO SECOND	Intervalo de 3 dias, 2 horas, 0 minutos e 45 segundos.
INTERVAL '-3 2:25:45' DAY TO SECOND	Intervalo negativo de 3 dias, 2 horas, 25 minutos e 45 segundos.
INTERVAL '1234 2:25:45' DAY(3) TO SECOND	Intervalo inválido, pois o número de dígitos nos dias ultrapassa a precisão especificada que é 3.
INTERVAL '123 2:25:45.123' DAY TO SECOND(2)	Intervalo inválido, pois o número de dígitos nos segundos fracionários ultrapassa a precisão especificada que é 2.

As instruções INSERT a seguir adicionam linhas na tabela promotions:

```
INSERT INTO promotions (promotion_id, name, duration)
VALUES (1, '10% off Z Files', INTERVAL '3' DAY);

INSERT INTO promotions (promotion_id, name, duration)
VALUES (2, '20% off Pop 3', INTERVAL '2' HOUR);

INSERT INTO promotions (promotion_id, name, duration)
VALUES (3, '30% off Modern Science', INTERVAL '25' MINUTE);

INSERT INTO promotions (promotion_id, name, duration)
VALUES (4, '20% off Tank War', INTERVAL '45' SECOND);

INSERT INTO promotions (promotion_id, name, duration)
VALUES (5, '10% off Chemistry', INTERVAL '3 2:25' DAY TO MINUTE);

INSERT INTO promotions (promotion_id, name, duration)
VALUES (6, '20% off Creative Yell', INTERVAL '3 2:25:45' DAY TO SECOND);

INSERT INTO promotions (promotion_id, name, duration)
VALUES (7, '15% off My Front Line',

INTERVAL '123 2:25:45.12' DAY(3) TO SECOND(2));
```

A consulta a seguir recupera as linhas da tabela promotions; observe a formatação dos valores de duration:

```
SELECT *
FROM promotions;
```

PROMOTION_ID	NAME	DURATION
1	10% off Z Files	+003 00:00:00.0000
2	20% off Pop 3	+000 02:00:00.0000
3	30% off Modern Science	+000 00:25:00.0000
4	20% off Tank War	+000 00:00:45.0000
5	10% off Chemistry	+003 02:25:00.0000
6	20% off Creative Yell	+003 02:25:45.0000
7	15% off My Front Line	+123 02:25:45.1200

Funções de intervalo de tempo

Existem várias funções que permitem obter e processar intervalos de tempo; elas estão mostradas na Tabela 5-9. Você irá aprender mais sobre as funções mostradas na Tabela 5-9 nas seções a seguir.

NUMTODSINTERVAL()

NUMTODSINTERVAL(*x*, *unidade\_intervalo*) converte o número *x* em um valor INTERVAL DAY TO SECOND. O intervalo de *x* é especificado em *unidade\_intervalo*, que pode ser DAY, HOUR, MINUTE ou SECOND. Por exemplo, a consulta a seguir converte vários números em intervalos de tempo usando NUMTODSINTERVAL():

Tabela 5-9    Funções de intervalo de tempo

Função	Descrição
NUMTODSINTERVAL( <i>x</i> , <i>unidade_intervalo</i> )	Converte o número <i>x</i> em um valor INTERVAL DAY TO SECOND. O intervalo de <i>x</i> é especificado em <i>unidade_intervalo</i> , que pode ser DAY, HOUR, MINUTE ou SECOND.
NUMTOYMINTERVAL( <i>x</i> , <i>unidade_intervalo</i> )	Converte o número <i>x</i> em um valor INTERVAL YEAR TO MONTH. O intervalo de <i>x</i> é especificado em <i>unidade_intervalo</i> , que pode ser YEAR ou MONTH.
TO_DSINTERVAL( <i>x</i> )	Converte a string <i>x</i> em um valor INTERVAL DAY TO SECOND.
TO_YMINTERVAL( <i>x</i> )	Converte a string <i>x</i> em um valor INTERVAL YEAR TO MONTH.

```
SELECT
  NUMTODSINTERVAL(1.5, 'DAY'),
  NUMTODSINTERVAL(3.25, 'HOUR'),
  NUMTODSINTERVAL(5, 'MINUTE'),
  NUMTODSINTERVAL(10.123456789, 'SECOND')
FROM dual;

NUMTODSINTERVAL(1.5,'DAY')
-----
NUMTODSINTERVAL(3.25,'HOUR')
-----
NUMTODSINTERVAL(5,'MINUTE')
-----
NUMTODSINTERVAL(10.123456789,'SECOND')
-----
+0000000001 12:00:00.000000000
+0000000000 03:15:00.000000000
+0000000000 00:05:00.000000000
+0000000000 00:00:10.123456789
```

**NUMTOYMINTERVAL()**

NUMTOYMINTERVAL(*x*, *unidade\_intervalo*) converte o número *x* em um valor INTERVAL YEAR TO MONTH. O intervalo de *x* é especificado em *unidade\_intervalo*, que pode ser YEAR ou MONTH. Por exemplo, a consulta a seguir converte dois números em intervalos de tempo usando NUMTOYMINTERVAL():

```
SELECT
  NUMTOYMINTERVAL(1.5, 'YEAR'),
  NUMTOYMINTERVAL(3.25, 'MONTH')
FROM dual;

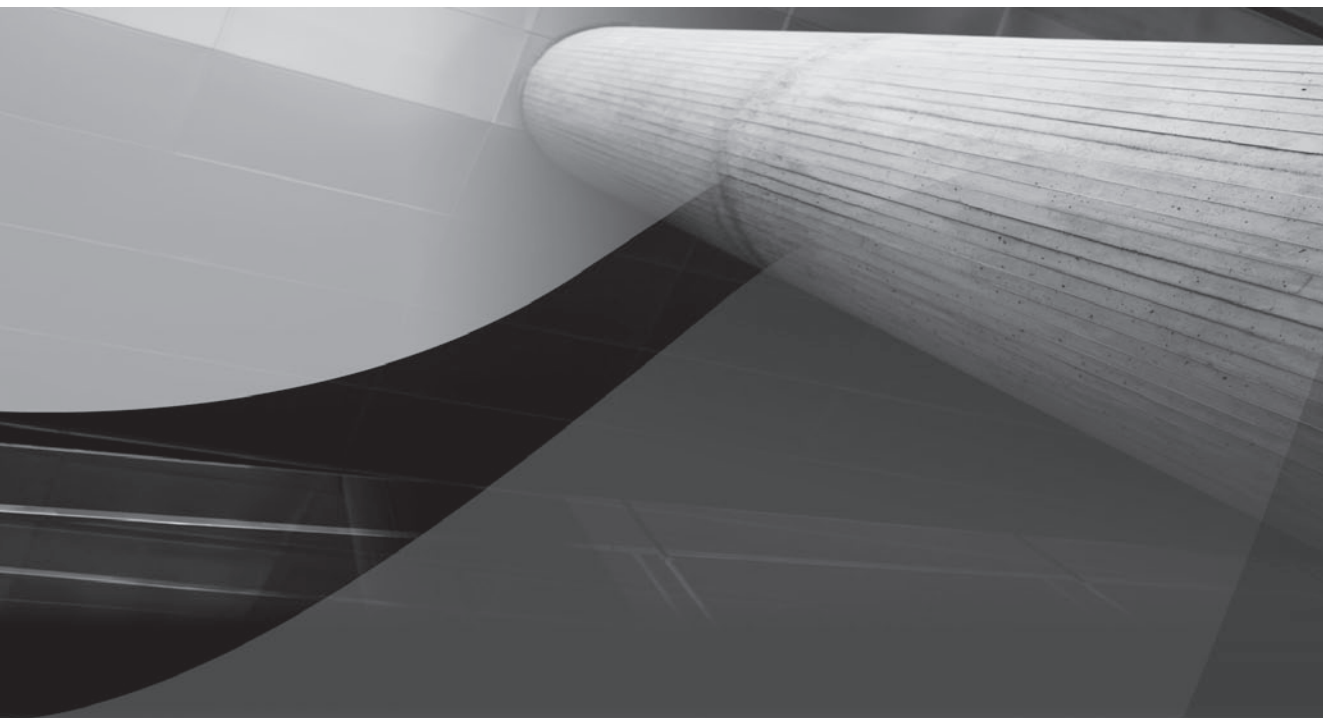
NUMTOYMINTERVAL(1.5,'YEAR')
-----
NUMTOYMINTERVAL(3.25,'MONTH')
-----
+0000000001-06
+0000000000-03
```

## RESUMO

Neste capítulo, você aprendeu que:

- É possível armazenar uma data/horário usando o tipo `DATE`. O tipo `DATE` armazena o século, todos os quatro dígitos de um ano, o mês, o dia, a hora (no formato de 24 horas), o minuto e o segundo.
- É possível usar `TO_CHAR()` e `TO_DATE()` para converter entre strings e datas e horas.
- O banco de dados Oracle sempre armazena todos os quatro dígitos de um ano e interpreta anos de dois dígitos usando um conjunto de regras. Você sempre deve especificar todos os quatro dígitos do ano.
- Existem várias funções que processam datas e horas. Um exemplo é `ADD_MONTHS(x, y)`, que retorna o resultado da adição de  $y$  meses em  $x$ .
- O Oracle Database 9i introduziu a capacidade de usar diferentes fusos horários. Um fuso horário é uma diferença em relação à hora de Greenwich, Inglaterra. A hora de Greenwich era conhecida como horário médio de Greenwich (GMT), mas agora é conhecida como Tempo Universal Coordenado (UTC). Você especifica um fuso horário usando uma diferença em relação a UTC ou o nome da região (por exemplo, PST).
- O Oracle Database 9i introduziu a capacidade de armazenar timestamps. Um timestamp armazena o século, todos os quatro dígitos de um ano, o mês, o dia, a hora (no formato de 24 horas), o minuto e o segundo. A vantagem de um timestamp em relação a um tipo `DATE` é que um timestamp pode armazenar um segundo fracionário e um fuso horário.
- O Oracle Database 9i introduziu a capacidade de manipular intervalos de tempo, os quais permitem armazenar um período de tempo. Um exemplo de intervalo de tempo é 1 ano e 3 meses.

No próximo capítulo, você irá aprender a aninhar uma consulta dentro de outra.



# CAPÍTULO 6

Subconsultas

Todas as consultas vistas até aqui continham apenas uma instrução `SELECT`. Neste capítulo, você vai aprender:

- Como colocar uma instrução `SELECT` interna dentro de uma instrução `SELECT`, `UPDATE` ou `DELETE` externa. A instrução `SELECT` interna é conhecida como *subconsulta*.
- As características dos diferentes tipos de subconsultas.
- Como as subconsultas permitem construir instruções muito complexas a partir de componentes simples.

## TIPOS DE SUBCONSULTAS

Existem dois tipos básicos de subconsultas:

- As **subconsultas de uma única linha** retornam zero ou uma linha para a instrução SQL externa. Existe um caso especial de subconsulta de uma linha que contém exatamente uma coluna; esse tipo de subconsulta é chamada de *subconsulta escalar*.
- As **subconsultas de várias linhas** retornam uma ou mais linhas para a instrução SQL externa.

Além disso, existem três tipos de subconsultas que podem retornar uma ou várias linhas:

- As **subconsultas de várias colunas** retornam mais de uma coluna para a instrução SQL externa.
- As **subconsultas correlacionadas** referenciam uma ou mais colunas na instrução SQL externa. Elas são chamadas de subconsultas “correlacionadas” porque são relacionadas à instrução SQL externa por meio das mesmas colunas.
- As **subconsultas aninhadas** são colocadas dentro de outra subconsulta. Você pode aninhar subconsultas até uma profundidade de 255.

Você vai aprender sobre cada um desses tipos de subconsultas e como adicioná-las em instruções `SELECT`, `UPDATE` e `DELETE`. Primeiro, vamos aprender a escrever subconsultas de uma única linha.

## ESCREVENDO SUBCONSULTAS DE UMA ÚNICA LINHA

Uma subconsulta de uma única linha é aquela que retorna zero ou uma linha para a instrução SQL externa. Você pode colocar uma subconsulta em uma cláusula `WHERE`, em uma cláusula `HAVING` ou em uma cláusula `FROM` de uma instrução `SELECT`. Neste capítulo, você verá alguns erros que pode encontrar ao executar subconsultas.

### Subconsultas em uma cláusula `WHERE`

É possível colocar uma subconsulta na cláusula `WHERE` de outra consulta. A consulta a seguir contém uma subconsulta colocada em sua cláusula `WHERE`; observe que a subconsulta é colocada entre parênteses (...):

```
SELECT first_name, last_name
FROM customers
WHERE customer_id =
    (SELECT customer_id
     FROM customers)
```

```
WHERE last_name = 'Brown');
```

```
FIRST_NAME LAST_NAME
-----
John      Brown
```

Esse exemplo recupera os valores de `first_name` e `last_name` da linha da tabela `customers` cujo valor de `last_name` é Brown. Vamos analisar essa consulta; a subconsulta na cláusula `WHERE` é:

```
SELECT customer_id
FROM customers
WHERE last_name = 'Brown';
```

Essa subconsulta é executada primeiro (e apenas uma vez) e retorna o valor de `customer_id` da coluna cujo valor de `last_name` é Brown. O valor de `customer_id` para essa linha é 1, o qual é passado para a cláusula `WHERE` da consulta externa. Portanto, a consulta externa retorna o mesmo resultado da consulta a seguir:

```
SELECT first_name, last_name
FROM customers
WHERE customer_id = 1;
```

## Usando outros operadores de uma única linha

O exemplo de subconsulta mostrado no início da seção anterior usava o operador de igualdade (`=`) na cláusula `WHERE`. Você também pode usar outros operadores de comparação, como `<>`, `<`, `>`, `<=` e `>=`, em uma subconsulta de uma única linha. O exemplo a seguir usa `>` na cláusula `WHERE` da consulta externa; a subconsulta usa a função `AVG()` para obter o preço médio dos produtos, o qual é passado para a cláusula `WHERE` da consulta externa. A consulta inteira retorna os valores de `product_id`, `name` e `price` dos produtos cujo preço é maior do que a média de preços.

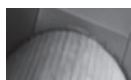
```
SELECT product_id, name, price
FROM products
WHERE price >
      (SELECT AVG(price)
       FROM products);
```

PRODUCT_ID	NAME	PRICE
1	Modern Science	19.95
2	Chemistry	30
3	Supernova	25.99
5	Z Files	49.99

Vamos decompor o exemplo para entender como ele funciona. O exemplo a seguir mostra a execução da subconsulta sozinha:

```
SELECT AVG(price)
FROM products;
```

```
AVG(PRICE)
-----
19.7308333
```

**NOTA**

*Essa subconsulta é um exemplo de subconsulta escalar, pois retorna exatamente uma linha contendo uma coluna. O valor retornado por uma subconsulta escalar é tratado como um único valor escalar.*

O valor 19.7308333 retornado pela subconsulta é utilizado na cláusula `WHERE` da consulta externa, a qual é, portanto, equivalente a:

```
SELECT product_id, name, price
FROM products
WHERE price > 19.7308333;
```

## Subconsultas em uma cláusula HAVING

Conforme foi visto no Capítulo 4, a cláusula `HAVING` é usada para filtrar grupos de linhas. Você pode colocar uma subconsulta na cláusula `HAVING` de uma consulta externa. Isso permite filtrar grupos de linhas com base no resultado retornado por sua subconsulta.

O exemplo a seguir usa uma subconsulta na cláusula `HAVING` da consulta externa. O exemplo recupera o valor de `product_type_id` e o preço médio dos produtos cujo preço médio é menor do que o máximo da média dos grupos do mesmo tipo de produto:

```
SELECT product_type_id, AVG(price)
FROM products
GROUP BY product_type_id
HAVING AVG(price) <
    (SELECT MAX(AVG(price))
     FROM products
     GROUP BY product_type_id)
ORDER BY product_type_id;
```

PRODUCT_TYPE_ID	AVG(PRICE)
1	24.975
3	13.24
4	13.99
	13.49

Observe que a subconsulta usa `AVG()` para calcular primeiro o preço médio para cada tipo de produto. Então, o resultado retornado por `AVG()` é passado para `MAX()`, que retorna o máximo das médias. Vamos analisar o exemplo para entender como ele funciona. O exemplo a seguir mostra a saída da subconsulta quando ela é executada sozinha:

```
SELECT MAX(AVG(price))
FROM products
GROUP BY product_type_id;
```

MAX(AVG(PRICE))
26.22



O valor 26.22 é usado na cláusula `HAVING` da consulta externa para filtrar as linhas do grupo para aquelas que têm um preço médio menor do que 26,22. A consulta a seguir mostra uma versão da consulta externa que recupera o valor de `product_type_id` e o preço médio dos produtos agrupados por `product_type_id`:

```
SELECT product_type_id, AVG(price)
FROM products
GROUP BY product_type_id
ORDER BY product_type_id;
```

```
PRODUCT_TYPE_ID  AVG(PRICE)
-----
1                24.975
2                26.22
3                13.24
4                13.99
                13.49
```

Os grupos com valor de `product_type_id` igual a 1, 3, 4 e nulo têm um preço médio menor do que 26,22. Conforme o esperado, esses são os mesmos grupos retornados pela consulta do início desta seção.

## Subconsultas em uma cláusula FROM (visões inline)

Você pode colocar uma subconsulta na cláusula `FROM` de uma consulta externa. Esses tipos de subconsultas também são conhecidos como *visões inline*, pois a subconsulta fornece dados em linha com a cláusula `FROM`. O exemplo simples a seguir recupera os produtos cujo valor de `product_id` é menor do que 3:

```
SELECT product_id
FROM
  (SELECT product_id
   FROM products
   WHERE product_id < 3);
```

```
PRODUCT_ID
-----
1
2
```

Note que a subconsulta retorna as linhas da tabela `products` cujo valor de `product_id` é menor do que 3 para a consulta externa, a qual então recupera e exibe esses valores de `product_id`. No que diz respeito à cláusula `FROM` da consulta externa, a saída da subconsulta é apenas outra fonte de dados.

O exemplo a seguir é mais útil e recupera os valores de `product_id` e `price` da tabela `products` na consulta externa e a subconsulta recupera o número de vezes que um produto foi comprado:

```
SELECT prds.product_id, price, purchases_data.product_count
FROM products prds,
  (SELECT product_id, COUNT(product_id) product_count
```

```

FROM purchases
GROUP BY product_id) purchases_data
WHERE prds.product_id = purchases_data.product_id;

```

PRODUCT_ID	PRICE	PRODUCT_COUNT
1	19.95	4
2	30	4
3	25.99	1

Note que a subconsulta recupera os valores de `product_id` e `COUNT (product_id)` da tabela `purchases` e os retorna para a consulta externa. A saída da subconsulta é apenas outra fonte de dados para a cláusula `FROM` da consulta externa.

## Erros que você pode encontrar

Nesta seção, você verá alguns erros que poderá encontrar. Especificamente, verá que uma subconsulta de uma única linha pode retornar no máximo uma linha e que uma subconsulta não pode conter uma cláusula `ORDER BY`.

### As subconsultas de uma única linha podem retornar no máximo uma linha

Se sua subconsulta retornar mais de uma linha, você irá obter o seguinte erro:

```
ORA-01427: single-row subquery returns more than one row.
```

Por exemplo, a subconsulta da instrução a seguir tenta passar várias linhas para o operador de igualdade (=) na consulta externa:

```

SQL> SELECT product_id, name
2   FROM products
3   WHERE product_id =
4     (SELECT product_id
5      FROM products
6      WHERE name LIKE '%e%');

(SELECT product_id
*)
ERROR at line 4:
ORA-01427: single-row subquery returns more than one row

```

Existem nove linhas na tabela `products` cujos nomes contêm a letra `e` e a subconsulta tenta passar essas linhas para o operador de igualdade na consulta externa. Como o operador de igualdade só pode manipular uma linha, a consulta é inválida e um erro é retornado. Você vai aprender a retornar várias linhas de uma subconsulta na seção “Escrevendo subconsultas de várias linhas”.

### As subconsultas não podem conter uma cláusula `ORDER BY`

Uma subconsulta não pode conter uma cláusula `ORDER BY`. Em vez disso, toda classificação deve ser feita na consulta externa. Por exemplo, a consulta externa a seguir tem uma cláusula `ORDER BY` no final, que classifica os valores de `product_id` em ordem decrescente:

```
SELECT product_id, name, price
FROM products
WHERE price >
      (SELECT AVG(price)
       FROM products)
ORDER BY product_id DESC;
```

PRODUCT_ID	NAME	PRICE
5	Z Files	49.99
3	Supernova	25.99
2	Chemistry	30
1	Modern Science	19.95

## ESCREVENDO SUBCONSULTAS DE VÁRIAS LINHAS

Uma subconsulta de várias linhas retorna uma ou mais linhas para uma instrução SQL externa. Para tratar de uma subconsulta que retorna várias linhas, sua consulta externa pode usar o operador `IN`, `ANY` ou `ALL`. Conforme vimos no Capítulo 2, você pode usar esses operadores para verificar se o valor de uma coluna está contido em uma lista de valores; por exemplo:

```
SELECT product_id, name
FROM products
WHERE product_id IN (1, 2, 3);
```

PRODUCT_ID	NAME
1	Modern Science
2	Chemistry
3	Supernova

Conforme será visto nesta seção, a lista de valores pode vir de uma subconsulta.

### NOTA

*Também é possível usar o operador `EXISTS` para verificar se um valor está em uma lista retornada por uma subconsulta correlacionada. Você irá aprender sobre isso na seção “Escrevendo subconsultas correlacionadas”.*

## Usando `IN` em uma subconsulta de várias linhas

Como vimos no Capítulo 2, você usa `IN` para verificar se um valor está em uma lista de valores especificada. A lista de valores pode vir dos resultados retornados por uma subconsulta. Você também pode usar `NOT IN` para executar a lógica oposta de `IN`: verificar se um valor não está em uma lista de valores especificada.

O exemplo simples a seguir usa `IN` para verificar se um valor de `product_id` está na lista de valores retornada pela subconsulta; a subconsulta retorna o valor de `product_id` dos produtos cujo nome contém a letra `e`:

```
SELECT product_id, name
FROM products
WHERE product_id IN
```

```
(SELECT product_id
FROM products
WHERE name LIKE '%e%');
```

```
PRODUCT_ID NAME
-----
1 Modern Science
2 Chemistry
3 Supernova
5 Z Files
6 2412: The Return
7 Space Force 9
8 From Another Planet
11 Creative Yell
12 My Front Line
```

O exemplo a seguir usa `NOT IN` para obter os produtos que não estão na tabela `purchases`:

```
SELECT product_id, name
FROM products
WHERE product_id NOT IN
  (SELECT product_id
   FROM purchases);
```

```
PRODUCT_ID NAME
-----
4 Tank War
5 Z Files
6 2412: The Return
7 Space Force 9
8 From Another Planet
9 Classical Music
10 Pop 3
11 Creative Yell
12 My Front Line
```

## Usando ANY em uma subconsulta de várias linhas

O operador `ANY` é usado para comparar um valor com qualquer valor presente em uma lista. Você deve colocar um operador `=`, `<>`, `<`, `>`, `<=` ou `>=` antes de `ANY` em sua consulta. O exemplo a seguir usa `ANY` para obter os funcionários cujo salário é menor do que qualquer um dos salários mais baixos da tabela `salary_grades`:

```
SELECT employee_id, last_name
FROM employees
WHERE salary < ANY
  (SELECT low_salary
   FROM salary_grades);
```

```
EMPLOYEE_ID LAST_NAME
-----
2 Johnson
3 Hobbs
4 Jones
```

## Usando ALL em uma subconsulta de várias linhas

O operador `ALL` é usado para comparar um valor com todos os valores presentes em uma lista. Você deve colocar um operador `=`, `<>`, `<`, `>`, `<=` ou `>=` antes de `ALL` em sua consulta. O exemplo a seguir usa `ALL` para obter os funcionários cujo salário é maior do que todos os salários mais altos da tabela `salary_grades`:

```
SELECT employee_id, last_name
FROM employees
WHERE salary > ALL
  (SELECT high_salary
   FROM salary_grades);
```

no rows selected

Nenhum funcionário tem salário maior do que o salário mais alto.

## ESCREVENDO SUBCONSULTAS DE VÁRIAS COLUNAS

As subconsultas vistas até aqui retornaram linhas contendo apenas uma coluna. Você não está limitado a uma única coluna: é possível escrever subconsultas que retornam várias colunas. O exemplo a seguir recupera os produtos com o menor preço para cada grupo de tipo de produto:

```
SELECT product_id, product_type_id, name, price
FROM products
WHERE (product_type_id, price) IN
  (SELECT product_type_id, MIN(price)
   FROM products
   GROUP BY product_type_id);
```

PRODUCT_ID	PRODUCT_TYPE_ID	NAME	PRICE
1	1	Modern Science	19.95
4	2	Tank War	13.95
8	3	From Another Planet	12.99
9	4	Classical Music	10.99

Note que a subconsulta retorna o valor de `product_type_id` e o valor de `price` mínimo para cada grupo de produtos e que eles são comparados com os valores de `product_type_id` e `price` para cada produto na cláusula `WHERE` da consulta externa.

## ESCREVENDO SUBCONSULTAS CORRELACIONADAS

Uma subconsulta correlacionada referencia uma ou mais colunas na instrução SQL externa. Elas são chamadas de *subconsultas correlacionadas* porque são relacionadas à instrução SQL externa por meio das mesmas colunas.

Normalmente, você usa uma subconsulta correlacionada quando precisa de uma resposta para uma pergunta que depende de um valor em cada linha contida em uma consulta externa. Por exemplo, talvez você queira ver se existe uma relação entre os dados, mas não se preocupa com o número de linhas retornadas pela subconsulta; isto é, quer apenas verificar se *alguma* linha é retornada, mas não se importa com a quantidade retornada.

Uma subconsulta correlacionada é executada uma vez para cada linha na consulta externa; isso difere da subconsulta não correlacionada, que é executada apenas uma vez antes da execução

da consulta externa. Além disso, uma subconsulta correlacionada pode trabalhar com valores nulos. Nas seções a seguir, você verá exemplos que ilustram esses conceitos.

### Exemplo de subconsulta correlacionada

A subconsulta correlacionada a seguir recupera os produtos que têm preço maior do que a média para seu tipo de produto:

```
SELECT product_id, product_type_id, name, price
FROM products outer
WHERE price >
  (SELECT AVG(price)
   FROM products inner
   WHERE inner.product_type_id = outer.product_type_id);
```

PRODUCT_ID	PRODUCT_TYPE_ID	NAME	PRICE
2	1	Chemistry	30
5	2	Z Files	49.99
7	3	Space Force 9	13.49
10	4	Pop 3	15.99
11	4	Creative Yell	14.99

Note que usamos o apelido `outer` para rotular a consulta externa e o apelido `inner` para a subconsulta interna. A referência à coluna `product_type_id` nas partes interna e externa é o que torna a subconsulta interna correlacionada à consulta externa. Além disso, a subconsulta retorna uma única linha contendo o preço médio do produto.

Em uma subconsulta correlacionada, cada linha da consulta externa é passada por vez para a subconsulta. A subconsulta lê uma linha de cada vez da consulta externa e a aplica na subconsulta até que todas as linhas da consulta externa sejam processadas. Então, os resultados da consulta inteira são retornados.

No exemplo anterior, a consulta externa recupera cada linha da tabela `products` e passa para a consulta interna. Cada linha é lida pela consulta interna, a qual calcula o preço médio de cada produto onde o valor de `product_type_id` na consulta interna é igual ao valor de `product_type_id` na consulta externa.

### Usando EXISTS e NOT EXISTS em uma subconsulta correlacionada

O operador `EXISTS` verifica a existência de linhas retornadas por uma subconsulta. Embora você possa usar `EXISTS` em subconsultas não correlacionadas, em geral ele é utilizado em subconsultas correlacionadas. O operador `NOT EXISTS` executa a lógica oposta de `EXISTS`: ele verifica se linhas não existem nos resultados retornados por uma subconsulta.

#### Usando EXISTS em uma subconsulta correlacionada

O exemplo a seguir usa `EXISTS` para recuperar funcionários que gerenciam outros funcionários; observe que não nos preocupamos com a quantidade de linhas retornadas pela subconsulta; só nos preocupamos em saber se alguma linha é retornada:

```
SELECT employee_id, last_name
FROM employees outer
WHERE EXISTS
  (SELECT employee_id
```

```

FROM employees inner
WHERE inner.manager_id = outer.employee_id);

EMPLOYEE_ID LAST_NAME
-----
1 Smith
2 Johnson

```

Como EXISTS apenas verifica a existência de linhas retornadas pela subconsulta, uma subconsulta não precisa retornar uma coluna — ela pode retornar apenas um valor literal. Essa característica pode melhorar o desempenho de sua consulta. Por exemplo, a consulta a seguir reescreve o exemplo anterior com a subconsulta retornando o valor literal 1:

```

SELECT employee_id, last_name
FROM employees outer
WHERE EXISTS
  (SELECT 1
   FROM employees inner
   WHERE inner.manager_id = outer.employee_id);

EMPLOYEE_ID LAST_NAME
-----
1 Smith
2 Johnson

```

Desde que a subconsulta retorne uma ou mais linhas, EXISTS retornará verdadeiro; se a subconsulta não retornar uma linha, EXISTS retornará falso. Nos exemplos, não nos preocupamos com a quantidade de linhas retornadas pela subconsulta: focamos apenas em se alguma linha (ou nenhuma linha) é retornada, de modo que EXISTS retorne verdadeiro (ou falso). Como a consulta externa exige pelo menos uma coluna, o valor literal 1 é retornado pela subconsulta no exemplo anterior.

### Usando NOT EXISTS em uma subconsulta correlacionada

O exemplo a seguir usa NOT EXISTS para recuperar os produtos que não foram comprados:

```

SELECT product_id, name
FROM products outer
WHERE NOT EXISTS
  (SELECT 1
   FROM purchases inner
   WHERE inner.product_id = outer.product_id);

PRODUCT_ID NAME
-----
4 Tank War
5 Z Files
6 2412: The Return
7 Space Force 9
8 From Another Planet
9 Classical Music
10 Pop 3
11 Creative Yell
12 My Front Line

```

**EXISTS e NOT EXISTS versus IN e NOT IN**

Na seção “Usando IN em uma subconsulta de várias linhas”, você viu como o operador IN é usado para verificar se um valor está contido em uma lista. EXISTS é diferente de IN: EXISTS verifica apenas a existência de linhas, enquanto IN verifica os valores reais.

**DICA**

*Normalmente, EXISTS oferece um desempenho melhor do que IN em subconsultas. Portanto, quando possível, você deve usar EXISTS em vez de IN.*

É preciso ter cuidado ao escrever consultas que utilizam NOT EXISTS ou NOT IN. Quando uma lista de valores contém um valor nulo, NOT EXISTS retorna verdadeiro, mas NOT IN retorna falso. Considere o exemplo a seguir, que usa NOT EXISTS e recupera os tipos de produtos que não têm produtos desse tipo na tabela products:

```
SELECT product_type_id, name
FROM product_types outer
WHERE NOT EXISTS
  (SELECT 1
   FROM products inner
   WHERE inner.product_type_id = outer.product_type_id);

PRODUCT_TYPE_ID NAME
-----
5 Magazine
```

Observe que uma linha é retornada por esse exemplo. O exemplo a seguir reescreve a consulta anterior para usar NOT IN; note que nenhuma linha é retornada:

```
SELECT product_type_id, name
FROM product_types
WHERE product_type_id NOT IN
  (SELECT product_type_id
   FROM products);

no rows selected
```

Nenhuma linha é retornada porque a subconsulta retorna uma lista de valores product\_id, um dos quais é nulo (o valor de product\_type\_id do produto nº 12 é nulo). Por isso, NOT IN na consulta externa retorna falso e, portanto, nenhuma linha é retornada. Você pode contornar isso usando a função NVL() para converter valores nulos em valores não nulos. No exemplo a seguir, NVL() é usada para converter valores nulos de product\_type\_id em 0:

```
SELECT product_type_id, name
FROM product_types
WHERE product_type_id NOT IN
  (SELECT NVL(product_type_id, 0)
   FROM products);

PRODUCT_TYPE_ID NAME
-----
5 Magazine
```

Desta vez a linha aparece.



Esses exemplos ilustram outra diferença entre as subconsultas correlacionadas e não correlacionadas: uma consulta correlacionada pode trabalhar com valores nulos.

## ESCREVENDO SUBCONSULTAS ANINHADAS

Você pode aninhar subconsultas dentro de outras subconsultas até uma profundidade de 255. Essa técnica deve ser usada com moderação — você pode verificar que sua consulta tem desempenho melhor usando junções de tabela. O exemplo a seguir contém uma subconsulta aninhada; observe que ela está contida dentro de uma subconsulta, que por sua vez está contida em uma consulta externa:

```
SELECT product_type_id, AVG(price)
FROM products
GROUP BY product_type_id
HAVING AVG(price) <
    (SELECT MAX(AVG(price))
     FROM products
     WHERE product_type_id IN
        (SELECT product_id
         FROM purchases
         WHERE quantity > 1)
     GROUP BY product_type_id)
ORDER BY product_type_id;
```

PRODUCT_TYPE_ID	AVG(PRICE)
1	24.975
3	13.24
4	13.99
	13.49

Esse exemplo é bastante complexo e contém três consultas: uma subconsulta aninhada, uma subconsulta e a consulta externa. Essas partes da consulta são executadas nessa ordem. Vamos decompor o exemplo nas três partes e examinar os resultados retornados. A subconsulta aninhada é:

```
SELECT product_id
FROM purchases
WHERE quantity > 1
```

Essa subconsulta retorna o valor de `product_id` dos produtos que foram comprados mais de uma vez. As linhas retornadas por essa subconsulta são:

```
PRODUCT_ID
-----
2
1
```

A subconsulta que recebe essa saída é:

```
SELECT MAX(AVG(price))
FROM products
WHERE product_type_id IN
    (... saída da subconsulta aninhada...)
GROUP BY product_type_id
```

Essa subconsulta retorna o maior preço médio dos produtos retornados pela subconsulta aninhada. A linha retornada é:

```
MAX (AVG (PRICE))
-----
26.22
```

Essa linha é retornada para a seguinte consulta externa:

```
SELECT product_type_id, AVG(price)
FROM products
GROUP BY product_type_id
HAVING AVG(price) <
    (... saída da subconsulta...)
ORDER BY product_type_id;
```

Essa consulta retorna o valor de `product_type_id` e o preço médio dos produtos que são menores do que o preço médio retornado pela subconsulta. As linhas retornadas são:

```
PRODUCT_TYPE_ID  AVG(PRICE)
-----
1                24.975
3                13.24
4                13.99
                13.49
```

Essas são as linhas retornadas pela consulta completa mostrada no início desta seção.

## ESCREVENDO INSTRUÇÕES UPDATE E DELETE CONTENDO SUBCONSULTAS

Até aqui, você viu apenas subconsultas contidas em uma instrução `SELECT`. Conforme verá nesta seção, você também pode colocar subconsultas dentro de instruções `UPDATE` e `DELETE`.

### Escrevendo uma instrução UPDATE contendo uma subconsulta

Em uma instrução `UPDATE`, você pode definir uma coluna com o resultado retornado por uma subconsulta de uma linha. Por exemplo, a instrução `UPDATE` a seguir define o salário do funcionário nº 4 como a média dos níveis salariais altos retornada por uma subconsulta:

```
UPDATE employees
SET salary =
    (SELECT AVG(high_salary)
     FROM salary_grades)
WHERE employee_id = 4;

1 row updated.
```

Fazer isso aumenta o salário do funcionário nº 4 de US\$ 500.000 para US\$ 625.000 (essa é a média dos salários altos da tabela `salary_grades`).

**NOTA**

*Se você executar a instrução UPDATE, lembre-se de executar uma instrução ROLLBACK para desfazer a alteração. Desse modo, seus resultados irão corresponder àqueles mostrados posteriormente neste livro.*

## Escrevendo uma instrução DELETE contendo uma subconsulta

Você pode usar as linhas retornadas por uma subconsulta na cláusula WHERE de uma instrução DELETE. Por exemplo, a instrução DELETE a seguir remove o funcionário cujo salário é maior do que a média dos níveis salariais altos retornada por uma subconsulta:

```
DELETE FROM employees
WHERE salary >
      (SELECT AVG(high_salary)
        FROM salary_grades);
```

1 row deleted.

Essa instrução DELETE remove o funcionário nº 1.

**NOTA**

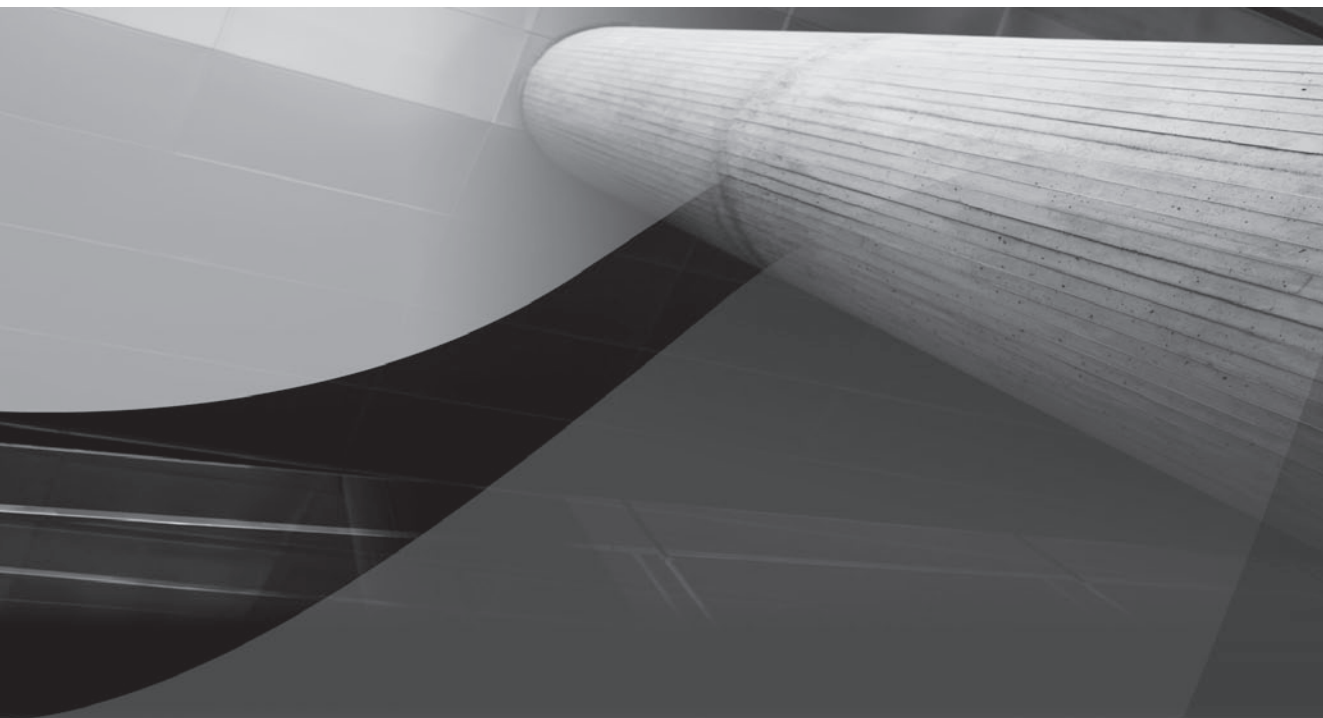
*Se você executar a instrução DELETE, execute uma instrução ROLLBACK para reverter a remoção da linha.*

## RESUMO

Neste capítulo, você aprendeu que:

- Uma subconsulta é uma consulta colocada dentro de uma instrução SELECT, UPDATE ou DELETE.
- As subconsultas de uma única linha retornam zero ou uma linha.
- As subconsultas de várias linhas retornam uma ou mais linhas.
- As subconsultas de várias colunas retornam mais de uma coluna.
- As subconsultas correlacionadas referenciam uma ou mais colunas na instrução SQL externa.
- As subconsultas aninhadas são subconsultas colocadas dentro de outra subconsulta.

No próximo capítulo, você vai aprender sobre as consultas avançadas.



# CAPÍTULO 7

Consultas avançadas

Neste capítulo, você vai aprender a:

- Usar os operadores de conjunto, que permitem combinar as linhas retornadas por duas ou mais consultas.
- Usar a função `TRANSLATE()` para transformar os caracteres de uma string em caracteres de outra string.
- Usar a função `DECODE()` para procurar determinado valor em um conjunto de valores.
- Usar a expressão `CASE` para executar lógica if-then-else em SQL.
- Executar consultas em dados hierárquicos.
- Usar as cláusulas `ROLLUP` e `CUBE` para obter subtotais e totais de grupos de linhas.
- Tirar proveito das funções analíticas, que efetuam cálculos complexos, como encontrar o tipo de produto mais vendido em cada mês, os principais vendedores etc.
- Efetuar cálculos entre linhas com a cláusula `MODEL`.
- Usar as novas cláusulas `PIVOT` e `UNPIVOT` do Oracle Database 11g, que são úteis para ver tendências globais em grandes volumes de dados.

Vamos começar examinando os operadores de conjunto.

## USANDO OS OPERADORES DE CONJUNTO

Os operadores de conjunto permitem combinar as linhas retornadas por duas ou mais consultas. A Tabela 7-1 mostra os quatro operadores de conjunto. Lembre a seguinte restrição ao usar um operador de conjunto: *o número de colunas e os tipos de coluna retornados pelas consultas devem corresponder, embora os nomes de coluna possam ser diferentes*. Você vai aprender a usar cada um dos operadores de conjunto mostrados na Tabela 7-1 em breve, mas primeiro veja as tabelas de exemplo utilizadas nesta seção.

**Tabela 7-1** Operadores de conjunto

Operador	Descrição
<code>UNION ALL</code>	Retorna todas as linhas recuperadas pelas consultas, incluindo as linhas duplicadas.
<code>UNION</code>	Retorna todas as linhas não duplicadas recuperadas pelas consultas.
<code>INTERSECT</code>	Retorna as linhas recuperadas pelas duas consultas.
<code>MINUS</code>	Retorna as linhas restantes, quando as linhas recuperadas pela segunda consulta são subtraídas das linhas recuperadas pela primeira.

## As tabelas de exemplo

As tabelas `products` e `more_products` são criadas pelo script `store_schema.sql` com as seguintes instruções:

```
CREATE TABLE products (
    product_id INTEGER
    CONSTRAINT products_pk PRIMARY KEY,
    product_type_id INTEGER
    CONSTRAINT products_fk_product_types
    REFERENCES product_types(product_type_id),
    name VARCHAR2(30) NOT NULL,
    description VARCHAR2(50),
    price NUMBER(5, 2)
);

CREATE TABLE more_products (
    prd_id INTEGER
    CONSTRAINT more_products_pk PRIMARY KEY,
    prd_type_id INTEGER
    CONSTRAINT more_products_fk_product_types
    REFERENCES product_types(product_type_id),
    name VARCHAR2(30) NOT NULL,
    available CHAR(1)
);
```

A consulta a seguir recupera as colunas `product_id`, `product_type_id` e `name` da tabela `products`:

```
SELECT product_id, product_type_id, name
FROM products;
```

PRODUCT_ID	PRODUCT_TYPE_ID	NAME
1	1	Modern Science
2	1	Chemistry
3	2	Supernova
4	2	Tank War
5	2	Z Files
6	2	2412: The Return
7	3	Space Force 9
8	3	From Another Planet
9	4	Classical Music
10	4	Pop 3
11	4	Creative Yell
12		My Front Line

A próxima consulta recupera as colunas `prd_id`, `prd_type_id` e `name` da tabela `more_products`:

```
SELECT prd_id, prd_type_id, name
FROM more_products;
```

PRD_ID	PRD_TYPE_ID	NAME
1	1	Modern Science
2	1	Chemistry
3		Supernova
4	2	Lunar Landing
5	2	Submarine

## Usando o operador UNION ALL

O operador `UNION ALL` retorna todas as linhas recuperadas pelas consultas, incluindo as linhas duplicadas. A consulta a seguir usa `UNION ALL`; observe que todas as linhas de `products` e `more_products` são recuperadas, incluindo as duplicadas:

```
SELECT product_id, product_type_id, name
FROM products
UNION ALL
SELECT prd_id, prd_type_id, name
FROM more_products;
```

PRODUCT_ID	PRODUCT_TYPE_ID	NAME
1	1	Modern Science
2	1	Chemistry
3	2	Supernova
4	2	Tank War
5	2	Z Files
6	2	2412: The Return
7	3	Space Force 9
8	3	From Another Planet
9	4	Classical Music
10	4	Pop 3
11	4	Creative Yell
12		My Front Line
1	1	Modern Science
2	1	Chemistry
3		Supernova
4	2	Lunar Landing
5	2	Submarine

17 rows selected.

Você pode classificar as linhas usando a cláusula `ORDER BY` seguida da posição da coluna. O próximo exemplo usa `ORDER BY 1` para classificar as linhas pela primeira coluna recuperada pelas duas consultas (`product_id` e `prd_id`):

```
SELECT product_id, product_type_id, name
FROM products
UNION ALL
SELECT prd_id, prd_type_id, name
FROM more_products
ORDER BY 1;
```

PRODUCT_ID	PRODUCT_TYPE_ID	NAME
1	1	Modern Science
1	1	Modern Science
2	1	Chemistry
2	1	Chemistry
3	2	Supernova
3		Supernova
4	2	Tank War
4	2	Lunar Landing
5	2	Z Files
5	2	Submarine
6	2	2412: The Return
7	3	Space Force 9
8	3	From Another Planet
9	4	Classical Music
10	4	Pop 3
11	4	Creative Yell
12		My Front Line

17 rows selected.

## Usando o operador UNION

O operador `UNION` retorna somente as linhas não duplicadas recuperadas pelas consultas, como mostra o exemplo a seguir. Observe que as linhas duplicadas “Modern Science” e “Chemistry” não são recuperadas e, portanto, somente 15 linhas são retornadas:

```
SELECT product_id, product_type_id, name
FROM products
UNION
SELECT prd_id, prd_type_id, name
FROM more_products;
```

PRODUCT_ID	PRODUCT_TYPE_ID	NAME
1	1	Modern Science
2	1	Chemistry
3	2	Supernova
3		Supernova
4	2	Lunar Landing
4	2	Tank War
5	2	Submarine
5	2	Z Files
6	2	2412: The Return
7	3	Space Force 9
8	3	From Another Planet
9	4	Classical Music
10	4	Pop 3
11	4	Creative Yell
12		My Front Line

15 rows selected.



## Usando o operador INTERSECT

O operador `INTERSECT` retorna somente as linhas recuperadas pelas duas consultas. O exemplo a seguir usa `INTERSECT`; observe que são retornadas as linhas “Modern Science” e “Chemistry”:

```
SELECT product_id, product_type_id, name
FROM products
INTERSECT
SELECT prd_id, prd_type_id, name
FROM more_products;

PRODUCT_ID PRODUCT_TYPE_ID NAME
-----
          1              1 Modern Science
          2              1 Chemistry
```

## Usando o operador MINUS

O operador `MINUS` retorna as linhas restantes, quando as linhas recuperadas pela segunda consulta são subtraídas das linhas recuperadas pela primeira. O exemplo a seguir usa `MINUS`; observe que as linhas de `more_products` são subtraídas de `products` e as linhas restantes são retornadas:

```
SELECT product_id, product_type_id, name
FROM products
MINUS
SELECT prd_id, prd_type_id, name
FROM more_products;

PRODUCT_ID PRODUCT_TYPE_ID NAME
-----
          3              2 Supernova
          4              2 Tank War
          5              2 Z Files
          6              2 2412: The Return
          7              3 Space Force 9
          8              3 From Another Planet
          9              4 Classical Music
         10              4 Pop 3
         11              4 Creative Yell
         12              My Front Line
```

10 rows selected.

## Combinando operadores de conjunto

Você pode combinar mais de duas consultas com vários operadores de conjunto, com os resultados retornados de um operador alimentando o operador seguinte. Por padrão, os operadores de conjunto são avaliados de cima para baixo, mas você deve indicar a ordem usando parênteses, para o caso da Oracle Corporation alterar esse comportamento padrão em futuras versões do software. Nos exemplos desta seção, vamos usar a tabela `product_changes` a seguir (criada pelo script `store_schema.sql`):

```
CREATE TABLE product_changes (
  product_id INTEGER
    CONSTRAINT prod_changes_pk PRIMARY KEY,
  product_type_id INTEGER
    CONSTRAINT prod_changes_fk_product_types
    REFERENCES product_types(product_type_id),
  name VARCHAR2(30) NOT NULL,
  description VARCHAR2(50),
  price NUMBER(5, 2)
);
```

A consulta a seguir retorna as colunas `product_id`, `product_type_id` e `name` da tabela `product_changes`:

```
SELECT product_id, product_type_id, name
FROM product_changes;
```

PRODUCT_ID	PRODUCT_TYPE_ID	NAME
1	1	Modern Science
2	1	New Chemistry
3	1	Supernova
13	2	Lunar Landing
14	2	Submarine
15	2	Airplane

A próxima consulta:

- Usa o operador `UNION` para combinar os resultados das tabelas `products` e `more_products`. (O operador `UNION` retorna somente as linhas não duplicadas recuperadas pelas consultas.)
- Usa o operador `INTERSECT` para combinar os resultados do operador `UNION` anterior com os resultados da tabela `product_changes`. (O operador `INTERSECT` retorna somente as linhas recuperadas pelas duas consultas.)
- Usa parênteses para indicar a ordem de avaliação, que é: (1) a operação `UNION` entre as tabelas `products` e `more_products`; (2) a operação `INTERSECT`.

```
(SELECT product_id, product_type_id, name
FROM products
UNION
SELECT prd_id, prd_type_id, name
FROM more_products)
INTERSECT
SELECT product_id, product_type_id, name
FROM product_changes;
```

PRODUCT_ID	PRODUCT_TYPE_ID	NAME
1	1	Modern Science

Na consulta a seguir, modificamos os parênteses para que a operação `INTERSECT` seja executada primeiro; observe os resultados diferentes retornados pela consulta, em comparação ao exemplo anterior:

```
SELECT product_id, product_type_id, name
FROM products
UNION
  (SELECT prd_id, prd_type_id, name
FROM more_products
INTERSECT
SELECT product_id, product_type_id, name
FROM product_changes);
```

PRODUCT_ID	PRODUCT_TYPE_ID	NAME
1	1	Modern Science
2	1	Chemistry
3	2	Supernova
4	2	Tank War
5	2	Z Files
6	2	2412: The Return
7	3	Space Force 9
8	3	From Another Planet
9	4	Classical Music
10	4	Pop 3
11	4	Creative Yell
12		My Front Line

Isso conclui a discussão sobre operadores de conjunto.

## USANDO A FUNÇÃO `TRANSLATE()`

`TRANSLATE(x, da_string, para_string)` converte as ocorrências dos caracteres em *da\_string* encontrados em *x* nos caracteres correspondentes em *para\_string*. É fácil entender como `TRANSLATE()` funciona vendo alguns exemplos. O exemplo a seguir usa `TRANSLATE()` para deslocar quatro casas para a direita cada caractere na string `SECRET MESSAGE: MEET ME IN THE PARK`; A torna-se E, B torna-se F e assim por diante:

```
SELECT TRANSLATE('SECRET MESSAGE: MEET ME IN THE PARK',
  'ABCDEFGHJKLMNOPQRSTUVWXYZ',
  'EFGHIJKLMNOPQRSTUVWXYZABCD')
FROM dual;
```

```
TRANSLATE('SECRETMESSAGE:MEETMEINTH
-----
WIGVIX QIWWEKI: QIIX QI MR XLI TEVO
```

O exemplo a seguir pega a saída do exemplo anterior e desloca os caracteres quatro casas para a esquerda; E torna-se A, F torna-se B e assim por diante:

```
SELECT TRANSLATE('WIGVIX QIWWEKI: QIIX QI MR XLI TEVO',
  'EFGHIJKLMNOPQRSTUVWXYZABCD',
```

```
'ABCDEFGHIJKLMNOPQRSTUVWXYZ')
FROM dual;

TRANSLATE('WIGVIXQIWWEKI:QIIXQIMRXL
-----
SECRET MESSAGE: MEET ME IN THE PARK
```

Naturalmente, você pode passar valores de coluna para `TRANSLATE()`. O exemplo a seguir passa a coluna `name` da tabela `products` para `TRANSLATE()`, que desloca as letras dos nomes de produto quatro casas para a direita:

```
SELECT product_id, TRANSLATE(name,
  'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz',
  'EFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMnopqrstuvwxyzabcd')
FROM products;

PRODUCT_ID TRANSLATE(NAME, 'ABCDEFGHIJKLMNOPQRSTUVWXYZ
-----
1 Qshivr Wgmirgi
2 Gliqmwxcv
3 Wytivrsze
4 Xero Aev
5 D Jmpiw
6 2412: Xli Vixyvr
7 Wtegi Jsvgi 9
8 Jvsq Ersxliv Tperix
9 Gpewwmgep Qywmg
10 Tst 3
11 Gviexmzi Cipp
12 Qc Jvsrx Pmri
```

Você também pode usar `TRANSLATE()` para converter números. O exemplo a seguir pega o número 12345 e converte 5 em 6, 4 em 7, 3 em 8, 2 em 9 e 1 em 0:

```
SELECT TRANSLATE(12345,
  54321,
  67890)
FROM dual;

TRANS
-----
09876
```

## USANDO A FUNÇÃO DECODE()

`DECODE(valor, valor_pesquisa, resultado, valor_padrão)` compara *valor* com *valor\_pesquisa*. Se os valores são iguais, `DECODE()` retorna *resultado*; caso contrário, *valor\_padrão* é retornado. `DECODE()` permite executar lógica if-then-else em SQL, sem a necessidade de usar PL/SQL. Cada um dos parâmetros de `DECODE()` pode ser uma coluna, um valor literal, uma função ou uma subconsulta.

**NOTA**

*DECODE () é uma função proprietária da Oracle e, portanto, você deve usar expressões CASE em seu lugar, caso esteja utilizando Oracle Database 9i e versões superiores (você vai aprender sobre expressões CASE na próxima seção). A função DECODE () é mencionada aqui porque você pode encontrá-la quando usar bancos de dados Oracle mais antigos.*

O exemplo a seguir ilustra o uso de DECODE () com valores literais; DECODE () retorna 2 (1 é comparado com 1 e, como eles são iguais, 2 é retornado):

```
SELECT DECODE(1, 1, 2, 3)
FROM dual;
```

```
DECODE(1,1,2,3)
-----
                2
```

O exemplo a seguir usa DECODE () para comparar 1 com 2 e, como eles não são iguais, 3 é retornado:

```
SELECT DECODE(1, 2, 1, 3)
FROM dual;
```

```
DECODE(1,2,1,3)
-----
                3
```

O próximo exemplo compara a coluna available da tabela more\_products; se available é igual a Y, a string 'Product is available' é retornada; caso contrário, 'Product is not available' é retornada:

```
SELECT prd_id, available,
       DECODE(available, 'Y', 'Product is available',
              'Product is not available')
FROM more_products;
```

```
PRD_ID A DECODE(AVAILABLE, 'Y', 'PR
-----
1 Y Product is available
2 Y Product is available
3 N Product is not available
4 N Product is not available
5 Y Product is available
```

Você pode passar vários parâmetros de pesquisa e resultado para DECODE (), como mostrado no exemplo a seguir, que retorna a coluna product\_type\_id como o nome do tipo de produto:

```
SELECT product_id, product_type_id,
       DECODE(product_type_id,
              1, 'Book',
              2, 'Video',
              3, 'DVD',
              4, 'CD',
              'Magazine')
```

**FROM products;**

PRODUCT_ID	PRODUCT_TYPE_ID	DECODE (P
1	1	Book
2	1	Book
3	2	Video
4	2	Video
5	2	Video
6	2	Video
7	3	DVD
8	3	DVD
9	4	CD
10	4	CD
11	4	CD
12		Magazine

Note que:

- Se `product_type_id` é 1, Book é retornado.
- Se `product_type_id` é 2, Video é retornado.
- Se `product_type_id` é 3, DVD é retornado.
- Se `product_type_id` é 4, CD é retornado.
- Se `product_type_id` é qualquer outro valor, Magazine é retornado.

## USANDO A EXPRESSÃO CASE

A expressão `CASE` executa a lógica if-then-else em SQL e é suportada no Oracle Database 9i e versões superiores. A expressão `CASE` funciona de maneira semelhante a `DECODE()`, mas você deve usar `CASE` porque ela é compatível com o padrão ANSI e faz parte do padrão SQL/92. Além disso, a expressão `CASE` é mais fácil de ler. Existem dois tipos de expressões `CASE`:

- Expressões case simples, que usam expressões para determinar o valor retornado
- Expressões case pesquisadas, que usam condições para determinar o valor retornado

Você vai aprender sobre esses dois tipos de expressões `CASE` a seguir.

### Usando expressões CASE simples

As expressões `CASE` simples usam expressões incorporadas para determinar o valor a ser retornado. Elas têm a seguinte sintaxe:

```
CASE expressão_pesquisa
  WHEN expressão1 THEN resultado1
  WHEN expressão2 THEN resultado2
  ...
  WHEN expressãoN THEN resultadoN
  ELSE resultado_padrão
END
```

onde

- *expressão\_pesquisa* é a expressão a ser avaliada.
- *expressão1*, *expressão2*, ..., *expressãoN* são as expressões a serem avaliadas em relação à *expressão\_pesquisa*.
- *resultado1*, *resultado2*, ..., *resultadoN* são os resultados retornados (um para cada expressão possível). Se *expressão1* for avaliada como *expressão\_pesquisa*, o *resultado1* será retornado e do mesmo modo para as outras expressões.
- *resultado\_padrão* é retornado quando nenhuma expressão correspondente é encontrada.

O exemplo a seguir mostra uma expressão CASE simples que retorna os tipos de produto como nomes:

```
SELECT product_id, product_type_id,  
       CASE product_type_id  
         WHEN 1 THEN 'Book'  
         WHEN 2 THEN 'Video'  
         WHEN 3 THEN 'DVD'  
         WHEN 4 THEN 'CD'  
         ELSE 'Magazine'  
       END  
FROM products;
```

PRODUCT_ID	PRODUCT_TYPE_ID	CASEPROD
1	1	Book
2	1	Book
3	2	Video
4	2	Video
5	2	Video
6	2	Video
7	3	DVD
8	3	DVD
9	4	CD
10	4	CD
11	4	CD
12		Magazine

## Usando expressões CASE pesquisadas

As expressões CASE pesquisadas utilizam condições para determinar o valor retornado. Elas têm a seguinte sintaxe:

```
CASE  
  WHEN condição1 THEN resultado1  
  WHEN condição2 THEN resultado2  
  ...  
  WHEN condiçãoN THEN resultadoN  
  ELSE resultado_padrão  
END
```

onde

- *condição1, condição2, ..., condiçãoN* são as expressões a serem avaliadas.
- *resultado1, resultado2, ..., resultadoN* são os resultados retornados (um para cada condição possível). Se *condição1* é verdadeira, o *resultado1* é retornado e do mesmo modo para as outras expressões.
- *resultado\_padrão* é retornado quando nenhuma condição retorna verdadeiro.

O exemplo a seguir ilustra o uso de uma expressão CASE pesquisada:

```
SELECT product_id, product_type_id,
CASE
  WHEN product_type_id = 1 THEN 'Book'
  WHEN product_type_id = 2 THEN 'Video'
  WHEN product_type_id = 3 THEN 'DVD'
  WHEN product_type_id = 4 THEN 'CD'
  ELSE 'Magazine'
END
FROM products;
```

PRODUCT_ID	PRODUCT_TYPE_ID	CASEPROD
1	1	Book
2	1	Book
3	2	Video
4	2	Video
5	2	Video
6	2	Video
7	3	DVD
8	3	DVD
9	4	CD
10	4	CD
11	4	CD
12		Magazine

Você pode usar operadores em uma expressão CASE pesquisada, como mostra o exemplo a seguir:

```
SELECT product_id, price,
CASE
  WHEN price > 15 THEN 'Expensive'
  ELSE 'Cheap'
END
FROM products;
```

PRODUCT_ID	PRICE	CASEWHENP
1	19.95	Expensive
2	30	Expensive
3	25.99	Expensive
4	13.95	Cheap
5	49.99	Expensive



6	14.95	Cheap
7	13.49	Cheap
8	12.99	Cheap
9	10.99	Cheap
10	15.99	Expensive
11	14.99	Cheap
12	13.49	Cheap

Exemplos mais avançados de expressões CASE são mostrados posteriormente neste capítulo e no Capítulo 16.

## CONSULTAS HIERÁRQUICAS

É comum organizar dados de forma hierárquica, por exemplo, as pessoas que trabalham em uma empresa, uma árvore genealógica e as peças que compõem um motor. Nesta seção, serão abordadas consultas que acessam uma hierarquia de funcionários que trabalham em nossa loja imaginária.

### Os dados de exemplo

Você vai ver o uso de uma tabela chamada `more_employees`, que é criada pelo script `store_schema.sql`, como segue:

```
CREATE TABLE more_employees (  
    employee_id INTEGER  
        CONSTRAINT more_employees_pk PRIMARY KEY,  
    manager_id INTEGER  
        CONSTRAINT more_empl_fk_fk_more_empl  
        REFERENCES more_employees(employee_id),  
    first_name VARCHAR2(10) NOT NULL,  
    last_name VARCHAR2(10) NOT NULL,  
    title VARCHAR2(20),  
    salary NUMBER(6, 0)  
);
```

A coluna `manager_id` é uma auto-referência à coluna `employee_id` da tabela `more_employees`; `manager_id` indica o gerente de um funcionário (se houver). A consulta a seguir retorna as linhas de `more_employees`:

```
SELECT *  
FROM more_employees;
```

EMPLOYEE_ID	MANAGER_ID	FIRST_NAME	LAST_NAME	TITLE	SALARY
1		James	Smith	CEO	800000
2	1	Ron	Johnson	Sales Manager	600000
3	2	Fred	Hobbs	Sales Person	200000
4	1	Susan	Jones	Support Manager	500000
5	2	Rob	Green	Sales Person	40000
6	4	Jane	Brown	Support Person	45000
7	4	John	Grey	Support Manager	30000
8	7	Jean	Blue	Support Person	29000

9	6	Henry	Heyson	Support Person	30000
10	1	Kevin	Black	Ops Manager	100000
11	10	Keith	Long	Ops Person	50000
12	10	Frank	Howard	Ops Person	45000
13	10	Doreen	Penn	Ops Person	47000

É difícil identificar os relacionamentos dos funcionários a partir desses dados. A Figura 7-1 mostra as relações de forma gráfica; os elementos — ou *nós* — formam uma árvore. Árvores de nós têm os seguintes termos técnicos associados:

- **Nó raiz** A raiz é o nó que está no topo da árvore. Na Figura 7-1, o nó raiz é James Smith, o diretor executivo.
- **Nó pai** Um pai é um nó que tem um ou mais nós debaixo dele. Por exemplo, o nó James Smith é o pai dos seguintes nós: Ron Johnson, Susan Jones e Kevin Black.
- **Nó filho** Um filho é um nó que tem um nó pai acima dele. Por exemplo, o nó pai de Ron Johnson é James Smith.
- **Nó folha** Uma folha é um nó que não tem filhos. Por exemplo, Fred Hobbs e Rob Green são nós folha.

As cláusulas `CONNECT BY` e `START WITH` de uma instrução `SELECT` executam consultas hierárquicas, conforme descrito a seguir.

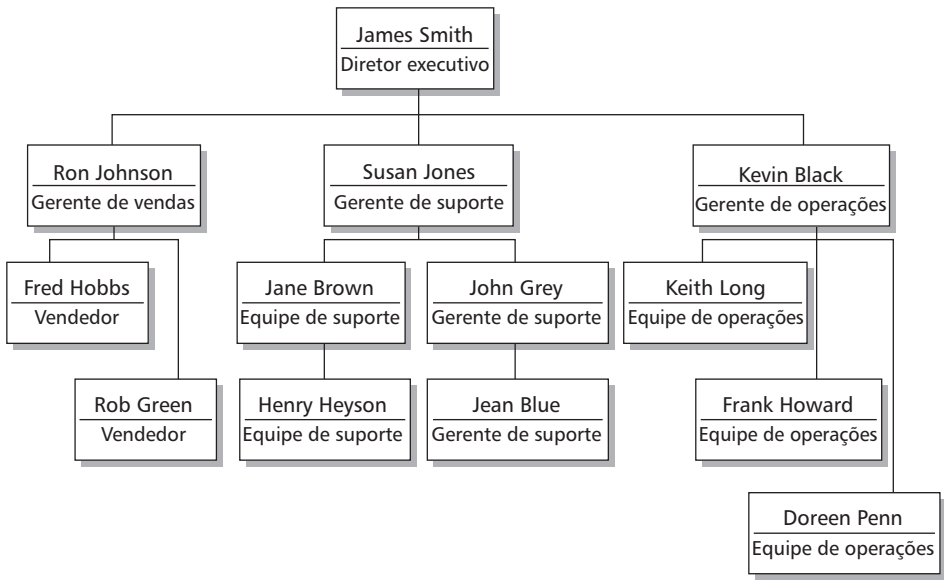


Figura 7-1 Relacionamento dos funcionários.

## Usando as cláusulas CONNECT BY e START WITH

A sintaxe das cláusulas CONNECT BY e START WITH de uma instrução SELECT é:

```
SELECT [LEVEL], coluna, expressão,...  
FROM tabela  
[WHERE cláusula_where]  
[[START WITH condição_inicial] [CONNECT BY PRIOR condição_anterior]];
```

onde

- LEVEL é uma pseudocoluna que indica o nível que você está dentro de uma árvore. LEVEL retorna 1 para um nó raiz, 2 para um filho da raiz e assim por diante.
- condição\_inicial especifica onde iniciar a consulta hierárquica. Você deve especificar uma cláusula START WITH quando escrever uma consulta hierárquica. Um exemplo de condição\_inicial é employee\_id = 1, que especifica que a consulta começa no funcionário nº 1.
- condição\_anterior especifica a relação entre as linhas pai e filho. Você deve especificar uma cláusula CONNECT BY PRIOR ao escrever uma consulta hierárquica. Um exemplo de condição\_anterior é employee\_id = manager\_id, que especifica que a relação é entre o employee\_id pai e o manager\_id filho — isto é, manager\_id do filho aponta para employee\_id do pai.

A consulta a seguir ilustra o uso das cláusulas START WITH e CONNECT BY PRIOR; observe que a primeira linha contém os detalhes de James Smith (funcionário nº 1), a segunda linha contém os detalhes de Ron Johnson, cujo valor de manager\_id é 1 e assim por diante:

```
SELECT employee_id, manager_id, first_name, last_name  
FROM more_employees  
START WITH employee_id = 1  
CONNECT BY PRIOR employee_id = manager_id;
```

EMPLOYEE_ID	MANAGER_ID	FIRST_NAME	LAST_NAME
1		James	Smith
2	1	Ron	Johnson
3	2	Fred	Hobbs
5	2	Rob	Green
4	1	Susan	Jones
6	4	Jane	Brown
9	6	Henry	Heyson
7	4	John	Grey
8	7	Jean	Blue
10	1	Kevin	Black
11	10	Keith	Long
12	10	Frank	Howard
13	10	Doreen	Penn

## Usando a pseudocoluna LEVEL

A próxima consulta ilustra o uso da pseudocoluna LEVEL para exibir o nível na árvore:

```
SELECT LEVEL, employee_id, manager_id, first_name, last_name
FROM more_employees
START WITH employee_id = 1
CONNECT BY PRIOR employee_id = manager_id
ORDER BY LEVEL;
```

LEVEL	EMPLOYEE_ID	MANAGER_ID	FIRST_NAME	LAST_NAME
1	1		James	Smith
2	2	1	Ron	Johnson
2	4	1	Susan	Jones
2	10	1	Kevin	Black
3	3	2	Fred	Hobbs
3	7	4	John	Grey
3	12	10	Frank	Howard
3	13	10	Doreen	Penn
3	11	10	Keith	Long
3	5	2	Rob	Green
3	6	4	Jane	Brown
4	9	6	Henry	Heyson
4	8	7	Jean	Blue

A próxima consulta usa a função COUNT () e LEVEL para obter o número de níveis na árvore:

```
SELECT COUNT(DISTINCT LEVEL)
FROM more_employees
START WITH employee_id = 1
CONNECT BY PRIOR employee_id = manager_id;

COUNT(DISTINCTLEVEL)
-----
4
```

## Formatando os resultados de uma consulta hierárquica

Você pode formatar os resultados de uma consulta hierárquica usando LEVEL e a função LPAD (), que preenche os valores com caracteres à esquerda. A consulta a seguir usa LPAD ( ' ', 2 \* LEVEL - 1) para preencher um total de 2 \* LEVEL - 1 espaços à esquerda; o resultado fará a endentação do nome do funcionário com espaços de acordo com seu valor de LEVEL (isto é, LEVEL 1 não é preenchido, LEVEL 2 é preenchido com dois espaços, LEVEL 3 com quatro espaços etc.):

```
SET PAGESIZE 999
COLUMN employee FORMAT A25
SELECT LEVEL,
       LPAD(' ', 2 * LEVEL - 1) || first_name || ' ' || last_name AS employee
FROM more_employees
START WITH employee_id = 1
CONNECT BY PRIOR employee_id = manager_id;

LEVEL EMPLOYEE
-----
1 James Smith
2   Ron Johnson
```

```

3    Fred Hobbs
3    Rob Green
2   Susan Jones
3    Jane Brown
4      Henry Heyson
3    John Grey
4      Jean Blue
2   Kevin Black
3      Keith Long
3      Frank Howard
3      Doreen Penn

```

As relações entre os funcionários são fáceis de identificar a partir desses resultados.

## Começando em um nó que não é o raiz

Não é preciso começar no nó raiz ao percorrer uma árvore: você pode começar em qualquer nó, usando a cláusula `START WITH`. A consulta a seguir começa com Susan Jones; observe que `LEVEL` retorna 1 para Susan Jones, 2 para Jane Brown e assim por diante:

```

SELECT LEVEL,
       LPAD(' ', 2 * LEVEL - 1) || first_name || ' ' || last_name AS employee
FROM more_employees
START WITH last_name = 'Jones'
CONNECT BY PRIOR employee_id = manager_id;

LEVEL  EMPLOYEE
-----
1 Susan Jones
2   Jane Brown
3     Henry Heyson
2   John Grey
3     Jean Blue

```

Se a loja tivesse mais de um funcionário com o mesmo sobrenome, você poderia usar simplesmente o valor de `employee_id` na cláusula `START WITH` da consulta. Por exemplo, a consulta a seguir usa o valor 4 para `employee_id` de Susan Jones:

```

SELECT LEVEL,
       LPAD(' ', 2 * LEVEL - 1) || first_name || ' ' || last_name AS employee
FROM more_employees
START WITH employee_id = 4
CONNECT BY PRIOR employee_id = manager_id;

```

Essa consulta retorna as mesmas linhas da anterior.

## Usando uma subconsulta em uma cláusula START WITH

Você pode usar uma subconsulta em uma cláusula `START WITH`. Por exemplo, a consulta a seguir usa uma subconsulta para selecionar o valor de `employee_id` cujo nome é Kevin Black; esse valor de `employee_id` é passado para a cláusula `START WITH`:

```

SELECT LEVEL,
       LPAD(' ', 2 * LEVEL - 1) || first_name || ' ' || last_name AS employee

```

```

FROM more_employees
START WITH employee_id = (
  SELECT employee_id
  FROM more_employees
  WHERE first_name = 'Kevin'
  AND last_name = 'Black'
)
CONNECT BY PRIOR employee_id = manager_id;

```

```

      LEVEL EMPLOYEE
-----
      1 Kevin Black
      2 Keith Long
      2 Frank Howard
      2 Doreen Penn

```

## Percorrendo a árvore para cima

Não é preciso percorrer uma árvore para baixo, dos pais para os filhos: é possível começar em um filho e percorrer para cima. Para tanto, basta trocar as colunas filho e pai na cláusula `CONNECT BY PRIOR`. Por exemplo, `CONNECT BY PRIOR manager_id = employee_id` conecta o valor de `manager_id` do filho no valor de `employee_id` do pai. A consulta a seguir começa com Jean Blue e percorre a árvore para cima, até chegar a James Smith; observe que `LEVEL` retorna 1 para Jean Blue, 2 para John Grey e assim por diante:

```

SELECT LEVEL,
  LPAD(' ', 2 * LEVEL - 1) || first_name || ' ' || last_name AS employee
FROM more_employees
START WITH last_name = 'Blue'
CONNECT BY PRIOR manager_id = employee_id;

```

```

      LEVEL EMPLOYEE
-----
      1 Jean Blue
      2 John Grey
      3 Susan Jones
      4 James Smith

```

## Eliminando nós e ramos de uma consulta hierárquica

Você pode eliminar um nó específico de uma árvore de consulta usando uma cláusula `WHERE`. A consulta a seguir elimina Ron Johnson dos resultados, usando `WHERE last_name != 'Johnson'`:

```

SELECT LEVEL,
  LPAD(' ', 2 * LEVEL - 1) || first_name || ' ' || last_name AS employee
FROM more_employees
WHERE last_name != 'Johnson'
START WITH employee_id = 1
CONNECT BY PRIOR employee_id = manager_id;

```

```

      LEVEL EMPLOYEE
-----
      1 James Smith
      3 Fred Hobbs

```

```

3      Rob Green
2      Susan Jones
3      Jane Brown
4      Henry Heyson
3      John Grey
4      Jean Blue
2      Kevin Black
3      Keith Long
3      Frank Howard
3      Doreen Penn

```

Embora Ron Johnson seja eliminado dos resultados, seus funcionários Fred Hobbs e Rob Green ainda estão incluídos. Para eliminar um ramo inteiro de nós dos resultados de uma consulta, adicione uma cláusula `AND` em sua cláusula `CONNECT BY PRIOR`. Por exemplo, a consulta a seguir usa `AND last_name!= 'Johnson'` para eliminar Ron Johnson e todos os seus funcionários dos resultados:

```

SELECT LEVEL,
       LPAD(' ', 2 * LEVEL - 1) || first_name || ' ' || last_name AS employee
FROM more_employees
START WITH employee_id = 1
CONNECT BY PRIOR employee_id = manager_id
AND last_name!= 'Johnson';

```

```

LEVEL EMPLOYEE
-----

```

```

1  James Smith
2   Susan Jones
3    Jane Brown
4   Henry Heyson
3    John Grey
4    Jean Blue
2   Kevin Black
3    Keith Long
3    Frank Howard
3    Doreen Penn

```

## Incluindo outras condições em uma consulta hierárquica

É possível incluir outras condições em uma consulta hierárquica usando uma cláusula `WHERE`. O exemplo a seguir usa uma cláusula `WHERE` para mostrar somente os funcionários cujos salários são menores ou iguais a US\$50.000:

```

SELECT LEVEL,
       LPAD(' ', 2 * LEVEL - 1) || first_name || ' ' || last_name AS employee,
       salary
FROM more_employees
WHERE salary <= 50000
START WITH employee_id = 1
CONNECT BY PRIOR employee_id = manager_id;

```

LEVEL	EMPLOYEE	SALARY
3	Rob Green	40000
3	Jane Brown	45000
4	Henry Heyson	30000
3	John Grey	30000
4	Jean Blue	29000
3	Keith Long	50000
3	Frank Howard	45000
3	Doreen Penn	47000

Isso conclui a discussão sobre consultas hierárquicas. Na próxima seção, você vai aprender sobre cláusulas de grupo avançadas.

## USANDO AS CLÁUSULAS GROUP BY ESTENDIDAS

Nesta seção, você vai aprender sobre:

- **ROLLUP**, que estende a cláusula **GROUP BY** para retornar uma linha contendo um subtotal para cada grupo de linhas, além de uma linha contendo um total geral para todos os grupos.
- **CUBE**, que estende a cláusula **GROUP BY** para retornar linhas contendo um subtotal para todas as combinações de colunas, além de uma linha contendo o total geral.

Antes, vamos ver as tabelas de exemplo utilizadas nesta seção.

### As tabelas de exemplo

As seguintes tabelas refinam a representação de funcionários em nossa loja imaginária:

- **divisions**, que armazena as divisões dentro da empresa
- **jobs**, que armazena os cargos dentro da empresa
- **employees2**, que armazena os funcionários

Essas tabelas são criadas pelo script `store_schema.sql`. A tabela **divisions** é criada usando a seguinte instrução:

```
CREATE TABLE divisions (
    division_id CHAR(3)
    CONSTRAINT divisions_pk PRIMARY KEY,
    name VARCHAR2(15) NOT NULL
);
```

A consulta a seguir recupera as linhas da tabela **divisions**:

```
SELECT *
FROM divisions;

DIV NAME
---
SAL Sales
```



OPE Operations  
SUP Support  
BUS Business

A tabela jobs é criada usando a seguinte instrução:

```
CREATE TABLE jobs (  
  job_id CHAR(3)  
  CONSTRAINT jobs_pk PRIMARY KEY,  
  name VARCHAR2(20) NOT NULL  
);
```

A próxima consulta recupera as linhas da tabela jobs:

```
SELECT *  
FROM jobs;  
  
JOB NAME  
--- -----  
WOR Worker  
MGR Manager  
ENG Engineer  
TEC Technologist  
PRE President
```

A tabela employees2 é criada usando a seguinte instrução:

```
CREATE TABLE employees2 (  
  employee_id INTEGER  
  CONSTRAINT employees2_pk PRIMARY KEY,  
  division_id CHAR(3)  
  CONSTRAINT employees2_fk_divisions  
  REFERENCES divisions(division_id),  
  job_id CHAR(3) REFERENCES jobs(job_id),  
  first_name VARCHAR2(10) NOT NULL,  
  last_name VARCHAR2(10) NOT NULL,  
  salary NUMBER(6, 0)  
);
```

A consulta a seguir recupera as cinco primeiras linhas da tabela employees2:

```
SELECT *  
FROM employees2  
WHERE ROWNUM <= 5;
```

EMPLOYEE_ID	DIV	JOB	FIRST_NAME	LAST_NAME	SALARY
1	BUS	PRE	James	Smith	800000
2	SAL	MGR	Ron	Johnson	350000
3	SAL	WOR	Fred	Hobbs	140000
4	SUP	MGR	Susan	Jones	200000
5	SAL	WOR	Rob	Green	350000

## USANDO A CLÁUSULA ROLLUP

A cláusula `ROLLUP` estende `GROUP BY` para retornar uma linha contendo um subtotal para cada grupo de linhas, além de uma linha contendo um total para todos os grupos. Conforme foi visto no Capítulo 4, `GROUP BY` agrupa linhas em blocos com um valor de coluna comum. Por exemplo, a consulta a seguir usa `GROUP BY` para agrupar as linhas da tabela `employees2` por `department_id` e usa `SUM()` a fim de obter a soma dos salários para cada `division_id`:

```
SELECT division_id, SUM(salary)
FROM employees2
GROUP BY division_id
ORDER BY division_id;
```

DIV	SUM(SALARY)
BUS	1610000
OPE	1320000
SAL	4936000
SUP	1015000

### Passando uma única coluna para ROLLUP

A consulta a seguir reescreve o exemplo anterior para usar `ROLLUP`; observe a linha ao final, que contém os salários totais de todos os grupos:

```
SELECT division_id, SUM(salary)
FROM employees2
GROUP BY ROLLUP (division_id)
ORDER BY division_id;
```

DIV	SUM(SALARY)
BUS	1610000
OPE	1320000
SAL	4936000
SUP	1015000
	8881000

#### NOTA

*Se você precisa das linhas em uma ordem específica, deve usar uma cláusula `ORDER BY`. Você precisa fazer isso apenas para o caso de a Oracle Corporation decidir alterar a ordem padrão das linhas retornadas por `ROLLUP`.*

### Passando várias colunas para ROLLUP

É possível passar várias colunas para `ROLLUP` que, então, agrupa as linhas em blocos com os mesmos valores de coluna. O exemplo a seguir passa as colunas `division_id` e `job_id` da tabela `employees2` para `ROLLUP`, que agrupa as linhas por essas colunas; na saída, observe que os salários

são somados por `division_id` e `job_id` e que `ROLLUP` retorna uma linha com a soma dos salários em cada `division_id`, além de uma linha no final com o total geral dos salários:

```
SELECT division_id, job_id, SUM(salary)
FROM employees2
GROUP BY ROLLUP(division_id, job_id)
ORDER BY division_id, job_id;
```

```
DIV JOB  SUM(SALARY)
--- ---  -
BUS MGR      530000
BUS PRE      800000
BUS WOR      280000
BUS          1610000
OPE ENG      245000
OPE MGR      805000
OPE WOR      270000
OPE          1320000
SAL MGR      4446000
SAL WOR      490000
SAL          4936000
SUP MGR      465000
SUP TEC      115000
SUP WOR      435000
SUP          1015000
              8881000
```

### ***Alterando a posição das colunas passadas para ROLLUP***

O exemplo a seguir inverte `division_id` e `job_id`; isso faz `ROLLUP` calcular a soma dos salários para cada `job_id`:

```
SELECT job_id, division_id, SUM(salary)
FROM employees2
GROUP BY ROLLUP(job_id, division_id)
ORDER BY job_id, division_id;
```

```
JOB DIV  SUM(SALARY)
--- ---  -
ENG OPE      245000
ENG          245000
MGR BUS      530000
MGR OPE      805000
MGR SAL      4446000
MGR SUP      465000
MGR          6246000
PRE BUS      800000
PRE          800000
TEC SUP      115000
TEC          115000
WOR BUS      280000
WOR OPE      270000
```

```

WOR SAL      490000
WOR SUP      435000
WOR          1475000
            8881000

```

### Usando outras funções agregadas com ROLLUP

Você pode usar qualquer uma das funções agregadas com ROLLUP (para ver uma lista das principais funções agregadas, consulte a Tabela 4-8 no Capítulo 4). O exemplo a seguir usa AVG() para calcular a média de salários:

```

SELECT division_id, job_id, AVG(salary)
FROM employees2
GROUP BY ROLLUP(division_id, job_id)
ORDER BY division_id, job_id;

```

```

DIV JOB  AVG(SALARY)
--- ---  -
BUS MGR   176666.667
BUS PRE    800000
BUS WOR    280000
BUS       322000
OPE ENG    245000
OPE MGR    201250
OPE WOR    135000
OPE       188571.429
SAL MGR    261529.412
SAL WOR    245000
SAL       259789.474
SUP MGR    232500
SUP TEC    115000
SUP WOR    145000
SUP       169166.667
          240027.027

```

### Usando a cláusula CUBE

A cláusula CUBE estende GROUP BY para retornar as linhas que contêm um subtotal para todas as combinações de colunas, além de uma linha contendo o total geral. O exemplo a seguir passa division\_id e job\_id para CUBE, que agrupa as linhas por essas colunas:

```

SELECT division_id, job_id, SUM(salary)
FROM employees2
GROUP BY CUBE(division_id, job_id)
ORDER BY division_id, job_id;

```

```

DIV JOB  SUM(SALARY)
--- ---  -
BUS MGR   530000
BUS PRE    800000
BUS WOR    280000
BUS       1610000

```

OPE ENG	245000
OPE MGR	805000
OPE WOR	270000
OPE	1320000
SAL MGR	4446000
SAL WOR	490000
SAL	4936000
SUP MGR	465000
SUP TEC	115000
SUP WOR	435000
SUP	1015000
ENG	245000
MGR	6246000
PRE	800000
TEC	115000
WOR	1475000
	8881000

Note que os salários são somados por `division_id` e `job_id`. CUBE retorna uma linha com a soma dos salários para cada `division_id`, junto com a soma de todos os salários para cada `job_id` ao final, juntamente com uma linha que contém o total geral dos salários. O exemplo a seguir troca `division_id` e `job_id`:

```
SELECT job_id, division_id, SUM(salary)
FROM employees2
GROUP BY CUBE(job_id, division_id)
ORDER BY job_id, division_id;
```

JOB	DIV	SUM(SALARY)
---	---	-----
ENG	OPE	245000
ENG		245000
MGR	BUS	530000
MGR	OPE	805000
MGR	SAL	4446000
MGR	SUP	465000
MGR		6246000
PRE	BUS	800000
PRE		800000
TEC	SUP	115000
TEC		115000
WOR	BUS	280000
WOR	OPE	270000
WOR	SAL	490000
WOR	SUP	435000
WOR		1475000
	BUS	1610000
	OPE	1320000
	SAL	4936000
	SUP	1015000
		8881000

## Usando a função GROUPING()

A função `GROUPING()` aceita uma coluna e retorna 0 ou 1. `GROUPING()` retorna 1 quando o valor da coluna é nulo e retorna 0 quando o valor da coluna não é nulo. `GROUPING()` é usada somente em consultas que utilizam `ROLLUP` ou `CUBE`. `GROUPING()` é útil quando você quer exibir um valor no caso em que, de outro modo, seria retornado um valor nulo.

### Usando GROUPING() com uma única coluna em ROLLUP

Como foi visto na seção “Passando uma única coluna para `ROLLUP`”, a última linha no conjunto de resultados do exemplo continha o total dos salários:

```
SELECT division_id, SUM(salary)
FROM employees2
GROUP BY ROLLUP(division_id)
ORDER BY division_id;
```

```
DIV SUM(SALARY)
-----
BUS      1610000
OPE      1320000
SAL      4936000
SUP      1015000
          8881000
```

A coluna `division_id` da última linha é nula. Você pode usar a função `GROUPING()` para determinar se essa coluna é nula, como mostrado na consulta a seguir. Observe que `GROUPING()` retorna 0 para as linhas que têm valores de `division_id` não nulos e retorna 1 para a última linha, que tem um valor de `division_id` nulo:

```
SELECT GROUPING(division_id), division_id, SUM(salary)
FROM employees2
GROUP BY ROLLUP(division_id)
ORDER BY division_id;
```

```
GROUPING(DIVISION_ID) DIV SUM(SALARY)
-----
0 BUS      1610000
0 OPE      1320000
0 SAL      4936000
0 SUP      1015000
1          8881000
```

### Usando CASE para converter o valor retornado de GROUPING()

Você pode usar a expressão `CASE` para converter o valor 1 do exemplo anterior em um valor significativo. O exemplo a seguir usa `CASE` para converter 1 na string 'All divisions':

```
SELECT
CASE GROUPING(division_id)
WHEN 1 THEN 'All divisions'
ELSE division_id
END AS div,
SUM(salary)
```

```
FROM employees2
GROUP BY ROLLUP (division_id)
ORDER BY division_id;
```

DIV	SUM(SALARY)
-----	-----
BUS	1610000
OPE	1320000
SAL	4936000
SUP	1015000
All divisions	8881000

**Usando CASE e GROUPING() para converter vários valores de coluna**

O exemplo a seguir amplia a idéia de substituição de valores nulos para uma cláusula ROLLUP contendo várias colunas (division\_id e job\_id); observe que os valores de division\_id nulos são substituídos pela string 'All divisions' e que os valores de job\_id nulos são substituídos por 'All jobs':

```
SELECT
  CASE GROUPING (division_id)
    WHEN 1 THEN 'All divisions'
    ELSE division_id
  END AS div,
  CASE GROUPING (job_id)
    WHEN 1 THEN 'All jobs'
    ELSE job_id
  END AS job,
  SUM (salary)
FROM employees2
GROUP BY ROLLUP (division_id, job_id)
ORDER BY division_id, job_id;
```

DIV	JOB	SUM(SALARY)
-----	-----	-----
BUS	MGR	530000
BUS	PRE	800000
BUS	WOR	280000
BUS	All jobs	1610000
OPE	ENG	245000
OPE	MGR	805000
OPE	WOR	270000
OPE	All jobs	1320000
SAL	MGR	4446000
SAL	WOR	490000
SAL	All jobs	4936000
SUP	MGR	465000
SUP	TEC	115000
SUP	WOR	435000
SUP	All jobs	1015000
All divisions	All jobs	8881000

**Usando GROUPING() com CUBE**

Você pode usar a função GROUPING() com CUBE, como neste exemplo:

```
SELECT
  CASE GROUPING(division_id)
    WHEN 1 THEN 'All divisions'
    ELSE division_id
  END AS div,
  CASE GROUPING(job_id)
    WHEN 1 THEN 'All jobs'
    ELSE job_id
  END AS job,
  SUM(salary)
FROM employees2
GROUP BY CUBE(division_id, job_id)
ORDER BY division_id, job_id;
```

DIV	JOB	SUM(SALARY)
-----	-----	-----
BUS	MGR	530000
BUS	PRE	800000
BUS	WOR	280000
BUS	All jobs	1610000
OPE	ENG	245000
OPE	MGR	805000
OPE	WOR	270000
OPE	All jobs	1320000
SAL	MGR	4446000
SAL	WOR	490000
SAL	All jobs	4936000
SUP	MGR	465000
SUP	TEC	115000
SUP	WOR	435000
SUP	All jobs	1015000
All divisions	ENG	245000
All divisions	MGR	6246000
All divisions	PRE	800000
All divisions	TEC	115000
All divisions	WOR	1475000
All divisions	All jobs	8881000

Usando a cláusula GROUPING SETS

Você usa a cláusula GROUPING SETS para obter apenas as linhas de subtotal. O exemplo a seguir usa GROUPING SETS para obter os subtotais dos salários por division\_id e job\_id:

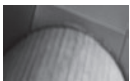
```
SELECT division_id, job_id, SUM(salary)
FROM employees2
GROUP BY GROUPING SETS(division_id, job_id)
ORDER BY division_id, job_id;
```

DIV	JOB	SUM(SALARY)
---	---	---
BUS		1610000
OPE		1320000



SAL	4936000
SUP	1015000
ENG	245000
MGR	6246000
PRE	800000
TEC	115000
WOR	1475000

Note que são retornados somente os subtotais para as colunas `division_id` e `job_id`; o total de todos os salários não é retornado. Você vai aprender a obter o total e os subtotais usando a função `GROUPING_ID()` na próxima seção.



#### DICA

Normalmente, a cláusula `GROUPING SETS` oferece melhor desempenho do que `CUBE`. Portanto, quando possível, você deve usar `GROUPING SETS` em vez de `CUBE`.

## Usando a função `GROUPING_ID()`

É possível usar a função `GROUPING_ID()` para filtrar linhas, utilizando uma cláusula `HAVING` para excluir aquelas que não contêm um subtotal ou total. A função `GROUPING_ID()` aceita uma ou mais colunas e retorna o equivalente decimal do vetor de bit `GROUPING`. O vetor de bit `GROUPING` é calculado combinando-se os resultados de uma chamada da função `GROUPING()` para cada coluna, em ordem.

### Calculando o vetor de bit `GROUPING`

Na seção “Usando a função `GROUPING()`”, você viu que `GROUPING()` retorna 1 quando o valor da coluna é nulo e retorna 0 quando o valor da coluna não é nulo; por exemplo:

- Se `division_id` e `job_id` são não nulos, `GROUPING()` retorna 0 para as duas colunas. O resultado de `division_id` é combinado com o resultado de `job_id`, fornecendo um vetor de bit igual a 00, cujo equivalente decimal é 0. Portanto, `GROUPING_ID()` retorna 0 quando `division_id` e `job_id` não são nulos.
- Se `division_id` não é nulo (o bit `GROUPING` é 0), mas `job_id` é nulo (o bit `GROUPING` é 1), o vetor de bit resultante é 01 e `GROUPING_ID()` retorna 1.
- Se `division_id` é nulo (o bit `GROUPING` é 1), mas `job_id` não é nulo (o bit `GROUPING` é 0), o vetor de bit resultante é 10 e `GROUPING_ID()` retorna 2.
- Se `division_id` e `job_id` são nulos (os dois bits `GROUPING` são 1), o vetor de bit é 11 e `GROUPING_ID()` retorna 3.

A tabela a seguir resume esses resultados.

<code>division_id</code>	<code>job_id</code>	Vetor de bit	Valor de retorno de <code>GROUPING_ID()</code>
não nulo	não nulo	00	0
não nulo	nulo	01	1
nulo	não nulo	10	2
nulo	nulo	11	3

Exemplo de consulta que ilustra o uso de GROUPING\_ID()

O exemplo a seguir passa division\_id e job\_id para GROUPING\_ID(); observe que a saída da função GROUPING\_ID() está de acordo com os valores retornados esperados, documentados na seção anterior:

```
SELECT
    division_id, job_id,
    GROUPING(dimension_id) AS DIV_GRP,
    GROUPING(job_id) AS JOB_GRP,
    GROUPING_ID(dimension_id, job_id) AS grp_id,
    SUM(salary)
FROM employees2
GROUP BY CUBE(dimension_id, job_id)
ORDER BY dimension_id, job_id;
```

DIV JOB	DIV_GRP	JOB_GRP	GRP_ID	SUM(SALARY)
BUS MGR	0	0	0	530000
BUS PRE	0	0	0	800000
BUS WOR	0	0	0	280000
BUS	0	1	1	1610000
OPE ENG	0	0	0	245000
OPE MGR	0	0	0	805000
OPE WOR	0	0	0	270000
OPE	0	1	1	1320000
SAL MGR	0	0	0	4446000
SAL WOR	0	0	0	490000
SAL	0	1	1	4936000
SUP MGR	0	0	0	465000
SUP TEC	0	0	0	115000
SUP WOR	0	0	0	435000
SUP	0	1	1	1015000
ENG	1	0	2	245000
MGR	1	0	2	6246000
PRE	1	0	2	800000
TEC	1	0	2	115000
WOR	1	0	2	1475000
	1	1	3	8881000

Uma aplicação útil de GROUPING\_ID()

Uma aplicação útil de GROUPING\_ID() é a filtragem de linhas usando uma cláusula HAVING. A cláusula HAVING pode excluir as linhas que não contêm um subtotal um ou total, simplesmente verificando se GROUPING\_ID() retorna um valor maior do que 0. Por exemplo:

```
SELECT
    division_id, job_id,
    GROUPING_ID(dimension_id, job_id) AS grp_id,
    SUM(salary)
FROM employees2
GROUP BY CUBE(dimension_id, job_id)
```

```
HAVING GROUPING_ID(division_id, job_id) > 0
ORDER BY division_id, job_id;
```

DIV	JOB	GRP_ID	SUM(SALARY)
BUS		1	1610000
OPE		1	1320000
SAL		1	4936000
SUP		1	1015000
ENG		2	245000
MGR		2	6246000
PRE		2	800000
TEC		2	115000
WOR		2	1475000
		3	8881000

## Usando uma coluna várias vezes em uma cláusula GROUP BY

Você pode usar uma coluna muitas vezes em uma cláusula GROUP BY. Isso permite reorganizar seus dados ou apresentar diferentes agrupamentos de dados. Por exemplo, a consulta a seguir contém uma cláusula GROUP BY que usa division\_id duas vezes, uma para agrupar por division\_id e novamente em uma cláusula ROLLUP:

```
SELECT division_id, job_id, SUM(salary)
FROM employees2
GROUP BY division_id, ROLLUP(division_id, job_id);
```

DIV	JOB	SUM(SALARY)
BUS	MGR	530000
BUS	PRE	800000
BUS	WOR	280000
OPE	ENG	245000
OPE	MGR	805000
OPE	WOR	270000
SAL	MGR	4446000
SAL	WOR	490000
SUP	MGR	465000
SUP	TEC	115000
SUP	WOR	435000
BUS		1610000
OPE		1320000
SAL		4936000
SUP		1015000
BUS		1610000
OPE		1320000
SAL		4936000
SUP		1015000

Note, contudo, que as últimas quatro linhas são duplicatas das quatro linhas anteriores. É possível eliminar essas duplicatas usando a função GROUP\_ID(), que será abordada a seguir.

## Usando a função GROUP\_ID()

Você pode usar a função `GROUP_ID()` para remover linhas duplicadas retornadas por uma cláusula `GROUP BY`. Essa função não aceita parâmetros. Se existem  $n$  duplicatas em um agrupamento em particular, `GROUP_ID` retorna números no intervalo de 0 a  $n - 1$ . O exemplo a seguir reescreve a consulta mostrada na seção anterior para incluir a saída de `GROUP_ID()`; observe que `GROUP_ID()` retorna 0 para todas as linhas, exceto para as quatro últimas, que são duplicatas das quatro linhas anteriores, e que `GROUP_ID()` retorna 1:

```
SELECT division_id, job_id, GROUP_ID(), SUM(salary)
FROM employees2
GROUP BY division_id, ROLLUP(division_id, job_id);
```

DIV	JOB	GROUP_ID()	SUM(SALARY)
BUS	MGR	0	530000
BUS	PRE	0	800000
BUS	WOR	0	280000
OPE	ENG	0	245000
OPE	MGR	0	805000
OPE	WOR	0	270000
SAL	MGR	0	4446000
SAL	WOR	0	490000
SUP	MGR	0	465000
SUP	TEC	0	115000
SUP	WOR	0	435000
BUS		0	1610000
OPE		0	1320000
SAL		0	4936000
SUP		0	1015000
BUS		1	1610000
OPE		1	1320000
SAL		1	4936000
SUP		1	1015000

Você pode eliminar linhas duplicadas usando uma cláusula `HAVING` que só permita linhas cujo valor de `GROUP_ID()` seja 0, por exemplo:

```
SELECT division_id, job_id, GROUP_ID(), SUM(salary)
FROM employees2
GROUP BY division_id, ROLLUP(division_id, job_id)
HAVING GROUP_ID() = 0;
```

DIV	JOB	GROUP_ID()	SUM(SALARY)
BUS	MGR	0	530000
BUS	PRE	0	800000
BUS	WOR	0	280000
OPE	ENG	0	245000
OPE	MGR	0	805000
OPE	WOR	0	270000

SAL MGR	0	4446000
SAL WOR	0	490000
SUP MGR	0	465000
SUP TEC	0	115000
SUP WOR	0	435000
BUS	0	1610000
OPE	0	1320000
SAL	0	4936000
SUP	0	1015000

Isso conclui a discussão sobre cláusulas GROUP BY estendidas.

## USANDO AS FUNÇÕES ANALÍTICAS

O banco de dados tem muitas funções analíticas internas que permitem efetuar cálculos complexos, como encontrar o tipo de produto mais vendido em cada mês, os principais vendedores etc. As funções analíticas são organizadas nas seguintes categorias:

- As **funções de classificação** permitem calcular classificações, percentis e *n*-is (tercis, quartis etc.).
- As **funções de percentil inversas** permitem calcular o valor corresponde a um percentil.
- As **funções de janela** permitem calcular agregados acumulados e móveis.
- As **funções de relatório** permitem calcular fatias de mercado, por exemplo.
- As **funções de defasagem e avanço** permitem obter um valor em uma linha, onde essa linha está certo número de linhas distante da linha atual.
- As **funções de primeiro e último** permitem obter o primeiro e o último valor em um grupo ordenado.
- As **funções de regressão linear** permitem ajustar uma linha de regressão dos mínimos quadrados normais a um conjunto de pares numéricos.
- As **função de classificação hipotética e distribuição** permitem calcular a classificação e o percentil que uma nova linha teria se você a inserisse em uma tabela.

Você vai aprender sobre essas funções em breve, mas primeiro vamos examinar a tabela de exemplo.

### A tabela de exemplo

A tabela `all_sales` armazena a soma de todas as vendas por valor em dólares para um ano, mês, tipo de produto e funcionário específicos. A tabela `all_sales` é criada pelo script `store_schema.sql`, como segue:

```
CREATE TABLE all_sales (
  year INTEGER NOT NULL,
  month INTEGER NOT NULL,
  prd_type_id INTEGER
  CONSTRAINT all_sales_fk_product_types
  REFERENCES product_types(product_type_id),
  emp_id INTEGER
```

```
CONSTRAINT all_sales_fk_employees2
REFERENCES employees2(employee_id),
amount NUMBER(8, 2),
CONSTRAINT all_sales_pk PRIMARY KEY (
year, month, prd_type_id, emp_id
)
);
```

A tabela `all_sales` contém cinco colunas:

- **YEAR** armazena o ano em que as vendas ocorreram
- **MONTH** armazena o mês em que as vendas ocorreram (de 1 a 12)
- **PRD\_TYPE\_ID** armazena o valor de `product_type_id` do produto
- **EMP\_ID** armazena o valor de `employee_id` do funcionário que realizou as vendas
- **AMOUNT** armazena o valor total em dólares das vendas

A consulta a seguir recupera as 12 primeiras linhas da tabela `all_sales`:

```
SELECT *
FROM all_sales
WHERE ROWNUM <= 12;
```

YEAR	MONTH	PRD_TYPE_ID	EMP_ID	AMOUNT
2003	1	1	21	10034.84
2003	2	1	21	15144.65
2003	3	1	21	20137.83
2003	4	1	21	25057.45
2003	5	1	21	17214.56
2003	6	1	21	15564.64
2003	7	1	21	12654.84
2003	8	1	21	17434.82
2003	9	1	21	19854.57
2003	10	1	21	21754.19
2003	11	1	21	13029.73
2003	12	1	21	10034.84

**NOTA**

*Na verdade, a tabela `all_sales` contém muito mais linhas do que isso, mas por limitações de espaço não listamos todas elas aqui.*

Vamos examinar as funções de classificação a seguir.

**Usando as funções de classificação**

As funções de classificação calculam classificações, percentis e *n*-is. Elas estão mostradas na Tabela 7-2. Vamos começar examinando as funções `RANK()` e `DENSE_RANK()`.

**Tabela 7-2** As funções de classificação

Função	Descrição
<code>RANK()</code>	Retorna a classificação dos itens em um grupo. <code>RANK()</code> deixa uma lacuna na sequência de classificações para o caso de um empate.
<code>DENSE_RANK()</code>	Retorna a classificação dos itens em um grupo. <code>DENSE_RANK()</code> não deixa uma lacuna na sequência de classificações para o caso de um empate.
<code>CUME_DIST()</code>	Retorna a posição de um valor especificado em relação a um grupo de valores. <code>CUME_DIST()</code> é a abreviação de distribuição acumulada, em inglês.
<code>PERCENT_RANK()</code>	Retorna a classificação percentual de um valor em relação a um grupo de valores.
<code>NTILE()</code>	Retorna <i>n</i> -is: tercís, quartís etc.
<code>ROW_NUMBER()</code>	Retorna um número com cada linha em um grupo.

**Usando as funções `RANK()` e `DENSE_RANK()`**

Você usa `RANK()` e `DENSE_RANK()` para classificar itens em um grupo. A diferença entre essas duas funções está na maneira como elas tratam dos itens que empatam: `RANK()` deixa uma lacuna na sequência, já `DENSE_RANK()` não deixa lacunas. Por exemplo, se você classificasse as vendas pelo tipo de produto e dois tipos de produto empatassem em primeiro lugar, `RANK()` colocaria os dois tipos de produto no primeiro lugar, mas o tipo de produto seguinte ficaria em terceiro lugar. `DENSE_RANK()` também colocaria os dois tipos de produto em primeiro lugar, mas o tipo de produto seguinte ficaria em segundo.

A consulta a seguir ilustra o uso de `RANK()` e `DENSE_RANK()` para obter a classificação das vendas pelo tipo de produto para o ano de 2003; observe o uso da palavra-chave `OVER` na sintaxe, na chamada das funções `RANK()` e `DENSE_RANK()`:

```
SELECT
    prd_type_id, SUM(amount),
    RANK() OVER (ORDER BY SUM(amount) DESC) AS rank,
    DENSE_RANK() OVER (ORDER BY SUM(amount) DESC) AS dense_rank
FROM all_sales
WHERE year = 2003
AND amount IS NOT NULL
GROUP BY prd_type_id
ORDER BY prd_type_id;
```

PRD_TYPE_ID	SUM(AMOUNT)	RANK	DENSE_RANK
1	905081.84	1	1
2	186381.22	4	4
3	478270.91	2	2
4	402751.16	3	3

Note que as vendas do tipo de produto nº 1 são classificadas em primeiro lugar, as vendas do tipo de produto nº 2 são classificadas em quarto e assim por diante. Como não existem empates, `RANK()` e `DENSE_RANK()` retornam as mesmas classificações.

A tabela `all_sales` contém valores nulos na coluna `AMOUNT` para todas as linhas cuja coluna `PRD_TYPE_ID` é 5; a consulta anterior omite essas linhas por causa da inclusão da linha `"AND amount IS NOT NULL"` na cláusula `WHERE`. O exemplo a seguir inclui essas linhas, excluindo a linha `AND` da cláusula `WHERE`:

```
SELECT
    prd_type_id, SUM(amount),
    RANK() OVER (ORDER BY SUM(amount) DESC) AS rank,
    DENSE_RANK() OVER (ORDER BY SUM(amount) DESC) AS dense_rank
FROM all_sales
WHERE year = 2003
GROUP BY prd_type_id
ORDER BY prd_type_id;
```

PRD_TYPE_ID	SUM(AMOUNT)	RANK	DENSE_RANK
1	905081.84	2	2
2	186381.22	5	5
3	478270.91	3	3
4	402751.16	4	4
5		1	1

Note que a última linha contém um valor nulo para a soma da coluna `AMOUNT` e que `RANK()` e `DENSE_RANK()` retornam 1 para essa linha. Isso acontece porque, por padrão, `RANK()` e `DENSE_RANK()` atribuem a classificação mais alta, que é 1, para os valores nulos em classificações decrescentes (isto é, `DESC` é usado na cláusula `OVER`) e a classificação mais baixa em classificações crescentes (isto é, `ASC` é usado na cláusula `OVER`).

**Controlando a classificação de valores nulos com as cláusulas `NULLS FIRST` e `NULLS LAST`** Com uma função analítica, você pode controlar explicitamente se os valores nulos são os mais altos ou os mais baixos em um grupo usando `NULLS FIRST` ou `NULLS LAST`. O exemplo a seguir usa `NULLS LAST` para especificar que os valores nulos são os mais baixos:

```
SELECT
    prd_type_id, SUM(amount),
    RANK() OVER (ORDER BY SUM(amount) DESC NULLS LAST) AS rank,
    DENSE_RANK() OVER (ORDER BY SUM(amount) DESC NULLS LAST) AS dense_rank
FROM all_sales
WHERE year = 2003
GROUP BY prd_type_id
ORDER BY prd_type_id;
```

PRD_TYPE_ID	SUM(AMOUNT)	RANK	DENSE_RANK
1	905081.84	1	1
2	186381.22	4	4
3	478270.91	2	2
4	402751.16	3	3
5		5	5



**Usando a cláusula PARTITION BY com funções analíticas** Você usa a cláusula PARTITION BY com as funções analíticas quando precisa dividir os grupos em subgrupos. Por exemplo, se você precisa subdividir o valor das vendas por mês, pode usar PARTITION BY month, como mostrado na consulta a seguir:

```
SELECT
    prd_type_id, month, SUM(amount),
    RANK() OVER (PARTITION BY month ORDER BY SUM(amount) DESC) AS rank
FROM all_sales
WHERE year = 2003
AND amount IS NOT NULL
GROUP BY prd_type_id, month
ORDER BY prd_type_id, month;
```

PRD_TYPE_ID	MONTH	SUM(AMOUNT)	RANK
1	1	38909.04	1
1	2	70567.9	1
1	3	91826.98	1
1	4	120344.7	1
1	5	97287.36	1
1	6	57387.84	1
1	7	60929.04	2
1	8	75608.92	1
1	9	85027.42	1
1	10	105305.22	1
1	11	55678.38	1
1	12	46209.04	2
2	1	14309.04	4
2	2	13367.9	4
2	3	16826.98	4
2	4	15664.7	4
2	5	18287.36	4
2	6	14587.84	4
2	7	15689.04	3
2	8	16308.92	4
2	9	19127.42	4
2	10	13525.14	4
2	11	16177.84	4
2	12	12509.04	4
3	1	24909.04	2
3	2	15467.9	3
3	3	20626.98	3
3	4	23844.7	2
3	5	18687.36	3
3	6	19887.84	3
3	7	81589.04	1
3	8	62408.92	2
3	9	46127.42	3
3	10	70325.29	3
3	11	46187.38	2
3	12	48209.04	1

4	1	17398.43	3
4	2	17267.9	2
4	3	31026.98	2
4	4	16144.7	3
4	5	20087.36	2
4	6	33087.84	2
4	7	12089.04	4
4	8	58408.92	3
4	9	49327.42	2
4	10	75325.14	2
4	11	42178.38	3
4	12	30409.05	3

**Usando os operadores ROLLUP, CUBE e GROUPING SETS com funções analíticas** Você pode usar os operadores ROLLUP, CUBE e GROUPING SETS com as funções analíticas. A consulta a seguir usa ROLLUP e RANK() para obter as classificações das vendas por identificação de tipo de produto:

```
SELECT
    prd_type_id, SUM(amount),
    RANK() OVER (ORDER BY SUM(amount) DESC) AS rank
FROM all_sales
WHERE year = 2003
GROUP BY ROLLUP(prd_type_id)
ORDER BY prd_type_id;
```

PRD_TYPE_ID	SUM(AMOUNT)	RANK
1	905081.84	3
2	186381.22	6
3	478270.91	4
4	402751.16	5
5		1
	1972485.13	2

A próxima consulta usa CUBE e RANK() para obter todas as classificações de vendas por identificação de tipo de produto e identificação de funcionário:

```
SELECT
    prd_type_id, emp_id, SUM(amount),
    RANK() OVER (ORDER BY SUM(amount) DESC) AS rank
FROM all_sales
WHERE year = 2003
GROUP BY CUBE(prd_type_id, emp_id)
ORDER BY prd_type_id, emp_id;
```

PRD_TYPE_ID	EMP_ID	SUM(AMOUNT)	RANK
1	21	197916.96	19
1	22	214216.96	17
1	23	98896.96	26
1	24	207216.96	18
1	25	93416.96	28
1	26	93417.04	27

1		905081.84	9
2	21	20426.96	40
2	22	19826.96	41
2	23	19726.96	42
2	24	43866.96	34
2	25	32266.96	38
2	26	50266.42	31
2		186381.22	21
3	21	140326.96	22
3	22	116826.96	23
3	23	112026.96	24
3	24	34829.96	36
3	25	29129.96	39
3	26	45130.11	33
3		478270.91	10
4	21	108326.96	25
4	22	81426.96	30
4	23	92426.96	29
4	24	47456.96	32
4	25	33156.96	37
4	26	39956.36	35
4		402751.16	13
5	21		1
5	22		1
5	23		1
5	24		1
5	25		1
5	26		1
5			1
	21	466997.84	11
	22	432297.84	12
	23	323077.84	15
	24	333370.84	14
	25	187970.84	20
	26	228769.93	16
		1972485.13	8

A próxima consulta usa `GROUPING SETS` e `RANK()` para obter apenas as classificações de subtotal do valor das vendas:

```
SELECT
    prd_type_id, emp_id, SUM(amount),
    RANK() OVER (ORDER BY SUM(amount) DESC) AS rank
FROM all_sales
WHERE year = 2003
GROUP BY GROUPING SETS(prd_type_id, emp_id)
ORDER BY prd_type_id, emp_id;
```

PRD_TYPE_ID	EMP_ID	SUM(AMOUNT)	RANK
1		905081.84	2
2		186381.22	11

3	478270.91	3
4	402751.16	6
5		1
21	466997.84	4
22	432297.84	5
23	323077.84	8
24	333370.84	7
25	187970.84	10
26	228769.93	9

### Usando as funções CUME\_DIST() e PERCENT\_RANK()

CUME\_DIST() calcula a posição de um valor especificado em relação a um grupo de valores; CUME\_DIST() é a abreviação de distribuição acumulada, em inglês. PERCENT\_RANK() calcula a classificação percentual de um valor em relação a um grupo de valores. A consulta a seguir ilustra o uso de CUME\_DIST() e PERCENT\_RANK() para obter a distribuição acumulada e a classificação percentual das vendas:

```
SELECT
    prd_type_id, SUM(amount),
    CUME_DIST() OVER (ORDER BY SUM(amount) DESC) AS cume_dist,
    PERCENT_RANK() OVER (ORDER BY SUM(amount) DESC) AS percent_rank
FROM all_sales
WHERE year = 2003
GROUP BY prd_type_id
ORDER BY prd_type_id;
```

PRD_TYPE_ID	SUM(AMOUNT)	CUME_DIST	PERCENT_RANK
1	905081.84	.4	.25
2	186381.22	1	1
3	478270.91	.6	.5
4	402751.16	.8	.75
5		.2	0

### Usando a função NTILE()

NTILE(*depósitos*) calcula *n*-is (tercis, quartis etc.); *depósitos* especifica o número de “depósitos” nos quais os grupos de linhas são colocados. Por exemplo, NTILE(2) especifica dois depósitos e, portanto, divide as linhas em dois grupos; NTILE(4) divide os grupos em quatro depósitos e, portanto, divide as linhas em quatro grupos. A consulta a seguir ilustra o uso de NTILE(); observe que 4 é passado para NTILE() para dividir os grupos de linhas em quatro depósitos:

```
SELECT
    prd_type_id, SUM(amount),
    NTILE(4) OVER (ORDER BY SUM(amount) DESC) AS ntile
FROM all_sales
WHERE year = 2003
AND amount IS NOT NULL
GROUP BY prd_type_id
ORDER BY prd_type_id;
```

PRD_TYPE_ID	SUM(AMOUNT)	NTILE
1	905081.84	1
2	186381.22	4
3	478270.91	2
4	402751.16	3

### Usando a função ROW\_NUMBER()

ROW\_NUMBER() retorna um número com cada linha em um grupo, começando em 1. A consulta a seguir ilustra o uso de ROW\_NUMBER():

```
SELECT
    prd_type_id, SUM(amount),
    ROW_NUMBER() OVER (ORDER BY SUM(amount) DESC) AS row_number
FROM all_sales
WHERE year = 2003
GROUP BY prd_type_id
ORDER BY prd_type_id;
```

PRD_TYPE_ID	SUM(AMOUNT)	ROW_NUMBER
1	905081.84	2
2	186381.22	5
3	478270.91	3
4	402751.16	4
5		1

Isso conclui a discussão sobre funções de classificação.

### Usando as funções de percentil inversas

Na seção “Usando as funções CUME\_DIST() e PERCENT\_RANK()”, você viu que CUME\_DIST() calcula a posição de um valor especificado em relação a um grupo de valores. Também viu que PERCENT\_RANK() calcula a classificação percentual de um valor em relação a um grupo de valores.

Nesta seção, você verá como utilizar as funções de percentil inversas para obter o valor correspondente a um percentil. Existem duas funções de percentil inversas: PERCENTILE\_DISC(x) e PERCENTILE\_CONT(x). Elas funcionam de maneira inversa a CUME\_DIST() e PERCENT\_RANK(). PERCENTILE\_DISC(x) examina os valores da distribuição acumulada em cada grupo até encontrar um que seja maior ou igual a x. PERCENTILE\_CONT(x) examina os valores da classificação percentual em cada grupo até encontrar um que seja maior ou igual a x.

A consulta a seguir ilustra o uso de PERCENTILE\_CONT() e PERCENTILE\_DISC() para obter a soma do valor cujo percentil é maior ou igual a 0,6:

```
SELECT
    PERCENTILE_CONT(0.6) WITHIN GROUP (ORDER BY SUM(amount) DESC)
    AS percentil_cont,
    PERCENTILE_DISC(0.6) WITHIN GROUP (ORDER BY SUM(amount) DESC)
    AS percentil_disc
FROM all_sales
WHERE year = 2003
GROUP BY prd_type_id;
```

PERCENTILE_CONT	PERCENTILE_DISC
417855.11	402751.16

Se você comparar a soma dos valores mostrada nesses resultados com aquela mostrada na seção anterior, verá que as somas correspondem àquelas cuja distribuição acumulada e classificação percentual são 0,6 e 0,75, respectivamente.

## Usando as funções de janela

As funções de janela são usadas para calcular, por exemplo, somas acumuladas e médias móveis dentro de um intervalo especificado de linhas, um intervalo de valores ou um intervalo de tempo. Uma consulta retorna um conjunto de linhas conhecido como conjunto de resultados. O termo “janela” é usado para descrever um subconjunto das linhas dentro do conjunto de resultados. Então, o subconjunto das linhas “vistas” pela janela é processado pelas funções de janela, as quais retornam um valor. É possível definir o início e o fim da janela.

Você pode usar uma janela com as seguintes funções: `SUM()`, `AVG()`, `MAX()`, `MIN()`, `COUNT()`, `VARIANCE()` e `STDDEV()`; essas funções foram estudadas no Capítulo 4. Também é possível usar uma janela com `FIRST_VALUE()` e `LAST_VALUE()`, que retornam o primeiro e o último valores em uma janela. (Você aprenderá mais sobre as funções `FIRST_VALUE()` e `LAST_VALUE()` na seção “Obtendo a primeira e a última linhas com `FIRST_VALUE()` e `LAST_VALUE()`”).

Na próxima seção, você verá como fazer uma soma acumulada e como tirar uma média móvel e uma média centralizada.

### Fazendo uma soma acumulada

A consulta a seguir faz uma soma acumulada para calcular o valor acumulado das vendas de 2003, começando em janeiro e terminando em dezembro. Observe que cada valor de vendas mensais é somado ao valor acumulado, que aumenta após cada mês:

```
SELECT
    month, SUM(amount) AS month_amount,
    SUM(SUM(amount)) OVER
        (ORDER BY month ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
        AS cumulative_amount
FROM all_sales
WHERE year = 2003
GROUP BY month
ORDER BY month;
```

MONTH	MONTH_AMOUNT	CUMULATIVE_AMOUNT
1	95525.55	95525.55
2	116671.6	212197.15
3	160307.92	372505.07
4	175998.8	548503.87
5	154349.44	702853.31
6	124951.36	827804.67
7	170296.16	998100.83
8	212735.68	1210836.51
9	199609.68	1410446.19
10	264480.79	1674926.98
11	160221.98	1835148.96
12	137336.17	1972485.13

Essa consulta usa a seguinte expressão para calcular o agregado acumulado:

```
SUM(SUM(amount)) OVER
  (ORDER BY month ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
  AS cumulative_amount
```

Vamos decompor essa expressão:

- `SUM(amount)` calcula a soma de um valor. A função `SUM()` externa calcula o valor acumulado.
- `ORDER BY month` ordena por mês as linhas lidas pela consulta.
- `ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW` define o início e o fim da janela. O início é definido como `UNBOUNDED PRECEDING`, o que significa que o início da janela é fixado na primeira linha do conjunto de resultados retornado pela consulta. O fim da janela é definido como `CURRENT ROW`; `CURRENT ROW` representa a linha atual do conjunto de resultados que está sendo processada, sendo que o fim da janela desliza uma linha para baixo depois que a função `SUM()` externa calcula e retorna o valor acumulado atual.

A consulta inteira calcula e retorna o total acumulado dos valores das vendas, começando no mês 1 e somando o valor das vendas do mês 2, depois do mês 3 e assim por diante, até (e incluindo) o mês 12. O início da janela é fixado no mês 1, mas a parte inferior da janela move-se uma linha para baixo no conjunto de resultados, depois que os valores das vendas de cada mês são somados ao total acumulado. Isso continua até que a última linha do conjunto de resultados seja processada pelas funções de janela e `SUM()`.

Não confunda o fim da janela com o fim do conjunto de resultados. No exemplo anterior, o fim da janela desliza uma linha para baixo no conjunto de resultados à medida que cada linha é processada (isto é, a soma do valor das vendas desse mês é adicionada ao total acumulado). No exemplo, o fim da janela começa na primeira linha, a soma do valor das vendas desse mês é adicionada ao total acumulado e, então, o fim da janela se move uma linha para baixo, para a segunda linha. Nesse ponto, a janela vê duas linhas. A soma do valor das vendas desse mês é adicionada ao total acumulado e o fim da janela se move uma linha para baixo, para a terceira linha. Nesse ponto, a janela vê três linhas. Isso continua até a 12ª linha ser processada. Nesse ponto, a janela vê 12 linhas.

A consulta a seguir usa uma soma acumulada para calcular o valor acumulado das vendas, começando em junho de 2003 (mês 6) e terminando em dezembro de 2003 (mês 12):

```
SELECT
  month, SUM(amount) AS month_amount,
  SUM(SUM(amount)) OVER
    (ORDER BY month ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS
    cumulative_amount
FROM all_sales
WHERE year = 2003
AND month BETWEEN 6 AND 12
GROUP BY month
ORDER BY month;
```

MONTH	MONTH_AMOUNT	CUMULATIVE_AMOUNT
6	124951.36	124951.36
7	170296.16	295247.52

8	212735.68	507983.2
9	199609.68	707592.88
10	264480.79	972073.67
11	160221.98	1132295.65
12	137336.17	1269631.82

### Calculando a média móvel

A consulta a seguir calcula a média móvel do valor das vendas entre o mês atual e os três meses anteriores:

```
SELECT
    month, SUM(amount) AS month_amount,
    AVG(SUM(amount)) OVER
        (ORDER BY month ROWS BETWEEN 3 PRECEDING AND CURRENT ROW)
        AS moving_average
FROM all_sales
WHERE year = 2003
GROUP BY month
ORDER BY month;
```

MONTH	MONTH_AMOUNT	MOVING_AVERAGE
1	95525.55	95525.55
2	116671.6	106098.575
3	160307.92	124168.357
4	175998.8	137125.968
5	154349.44	151831.94
6	124951.36	153901.88
7	170296.16	156398.94
8	212735.68	165583.16
9	199609.68	176898.22
10	264480.79	211780.578
11	160221.98	209262.033
12	137336.17	190412.155

Note que a consulta usa a seguinte expressão para calcular a média móvel:

```
AVG(SUM(amount)) OVER
    (ORDER BY month ROWS BETWEEN 3 PRECEDING AND CURRENT ROW)
    AS moving_average
```

Vamos decompor essa expressão:

- `SUM(amount)` calcula a soma de um valor. A função `AVG()` externa calcula a média.
- `ORDER BY month` ordena por mês as linhas lidas pela consulta.
- `ROWS BETWEEN 3 PRECEDING AND CURRENT ROW` define o início da janela como incluindo as três linhas que precedem a linha atual; o fim da janela é a linha atual que está sendo processada.

Então, a expressão inteira calcula a média móvel do valor das vendas entre o mês atual e os três meses anteriores. Como para os dois primeiros meses estão disponíveis menos dados do que os três meses completos, a média móvel é baseada somente nos meses disponíveis.



Tanto o início como o fim da janela começam na linha nº 1 lida pela consulta. O fim da janela se move para baixo depois que cada linha é processada. O início da janela se moverá para baixo somente depois que a linha nº 4 tiver sido processada, e se moverá uma linha para baixo depois que cada linha for processada, até que a última linha do conjunto de resultados seja lida.

### Calculando a média centralizada

A consulta a seguir calcula a média móvel do valor das vendas centralizada entre o mês anterior e o seguinte em relação ao mês atual:

```
SELECT
    month, SUM(amount) AS month_amount,
    AVG(SUM(amount)) OVER
        (ORDER BY month ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)
    AS moving_average
FROM all_sales
WHERE year = 2003
GROUP BY month
ORDER BY month;
```

MONTH	MONTH_AMOUNT	MOVING_AVERAGE
1	95525.55	106098.575
2	116671.6	124168.357
3	160307.92	150992.773
4	175998.8	163552.053
5	154349.44	151766.533
6	124951.36	149865.653
7	170296.16	169327.733
8	212735.68	194213.84
9	199609.68	225608.717
10	264480.79	208104.15
11	160221.98	187346.313
12	137336.17	148779.075

Note que a consulta usa a seguinte expressão para calcular a média móvel:

```
AVG(SUM(amount)) OVER
    (ORDER BY month ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)
    AS moving_average
```

Vamos decompor essa expressão:

- `SUM(amount)` calcula a soma de um valor. A função `AVG()` externa calcula a média.
- `ORDER BY month` ordena por mês as linhas lidas pela consulta.
- `ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING` define o início da janela como incluindo a linha anterior à linha que está sendo processada. O fim da janela é a linha após a linha atual.

Então, a expressão inteira calcula a média móvel do valor das vendas entre o mês atual e o anterior. Como para o primeiro e para o último mês estão disponíveis menos do que três meses de dados, a média móvel é baseada somente nos meses disponíveis.

O início da janela começa na linha nº 1 lida pela consulta. O fim da janela começa na linha nº 2 e se move para baixo depois que cada linha é processada. O início da janela se moverá para baixo somente quando a linha nº 2 tiver sido processada. O processamento continua até que a última linha lida pela consulta seja processada.

### **Obtendo a primeira e a última linhas com FIRST\_VALUE() e LAST\_VALUE()**

Você usa as funções FIRST\_VALUE() e LAST\_VALUE() para obter a primeira e a última linhas de uma janela. A consulta a seguir usa FIRST\_VALUE() e LAST\_VALUE() para obter o valor das vendas do mês anterior e do seguinte:

```
SELECT
    month, SUM(amount) AS month_amount,
    FIRST_VALUE(SUM(amount)) OVER
        (ORDER BY month ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)
    AS previous_month_amount,
    LAST_VALUE(SUM(amount)) OVER
        (ORDER BY month ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)
    AS next_month_amount
FROM all_sales
WHERE year = 2003
GROUP BY month
ORDER BY month;
```

MONTH	MONTH_AMOUNT	PREVIOUS_MONTH_AMOUNT	NEXT_MONTH_AMOUNT
1	95525.55	95525.55	116671.6
2	116671.6	95525.55	160307.92
3	160307.92	116671.6	175998.8
4	175998.8	160307.92	154349.44
5	154349.44	175998.8	124951.36
6	124951.36	154349.44	170296.16
7	170296.16	124951.36	212735.68
8	212735.68	170296.16	199609.68
9	199609.68	212735.68	264480.79
10	264480.79	199609.68	160221.98
11	160221.98	264480.79	137336.17
12	137336.17	160221.98	137336.17

A próxima consulta divide o valor das vendas do mês atual pelo valor das vendas do mês anterior (rotulado como curr\_div\_prev) e também divide o valor das vendas do mês atual pelo valor das vendas do mês seguinte (rotulado como curr\_div\_next):

```
SELECT
    month, SUM(amount) AS month_amount,
    SUM(amount)/FIRST_VALUE(SUM(amount)) OVER
        (ORDER BY month ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)
    AS curr_div_prev,
    SUM(amount)/LAST_VALUE(SUM(amount)) OVER
        (ORDER BY month ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)
    AS curr_div_next
FROM all_sales
WHERE year = 2003
```

```
GROUP BY month
ORDER BY month;
```

MONTH	MONTH_AMOUNT	CURR_DIV_PREV	CURR_DIV_NEXT
1	95525.55	1	.818755807
2	116671.6	1.22136538	.727796855
3	160307.92	1.37400978	.910846665
4	175998.8	1.09787963	1.14026199
5	154349.44	.876991434	1.23527619
6	124951.36	.809535558	.733729756
7	170296.16	1.36289961	.800505867
8	212735.68	1.24921008	1.06575833
9	199609.68	.93829902	.754722791
10	264480.79	1.3249898	1.65071478
11	160221.98	.605798175	1.16664081
12	137336.17	.857161858	1

Isso conclui a discussão sobre funções de janela.


## Usando as funções de relatório

As funções de relatório são usadas para efetuar cálculos entre grupos e partições dentro de grupos. É possível fazer relatórios com as seguintes funções: SUM(), AVG(), MAX(), MIN(), COUNT(), VARIANCE() e STDDEV(). Você também pode usar a função RATIO\_TO\_REPORT() para calcular a proporção de um valor em relação à soma de um conjunto de valores. Nesta seção, você verá como fazer um relatório sobre uma soma e usar a função RATIO\_TO\_REPORT().

### Relatório sobre uma soma

Para os três primeiros meses de 2003, a consulta a seguir relata:

- A soma total de todas as vendas para todos os três meses (rotulada como total\_month\_amount).
- A soma total de todas as vendas para todos os tipos de produto (rotulada como total\_product\_type\_amount).



```
SELECT
    month, prd_type_id,
    SUM(SUM(amount)) OVER (PARTITION BY month)
    AS total_month_amount,
    SUM(SUM(amount)) OVER (PARTITION BY prd_type_id)
    AS total_product_type_amount
FROM all_sales
WHERE year = 2003
AND month <= 3
GROUP BY month, prd_type_id
ORDER BY month, prd_type_id;
```

MONTH	PRD_TYPE_ID	TOTAL_MONTH_AMOUNT	TOTAL_PRODUCT_TYPE_AMOUNT
1	1	95525.55	201303.92
1	2	95525.55	44503.92

1	3	95525.55	61003.92
1	4	95525.55	65693.31
1	5	95525.55	
2	1	116671.6	201303.92
2	2	116671.6	44503.92
2	3	116671.6	61003.92
2	4	116671.6	65693.31
2	5	116671.6	
3	1	160307.92	201303.92
3	2	160307.92	44503.92
3	3	160307.92	61003.92
3	4	160307.92	65693.31
3	5	160307.92	

Note que a consulta usa a seguinte expressão para relatar a soma total de todas as vendas para todos os meses (rotulada como `total_month_amount`):

```
SUM(SUM(amount)) OVER (PARTITION BY month)
AS total_month_amount
```

Vamos decompor essa expressão:

- `SUM(amount)` calcula a soma de um valor. A função `SUM()` externa calcula a soma total.
- `OVER (PARTITION BY month)` faz a função `SUM()` externa calcular a soma para cada mês.

A consulta anterior também usa a seguinte expressão para relatar a soma total de todas as vendas para todos os tipos de produto (rotulada como `total_product_type_amount`):

```
SUM(SUM(amount)) OVER (PARTITION BY prd_type_id)
AS total_product_type_amount
```

Vamos decompor essa expressão:

- `SUM(amount)` calcula a soma de um valor. A função `SUM()` externa calcula a soma total.
- `OVER (PARTITION BY prd_type_id)` faz a função `SUM()` externa calcular a soma para cada tipo de produto.

### Usando a função `RATIO_TO_REPORT()`

Você usa a função `RATIO_TO_REPORT()` para calcular a proporção de um valor em relação à soma de um conjunto de valores. Para os três primeiros meses de 2003, a consulta a seguir relata:

- A soma do valor das vendas por tipo de produto para cada mês (rotulada como `prd_type_amount`).
- A proporção do valor das vendas do tipo de produto em relação às vendas do mês inteiro (rotulada como `prd_type_ratio`), que é calculada usando `RATIO_TO_REPORT()`.

```
SELECT
    month, prd_type_id,
    SUM(amount) AS prd_type_amount,
    RATIO_TO_REPORT(SUM(amount)) OVER (PARTITION BY month) AS prd_type_ratio
```

```
FROM all_sales
WHERE year = 2003
AND month <= 3
GROUP BY month, prd_type_id
ORDER BY month, prd_type_id;
```

MONTH	PRD_TYPE_ID	PRD_TYPE_AMOUNT	PRD_TYPE_RATIO
1	1	38909.04	.40731553
1	2	14309.04	.149792804
1	3	24909.04	.260757881
1	4	17398.43	.182133785
1	5		
2	1	70567.9	.604842138
2	2	13367.9	.114577155
2	3	15467.9	.132576394
2	4	17267.9	.148004313
2	5		
3	1	91826.98	.57281624
3	2	16826.98	.104966617
3	3	20626.98	.128670998
3	4	31026.98	.193546145
3	5		

Note que a consulta usa a seguinte expressão para calcular a proporção (rotulada como `prd_type_ratio`):

```
RATIO_TO_REPORT(SUM(amount)) OVER (PARTITION BY month) AS prd_type_ratio
```

Vamos decompor essa expressão:

- `SUM(amount)` calcula a soma do valor das vendas.
- `OVER (PARTITION BY month)` faz a função `SUM()` externa calcular a soma do valor das vendas para cada mês.
- A proporção é calculada dividindo-se a soma do valor das vendas para cada tipo de produto pela soma do valor das vendas do mês inteiro.

Isso conclui a discussão sobre funções de relatório.

## Usando as funções LAG() e LEAD()

Você usa as funções `LAG()` e `LEAD()` para obter um valor em uma linha onde essa linha está uma determinada quantidade de linhas distante da atual. A consulta a seguir usa `LAG()` e `LEAD()` para obter o valor das vendas do mês anterior e do seguinte:

```
SELECT
    month, SUM(amount) AS month_amount,
    LAG(SUM(amount), 1) OVER (ORDER BY month) AS previous_month_amount,
    LEAD(SUM(amount), 1) OVER (ORDER BY month) AS next_month_amount
FROM all_sales
WHERE year = 2003
```

```
GROUP BY month
ORDER BY month;
```

MONTH	MONTH_AMOUNT	PREVIOUS_MONTH_AMOUNT	NEXT_MONTH_AMOUNT
1	95525.55		116671.6
2	116671.6	95525.55	160307.92
3	160307.92	116671.6	175998.8
4	175998.8	160307.92	154349.44
5	154349.44	175998.8	124951.36
6	124951.36	154349.44	170296.16
7	170296.16	124951.36	212735.68
8	212735.68	170296.16	199609.68
9	199609.68	212735.68	264480.79
10	264480.79	199609.68	160221.98
11	160221.98	264480.79	137336.17
12	137336.17	160221.98	

Note que a consulta usa as seguintes expressões para obter as vendas do mês anterior e do seguinte:

```
LAG(SUM(amount), 1) OVER (ORDER BY month) AS previous_month_amount,
LEAD(SUM(amount), 1) OVER (ORDER BY month) AS next_month_amount
```

LAG(SUM(amount), 1) obtém a soma do valor da linha anterior. LEAD(SUM(amount), 1) obtém a soma do valor da linha seguinte.

Usando as funções FIRST e LAST

Você usa as funções FIRST e LAST para obter o primeiro e o último valores em um grupo ordenado. É possível usar FIRST e LAST com as seguintes funções: MIN(), MAX(), COUNT(), SUM(), AVG(), STDDEV() e VARIANCE(). A consulta a seguir usa FIRST e LAST para obter os meses de 2003 que tiveram as vendas mais altas e mais baixas:

```
SELECT
  MIN(month) KEEP (DENSE_RANK FIRST ORDER BY SUM(amount))
  AS highest_sales_month,
  MIN(month) KEEP (DENSE_RANK LAST ORDER BY SUM(amount))
  AS lowest_sales_month
FROM all_sales
WHERE year = 2003
GROUP BY month
ORDER BY month;
```

HIGHEST_SALES_MONTH	LOWEST_SALES_MONTH
1	10

Usando as funções de regressão linear

As funções de regressão linear ajustam uma linha de regressão dos mínimos quadrados normais em um conjunto de pares numéricos. É possível usar as funções de regressão linear como funções agre-

gadas, de janela ou de relatório. A tabela a seguir mostra as funções de regressão linear. Na sintaxe da função, *y* é interpretado pelas funções como uma variável que depende de *x*.

Função	Descrição
REGR_AVGX ( <i>y</i> , <i>x</i> )	Retorna a média de <i>x</i> após eliminar os pares <i>x</i> e <i>y</i> onde <i>x</i> ou <i>y</i> é nulo
REGR_AVGY ( <i>y</i> , <i>x</i> )	Retorna a média de <i>y</i> após eliminar os pares <i>x</i> e <i>y</i> onde <i>x</i> ou <i>y</i> é nulo
REGR_COUNT ( <i>y</i> , <i>x</i> )	Retorna o número de pares numéricos não-nulos que são usados para ajustar a linha de regressão
REGR_INTERCEPT ( <i>y</i> , <i>x</i> )	Retorna a interseção no eixo <i>y</i> da linha de regressão
REGR_R2 ( <i>y</i> , <i>x</i> )	Retorna o coeficiente de determinação (ao quadrado de R) da linha de regressão
REGR_SLOPE ( <i>y</i> , <i>x</i> )	Retorna a inclinação da linha de regressão
REGR_SXX ( <i>y</i> , <i>x</i> )	Retorna REG_COUNT ( <i>y</i> , <i>x</i> ) * VAR_POP ( <i>x</i> )
REGR_SXY ( <i>y</i> , <i>x</i> )	Retorna REG_COUNT ( <i>y</i> , <i>x</i> ) * COVAR_POP ( <i>y</i> , <i>x</i> )
REGR_SYY ( <i>y</i> , <i>x</i> )	Retorna REG_COUNT ( <i>y</i> , <i>x</i> ) * VAR_POP ( <i>y</i> )

A consulta a seguir mostra o uso das funções de regressão linear:

```
SELECT
  prd_type_id,
  REGR_AVGX(amount, month) AS avgx,
  REGR_AVGY(amount, month) AS avgy,
  REGR_COUNT(amount, month) AS count,
  REGR_INTERCEPT(amount, month) AS inter,
  REGR_R2(amount, month) AS r2,
  REGR_SLOPE(amount, month) AS slope,
  REGR_SXX(amount, month) AS sxx,
  REGR_SXY(amount, month) AS sxy,
  REGR_SYY(amount, month) AS syy
FROM all_sales
WHERE year = 2003
GROUP BY prd_type_id;
```

PRD_TYPE_ID	AVGX	AVGY	COUNT	INTER	R2
-----	-----	-----	-----	-----	-----
SLOPE	SXX	SXY	SYY		
-----	-----	-----	-----		
1	6.5 12570.5811		72	13318.4543	.003746289
-115.05741	858 -98719.26	3031902717			
2	6.5 2588.62806		72	2608.11268	.0000508
-2.997634	858 -2571.97	151767392			
3	6.5 6642.65153		72	2154.23119	.126338815

```
690.526206      858 592471.485 3238253324
                4      6.5 5593.76611      72 2043.47164 .128930297

546.199149      858 468638.87 1985337488
                5                0
```

Usando as funções de classificação hipotética e distribuição

As funções de classificação hipotética e distribuição calculam a classificação e o percentil que uma nova linha teria se fosse inserida em uma tabela. É possível efetuar cálculos hipotéticos com as seguintes funções: `RANK()`, `DENSE_RANK()`, `PERCENT_RANK()` e `CUME_DIST()`.

Um exemplo de função hipotética será dado após a consulta a seguir, que usa `RANK()` e `PERCENT_RANK()` para obter a classificação e a classificação percentual das vendas por tipo de produto de 2003:

```
SELECT
  prd_type_id, SUM(amount),
  RANK() OVER (ORDER BY SUM(amount) DESC) AS rank,
  PERCENT_RANK() OVER (ORDER BY SUM(amount) DESC) AS percent_rank
FROM all_sales
WHERE year = 2003
AND amount IS NOT NULL
GROUP BY prd_type_id
ORDER BY prd_type_id;
```

PRD_TYPE_ID	SUM(AMOUNT)	RANK	PERCENT_RANK
1	905081.84	1	0
2	186381.22	4	1
3	478270.91	2	.333333333
4	402751.16	3	.666666667

A próxima consulta mostra a classificação hipotética e a classificação percentual de um valor de vendas de US\$500.000:

```
SELECT
  RANK(500000) WITHIN GROUP (ORDER BY SUM(amount) DESC)
  AS rank,
  PERCENT_RANK(500000) WITHIN GROUP (ORDER BY SUM(amount) DESC)
  AS percent_rank
FROM all_sales
WHERE year = 2003
AND amount IS NOT NULL
GROUP BY prd_type_id
ORDER BY prd_type_id;
```

RANK	PERCENT_RANK
2	.25

Como você pode ver, a classificação hipotética e a classificação percentual de um valor de vendas de US\$500.000 são 2 e 0,25. Isso conclui a discussão sobre funções hipotéticas.



## USANDO A CLÁUSULA MODEL

A cláusula `MODEL` foi introduzida com o Oracle Database 10g e permite efetuar cálculos entre linhas e acessar uma coluna em uma linha, como uma célula em uma matriz. Isso permite efetuar cálculos de maneira semelhante aos de uma planilha eletrônica. Por exemplo, a tabela `all_sales` contém informações de vendas dos meses de 2003. Você pode usar a cláusula `MODEL` para calcular as vendas nos meses futuros com base nas vendas de 2003.

### Um exemplo da cláusula MODEL

O modo mais fácil de aprender a usar a cláusula `MODEL` é vendo um exemplo. A consulta a seguir recupera o valor das vendas de cada mês de 2003, feitas pelo funcionário nº 21 para os tipos de produto nº 1 e nº 2 e calcula as vendas previstas para janeiro, fevereiro e março de 2004 com base nas vendas de 2003:

```
SELECT prd_type_id, year, month, sales_amount
FROM all_sales
WHERE prd_type_id BETWEEN 1 AND 2
AND emp_id = 21
MODEL
PARTITION BY (prd_type_id)
DIMENSION BY (month, year)
MEASURES (amount sales_amount) (
  sales_amount[1, 2004] = sales_amount[1, 2003],
  sales_amount[2, 2004] = sales_amount[2, 2003] + sales_amount[3, 2003],
  sales_amount[3, 2004] = ROUND(sales_amount[3, 2003] * 1.25, 2)
)
ORDER BY prd_type_id, year, month;
```

Vamos decompor essa consulta:

- `PARTITION BY (prd_type_id)` especifica que os resultados são particionados por `prd_type_id`.
- `DIMENSION BY (month, year)` especifica que as dimensões da matriz são `month` e `year`. Isso significa que uma célula na matriz é acessada especificando-se um mês e um ano.
- `MEASURES (amount sales_amount)` especifica que cada célula da matriz contém um valor e que o nome da matriz é `sales_amount`. Para acessar a célula na matriz `sales_amount` para janeiro de 2003, você usa `sales_amount[1, 2003]`, o que retorna o valor das vendas desse mês e ano.
- Após `MEASURES` aparecem três linhas que calculam as vendas futuras para janeiro, fevereiro e março de 2004:
  - `sales_amount[1, 2004] = sales_amount[1, 2003]` define o valor das vendas para janeiro de 2004 como o valor de janeiro de 2003.
  - `sales_amount[2, 2004] = sales_amount[2, 2003] + sales_amount[3, 2003]` define o valor das vendas para fevereiro de 2004 como o valor de fevereiro de 2003 mais março de 2003.

- `sales_amount[3, 2004] = ROUND(sales_amount[3, 2003] * 1.25, 2)` define o valor das vendas para março de 2004 como o valor arredondado do valor das vendas de março de 2003 multiplicado por 1,25.
- `ORDER BY prd_type_id, year, month` simplesmente ordena os resultados retornados pela consulta inteira.

A saída da consulta está mostrada na listagem a seguir; observe que os resultados contêm os valores das vendas de todos os meses de 2003 para os tipos de produto nº 1 e nº 2, mais os valores das vendas previstos para os três primeiros meses de 2004 (que aparecem em negrito para se destacarem):

PRD_TYPE_ID	YEAR	MONTH	SALES_AMOUNT
1	2003	1	10034.84
1	2003	2	15144.65
1	2003	3	20137.83
1	2003	4	25057.45
1	2003	5	17214.56
1	2003	6	15564.64
1	2003	7	12654.84
1	2003	8	17434.82
1	2003	9	19854.57
1	2003	10	21754.19
1	2003	11	13029.73
1	2003	12	10034.84
1	<b>2004</b>	<b>1</b>	<b>10034.84</b>
1	<b>2004</b>	<b>2</b>	<b>35282.48</b>
1	<b>2004</b>	<b>3</b>	<b>25172.29</b>
2	2003	1	1034.84
2	2003	2	1544.65
2	2003	3	2037.83
2	2003	4	2557.45
2	2003	5	1714.56
2	2003	6	1564.64
2	2003	7	1264.84
2	2003	8	1734.82
2	2003	9	1854.57
2	2003	10	2754.19
2	2003	11	1329.73
2	2003	12	1034.84
2	<b>2004</b>	<b>1</b>	<b>1034.84</b>
2	<b>2004</b>	<b>2</b>	<b>3582.48</b>
2	<b>2004</b>	<b>3</b>	<b>2547.29</b>

## Usando notação posicional e simbólica para acessar células

No exemplo anterior, você viu como acessar uma célula em uma matriz usando a seguinte notação: `sales_amount[1, 2004]`, onde 1 é o mês e 2004 é o ano. Isso é referido como notação posicional, pois o significado das dimensões é determinado pela sua posição: a primeira posição contém o mês e a segunda posição contém o ano.

Você também pode usar notação simbólica para indicar explicitamente o significado das dimensões, como por exemplo, em `sales_amount [month=1, year=2004]`. A consulta a seguir reescreve a consulta anterior para usar notação simbólica:

```
SELECT prd_type_id, year, month, sales_amount
FROM all_sales
WHERE prd_type_id BETWEEN 1 AND 2
AND emp_id = 21
MODEL
PARTITION BY (prd_type_id)
DIMENSION BY (month, year)
MEASURES (amount sales_amount) (
  sales_amount [month=1, year=2004] = sales_amount [month=1, year=2003],
  sales_amount [month=2, year=2004] =
    sales_amount [month=2, year=2003] + sales_amount [month=3, year=2003],
  sales_amount [month=3, year=2004] =
    ROUND(sales_amount [month=3, year=2003] * 1.25, 2)
)
ORDER BY prd_type_id, year, month;
```

Ao usar notação posicional ou simbólica, é importante saber da maneira diferente como elas tratam os valores nulos nas dimensões. Por exemplo, `sales_amount [null, 2003]` retorna o valor cujo mês é nulo e o ano é 2003, mas `sales_amount [month=null, year=2004]` não acessaria uma célula válida, pois `null=null` sempre retorna falso.

## Acessando um intervalo de células com BETWEEN e AND

Você pode acessar um intervalo de células usando as palavras-chave `BETWEEN` e `AND`. Por exemplo, a expressão a seguir define o valor das vendas de janeiro de 2004 como a média arredondada das vendas entre janeiro e março de 2003:

```
sales_amount [1, 2004] =
  ROUND(AVG(sales_amount) [month BETWEEN 1 AND 3, 2003], 2)
```

A consulta a seguir mostra o uso dessa expressão:

```
SELECT prd_type_id, year, month, sales_amount
FROM all_sales
WHERE prd_type_id BETWEEN 1 AND 2
AND emp_id = 21
MODEL
PARTITION BY (prd_type_id)
DIMENSION BY (month, year)
MEASURES (amount sales_amount) (
  sales_amount [1, 2004] =
    ROUND(AVG(sales_amount) [month BETWEEN 1 AND 3, 2003], 2)
)
ORDER BY prd_type_id, year, month;
```

## Acessando todas as células com ANY e IS ANY

Você pode acessar todas as células de uma matriz usando os predicados `ANY` e `IS ANY`. `ANY` é usado com notação posicional e `IS ANY` com notação simbólica. Por exemplo, a expressão a seguir define o valor das vendas de janeiro de 2004 como a soma arredondada das vendas de todos os meses e anos:

```
sales_amount[1, 2004] =
  ROUND(SUM(sales_amount) [ANY, year IS ANY], 2)
```

A consulta a seguir mostra o uso dessa expressão:

```
SELECT prd_type_id, year, month, sales_amount
FROM all_sales
WHERE prd_type_id BETWEEN 1 AND 2
AND emp_id = 21
MODEL
PARTITION BY (prd_type_id)
DIMENSION BY (month, year)
MEASURES (amount sales_amount) (
  sales_amount[1, 2004] =
    ROUND(SUM(sales_amount) [ANY, year IS ANY], 2)
)
ORDER BY prd_type_id, year, month;
```

## Obtendo o valor atual de uma dimensão com CURRENTV()

Você pode obter o valor atual de uma dimensão usando a função `CURRENTV()`. Por exemplo, a expressão a seguir define o valor das vendas do primeiro mês de 2004 como 1,25 vezes as vendas do mesmo mês em 2003; observe o uso de `CURRENTV()` para obter o mês atual, que é 1:

```
sales_amount[1, 2004] =
  ROUND(sales_amount[CURRENTV(), 2003] * 1.25, 2)
```

A consulta a seguir mostra o uso dessa expressão:

```
SELECT prd_type_id, year, month, sales_amount
FROM all_sales
WHERE prd_type_id BETWEEN 1 AND 2
AND emp_id = 21
MODEL
PARTITION BY (prd_type_id)
DIMENSION BY (month, year)
MEASURES (amount sales_amount) (
  sales_amount[1, 2004] =
    ROUND(sales_amount[CURRENTV(), 2003] * 1.25, 2)
)
ORDER BY prd_type_id, year, month;
```

A saída dessa consulta é a seguinte (os valores de 2004 foram destacados em negrito):

PRD_TYPE_ID	YEAR	MONTH	SALES_AMOUNT
1	2003	1	10034.84
1	2003	2	15144.65
1	2003	3	20137.83
1	2003	4	25057.45
1	2003	5	17214.56
1	2003	6	15564.64
1	2003	7	12654.84

1	2003	8	17434.82
1	2003	9	19854.57
1	2003	10	21754.19
1	2003	11	13029.73
1	2003	12	10034.84
<b>1</b>	<b>2004</b>	<b>1</b>	<b>12543.55</b>
2	2003	1	1034.84
2	2003	2	1544.65
2	2003	3	2037.83
2	2003	4	2557.45
2	2003	5	1714.56
2	2003	6	1564.64
2	2003	7	1264.84
2	2003	8	1734.82
2	2003	9	1854.57
2	2003	10	2754.19
2	2003	11	1329.73
2	2003	12	1034.84
<b>2</b>	<b>2004</b>	<b>1</b>	<b>1293.55</b>

## Acessando células com um loop FOR

Você pode acessar células usando um loop FOR. Por exemplo, a expressão a seguir define o valor das vendas dos três primeiros meses de 2004 como 1,25 vezes as vendas dos mesmos meses de 2003; observe o uso do loop FOR e da palavra-chave INCREMENT, que especifica o valor para incrementar month durante cada iteração do loop:

```
sales_amount[FOR month FROM 1 TO 3 INCREMENT 1, 2004] =  
    ROUND(sales_amount[CURRENTV(), 2003] * 1.25, 2)
```

A consulta a seguir mostra o uso dessa expressão:

```
SELECT prd_type_id, year, month, sales_amount  
FROM all_sales  
WHERE prd_type_id BETWEEN 1 AND 2  
AND emp_id = 21  
MODEL  
PARTITION BY (prd_type_id)  
DIMENSION BY (month, year)  
MEASURES (amount sales_amount) (  
    sales_amount[FOR month FROM 1 TO 3 INCREMENT 1, 2004] =  
        ROUND(sales_amount[CURRENTV(), 2003] * 1.25, 2)  
)  
ORDER BY prd_type_id, year, month;
```

A saída dessa consulta é a seguinte (os valores de 2004 foram destacados em negrito):

PRD_TYPE_ID	YEAR	MONTH	SALES_AMOUNT
1	2003	1	10034.84
1	2003	2	15144.65
1	2003	3	20137.83
1	2003	4	25057.45

1	2003	5	17214.56
1	2003	6	15564.64
1	2003	7	12654.84
1	2003	8	17434.82
1	2003	9	19854.57
1	2003	10	21754.19
1	2003	11	13029.73
1	2003	12	10034.84
1	2004	1	12543.55
1	2004	2	18930.81
1	2004	3	25172.29
2	2003	1	1034.84
2	2003	2	1544.65
2	2003	3	2037.83
2	2003	4	2557.45
2	2003	5	1714.56
2	2003	6	1564.64
2	2003	7	1264.84
2	2003	8	1734.82
2	2003	9	1854.57
2	2003	10	2754.19
2	2003	11	1329.73
2	2003	12	1034.84
2	2004	1	1293.55
2	2004	2	1930.81
2	2004	3	2547.29

## Tratando de valores nulos e ausentes

Nesta seção, você irá aprender a tratar de valores nulos e ausentes usando a cláusula `MODEL`.

### Usando `IS PRESENT`

`IS PRESENT` retorna verdadeiro se a linha especificada pela referência de célula existia antes da execução da cláusula `MODEL`. Por exemplo:

```
sales_amount[CURRENTV(), 2003] IS PRESENT
```

retornará verdadeiro se `sales_amount[CURRENTV(), 2003]` existe.

A expressão a seguir define o valor das vendas dos três primeiros meses de 2004 como 1,25 multiplicado pelas vendas dos mesmos meses de 2003:

```
sales_amount[FOR month FROM 1 TO 3 INCREMENT 1, 2004] =
  CASE WHEN sales_amount[CURRENTV(), 2003] IS PRESENT THEN
    ROUND(sales_amount[CURRENTV(), 2003] * 1.25, 2)
  ELSE
    0
  END
```

A consulta a seguir mostra o uso dessa expressão:

```
SELECT prd_type_id, year, month, sales_amount
FROM all_sales
WHERE prd_type_id BETWEEN 1 AND 2
```

```

AND emp_id = 21
MODEL
PARTITION BY (prd_type_id)
DIMENSION BY (month, year)
MEASURES (amount sales_amount) (
  sales_amount[FOR month FROM 1 TO 3 INCREMENT 1, 2004] =
    CASE WHEN sales_amount[CURRENTV(), 2003] IS PRESENT THEN
      ROUND(sales_amount[CURRENTV(), 2003] * 1.25, 2)
    ELSE
      0
    END
)
ORDER BY prd_type_id, year, month;

```

A saída dessa consulta é a mesma do exemplo da seção anterior.

### Usando PRESENTV()

PRESENTV(*célula*, *expr1*, *expr2*) retorna a expressão *expr1* se a linha especificada pela referência de *célula* existia antes da execução da cláusula MODEL. Se a linha não existia, a expressão *expr2* é retornada. Por exemplo:

```

PRESENTV(sales_amount[CURRENTV(), 2003],
  ROUND(sales_amount[CURRENTV(), 2003] * 1.25, 2), 0)

```

retornará o valor das vendas arredondado, se sales\_amount[CURRENTV(), 2003] existe; caso contrário, 0 será retornado.

A consulta a seguir mostra o uso dessa expressão:

```

SELECT prd_type_id, year, month, sales_amount
FROM all_sales
WHERE prd_type_id BETWEEN 1 AND 2
AND emp_id = 21
MODEL
PARTITION BY (prd_type_id)
DIMENSION BY (month, year)
MEASURES (amount sales_amount) (
  sales_amount[FOR month FROM 1 TO 3 INCREMENT 1, 2004] =
    PRESENTV(sales_amount[CURRENTV(), 2003],
      ROUND(sales_amount[CURRENTV(), 2003] * 1.25, 2), 0)
)
ORDER BY prd_type_id, year, month;

```

### Usando PRESENTNNV()

PRESENTNNV(*célula*, *expr1*, *expr2*) retorna a expressão *expr1* se a linha especificada pela referência de *célula* existia antes da execução da cláusula MODEL e o valor da célula não é nulo. Se a linha não existe ou o valor da célula é nulo, a expressão *expr2* é retornada. Por exemplo,

```

PRESENTNNV(sales_amount[CURRENTV(), 2003],
  ROUND(sales_amount[CURRENTV(), 2003] * 1.25, 2), 0)

```

retornará o valor das vendas arredondado, se `sales_amount [CURRENTV(), 2003]` existe e não é nulo; caso contrário, 0 será retornado.

### Usando IGNORE NAV e KEEP NAV

IGNORE NAV retorna:

- 0 para valores numéricos nulos ou ausentes.
- Uma string vazia para valores de string nulos ou ausentes.
- 01-JAN-2000 para valores de data nulos ou ausentes.
- Nulo para todos os outros tipos de banco de dados.

KEEP NAV retorna nulo para valores numéricos nulos ou ausentes. KEEP NAV é o padrão. A consulta a seguir mostra o uso de IGNORE NAV:

```
SELECT prd_type_id, year, month, sales_amount
FROM all_sales
WHERE prd_type_id BETWEEN 1 AND 2
AND emp_id = 21
MODEL IGNORE NAV
PARTITION BY (prd_type_id)
DIMENSION BY (month, year)
MEASURES (amount sales_amount) (
  sales_amount[FOR month FROM 1 TO 3 INCREMENT 1, 2004] =
    ROUND(sales_amount[CURRENTV(), 2003] * 1.25, 2)
)
ORDER BY prd_type_id, year, month;
```

### Atualizando células existentes

Por padrão, se a célula referenciada no lado esquerdo de uma expressão existe, ela é atualizada. Se a célula não existe, uma nova linha é criada na matriz. Você pode mudar esse comportamento padrão usando RULES UPDATE, que especifica que, se a célula não existe, não será criada uma nova linha. A consulta a seguir mostra o uso de RULES UPDATE:

```
SELECT prd_type_id, year, month, sales_amount
FROM all_sales
WHERE prd_type_id BETWEEN 1 AND 2
AND emp_id = 21
MODEL
PARTITION BY (prd_type_id)
DIMENSION BY (month, year)
MEASURES (amount sales_amount)
RULES UPDATE (
  sales_amount[FOR month FROM 1 TO 3 INCREMENT 1, 2004] =
    ROUND(sales_amount[CURRENTV(), 2003] * 1.25, 2)
)
ORDER BY prd_type_id, year, month;
```



Como as células de 2004 não existem e `RULES UPDATE` é usado, nenhuma linha nova é criada na matriz para 2004; portanto, a consulta não retorna linhas para 2004. A listagem a seguir mostra a saída da consulta — observe que não existe uma linha para 2004:

PRD_TYPE_ID	YEAR	MONTH	SALES_AMOUNT
1	2003	1	10034.84
1	2003	2	15144.65
1	2003	3	20137.83
1	2003	4	25057.45
1	2003	5	17214.56
1	2003	6	15564.64
1	2003	7	12654.84
1	2003	8	17434.82
1	2003	9	19854.57
1	2003	10	21754.19
1	2003	11	13029.73
1	2003	12	10034.84
2	2003	1	1034.84
2	2003	2	1544.65
2	2003	3	2037.83
2	2003	4	2557.45
2	2003	5	1714.56
2	2003	6	1564.64
2	2003	7	1264.84
2	2003	8	1734.82
2	2003	9	1854.57
2	2003	10	2754.19
2	2003	11	1329.73
2	2003	12	1034.84

## USANDO AS CLÁUSULAS PIVOT E UNPIVOT

A cláusula `PIVOT` é novidade do Oracle Database 11g e permite transformar linhas em colunas na saída de uma consulta e, ao mesmo tempo, executar uma função de agregação nos dados. O Oracle Database 11g também tem a cláusula `UNPIVOT` que transforma colunas em linhas na saída de uma consulta. `PIVOT` e `UNPIVOT` são úteis para ver tendências globais em grandes volumes de dados, como tendências em vendas durante um período de tempo. Você vai ver consultas que mostram o uso de `PIVOT` e `UNPIVOT` nas seções a seguir.

### Um exemplo simples da cláusula PIVOT

O modo mais fácil de aprender a usar a cláusula `PIVOT` é vendo um exemplo. A consulta a seguir mostra o valor total das vendas dos tipos de produto nº 1, nº 2 e nº 3 para os quatro primeiros meses de 2003; observe que as células na saída da consulta mostram a soma dos valores das vendas para cada tipo de produto em cada mês:

```

SELECT *
FROM (
    SELECT month, prd_type_id, amount
    FROM all_sales
    WHERE year = 2003
    AND prd_type_id IN (1, 2, 3)
)
PIVOT (
    SUM(amount) FOR month IN (1 AS JAN, 2 AS FEB, 3 AS MAR, 4 AS APR)
)
ORDER BY prd_type_id;

```

PRD_TYPE_ID	JAN	FEB	MAR	APR
1	38909.04	70567.9	91826.98	120344.7
2	14309.04	13367.9	16826.98	15664.7
3	24909.04	15467.9	20626.98	23844.7

Começando na primeira linha da saída, você pode ver que:

- U\$38.909,04 do tipo de produto nº 1 foram vendidos em janeiro
- US\$70.567,90 do tipo de produto nº 1 foram vendidos em fevereiro
- ...e assim por diante para o restante da primeira linha

A segunda linha da saída mostra que:

- U\$14.309,04 do tipo de produto nº 2 foram vendidos em janeiro
- US\$13.367,90 do tipo de produto nº 2 foram vendidos em fevereiro
- ...e assim por diante para o restante da saída

## NOTA

A cláusula PIVOT é uma ferramenta poderosa que permite ver tendências em vendas de tipos de produtos em um período de meses. Com base nessas tendências, uma loja real poderia utilizar a informação para alterar sua tática de vendas e formular novas campanhas de marketing.

A instrução SELECT anterior tem a seguinte estrutura:

```

SELECT *
FROM (
    consulta_interna
)
PIVOT (
    função_agregada FOR coluna_pivô IN (lista_de_valores)
)
ORDER BY...;

```

Vamos decompor o exemplo anterior nos elementos estruturais:

- Existe uma consulta interna e uma externa. A consulta interna obtém o mês, o tipo de produto e o valor da tabela `all_sales` e passa os resultados para a consulta externa.
- `SUM(amount) FOR month IN (1 AS JAN, 2 AS FEB, 3 AS MAR, 4 AS APR)` é a linha na cláusula `PIVOT`.
- A função `SUM()` soma os valores das vendas para os tipos de produto nos quatro primeiros meses (os meses são listados na parte `IN`). Em vez de retornar os meses como 1, 2, 3 e 4 na saída, a parte `AS` renomeia os números como `JAN`, `FEB`, `MAR` e `APR` para tornar os meses mais legíveis na saída.
- A coluna `month` da tabela `all_sales` é usada como coluna pivô. Isso significa que os meses aparecem como colunas na saída. Na verdade, as linhas são rotacionadas — ou *pivoteadas* — para apresentar os meses como colunas.
- Ao final do exemplo, a linha `ORDER BY prd_type_id` simplesmente ordena os resultados pelo tipo de produto.

## Usando pivô em várias colunas

Você pode pivotar várias colunas colocando-as na parte `FOR` da cláusula `PIVOT`. O exemplo a seguir efetua o pivot nas colunas `month` e `prd_type_id`, que são referenciadas na parte `FOR`; observe que a lista de valores na parte `IN` da cláusula `PIVOT` contém um valor para as colunas `month` e `prd_type_id`:

```
SELECT *
FROM (
  SELECT month, prd_type_id, amount
  FROM all_sales
  WHERE year = 2003
  AND prd_type_id IN (1, 2, 3)
)
PIVOT (
  SUM(amount) FOR (month, prd_type_id) IN (
    (1, 2) AS JAN_PRDTYPE2,
    (2, 3) AS FEB_PRDTYPE3,
    (3, 1) AS MAR_PRDTYPE1,
    (4, 2) AS APR_PRDTYPE2
  )
);
```

JAN_PRDTYPE2	FEB_PRDTYPE3	MAR_PRDTYPE1	APR_PRDTYPE2
14309.04	15467.9	91826.98	15664.7

As células na saída mostram a soma dos valores das vendas para cada tipo de produto no mês especificado (o tipo de produto e o mês da consulta são colocados na lista de valores, na parte `IN`). Como você pode ver na saída da consulta, foram obtidos os seguintes valores de vendas:

- US\$14.309,04 do tipo de produto nº 2 em janeiro

- US\$15.467,90 do tipo de produto nº 3 em fevereiro
- US\$91.826,98 do tipo de produto nº 1 em março
- US\$15.664,70 do tipo de produto nº 2 em abril

Você pode colocar quaisquer valores na parte `IN` para obter os valores que interessem. No exemplo a seguir, os valores dos tipos de produto são embaralhados na parte `IN` para obter as vendas desses tipos de produto nos meses especificados:

```
SELECT *
FROM (
    SELECT month, prd_type_id, amount
    FROM all_sales
    WHERE year = 2003
    AND prd_type_id IN (1, 2, 3)
)
PIVOT (
    SUM(amount) FOR (month, prd_type_id) IN (
        (1, 1) AS JAN_PRDTYPE1,
        (2, 2) AS FEB_PRDTYPE2,
        (3, 3) AS MAR_PRDTYPE3,
        (4, 1) AS APR_PRDTYPE1
    )
);
```

JAN_PRDTYPE1	FEB_PRDTYPE2	MAR_PRDTYPE3	APR_PRDTYPE1
38909.04	13367.9	20626.98	120344.7

Como você pode ver a partir dessa saída, foram obtidos os seguintes valores de vendas:

- US\$38.909,04 do tipo de produto nº 1 em janeiro
- US\$13.367,90 do tipo de produto nº 2 em fevereiro
- US\$20.626,98 do tipo de produto nº 3 em março
- US\$120.344,70 do tipo de produto nº 1 em abril

## Usando várias funções agregadas em um pivô

Você pode usar várias funções agregadas em um pivô. Por exemplo, a consulta a seguir usa `SUM()` para obter as vendas totais dos tipos de produto em janeiro e fevereiro e `AVG()` para obter as médias das vendas:

```
SELECT *
FROM (
    SELECT month, prd_type_id, amount
    FROM all_sales
    WHERE year = 2003
    AND prd_type_id IN (1, 2, 3)
)
```

```

PIVOT (
  SUM(amount) AS sum_amount,
  AVG(amount) AS avg_amount
  FOR (month) IN (
    1 AS JAN, 2 AS FEB
  )
)
ORDER BY prd_type_id;

```

PRD_TYPE_ID	JAN_SUM_AMOUNT	JAN_AVG_AMOUNT	FEB_SUM_AMOUNT	FEB_AVG_AMOUNT
1	38909.04	6484.84	70567.9	11761.3167
2	14309.04	2384.84	13367.9	2227.98333
3	24909.04	4151.50667	15467.9	2577.98333

Como você pode ver, para o tipo de produto nº 1 a primeira linha da saída mostra:

- Um total de US\$38.909,04 e uma média de US\$6.484,84 vendidos em janeiro
- Um total de US\$70.567,90 e uma média de US\$11.761,32 vendidos em fevereiro

Para o tipo de produto nº 2, a segunda linha da saída mostra:

- Um total de US\$14.309,04 e uma média de US\$2.384,84 vendidos em janeiro
- Um total de US\$13.367,90 e uma média de US\$2.227,98 vendidos em fevereiro

...e assim por diante para o restante da saída.

## Usando a cláusula UNPIVOT

A cláusula UNPIVOT transforma colunas em linhas. Os exemplos desta seção usam a tabela a seguir, chamada `pivot_sales_data` (criada pelo script `store_schema.sql`); `pivot_sales_data` é preenchida por uma consulta que retorna uma versão pivoteada dos dados de vendas:

```

CREATE TABLE pivot_sales_data AS
SELECT *
FROM (
  SELECT month, prd_type_id, amount
  FROM all_sales
  WHERE year = 2003
  AND prd_type_id IN (1, 2, 3)
)
PIVOT (
  SUM(amount) FOR month IN (1 AS JAN, 2 AS FEB, 3 AS MAR, 4 AS APR)
)
ORDER BY prd_type_id;

```

A consulta a seguir retorna o conteúdo da tabela `pivot_sales_data`:

```

SELECT *
FROM pivot_sales_data;

```

PRD_TYPE_ID	JAN	FEB	MAR	APR
1	38909.04	70567.9	91826.98	120344.7
2	14309.04	13367.9	16826.98	15664.7
3	24909.04	15467.9	20626.98	23844.7

A próxima consulta usa UNPIVOT para obter os dados das vendas em uma forma sem uso de pivô:

```
SELECT *
FROM pivot_sales_data
UNPIVOT (
    amount FOR month IN (JAN, FEB, MAR, APR)
)
ORDER BY prd_type_id;
```

PRD_TYPE_ID	MON	AMOUNT
1	JAN	38909.04
1	FEB	70567.9
1	MAR	91826.98
1	APR	120344.7
2	JAN	14309.04
2	FEB	13367.9
2	APR	15664.7
2	MAR	16826.98
3	JAN	24909.04
3	MAR	20626.98
3	FEB	15467.9
3	APR	23844.7

Note que a consulta rotaciona os dados pivotados. Por exemplo, os totais das vendas mensais que aparecem nas linhas horizontais de pivot\_sales\_data são mostrados na coluna vertical AMOUNT.

#### DICA

Considere o uso de UNPIVOT quando você tiver uma consulta que retorna linhas com muitas colunas e quiser ver essas colunas como linhas.

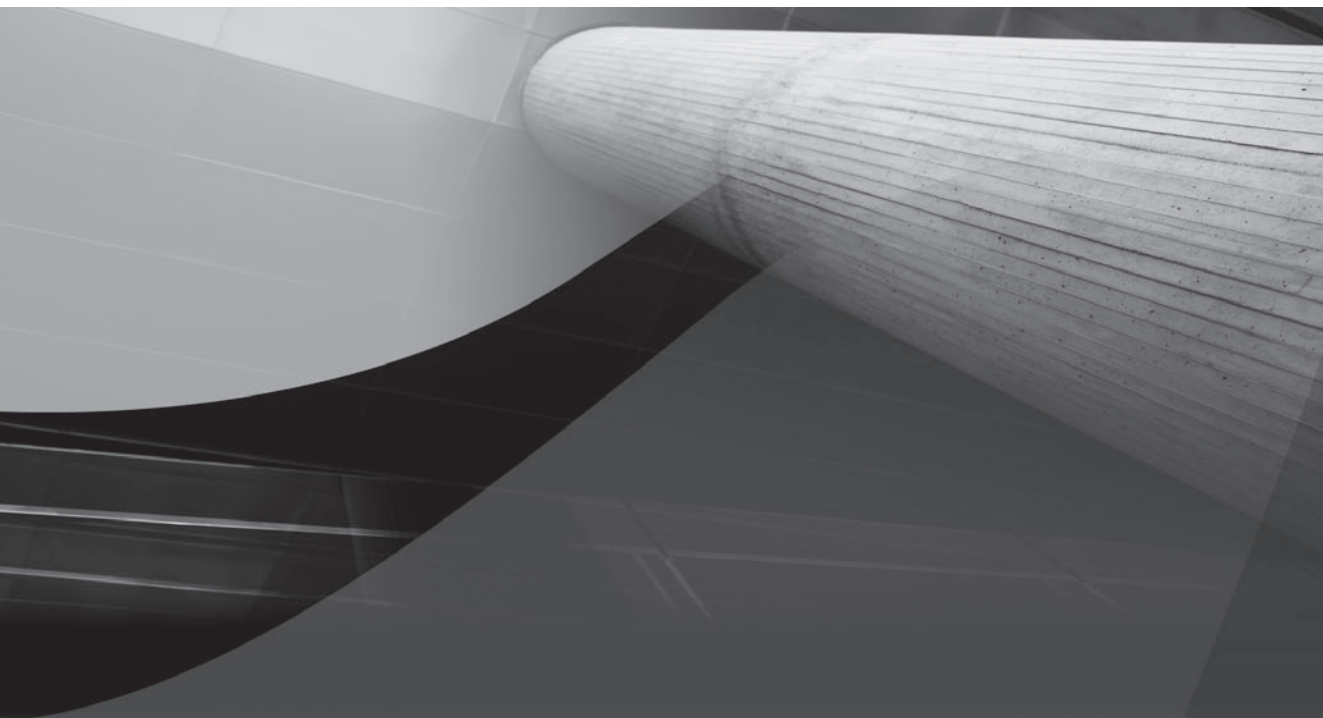
## RESUMO

Neste capítulo, você aprendeu que:

- Os operadores de conjunto (UNION ALL, UNION, INTERSECT e MINUS) permitem combinar as linhas retornadas por duas ou mais consultas.
- TRANSLATE(*x*, *da\_string*, *para\_string*) transforma os caracteres de uma string nos caracteres da outra string.
- DECODE(*valor*, *valor\_pesquisa*, *resultado*, *valor\_padrão*) compara *valor* com *valor\_pesquisa*. Se os valores são iguais, DECODE() retorna *valor\_pesquisa*; caso contrário, *valor\_padrão* é retornado. DECODE() permite executar lógica if-then-else em SQL.

- `CASE` é semelhante a `DECODE()`. Você deve usar `CASE`, pois é compatível com o padrão ANSI.
- Consultas podem ser executadas em dados organizados em uma hierarquia.
- `ROLLUP` estende a cláusula `GROUP BY` para retornar uma linha contendo um subtotal para cada grupo de linhas, além de uma linha contendo um total geral para todos os grupos.
- `CUBE` estende a cláusula `GROUP BY` para retornar as linhas que contêm um subtotal para todas as combinações de colunas, além de uma linha contendo o total geral.
- O banco de dados tem muitas funções analíticas internas que permitem efetuar cálculos complexos, como encontrar o tipo de produto mais vendido em cada mês, os principais vendedores etc.
- A cláusula `MODEL` efetua cálculos entre linhas e permite tratar dados de tabela como uma matriz. Isso permite efetuar cálculos de maneira semelhante aos de uma planilha eletrônica.
- As cláusulas `PIVOT` e `UNPIVOT` do Oracle Database 11g são úteis para ver tendências globais em grandes volumes de dados.

No próximo capítulo, você vai aprender sobre a alteração do conteúdo de uma tabela.



# CAPÍTULO

# 8

Alterando o conteúdo  
de tabelas



Neste capítulo, você vai aprender sobre a alteração do conteúdo de tabelas, especificamente sobre:

- Como adicionar, modificar e remover linhas usando as instruções `INSERT`, `UPDATE` e `DELETE`
- Como as transações de banco de dados podem consistir em várias instruções `INSERT`, `UPDATE` e `DELETE`
- Como tornar os resultados de suas transações permanentes usando a instrução `COMMIT` ou desfazer seus resultados inteiramente, usando a instrução `ROLLBACK`
- Como um banco de dados Oracle pode processar várias transações ao mesmo tempo
- Como utilizar consultas flashback para ver as linhas como eram originalmente antes das alterações feitas

## ADICIONANDO LINHAS COM A INSTRUÇÃO INSERT

A instrução `INSERT` é usada para adicionar linhas em uma tabela. Nessa instrução, você pode especificar o seguinte:

- A tabela na qual a linha será inserida
- Uma lista de colunas para as quais você deseja especificar valores
- Uma lista de valores para armazenar nas colunas especificadas

Ao adicionar uma linha, normalmente você fornece um valor para a chave primária e todas as outras colunas que são definidas como `NOT NULL`. Não é necessário especificar valores para colunas `NULL`; por padrão, elas serão configuradas como nulas. Você pode descobrir quais colunas estão definidas como `NOT NULL` usando o comando `DESCRIBE` do `SQL*Plus`. O exemplo a seguir descreve a tabela `customers`:

<b>DESCRIBE customers</b>		
Name	Null?	Type
-----	-----	-----
CUSTOMER_ID	NOT NULL	NUMBER (38)
FIRST_NAME	NOT NULL	VARCHAR2 (10)
LAST_NAME	NOT NULL	VARCHAR2 (10)
DOB		DATE
PHONE		VARCHAR2 (12)

As colunas `customer_id`, `first_name` e `last_name` são `NOT NULL`, significando que um valor deve ser fornecido para elas. As colunas `dob` e `phone` não exigem um valor: se você omitir esses valores ao adicionar uma linha, as colunas serão configuradas como nulas.

A instrução `INSERT` a seguir adiciona uma linha na tabela `customers`. Note que a ordem dos valores na cláusula `VALUES` corresponde à ordem na qual as colunas são especificadas na lista de colunas. Observe também que a instrução tem três partes: o nome da tabela, a lista de colunas e os valores a serem adicionados.

```
INSERT INTO customers (
  customer_id, first_name, last_name, dob, phone
) VALUES (
  6, 'Fred', 'Brown', '01-JAN-1970', '800-555-1215'
);
```

1 row created.

O SQL\*Plus responde que uma linha foi criada. Você pode verificar isso executando a instrução `SELECT` a seguir:

```
SELECT *
FROM customers;
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB	PHONE
1	John	Brown	01-JAN-65	800-555-1211
2	Cynthia	Green	05-FEB-68	800-555-1212
3	Steve	White	16-MAR-71	800-555-1213
4	Gail	Black		800-555-1214
5	Doreen	Blue	20-MAY-70	
6	Fred	Brown	01-JAN-70	800-555-1215

Observe que agora a nova linha aparece nos resultados retornados pela consulta.

## Omitindo a lista de colunas

É possível omitir a lista de colunas ao fornecer valores para cada coluna, como neste exemplo:

```
INSERT INTO customers
VALUES (7, 'Jane', 'Green', '01-JAN-1970', '800-555-1216');
```

Quando você omite a lista de colunas, a ordem dos valores fornecidos deve corresponder à ordem das colunas conforme listadas na saída do comando `DESCRIBE`.

## Especificando um valor nulo para uma coluna

É possível especificar um valor nulo para uma coluna usando a palavra-chave `NULL`. Por exemplo, a instrução `INSERT` a seguir especifica um valor nulo para as colunas `dob` e `phone`:

```
INSERT INTO customers
VALUES (8, 'Sophie', 'White', NULL, NULL);
```

Quando você exibir essa linha usando uma consulta, não verá um valor para as colunas `dob` e `phone`, pois elas foram configuradas com valores nulos:

```
SELECT *
FROM customers
WHERE customer_id = 8;
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB	PHONE
8	Sophie	White		

Observe que os valores das colunas `dob` e `phone` estão em branco.

## Incluindo apóstrofos e aspas em um valor de coluna

Você pode incluir um apóstrofo e aspas em um valor de coluna. Por exemplo, a instrução `INSERT` a seguir especifica o sobrenome `O'Malley` para um novo cliente; observe o uso de dois apóstrofos no sobrenome, após a letra `O`:

```
INSERT INTO customers
VALUES (9, 'Kyle', 'O'Malley', NULL, NULL);
```

O exemplo a seguir especifica o nome `The "Great" Gatsby` para um novo produto:

```
INSERT INTO products (
  product_id, product_type_id, name, description, price
) VALUES (
  13, 1, 'The "Great" Gatsby', NULL, 12.99
);
```

## Copiando linhas de uma tabela para outra

Você pode copiar linhas de uma tabela para outra usando uma consulta no lugar dos valores de coluna na instrução `INSERT`. O número de colunas e os tipos de coluna na origem e no destino devem corresponder. O exemplo a seguir usa uma instrução `SELECT` para recuperar as colunas `first_name` e `last_name` do cliente nº 1 e fornece essas colunas para uma instrução `INSERT`:

```
INSERT INTO customers (customer_id, first_name, last_name)
SELECT 10, first_name, last_name
FROM customers
WHERE customer_id = 1;
```

Note que o valor de `customer_id` para a nova linha é configurado como 10.

### NOTA

*O Oracle Database 9i introduziu a instrução `MERGE`, a qual permite mesclar linhas de uma tabela em outra. `MERGE` é muito mais flexível do que combinar uma instrução `INSERT` e uma instrução `SELECT` para copiar linhas de uma tabela em outra. Você vai aprender sobre `MERGE` na seção “Mesclando linhas com `MERGE`”.*

## MODIFICANDO LINHAS COM A INSTRUÇÃO UPDATE

A instrução `UPDATE` modifica linhas em uma tabela. Ao usar essa instrução, normalmente você especifica as seguintes informações:

- O nome da tabela
- Uma cláusula `WHERE` especificando as linhas a serem alteradas
- Uma lista de nomes de colunas, junto com seus novos valores, especificados com a cláusula `SET`

Você pode alterar uma ou mais linhas usando a mesma instrução `UPDATE`. Se mais de uma linha for especificada, a mesma alteração será implementada para todas essas linhas. Por exemplo, a instrução `UPDATE` a seguir configura a coluna `last_name` como `Orange` para a linha cujo valor de `customer_id` é 2:

```
UPDATE customers
SET last_name = 'Orange'
WHERE customer_id = 2;
```

1 row updated.

O SQL\*Plus confirma que uma linha foi atualizada. Se a cláusula `WHERE` fosse omitida, todas as linhas seriam atualizadas. A consulta a seguir confirma que a alteração foi feita:

```
SELECT *
FROM customers
WHERE customer_id = 2;
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB	PHONE
2	Cynthia	Orange	05-FEB-68	800-555-1212

Você pode alterar várias linhas e várias colunas na mesma instrução `UPDATE`. Por exemplo, a instrução `UPDATE` a seguir eleva em 20% o preço de todos os produtos cujo preço atual é maior ou igual a US\$20. Ela também altera os nomes desses produtos para minúsculas:

```
UPDATE products
SET
  price = price * 1.20,
  name = LOWER(name)
WHERE
  price >= 20;
```

3 rows updated.

Três linhas foram atualizadas por essa instrução. A consulta a seguir confirma a alteração:

```
SELECT product_id, name, price
FROM products
WHERE price >= (20 * 1.20);
```

ID	NAME	PRICE
2	chemistry	36
3	supernova	31.19
5	z-files	59.99

#### NOTA

*Você também pode usar uma subconsulta com uma instrução `UPDATE`. Isso foi abordado no Capítulo 6, na seção “Escrevendo uma instrução `UPDATE` contendo uma subconsulta”.*

## A CLÁUSULA RETURNING

No Oracle Database 10g e versões superiores, você pode usar a cláusula `RETURNING` para retornar o valor de uma função agregada, como `AVG()`. As funções agregadas foram abordadas no Capítulo 4.

As seguintes tarefas são executadas pelo próximo exemplo:

- Declara uma variável chamada `average_product_price`
- Diminui o valor da coluna `price` das linhas na tabela `products` e salva o preço médio na variável `average_product_price` usando a cláusula `RETURNING`
- Imprime o valor da variável `average_product_price`

```
VARIABLE average_product_price NUMBER

UPDATE products
SET price = price * 0.75
RETURNING AVG(price) INTO :average_product_price;
12 rows updated.

PRINT average_product_price
AVERAGE_PRODUCT_PRICE
-----
16.1216667
```

## REMOVENDO LINHAS COM A INSTRUÇÃO DELETE

A instrução `DELETE` é usada para remover linhas de uma tabela. Geralmente, você deve especificar uma cláusula `WHERE` que limite as linhas que deseja excluir; se não fizer isso, *todas* as linhas serão excluídas. A instrução `DELETE` a seguir remove a linha da tabela `customers` cujo valor de `customer_id` é 10:

```
DELETE FROM customers
WHERE customer_id = 10;

1 row deleted.
```

O SQL\*Plus confirma que uma linha foi excluída. Você também pode usar uma subconsulta com uma instrução `DELETE`. Isso foi abordado no Capítulo 6, na seção “Escrevendo uma instrução `DELETE` contendo uma subconsulta”.

### NOTA

*Se você executou as instruções `INSERT`, `UPDATE` e `DELETE` anteriores, reverta-as usando `ROLLBACK` para que seus resultados correspondam àqueles mostrados no restante deste capítulo. Não se preocupe se você já se desconectou do banco de dados: basta executar novamente o script `store_schema.sql` para recriar tudo.*

## INTEGRIDADE DO BANCO DE DADOS

Quando você executa uma instrução DML (`INSERT`, `UPDATE` ou `DELETE`, por exemplo), o banco de dados garante que as linhas nas tabelas mantenham sua integridade. Isso significa que as alterações feitas nas linhas não afetam a chave primária e os relacionamentos de chave estrangeira das tabelas.

## Aplicação das restrições de chave primária

Vamos examinar alguns exemplos que mostram a aplicação de uma restrição de chave primária. A chave primária da tabela `customers` é a coluna `customer_id`, o que significa que todo valor armazenado nessa coluna deve ser exclusivo. Se você tentar inserir uma linha com um valor duplicado para uma chave primária, o banco de dados retornará o erro `ORA-00001`, como neste exemplo:

```
SQL> INSERT INTO customers (  
2   customer_id, first_name, last_name, dob, phone  
3 ) VALUES (  
4   1, 'Jason', 'Price', '01-JAN-60', '800-555-1211'  
5 );  
INSERT INTO customers (  
*  
ERROR at line 1:  
ORA-00001: unique constraint (STORE.CUSTOMERS_PK) violated
```

Se você tentar atualizar um valor de chave primária com um valor já existente na tabela, o banco de dados retornará o mesmo erro:

```
SQL> UPDATE customers  
2   SET customer_id = 1  
3   WHERE customer_id = 2;  
UPDATE customers  
*  
ERROR at line 1:  
ORA-00001: unique constraint (STORE.CUSTOMERS_PK) violated
```

## Aplicação das restrições de chave estrangeira

Um relacionamento de chave estrangeira é aquele no qual uma coluna de uma tabela é referenciada em outra. Por exemplo, a coluna `product_type_id` da tabela `products` referencia a coluna `product_type_id` da tabela `product_types`. A tabela `product_types` é conhecida como tabela *pai* e a tabela `products` é conhecida como tabela *filho*, refletindo a dependência da coluna `product_type_id` da tabela `products` em relação à coluna `product_type_id` da tabela `product_types`.

Se você tentar inserir uma linha na tabela `products` com um valor de `product_type_id` inexistente, o banco de dados retornará o erro `ORA-02291`. Esse erro indica que o banco de dados não conseguiu encontrar um valor de chave pai correspondente (a chave pai é a coluna `product_type_id` da tabela `product_types`). No exemplo a seguir, o erro é retornado porque não existe uma linha na tabela `product_types` cujo valor de `product_type_id` é 6:

```
SQL> INSERT INTO products (  
2   product_id, product_type_id, name, description, price  
3 ) VALUES (  
4   13, 6, 'Test', 'Test', NULL  
5 );  
INSERT INTO products (  
*  
ERROR at line 1:  
ORA-02291: integrity constraint (STORE.PRODUCTS_FK_PRODUCT_TYPES)  
violated - parent key not found
```

Da mesma forma, se você tentar atualizar o valor de `product_type_id` de uma linha na tabela `products` com um valor de chave pai inexistente, o banco de dados retornará o mesmo erro, como neste exemplo:

```
SQL> UPDATE products
  2 SET product_type_id = 6
  3 WHERE product_id = 1;
UPDATE products
*
ERROR at line 1:
ORA-02291: integrity constraint (STORE.PRODUCTS_FK_PRODUCT_TYPES)
violated - parent key not found
```

Por fim, se você tentar excluir uma linha na tabela pai, que tenha linhas filhos dependentes, o banco de dados retornará o erro ORA-02292. Por exemplo, se tentar excluir da tabela `product_types` a linha cujo valor de `product_type_id` é 1, o banco de dados retornará esse erro, pois a tabela `products` contém linhas cujo valor de `product_type_id` é 1:

```
SQL> DELETE FROM product_types
  2 WHERE product_type_id = 1;
DELETE FROM product_types
*
ERROR at line 1:
ORA-02292: integrity constraint (STORE.PRODUCTS_FK_PRODUCT_TYPES)
violated - child record found
```

Se o banco de dados permitisse essa exclusão, as linhas filho ficariam inválidas, pois não apontariam para valores válidos na tabela pai.

## USANDO VALORES PADRÃO

O Oracle Database 9i introduziu um recurso que permite definir um valor padrão para uma coluna. Por exemplo, a instrução a seguir cria uma tabela chamada `order_status`; a coluna `status` tem como valor padrão 'Order placed' e a coluna `last_modified` tem como valor padrão a data e hora retornadas por `SYSDATE`:

```
CREATE TABLE order_status (
  order_status_id INTEGER
  CONSTRAINT default_example_pk PRIMARY KEY,
  status VARCHAR2(20) DEFAULT 'Order placed' NOT NULL,
  last_modified DATE DEFAULT SYSDATE
);
```

### NOTA

*A tabela `order_status` é criada pelo script `store_schema.sql`. Isso significa que você não precisa digitar a instrução `CREATE TABLE` anterior. Além disso, você não precisa digitar as instruções `INSERT` mostradas nesta seção.*

Quando você adiciona uma nova linha na tabela `order_status`, mas não especifica os valores das colunas `status` e `last_modified`, essas colunas são configuradas com os valores padrão. Por exemplo, a instrução `INSERT` a seguir omite os valores das colunas `status` e `last_modified`:

```
INSERT INTO order_status (order_status_id)
VALUES (1);
```

A coluna `status` é configurada com o valor padrão 'Order placed' e a coluna `last_modified` é configurada com a data e hora atuais. Você pode anular os padrões, especificando um valor para as colunas, como mostrado no exemplo a seguir:

```
INSERT INTO order_status (order_status_id, status, last_modified)
VALUES (2, 'Order shipped', '10-JUN-2004');
```

A consulta a seguir recupera as linhas de `order_status`:

```
SELECT *
FROM order_status;
```

ORDER_STATUS_ID	STATUS	LAST_MODI
1	Order placed	25-JUL-07
2	Order shipped	10-JUN-04

Você pode redefinir uma coluna com o padrão usando a palavra-chave `DEFAULT` em uma instrução `UPDATE`. Por exemplo, a instrução `UPDATE` a seguir configura a coluna `status` com o padrão:

```
UPDATE order_status
SET status = DEFAULT
WHERE order_status_id = 2;
```

A consulta a seguir mostra a alteração feita por essa instrução `UPDATE`:

```
SELECT *
FROM order_status;
```

ORDER_STATUS_ID	STATUS	LAST_MODI
1	Order placed	25-JUL-07
2	Order placed	10-JUN-04

## MESCLANDO LINHAS COM MERGE

O Oracle Database 9i introduziu a instrução `MERGE`, que permite mesclar linhas de uma tabela em outra. Por exemplo, talvez você queira mesclar alterações dos produtos listados em uma tabela na tabela `products`. O esquema `store` contém uma tabela chamada `product_changes` que foi criada com a seguinte instrução `CREATE TABLE` em `store_schema.sql`:

```
CREATE TABLE product_changes (
  product_id INTEGER
    CONSTRAINT prod_changes_pk PRIMARY KEY,
  product_type_id INTEGER
    CONSTRAINT prod_changes_fk_product_types
      REFERENCES product_types (product_type_id),
  name VARCHAR2(30) NOT NULL,
```



```
description VARCHAR2(50),  
price NUMBER(5, 2)  
);
```

A consulta a seguir recupera as colunas `product_id`, `product_type_id`, `name` e `price` dessa tabela:

```
SELECT product_id, product_type_id, name, price  
FROM product_changes;
```

PRODUCT_ID	PRODUCT_TYPE_ID	NAME	PRICE
1	1	Modern Science	40
2	1	New Chemistry	35
3	1	Supernova	25.99
13	2	Lunar Landing	15.99
14	2	Submarine	15.99
15	2	Airplane	15.99

Digamos que você queira mesclar as linhas da tabela `product_changes` na tabela `products`, como segue:

- Para as linhas com valores de `product_id` correspondentes nas duas tabelas, atualize as linhas existentes em `products` com os valores de coluna de `product_changes`. Por exemplo, o produto nº 1 tem um preço em `product_changes` diferente do que existe em `products`; portanto, o preço do produto nº 1 deve ser atualizado na tabela `products`. Da mesma forma, o produto nº 2 tem nome e preço diferentes; portanto, os dois valores precisam ser atualizados em `products`. Por fim, o produto nº 3 tem um valor de `product_type_id` diferente e, portanto, esse valor deve ser atualizado em `products`.
- Para as linhas novas em `product_changes`, insira essas novas linhas na tabela `products`. Os produtos nº 13, 14 e 15 são novos em `product_changes` e, portanto, devem ser inseridos em `products`.

É mais fácil aprender a usar a instrução `MERGE` com um exemplo. O exemplo a seguir realiza a mesclagem conforme definido anteriormente:

```
MERGE INTO products p  
USING product_changes pc ON (  
    p.product_id = pc.product_id  
)  
WHEN MATCHED THEN  
    UPDATE  
    SET  
        p.product_type_id = pc.product_type_id,  
        p.name = pc.name,  
        p.description = pc.description,  
        p.price = pc.price  
WHEN NOT MATCHED THEN  
    INSERT (  
        p.product_id, p.product_type_id, p.name,  
        p.description, p.price  
    ) VALUES (  

```

```
pc.product_id, pc.product_type_id, pc.name,
pc.description, pc.price
);
```

6 rows merged.

### NOTA

Você encontrará um script chamado `merge_example.sql` no diretório SQL. Esse script contém a instrução `MERGE` anterior.

Observe os seguintes aspectos sobre a instrução `MERGE`:

- A cláusula `MERGE INTO` especifica o nome da tabela na qual as linhas serão mescladas. No exemplo, essa tabela é `products`, que recebeu o apelido `p`.
- A cláusula `USING... ON` especifica uma junção de tabela. No exemplo, a junção é feita nas colunas `product_id` das tabelas `products` e `product_changes`. A tabela `product_changes` também recebeu um apelido, `pc`.
- A cláusula `WHEN MATCHED THEN` especifica a ação a ser executada quando a cláusula `USING... ON` é satisfeita por uma linha. No exemplo, essa ação é uma instrução `UPDATE` que configura as colunas `product_type_id`, `name`, `description` e `price` da linha existente na tabela `products` com os valores de coluna da linha correspondente na tabela `product_changes`.
- A cláusula `WHEN NOT MATCHED THEN` especifica a ação a ser executada quando a cláusula `USING... ON` não é satisfeita para uma linha. No exemplo, essa ação é uma instrução `INSERT` que adiciona uma linha na tabela `products`, pegando os valores de coluna da linha na tabela `product_changes`.

Se você executar a instrução `MERGE` anterior, verá que ela relata que seis linhas são mescladas; essas são as linhas com valores de `product_id` 1, 2, 3, 13, 14 e 15. A consulta a seguir recupera as seis linhas mescladas da tabela `products`:

```
SELECT product_id, product_type_id, name, price
FROM products
WHERE product_id IN (1, 2, 3, 13, 14, 15);
```

PRODUCT_ID	PRODUCT_TYPE_ID	NAME	PRICE
1	1	Modern Science	40
2	1	New Chemistry	35
3	1	Supernova	25.99
13	2	Lunar Landing	15.99
14	2	Submarine	15.99
15	2	Airplane	15.99

As seguintes alterações foram feitas nessas linhas:

- O produto nº 1 tem um novo preço
- O produto nº 2 tem um novo nome e um novo preço

- O produto nº 3 tem uma nova identificação de tipo de produto
- Os produtos nº 13, 14 e 15 são novos

Agora que você já aprendeu a fazer alterações no conteúdo de tabelas, vamos passar para as transações de banco de dados.

## TRANSAÇÕES DE BANCO DE DADOS

Uma *transação* de banco de dados é um grupo de instruções SQL que executam uma *unidade lógica de trabalho*. Você pode considerar uma transação como um conjunto inseparável de instruções SQL cujos resultados devem se tornar permanentes no banco de dados como um todo (ou serem desfeitas como um todo).

Um exemplo de transação de banco de dados é uma transferência de dinheiro de uma conta bancária para outra. Uma instrução `UPDATE` subtrairia do valor total em dinheiro de uma conta e outra instrução `UPDATE` adicionaria dinheiro na outra conta. Tanto a subtração como a adição devem ser permanentemente registradas no banco de dados; caso contrário, dinheiro será perdido. Se há um problema na transferência de dinheiro, então tanto a subtração como a adição devem ser desfeitas. O exemplo simples esboçado neste parágrafo utiliza apenas duas instruções `UPDATE`, mas uma transação pode consistir em muitas instruções `INSERT`, `UPDATE` e `DELETE`.

### Confirmando e revertendo uma transação

Para registrar permanentemente os resultados obtidos pelas instruções SQL em uma transação, realize um *commit* usando a instrução SQL `COMMIT`. Se precisar desfazer os resultados, realize um *rollback* usando a instrução SQL `ROLLBACK`, que restaura todas as linhas como eram originalmente. O exemplo a seguir adiciona uma linha na tabela `customers` e depois torna a alteração permanente executando uma instrução `COMMIT`:

```
INSERT INTO customers
VALUES (6, 'Fred', 'Green', '01-JAN-1970', '800-555-1215');

1 row created.

COMMIT;

Commit complete.
```

O exemplo a seguir atualiza o cliente nº 1 e depois desfaz a alteração executando uma instrução `ROLLBACK`:

```
UPDATE customers
SET first_name = 'Edward'
WHERE customer_id = 1;

1 row updated.

ROLLBACK;

Rollback complete.
```

A consulta a seguir mostra a nova linha da instrução COMMIT:

```
SELECT *  
FROM customers;
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB	PHONE
1	John	Brown	01-JAN-65	800-555-1211
2	Cynthia	Green	05-FEB-68	800-555-1212
3	Steve	White	16-MAR-71	800-555-1213
4	Gail	Black		800-555-1214
5	Doreen	Blue	20-MAY-70	
6	Fred	Green	01-JAN-70	800-555-1215

Note que o cliente nº 6 se tornou permanente com a instrução COMMIT, mas a alteração realizada no nome do cliente nº 1 foi desfeita pela instrução ROLLBACK.

## Iniciando e terminando uma transação

Uma transação é uma unidade lógica de trabalho que permite dividir suas instruções SQL. Uma transação tem um início e um fim; ela começa quando um dos seguintes eventos ocorre:

- Você se conecta no banco de dados e executa uma instrução DML (INSERT, UPDATE ou DELETE).
- Uma transação anterior termina e você insere outra instrução DML.

Uma transação termina quando um dos seguintes eventos ocorre:

- Você executa uma instrução COMMIT ou ROLLBACK.
- Você executa uma instrução DDL, como uma instrução CREATE TABLE, caso em que uma instrução COMMIT é executada automaticamente.
- Você executa uma instrução DCL, como uma instrução GRANT, caso em que uma instrução COMMIT é executada automaticamente. Você vai aprender sobre GRANT no próximo capítulo.
- Você se desconecta do banco de dados. Se você sai do SQL\*Plus normalmente, digitando o comando EXIT, uma instrução COMMIT é executada de modo automático. Se o SQL\*Plus termina de forma incomum — por exemplo, se o computador em que o SQL\*Plus estava sendo executado falha —, uma instrução ROLLBACK é executada automaticamente. Isso se aplica a qualquer programa que acesse um banco de dados. Por exemplo, se você escrevesse um programa Java que acessasse um banco de dados e seu programa falhasse, uma instrução ROLLBACK seria executada automaticamente.
- Você executa uma instrução DML que falha, no caso em que uma instrução ROLLBACK é executada automaticamente para essa instrução DML individual.

### DICA

Não confirmar ou reverter explicitamente suas transações é uma má prática; portanto, execute uma instrução COMMIT ou ROLLBACK no final de suas transações.

## Savepoints (pontos de salvamento)

Também é possível definir um *savepoint* em qualquer lugar dentro de uma transação. Isso permite reverter alterações até esse *savepoint*. Os *savepoints* podem ser úteis para decompor transações muito longas, pois, se você cometer um erro após ter definido um *savepoint*, não precisará reverter a transação até o início. Entretanto, você deve usar *savepoints* com moderação: em vez disso, é melhor reestruturar sua transação em transações menores.

Você verá um exemplo de *savepoint* em breve, mas primeiro examine o preço atual dos produtos nº 4 e 5:

```
SELECT product_id, price
FROM products
WHERE product_id IN (4, 5);
```

PRODUCT_ID	PRICE
4	13.95
5	49.99

O preço do produto nº 4 é US\$13,95 e o preço do produto nº 5 é US\$49,99. A instrução `UPDATE` a seguir aumenta em 20% o preço do produto nº 4:

```
UPDATE products
SET price = price * 1.20
WHERE product_id = 4;
```

1 row updated.

A instrução a seguir define um *savepoint* chamado `save1`:

```
SAVEPOINT save1;
```

Savepoint created.

Todas as instruções DML executadas depois desse ponto podem ser revertidas para o *savepoint* e a alteração feita no produto nº 4 será mantida. A instrução `UPDATE` a seguir aumenta em 30% o preço do produto nº 5:

```
UPDATE products
SET price = price * 1.30
WHERE product_id = 5;
```

1 row updated.

A consulta a seguir obtém os preços dos dois produtos:

```
SELECT product_id, price
FROM products
WHERE product_id IN (4, 5);
```

PRODUCT_ID	PRICE
4	16.74
5	64.99

O preço do produto nº 4 é 20% maior e o preço do produto nº 5 é 30% maior. A instrução a seguir reverte a transação para o savepoint estabelecido anteriormente:

```
ROLLBACK TO SAVEPOINT save1;
```

Rollback complete.

Isso desfaz a alteração de preço do produto nº 5, mas deixou a alteração de preço do produto nº 4 intacta. A consulta a seguir mostra isso:

```
SELECT product_id, price
FROM products
WHERE product_id IN (4, 5);
```

PRODUCT_ID	PRICE
4	16.74
5	49.99

Conforme o esperado, o produto nº 4 manteve seu preço maior, mas o preço do produto nº 5 está de volta ao original. A instrução ROLLBACK a seguir desfaz a transação inteira:

```
ROLLBACK;
```

Rollback complete.

Isso desfaz a alteração realizada no preço do produto nº 4, conforme mostra a consulta a seguir:

```
SELECT product_id, price
FROM products
WHERE product_id IN (4, 5);
```

PRODUCT_ID	PRICE
4	13.95
5	49.99

## Propriedades de transação ACID

Anteriormente, definimos uma transação como uma *unidade lógica de trabalho*; isto é, um grupo de instruções SQL relacionadas que sofrem *commit* ou *rollback* como uma unidade. A definição mais rigorosa de transação da teoria de banco de dados diz que uma transação tem quatro propriedades fundamentais, conhecidas como propriedades *ACID* (das primeiras letras de cada propriedade da lista a seguir):

- **Atômica** As transações são atômicas, significando que as instruções SQL contidas em uma transação constituem uma única unidade de trabalho.
- **Consistente** As transações garantem que o estado do banco de dados permanece consistente, significando que o banco de dados está em um estado consistente quando uma transação começa e que ele termina em outro estado consistente quando a transação acaba.
- **Isolada** Transações separadas não devem interferir umas com as outras.
- **Durável** Uma vez que a transação sofreu *commit*, as alterações feitas no banco de dados são preservadas, mesmo que a máquina em que o software de banco de dados está sendo executado falhe posteriormente.

O software de banco de dados Oracle lida com essas propriedades ACID e tem amplos recursos de recuperação para restaurar bancos de dados depois de falhas de sistema.

## Transações concorrentes

O software de banco de dados Oracle suporta muitos usuários interagindo com um banco de dados, e cada usuário pode executar suas próprias transações simultaneamente. Essas transações são conhecidas como transações *concorrentes*.

Se os usuários estão executando transações que afetam a mesma tabela, os efeitos dessas transações são separados uns dos outros até que uma instrução `COMMIT` seja executada. A seguinte seqüência de eventos, baseada em duas transações chamadas T1 e T2 que acessam a tabela `customers`, ilustra a separação das transações:

1. T1 e T2 executam uma instrução `SELECT` que recupera todas as linhas da tabela `customers`.
2. T1 executa uma instrução `INSERT` para adicionar uma linha na tabela `customers`, mas não executa uma instrução `COMMIT`.
3. T2 executa outra instrução `SELECT` e recupera as mesmas linhas do passo 1. T2 não “vê” a nova linha adicionada por T1 no passo 2.
4. T1 finalmente executa uma instrução `COMMIT` para registrar permanentemente a nova linha adicionada no passo 2.
5. T2 executa outra instrução `SELECT` e finalmente “vê” a nova linha adicionada por T1.

Resumindo: T2 não vê as alterações feitas por T1 até que este efetue `commit` nas suas alterações. Esse é o nível de isolamento padrão entre transações, mas, conforme você vai aprender na seção “Níveis de isolamento de transação”, é possível mudar o nível de isolamento.

A Tabela 8-1 mostra exemplos de instruções SQL que ilustram melhor o funcionamento das transações concorrentes. A tabela mostra a ordem intercalada na qual as instruções são executadas por duas transações chamadas T1 e T2. T1 recupera linhas, adiciona uma linha e atualiza uma linha na tabela `customers`. T2 recupera linhas da tabela `customers`. T2 não vê as alterações feitas por T1 até que este confirme suas alterações. É possível digitar as instruções mostradas na Tabela 8-1 e ver seus resultados iniciando duas sessões SQL\*Plus separadas e conectando-se como o usuário `store` nas duas sessões; você digita as instruções nas sessões no SQL\*Plus na ordem intercalada mostrada na tabela.

## Bloqueio de transação

Para suportar transações concorrentes, o software de banco de dados Oracle precisa garantir que os dados das tabelas permaneçam válidos. Ele faz isso utilizando *bloqueios*. Considere o exemplo a seguir, no qual duas transações chamadas T1 e T2 tentam modificar o cliente nº 1 na tabela `customers`:

1. T1 executa uma instrução `UPDATE` para modificar o cliente nº 1, mas não executa uma instrução `COMMIT`. Diz-se que T1 “bloqueou” a linha.
2. T2 também tenta executar uma instrução `UPDATE` para modificar o cliente nº 1, mas como essa linha já está bloqueada por T1, T2 é impedido de obter um bloqueio nela. A instrução `UPDATE` de T2 precisa esperar até que T1 termine e libere o bloqueio na linha.
3. T1 termina, executando uma instrução `COMMIT`, liberando assim o bloqueio na linha.
4. T2 obtém o bloqueio na linha e a instrução `UPDATE` é executada. T2 mantém o bloqueio na linha até terminar.

Resumindo: uma transação não pode obter um bloqueio em uma linha enquanto outra transação já mantém o bloqueio nessa linha.

Tabela 8-1 Transações concorrentes

Transação 1 T1	Transação 2 T2
(1) <code>SELECT *</code> <code>FROM customers;</code>	(2) <code>SELECT *</code> <code>FROM customers;</code>
(3) <code>INSERT INTO customers (</code> <code>customer_id, first_name, last_name</code> <code>) VALUES (</code> <code>7, 'Jason', 'Price'</code> <code>);</code>	
(4) <code>UPDATE customers</code> <code>SET last_name = 'Orange'</code> <code>WHERE customer_id = 2;</code>	
(5) <code>SELECT *</code> <code>FROM customers;</code> O conjunto de resultados retornado contém a nova linha e a atualização.	(6) <code>SELECT *</code> <code>FROM customers;</code> O conjunto de resultados retornado não contém a nova linha nem a atualização feita por T1. Em vez disso, o conjunto de resultados contém as linhas originais recuperadas no passo 2.
(7) <code>COMMIT;</code> Isso confirma a nova linha e a atualização.	
	(8) <code>SELECT *</code> <code>FROM customers;</code> O conjunto de resultados retornado contém a nova linha e a atualização feita por T1 nos passos 3 e 4.



**NOTA**  
*O modo mais fácil de entender o bloqueio padrão é como segue: leitores não bloqueiam leitores, gravadores não bloqueiam leitores e gravadores só bloqueiam gravadores quando eles tentam modificar a mesma linha.*

Níveis de isolamento de transação

O nível de isolamento de transação é o grau com que as alterações feitas por uma transação são separadas das outras transações em execução concomitante. Antes de ver os vários níveis de isolamento de transação disponíveis, é preciso entender os tipos de problemas que podem ocorrer quando transações concorrentes tentam acessar as mesmas linhas em uma tabela.

Na lista a seguir, você verá exemplos de duas transações concorrentes chamadas T1 e T2 que estão acessando as mesmas linhas; aparecem listados os três tipos de problemas de processamento de transação em potencial:

- **Leituras fantasmas** T1 lê um conjunto de linhas retornadas por uma cláusula WHERE especificada. Então, T2 insere uma nova linha, a qual também satisfaz a cláusula WHERE da consulta usada anteriormente por T1. T1 lê as linhas novamente, usando a mesma consulta, mas agora vê a linha adicional que T2 acabou de inserir. Essa nova linha é



conhecida como “fantasma” porque, para T1, essa linha parece ter aparecido como que por magia.

- **Leituras que não podem ser repetidas** T1 lê uma linha e T2 atualiza a mesma linha que T1 acabou de ler. Então, T1 lê novamente a mesma linha e descobre que a linha que leu anteriormente agora está diferente. Isso é conhecido como leitura “que não pode ser repetida”, pois a linha lida originalmente por T1 foi alterada.
- **Leituras sujas** T1 atualiza uma linha, mas não efetua commit na atualização. Então, T2 lê a linha atualizada. T1 realiza um rollback, desfazendo a atualização anterior. Agora a linha que T2 acabou de ler não é mais válida (ela está “suja”), pois a atualização feita por T1 não estava confirmada quando a linha foi lida por T2.

Para lidar com esses problemas em potencial, os bancos de dados implementam vários níveis de isolamento de transação para evitar que transações concorrentes interfiram umas nas outras. O padrão SQL define os seguintes níveis de isolamento de transação, mostrados em ordem crescente de isolamento:

- **READ UNCOMMITTED** Leituras fantasmas, leituras que não podem ser repetidas e leituras sujas são permitidas.
- **READ COMMITTED** Leituras fantasmas e leituras que não podem ser repetidas são permitidas, mas leituras sujas não.
- **REPEATABLE READ** Leituras fantasmas são permitidas, mas leituras que não podem ser repetidas e leituras sujas não.
- **SERIALIZABLE** Leituras fantasmas, leituras que não podem ser repetidas e leituras sujas não são permitidas.

O software de banco de dados Oracle suporta os níveis de isolamento de transação `READ COMMITTED` e `SERIALIZABLE`. Ele não suporta os níveis `READ UNCOMMITTED` e `REPEATABLE READ`. O nível de isolamento de transação normal definido pelo padrão SQL é `SERIALIZABLE`, mas o padrão usado pelo banco de dados Oracle é `READ COMMITTED`, que é aceitável para quase todas as aplicações.



### CUIDADO

*Embora possa usar `SERIALIZABLE` com o banco de dados Oracle, isso pode aumentar o tempo de conclusão de suas instruções SQL. Você só deve usar `SERIALIZABLE` se for absolutamente necessário.*

O nível de isolamento de transação é configurado com a instrução `SET TRANSACTION`. Por exemplo, a instrução a seguir define o nível de isolamento de transação como `SERIALIZABLE`:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

Você verá um exemplo de transação que utiliza o nível de isolamento `SERIALIZABLE` a seguir.

### Exemplo de transação `SERIALIZABLE`

Nesta seção, você verá um exemplo que mostra o efeito da configuração do nível de isolamento de transação como `SERIALIZABLE`. O exemplo utiliza duas transações, T1 e T2. T1 tem o nível de isolamento padrão `READ COMMITTED`; T2 tem o nível de isolamento de transação `SERIALIZABLE`. T1 e T2 lêem as linhas da tabela `customers` e, então, T1 insere uma nova linha e atualiza uma linha existente na tabela `customers`. Como T2 é `SERIALIZABLE`, não vê a linha inserida nem a atualização feita na linha existente realizada por T1, mesmo *depois* de T1 efetuar um `commit` nas alterações. Isso porque ler a linha inserida seria uma leitura fantasma e ler a atualização seria uma leitura que não pode ser repetida, as quais não são permitidas pelas transações `SERIALIZABLE`.

A Tabela 8-2 mostra as instruções SQL que compõem T1 e T2 na ordem intercalada na qual elas devem ser executadas.

Tabela 8-2 Transações `SERIALIZABLE`

Transação 1 T1 (READ COMMITTED)	Transação 2 T2 (SERIALIZABLE)
	(1) <code>SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;</code>
(3) <code>SELECT *</code> <code>FROM customers;</code>	(2) <code>SELECT *</code> <code>FROM customers;</code>
(4) <code>INSERT INTO customers ( customer_id, first_name, last_name ) VALUES ( 8, 'Steve', 'Button' );</code>	
(5) <code>UPDATE customers SET last_name = 'Yellow' WHERE customer_id = 3;</code>	
(6) <code>COMMIT;</code>	
(7) <code>SELECT *</code> <code>FROM customers;</code> O conjunto de resultados retornado contém a nova linha e a atualização.	(8) <code>SELECT *</code> <code>FROM customers;</code> O conjunto de resultados retornado <i>ainda</i> não contém a nova linha nem a atualização feita por T1. Isso porque T2 é <code>SERIALIZABLE</code> .

# CONSULTAS FLASHBACK

Se você efetuar commit por engano em alterações e quiser ver as linhas como eram originalmente, pode usar uma consulta flashback. É possível utilizar os resultados da consulta flashback a fim de alterar as linhas manualmente para que tenham seus valores originais de volta, caso seja necessário.

As consultas flashback podem ser baseadas em uma data/horário ou em um número de alteração de sistema (SCN). O banco de dados utiliza SCNs para rastrear as alterações feitas nos dados e você pode usá-los para voltar a um SCN em particular no banco de dados.

## Concedendo o privilégio de usar flashbacks

Os flashbacks utilizam o pacote PL/SQL DBMS\_FLASHBACK, para o qual você precisa ter o privilégio EXECUTE para executar. O exemplo a seguir se conecta como o usuário sys e concede o privilégio EXECUTE em DBMS\_FLASHBACK para o usuário store:

```
CONNECT sys/change_on_install AS sysdba
GRANT EXECUTE ON SYS.DBMS_FLASHBACK TO store;
```

**NOTA**  
*Fale com o administrador do seu banco de dados, caso não seja capaz de executar essas instruções. Você vai aprender sobre privilégios no próximo capítulo e sobre pacotes PL/SQL no Capítulo 11.*

## Consultas flashback de tempo

O exemplo a seguir se conecta como store e recupera as colunas product\_id, name e price das primeiras cinco linhas da tabela products:

```
CONNECT store/store_password
SELECT product_id, name, price
FROM products
WHERE product_id <= 5;
```

PRODUCT_ID	NAME	PRICE
1	Modern Science	19.95
2	Chemistry	30
3	Supernova	25.99
4	Tank War	13.95
5	Z Files	49.99

**NOTA**  
*Se você vir preços diferentes para qualquer um desses produtos, execute novamente o arquivo store\_schema.sql.*

O exemplo a seguir reduz o preço dessas linhas, confirma a alteração e recupera as linhas novamente para que você possa ver os novos preços:

```
UPDATE products
SET price = price * 0.75
WHERE product_id <= 5;
```

```
COMMIT;
```

```
SELECT product_id, name, price
FROM products
WHERE product_id <= 5;
```

PRODUCT_ID	NAME	PRICE
1	Modern Science	14.96
2	Chemistry	22.5
3	Supernova	19.49
4	Tank War	10.46
5	Z Files	37.49

A instrução a seguir executa a procedure `DBMS_FLASHBACK.ENABLE_AT_TIME()`, que permite voltar para uma data/horário específica; observe que a procedure `DBMS_FLASHBACK.ENABLE_AT_TIME()` aceita uma data/horário e que o exemplo passa `SYSDATE - 10 / 1440` para a procedure (essa expressão é avaliada como uma data/horário dez minutos no passado):

```
EXECUTE DBMS_FLASHBACK.ENABLE_AT_TIME(SYSDATE - 10 / 1440);
```

#### NOTA

24 horas x 60 minutos por hora = 1440 minutos. Portanto, `SYSDATE - 10 / 1440` é uma data/horário dez minutos no passado.

Todas as consultas que você executar agora exibirão as linhas como elas eram há dez minutos. Supondo que você executou a instrução `UPDATE` anterior há menos de dez minutos, a consulta a seguir exibirá os preços como eram antes que fossem atualizados:

```
SELECT product_id, name, price
FROM products
WHERE product_id <= 5;
```

PRODUCT_ID	NAME	PRICE
1	Modern Science	19.95
2	Chemistry	30
3	Supernova	25.99
4	Tank War	13.95
5	Z Files	49.99

Para desativar o flashback, execute `DBMS_FLASHBACK.DISABLE()`, como mostrado no exemplo a seguir:

```
EXECUTE DBMS_FLASHBACK.DISABLE();
```

#### CUIDADO

Você deve desativar um flashback antes de poder ativá-lo novamente.

Agora, quando você executar consultas, as linhas serão recuperadas conforme existem atualmente, como mostrado a seguir:

```
SELECT product_id, name, price
FROM products
WHERE product_id <= 5;
```

PRODUCT_ID	NAME	PRICE
1	Modern Science	14.96
2	Chemistry	22.5
3	Supernova	19.49
4	Tank War	10.46
5	Z Files	37.49

### Consultas flashback com número de alteração de sistema

Flashbacks baseados em números de alteração de sistema (SCNs) podem ser mais precisos do que aqueles baseados no tempo, pois o banco de dados utiliza SCNs para monitorar as alterações feitas nos dados. Para obter o SCN atual, execute `DBMS_FLASHBACK.GET_SYSTEM_CHANGE_NUMBER()`, como mostrado no exemplo a seguir:

```
VARIABLE current_scn NUMBER

EXECUTE :current_scn := DBMS_FLASHBACK.GET_SYSTEM_CHANGE_NUMBER();

PRINT current_scn

CURRENT_SCN
-----
292111
```

O exemplo a seguir adiciona uma linha na tabela `products`, confirma a alteração e recupera a nova linha:

```
INSERT INTO products (
    product_id, product_type_id, name, description, price
) VALUES (
    15, 1, 'Physics', 'Textbook on physics', 39.95
);

COMMIT;

SELECT *
FROM products
WHERE product_id = 15;

PRODUCT_ID PRODUCT_TYPE_ID NAME
-----
15          1 Physics
Textbook on physics
PRICE
-----
39.95
```

O exemplo a seguir executa a procedure `DBMS_FLASHBACK.ENABLE_AT_SYSTEM_CHANGE_NUMBER()`, que permite voltar a um SCN; observe que essa procedure aceita um SCN e que o exemplo passa a variável `current_scn` para ela:

```
EXECUTE DBMS_FLASHBACK.ENABLE_AT_SYSTEM_CHANGE_NUMBER(:current_scn);
```

Todas as consultas que você executar agora exibirão as linhas como eram no SCN armazenado em `current_scn`, antes da execução da instrução `INSERT`. A consulta a seguir tenta obter a linha com valor de `product_id` igual a 15; ela falha, porque essa nova linha foi adicionada após o SCN armazenado em `current_scn`:

```
SELECT product_id  
FROM products  
WHERE product_id = 15;  
  
no rows selected
```

Para desativar o flashback, execute `DBMS_FLASHBACK.DISABLE()`, como mostrado no exemplo a seguir:

```
EXECUTE DBMS_FLASHBACK.DISABLE();
```

Se você executar a consulta anterior novamente, verá a nova linha adicionada pela instrução `INSERT`.

#### NOTA

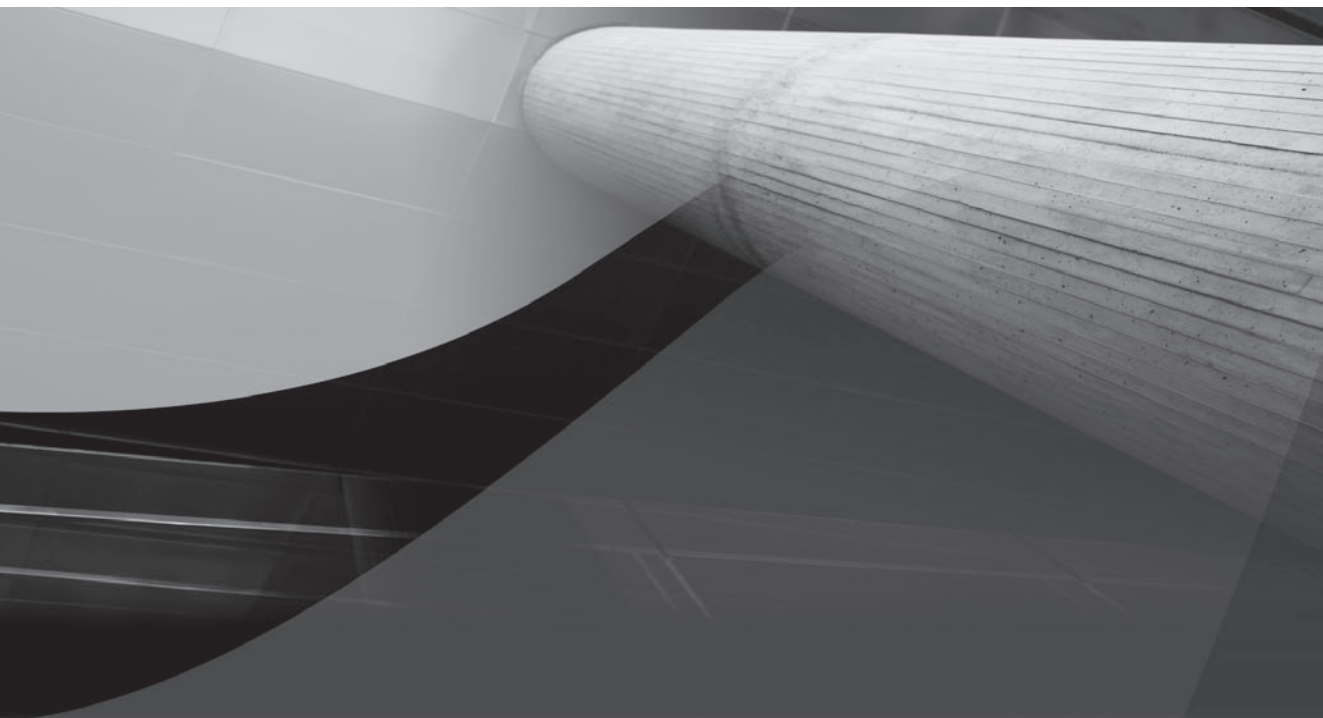
*Se você acompanhou os exemplos, execute novamente o script `store_schema.sql` para recriar tudo. Desse modo, os resultados de suas instruções SQL corresponderão aos que aparecem aqui, à medida que você avançar no restante deste livro.*

## RESUMO

Neste capítulo, você aprendeu:

- Como adicionar linhas usando a instrução `INSERT`
- Como modificar linhas usando a instrução `UPDATE`
- Como remover linhas usando a instrução `DELETE`
- Como o banco de dados mantém a integridade referencial por meio da aplicação de restrições
- Como usar a palavra-chave `DEFAULT` para especificar valores padrão para colunas
- Como mesclar linhas usando a instrução `MERGE`
- Que uma transação de banco de dados é um grupo de instruções SQL que compõem uma unidade lógica de trabalho
- Que o software de banco de dados Oracle pode manipular várias transações concorrentes
- Como usar consultas flashback para ver as linhas como eram originalmente antes de você alterá-las

No próximo capítulo, você vai aprender sobre usuários, privilégios e atribuições.

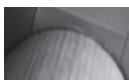


# CAPÍTULO 9

Usuários, privilégios e  
atribuições

Neste capítulo, você vai aprender:

- Mais sobre usuários
- Como os privilégios são usados para permitir que os usuários executem tarefas no banco de dados
- A explorar os dois tipos de privilégios: privilégios de sistema e privilégios de objeto
- Como os privilégios de sistema permitem realizar ações, como executar instruções DDL
- Como os privilégios de objeto permitem realizar ações, como executar instruções DML
- A explorar o agrupamento de privilégios em atribuições
- A fazer a auditoria da execução de instruções SQL



#### NOTA

*Se quiser acompanhar os exemplos, digite as instruções SQL mostradas neste capítulo; as instruções não estão contidas em um script.*

## USUÁRIOS

Nesta seção, você vai aprender a criar um usuário, a alterar a senha de um usuário e a remover um usuário. O termo “tablespace” é bastante usado neste capítulo. Os tablespaces são usados pelo banco de dados para armazenar objetos separados, os quais podem incluir tabelas, tipos, código PL/SQL etc. Normalmente, objetos relacionados são agrupados e armazenados no mesmo tablespace. Por exemplo, você poderia criar um aplicativo de entrada de pedidos e armazenar todos os objetos desse aplicativo em um único tablespace ou poderia criar um aplicativo de cadeia de abastecimento e armazenar os objetos desse aplicativo em um tablespace diferente. Para obter mais detalhes sobre tablespaces, consulte o manual *Oracle Database Concepts*, publicado pela Oracle Corporation.

### Criando um usuário

Para criar um usuário no banco de dados, use a instrução `CREATE USER`. A sintaxe simplificada da instrução `CREATE USER` é:

```
CREATE USER nome_usuario IDENTIFIED BY senha
[DEFAULT TABLESPACE tablespace_padrao]
[TEMPORARY TABLESPACE tablespace_temporario];
```

onde

- *nome\_usuario* é o nome do usuário do banco de dados.
- *senha* é a senha do usuário do banco de dados.
- *tablespace\_padrao* é o tablespace padrão onde os objetos do banco de dados são armazenados. Se você omitir um tablespace padrão, será utilizado o tablespace `SYSTEM` padrão, que sempre existe em um banco de dados.



- *tablespace\_temporário* é o tablespace padrão onde são armazenados os objetos temporários. Esses objetos incluem as tabelas temporárias, sobre as quais você vai aprender no próximo capítulo. Se você omitir um tablespace temporário, o tablespace *SYSTEM* padrão será usado.

O exemplo a seguir se conecta como *system* e cria um usuário chamado *jason* com a senha *price*:

```
CONNECT system/manager
CREATE USER jason IDENTIFIED BY price;
```

#### NOTA

*Se quiser acompanhar os exemplos, se conecte no banco de dados como um usuário privilegiado. No exemplo, utilizei o usuário *system*, que tem a senha *manager* no meu banco de dados.*

O exemplo a seguir cria um usuário chamado *henry* e especifica um tablespace padrão e um temporário:

```
CREATE USER henry IDENTIFIED BY hooray
DEFAULT TABLESPACE users
TEMPORARY TABLESPACE temp;
```

#### NOTA

*Se o seu banco de dados não tem tablespaces chamados *users* e *temp*, você pode pular este exemplo. O usuário *henry* não é usado em outras partes deste livro e foi incluído no exemplo apenas para ensinar como especificar tablespaces para um usuário. Você pode ver todos os tablespaces de um banco de dados conectando-se como o usuário *system* e executando a consulta `SELECT tablespace_name FROM dba_tablespace`.*

Um usuário precisa ter permissão para se movimentar em um banco de dados. Por exemplo, para se conectar no banco de dados, o usuário precisa receber permissão para criar uma sessão, que é o privilégio de sistema *CREATE SESSION*. As permissões são concedidas por um usuário privilegiado (*system*, por exemplo) utilizando a instrução *GRANT*. O exemplo a seguir concede a permissão *CREATE SESSION* a *jason*:

```
GRANT CREATE SESSION TO jason;
```

Agora o usuário *jason* poderá se conectar no banco de dados. O exemplo a seguir cria outros usuários que são utilizados neste capítulo e concede a eles o privilégio *CREATE SESSION*:

```
CREATE USER steve IDENTIFIED BY button;
CREATE USER gail IDENTIFIED BY seymour;
GRANT CREATE SESSION TO steve, gail;
```

## Alterando a senha de um usuário

É possível alterar a senha de um usuário utilizando a instrução *ALTER USER*. Por exemplo, a instrução a seguir altera a senha de *jason* para *marcus*:

```
ALTER USER jason IDENTIFIED BY marcus;
```

Você também pode alterar a senha do usuário que está conectado atualmente usando o comando `PASSWORD`. Depois que você digita `PASSWORD`, o SQL\*Plus pede para que digite a senha antiga e a nova senha duas vezes, para confirmar. O exemplo a seguir se conecta como `jason` e executa `PASSWORD`; observe que a senha é mascarada com o uso de asteriscos:

```
CONNECT jason/marcus
PASSWORD
Changing password for JASON
Old password: *****
New password: *****
Retype new password: *****
Password changed
```

## Excluindo um usuário

A instrução `DROP USER` exclui um usuário. O exemplo a seguir se conecta como `system` e usa `DROP USER` para excluir `jason`:

```
CONNECT system/manager
DROP USER jason;
```

### NOTA

*Você deve adicionar a palavra-chave `CASCADE` depois do nome do usuário na instrução `DROP USER`, caso o esquema desse usuário contenha objetos, como tabelas, por exemplo. Entretanto, antes de fazer isso, você deve certificar-se de que nenhum outro usuário precise acessar esses objetos.*

## PRIVILÉGIOS DE SISTEMA

Um *privilegio de sistema* permite ao usuário realizar certas ações dentro do banco de dados, como executar instruções DDL. Por exemplo, `CREATE TABLE` permite que o usuário crie uma tabela em seu esquema. Alguns dos privilégios de sistema comumente usados estão mostrados na Tabela 9-1.

### NOTA

*É possível obter a lista completa de privilégios de sistema no manual Oracle Database SQL Reference, publicado pela Oracle Corporation.*

Conforme verá posteriormente, os privilégios podem ser agrupados em *atribuições*. Duas atribuições úteis para conceder a um usuário são `CONNECT` e `RESOURCE`; `CONNECT` permite que o usuário se conecte no banco de dados; `RESOURCE` permite criar vários objetos de banco de dados, como tabelas, seqüências, código PL/SQL etc.

## Concedendo privilégio de sistema a um usuário

A instrução `GRANT` concede privilégio de sistema a um usuário. O exemplo a seguir concede alguns privilégios de sistema a `steve` (supondo que você ainda esteja conectado no banco de dados como `system`):

```
GRANT CREATE SESSION, CREATE USER, CREATE TABLE TO steve;
```

**Tabela 9-1** *Privilégios de sistema mais usados*

Privilégio de sistema	Permite que você...
CREATE SESSION	Conecta-se em um banco de dados.
CREATE SEQUENCE	Crie uma seqüência, que é uma série de números normalmente usados para preencher uma coluna de chave primária automaticamente. Você vai aprender sobre seqüências no próximo capítulo.
CREATE SYNONYM	Crie um sinônimo. Um sinônimo permite que você referencie uma tabela em outro esquema. Você vai aprender sobre sinônimos posteriormente neste capítulo.
CREATE TABLE	Crie uma tabela no esquema do usuário.
CREATE ANY TABLE	Crie uma tabela em qualquer esquema.
DROP TABLE	Exclua uma tabela do esquema do usuário.
DROP ANY TABLE	Exclua uma tabela de qualquer esquema.
CREATE PROCEDURE	Crie uma procedure armazenada.
EXECUTE ANY PROCEDURE	Execute uma procedure em qualquer esquema.
CREATE USER	Crie um usuário.
DROP USER	Exclua um usuário.
CREATE VIEW	Crie uma visão. Uma visão é uma consulta armazenada que permite acessar várias tabelas e colunas. Ela pode ser consultada da mesma forma que uma tabela. Você vai aprender sobre visões no próximo capítulo.

Você também pode usar `WITH ADMIN OPTION` para permitir que o usuário conceda um privilégio a outro usuário. O exemplo a seguir concede a `steve` o privilégio `EXECUTE ANY PROCEDURE` com a opção `ADMIN`:

```
GRANT EXECUTE ANY PROCEDURE TO steve WITH ADMIN OPTION;
```

`EXECUTE ANY PROCEDURE` pode então ser concedido por `steve` a outro usuário. O exemplo a seguir se conecta como `steve` e concede `EXECUTE ANY PROCEDURE` a `gail`:

```
CONNECT steve/button  
GRANT EXECUTE ANY PROCEDURE TO gail;
```

Você pode conceder um privilégio a todos os usuários, concedendo-o para `PUBLIC*`. O exemplo a seguir se conecta como `system` e concede `EXECUTE ANY PROCEDURE` para `PUBLIC`:

```
CONNECT system/manager  
GRANT EXECUTE ANY PROCEDURE TO PUBLIC;
```

Agora, todos os usuários do banco de dados têm o privilégio `EXECUTE ANY PROCEDURE`.

---

\* N. de R.T.: Conceder privilégios para `PUBLIC` significa conceder os privilégios para todos os usuários do banco de dados. `PUBLIC` é uma conta virtual de usuário que representa todos os usuários. Tenha o cuidado de conceder à `PUBLIC` apenas os privilégios que devem ser utilizados por todos os usuários.

## Verificando os privilégios de sistema concedidos a um usuário

Você pode verificar quais privilégios de sistema um usuário tem consultando `user_sys_privs`. A Tabela 9-2 descreve algumas das colunas existentes em `user_sys_privs`.



### NOTA

`user_sys_privs` faz parte do dicionário de dados do banco de dados Oracle. O dicionário de dados armazena informações sobre o banco de dados.

O exemplo a seguir se conecta como `steve` e consulta `user_sys_privs`:

```
CONNECT steve/button
SELECT *
FROM user_sys_privs
ORDER BY privilege;
```

USERNAME	PRIVILEGE	ADM
-----	-----	---
STEVE	CREATE SESSION	NO
STEVE	CREATE TABLE	NO
STEVE	CREATE USER	NO
PUBLIC	EXECUTE ANY PROCEDURE	NO
STEVE	EXECUTE ANY PROCEDURE	YES

O exemplo a seguir se conecta como `gail` e consulta `user_sys_privs`:

```
CONNECT gail/seymour
SELECT *
FROM user_sys_privs
ORDER BY privilege;
```

USERNAME	PRIVILEGE	ADM
-----	-----	---
GAIL	CREATE SESSION	NO
GAIL	EXECUTE ANY PROCEDURE	NO
PUBLIC	EXECUTE ANY PROCEDURE	NO

Observe que `gail` tem o privilégio `EXECUTE ANY PROCEDURE` que foi concedido anteriormente por `steve`.

**Tabela 9-2** Algumas colunas existentes em `user_sys_privs`

Coluna	Tipo	Descrição
username	VARCHAR2 (30)	Nome do usuário atual
privilege	VARCHAR2 (40)	O privilégio de sistema que o usuário possui
admin_option	VARCHAR2 (3)	Se o usuário é capaz de conceder o privilégio a outro usuário

## Utilizando privilégios de sistema

Uma vez que um usuário tenha recebido um privilégio de sistema, ele pode utilizá-lo para executar a tarefa especificada. Por exemplo, `steve` tem o privilégio `CREATE USER`; portanto, ele é capaz de criar um usuário:

```
CONNECT steve/button
CREATE USER roy IDENTIFIED BY williams;
```

Se `steve` tentasse usar um privilégio de sistema que não possui, o banco de dados retornaria o erro `ORA-01031: insufficient privileges`. Por exemplo, `steve` não tem o privilégio `DROP USER` e, no exemplo a seguir, ele tenta excluir `roy` e falha:

```
SQL> DROP USER roy;
DROP USER roy
*
ERROR at line 1:
ORA-01031: insufficient privileges
```

## Revogando privilégios de sistema de um usuário

Você revoga privilégios de sistema de um usuário utilizando `REVOKE`. O exemplo a seguir se conecta como `system` e revoga o privilégio `CREATE TABLE` de `steve`:

```
CONNECT system/manager
REVOKE CREATE TABLE FROM steve;
```

O exemplo a seguir revoga o privilégio `EXECUTE ANY PROCEDURE` de `steve`:

```
REVOKE EXECUTE ANY PROCEDURE FROM steve;
```

Quando você revoga `EXECUTE ANY PROCEDURE` de `steve` — que já passou esse privilégio para `gail` —, `gail` ainda mantém o privilégio:

```
CONNECT gail/seymour
SELECT *
FROM user_sys_privs
ORDER BY privilege;
```

USERNAME	PRIVILEGE	ADM
GAIL	CREATE SESSION	NO
GAIL	EXECUTE ANY PROCEDURE	NO
PUBLIC	EXECUTE ANY PROCEDURE	NO

## PRIVILÉGIOS DE OBJETO

Um *privilégio de objeto* permite que o usuário realize certas ações nos objetos do banco de dados, como executar instruções DML em tabelas. Por exemplo, `INSERT ON store.products` permite que o usuário insira linhas na tabela `products` do esquema `store`. Alguns dos privilégios de objeto mais usados estão mostrados na Tabela 9-3.

**Tabela 9-3** *Privilégios de objeto comumente usados*

Privilégio de objeto	Permite que o usuário...
SELECT	Realize uma seleção
INSERT	Realize uma inserção
UPDATE	Realize uma atualização
DELETE	Realize uma exclusão
EXECUTE	Execute uma procedure armazenada



**NOTA**

*Você pode obter a lista completa de privilégios de sistema no manual Oracle Database SQL Reference, publicado pela Oracle Corporation.*

### Concedendo privilégios de objeto a um usuário

A instrução `GRANT` concede um privilégio de objeto a um usuário. O exemplo a seguir se conecta como `store` e concede os privilégios de objeto `SELECT`, `INSERT` e `UPDATE` na tabela `products` a `steve`, junto com o privilégio `SELECT` na tabela `employees`:

```
CONNECT store/store_password
GRANT SELECT, INSERT, UPDATE ON store.products TO steve;
GRANT SELECT ON store.employees TO steve;
```

O exemplo a seguir concede a `steve` o privilégio `UPDATE` nas colunas `last_name` e `salary`:

```
GRANT UPDATE (last_name, salary) ON store.employees TO steve;
```

Você também pode usar a opção `GRANT` para permitir que um usuário conceda um privilégio a outro usuário. O exemplo a seguir concede a `steve` o privilégio `SELECT` na tabela `customers` com a opção `GRANT`:

```
GRANT SELECT ON store.customers TO steve WITH GRANT OPTION;
```



**NOTA**

*A opção `GRANT` permite que um usuário conceda um privilégio de objeto a outro usuário e a opção `ADMIN` permite que um usuário conceda um privilégio de sistema a outro usuário.*

Então, o privilégio `SELECT ON store.customers` pode ser concedido por `steve` a outro usuário. O exemplo a seguir se conecta como `steve` e concede esse privilégio a `gail`:

```
CONNECT steve/button
GRANT SELECT ON store.customers TO gail;
```

## Verificando os privilégios de objeto concedidos

Você pode verificar quais privilégios de objeto de tabela um usuário concedeu a outros usuários consultando `user_tab_privs_made`. A Tabela 9-4 documenta as colunas de `user_tab_privs_made`. O exemplo a seguir se conecta como `store` e consulta `user_tab_privs_made`. Como existem muitas linhas, limitamos as linhas recuperadas àquelas onde `table_name` é `PRODUCTS`:

```
CONNECT store/store_password
SELECT *
FROM user_tab_privs_made
WHERE table_name = 'PRODUCTS';
```

GRANTEE	TABLE_NAME		
GRANTOR	PRIVILEGE	GRA	HIE
STEVE	PRODUCTS		
STORE	INSERT	NO	NO
STEVE	PRODUCTS		
STORE	SELECT	NO	NO
STEVE	PRODUCTS		
STORE	UPDATE	NO	NO

Você pode verificar quais privilégios de objeto de coluna um usuário concedeu consultando `user_col_privs_made`. A Tabela 9-5 documenta as colunas de `user_col_privs_made`.

**Tabela 9-4** Algumas colunas de `user_tab_privs_made`

Coluna	Tipo	Descrição
grantee	VARCHAR2 (30)	O usuário a quem o privilégio foi concedido
table_name	VARCHAR2 (30)	Nome do objeto (como uma tabela, por exemplo) no qual o privilégio foi concedido
grantor	VARCHAR2 (30)	O usuário que concedeu o privilégio
privilege	VARCHAR2 (40)	O privilégio no objeto
grantable	VARCHAR2 (3)	Se o beneficiado pode conceder o privilégio a outro (YES ou NO)
hierarchy	VARCHAR2 (3)	Se o privilégio faz parte de uma hierarquia (YES ou NO)

**Tabela 9-5** Algumas colunas de `user_col_privs_made`

Coluna	Tipo	Descrição
grantee	VARCHAR2 (30)	O usuário a quem o privilégio foi concedido
table_name	VARCHAR2 (30)	Nome do objeto no qual o privilégio foi concedido
column_name	VARCHAR2 (30)	Nome do objeto no qual o privilégio foi concedido
grantor	VARCHAR2 (30)	O usuário que concedeu o privilégio
privilege	VARCHAR2 (40)	O privilégio no objeto
grantable	VARCHAR2 (3)	Se o beneficiário pode conceder o privilégio a outro (YES ou NO)

O exemplo a seguir consulta `user_col_privs_made`:

```
SELECT *
FROM user_col_privs_made;
```

GRANTEE	TABLE_NAME	
-----	-----	
COLUMN_NAME	GRANTOR	
-----	-----	
PRIVILEGE		GRA
-----		---
STEVE	EMPLOYEES	
LAST_NAME	STORE	
UPDATE		NO
STEVE	EMPLOYEES	
SALARY	STORE	
UPDATE		NO

### Verificando os privilégios de objeto recebidos

Você pode verificar quais privilégios de objeto em uma tabela um usuário recebeu consultando a tabela `user_tab_privs_rec`. A Tabela 9-6 documenta as colunas de `user_tab_privs_rec`. O exemplo a seguir se conecta como `steve` e consulta `user_tab_privs_rec`:



Tabela 9-6 Algumas colunas de user\_tab\_privs\_recd

Coluna	Tipo	Descrição
owner	VARCHAR2 (30)	O usuário que possui o objeto
table_name	VARCHAR2 (30)	Nome do objeto no qual o privilégio foi concedido
grantor	VARCHAR2 (30)	O usuário que concedeu o privilégio
privilege	VARCHAR2 (40)	Privilégio no objeto
grantable	VARCHAR2 (3)	Se o beneficiário pode conceder o privilégio a outro (YES ou NO)
hierarchy	VARCHAR2 (3)	Se o privilégio faz parte de uma hierarquia (YES ou NO)

```
CONNECT steve/button
SELECT *
FROM user_tab_privs_recd
ORDER BY privilege;
```

OWNER	TABLE_NAME		
GRANTOR	PRIVILEGE	GRA	HIE
STORE	PRODUCTS		
STORE	INSERT	NO	NO
STORE	CUSTOMERS		
STORE	SELECT	YES	NO
STORE	EMPLOYEES		
STORE	SELECT	NO	NO
STORE	PRODUCTS		
STORE	SELECT	NO	NO
STORE	PRODUCTS		
STORE	UPDATE	NO	NO

Você pode verificar quais privilégios de objeto de coluna um usuário recebeu consultando user\_col\_privs\_recd. A Tabela 9-7 documenta as colunas de user\_col\_privs\_recd.

**Tabela 9-7** Algumas colunas de `user_col_privs_rec`

Coluna	Tipo	Descrição
owner	VARCHAR2 (30)	O usuário que possui o objeto
table_name	VARCHAR2 (30)	Nome da tabela na qual o privilégio foi concedido
column_name	VARCHAR2 (30)	Nome da coluna na qual o privilégio foi concedido
grantor	VARCHAR2 (30)	O usuário que concedeu o privilégio
privilege	VARCHAR2 (40)	Privilégio no objeto
granttable	VARCHAR2 (3)	Se o beneficiário pode conceder o privilégio a outro (YES ou NO)

O exemplo a seguir consulta `user_col_privs_rec`:

```
SELECT *
FROM user_col_privs_rec;
```

OWNER	TABLE_NAME	
-----	-----	
COLUMN_NAME	GRANTOR	
-----	-----	
PRIVILEGE		GRA
-----		---
STORE	EMPLOYEES	
LAST_NAME	STORE	
UPDATE		NO
STORE	EMPLOYEES	
SALARY	STORE	
UPDATE		NO

## Utilizando privilégios de objeto

Depois que um usuário recebeu um privilégio de objeto, ele pode utilizá-lo para executar a tarefa especificada. Por exemplo, `steve` tem o privilégio `SELECT` em `store.customers`:

```
CONNECT steve/button
SELECT *
FROM store.customers;
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB	PHONE
-----	-----	-----	-----	-----
1	John	Brown	01-JAN-65	800-555-1211
2	Cynthia	Green	05-FEB-68	800-555-1212
3	Steve	White	16-MAR-71	800-555-1213
4	Gail	Black		800-555-1214
5	Doreen	Blue	20-MAY-70	

Se *steve* tentasse recuperar da tabela *purchases* — para a qual ele não tem qualquer permissão —, o banco de dados retornaria o erro `ORA-00942: table or view does not exist`:

```
SQL> SELECT *
      2 FROM store.purchases;
FROM store.purchases
      *
ERROR at line 2:
ORA-00942: table or view does not exist
```

## Sinônimos

Nos exemplos da seção anterior, você viu que pode acessar tabelas em outro esquema especificando o nome do esquema, seguido da tabela. Por exemplo, quando *steve* recuperou linhas da tabela *customers* no esquema *store*, ele executou uma consulta em *store.customers*. Você pode evitar a digitação do nome do esquema criando um *sinônimo* para uma tabela, com a instrução `CREATE SYNONYM`. Vejamos um exemplo. Primeiro, conecte-se como *system* e conceda o privilégio de sistema `CREATE SYNONYM` a *steve*:

```
CONNECT system/manager
GRANT CREATE SYNONYM TO steve;
```

Em seguida, conecte-se como *steve* e execute a instrução `CREATE SYNONYM` para criar um sinônimo para a tabela *store.customers*:

```
CONNECT steve/button
CREATE SYNONYM customers FOR store.customers;
```

Para recuperar linhas de *store.customers*, *steve* só precisa referenciar o sinônimo *customers* na cláusula `FROM` de uma instrução `SELECT`. Por exemplo:

```
SELECT *
FROM customers;
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB	PHONE
1	John	Brown	01-JAN-65	800-555-1211
2	Cynthia	Green	05-FEB-68	800-555-1212
3	Steve	White	16-MAR-71	800-555-1213
4	Gail	Black		800-555-1214
5	Doreen	Blue	20-MAY-70	

## Sinônimos públicos

Você também pode criar um sinônimo *público* para uma tabela, assim todos os usuários vêem o sinônimo. As seguintes tarefas

- Conectar-se como *system*
- Conceder o privilégio de sistema `CREATE PUBLIC SYNONYM` para *store*

- Conectar-se como store
- Criar um sinônimo público chamado products para store.products

são executadas pelas instruções a seguir:

```
CONNECT system/manager
GRANT CREATE PUBLIC SYNONYM TO store;
CONNECT store/store_password
CREATE PUBLIC SYNONYM products FOR store.products;
```

Se você se conectar como steve, que tem o privilégio SELECT em store.products, poderá recuperar linhas de store.products por meio do sinônimo público products:

```
CONNECT steve/button
SELECT *
FROM products;
```

Mesmo que um sinônimo público tenha sido criado para store.products, um usuário ainda precisa de privilégios de objeto nessa tabela para realmente acessá-la. Por exemplo, gail pode ver o sinônimo público products, mas não tem privilégios de objeto em store.products. Portanto, se gail tentar recuperar linhas de products, o banco de dados retornará o erro ORA-00942: table or view does not exist:

```
SQL> CONNECT gail/seymour
Connected.
SQL> SELECT * FROM products;
SELECT * FROM products
      *
ERROR at line 1:
ORA-00942: table or view does not exist
```

Se gail tivesse o privilégio de objeto SELECT na tabela store.products, a instrução SELECT anterior teria sucesso. Se um usuário tem outros privilégios de objeto, ele pode exercê-los por meio de um sinônimo. Por exemplo, se gail tivesse o privilégio de objeto INSERT na tabela store.products, poderia adicionar uma linha em store.products por meio do sinônimo products.

## Revogando privilégios de objeto

Você pode revogar privilégios de objeto usando REVOKE. O exemplo a seguir se conecta como store e revoga de steve o privilégio INSERT na tabela products:

```
CONNECT store/store_password
REVOKE INSERT ON products FROM steve;
```

O exemplo a seguir revoga de steve o privilégio SELECT na tabela customers:

```
REVOKE SELECT ON store.customers FROM steve;
```

Quando você revoga SELECT ON store.customers de steve — que já passou esse privilégio a gail —, gail também perde o privilégio.

## ATRIBUIÇÕES (ROLES)

Uma *atribuição* é um grupo de privilégios que você pode conceder a um usuário ou a outra atribuição. Os pontos a seguir resumem as vantagens e características das atribuições:

- Em vez de conceder um privilégio por vez diretamente a um usuário, você pode criar uma atribuição, conceder privilégios a essa atribuição e, então, conceder a atribuição a vários usuários e atribuições.
- Quando você adiciona ou exclui um privilégio de uma atribuição, todos os usuários e atribuições associados a esta atribuição recebem ou perdem automaticamente esse privilégio.
- Você pode conceder várias atribuições a um usuário ou a uma atribuição.
- Você pode designar uma senha para uma atribuição.

Como você pode ver a partir desses pontos, as atribuições podem ajudá-lo a gerenciar vários privilégios atribuídos a vários usuários.

### Criando atribuições

Para criar uma atribuição, você deve ter o privilégio de sistema `CREATE ROLE`. Conforme você verá em um exemplo posterior, o usuário `store` também precisa da capacidade de conceder o privilégio de sistema `CREATE USER` com a opção `ADMIN`. O exemplo a seguir se conecta como `system` e concede os privilégios necessários a `store`:

```
CONNECT system/manager
GRANT CREATE ROLE TO store;
GRANT CREATE USER TO store WITH ADMIN OPTION;
```

A Tabela 9-8 mostra as atribuições que você vai criar em breve. Você cria uma atribuição com a instrução `CREATE ROLE`. As instruções a seguir se conectam como `store` e criam as três atribuições mostradas na Tabela 9-8:

```
CONNECT store/store_password
CREATE ROLE product_manager;
CREATE ROLE hr_manager;
CREATE ROLE overall_manager IDENTIFIED by manager_password;
```

Observe que `overall_manager` tem a senha `manager_password`.

**Tabela 9-8** *Atribuições a serem criadas*

Nome da atribuição	Tem permissões para
<code>product_manager</code>	Executar operações <code>SELECT</code> , <code>INSERT</code> , <code>UPDATE</code> e <code>DELETE</code> nas tabelas <code>product_types</code> e <code>products</code> .
<code>hr_manager</code>	Executar operações <code>SELECT</code> , <code>INSERT</code> , <code>UPDATE</code> e <code>DELETE</code> nas tabelas <code>salary_grades</code> e <code>employees</code> . Além disso, <code>hr_manager</code> é capaz de criar usuários.
<code>overall_manager</code>	Executar operações <code>SELECT</code> , <code>INSERT</code> , <code>UPDATE</code> e <code>DELETE</code> em todas as tabelas mostradas nas atribuições anteriores; <code>overall_manager</code> terá as atribuições anteriores.

### Concedendo privilégios a atribuições

Você concede privilégios a uma atribuição com a instrução GRANT. É possível conceder privilégios de sistema e de objeto a uma atribuição, assim como conceder outra atribuição a uma atribuição. O exemplo a seguir concede os privilégios necessários às atribuições product\_manager e hr\_manager e concede essas duas atribuições à overall\_manager:

```
GRANT SELECT, INSERT, UPDATE, DELETE ON product_types TO product_manager;
GRANT SELECT, INSERT, UPDATE, DELETE ON products TO product_manager;
GRANT SELECT, INSERT, UPDATE, DELETE ON salary_grades TO hr_manager;
GRANT SELECT, INSERT, UPDATE, DELETE ON employees TO hr_manager;
GRANT CREATE USER TO hr_manager;
GRANT product_manager, hr_manager TO overall_manager;
```

### Concedendo atribuições a um usuário

Você concede uma atribuição a um usuário com o comando GRANT. O exemplo a seguir concede a atribuição overall\_manager a steve:

```
GRANT overall_manager TO steve;
```

### Verificando as atribuições concedidas a um usuário

Você pode verificar quais atribuições foram concedidas a um usuário consultando user\_role\_privs. A Tabela 9-9 define as colunas de user\_role\_privs. O exemplo a seguir se conecta como steve e consulta user\_role\_privs:

```
CONNECT steve/button
SELECT *
FROM user_role_privs;
```

USERNAME	GRANTED_ROLE	ADM	DEF	OS_
-----	-----	-----	-----	-----
STEVE	OVERALL_MANAGER	NO	YES	NO

Tabela 9-9 Algumas colunas de user\_role\_privs

Coluna	Tipo	Descrição
username	VARCHAR2 (30)	Nome do usuário a quem a atribuição foi concedida
granted_role	VARCHAR2 (30)	Nome da atribuição concedida ao usuário
admin_option	VARCHAR2 (3)	Se o usuário é capaz de conceder a atribuição a outro usuário ou a outra atribuição (YES ou NO)
default_role	VARCHAR2 (3)	Se a atribuição é ativada por padrão quando o usuário se conecta no banco de dados (YES ou NO)
os_granted	VARCHAR2 (3)	Se a atribuição foi concedida pelo sistema operacional (YES ou NO)

Um usuário que cria uma atribuição também recebe essa atribuição, por padrão. O exemplo a seguir se conecta como store e consulta user\_role\_privs:

```
CONNECT store/store_password
SELECT *
FROM user_role_privs;
```

USERNAME	GRANTED_ROLE	ADM	DEF	OS_
-----	-----	---	---	---
STORE	CONNECT	NO	YES	NO
STORE	HR_MANAGER	YES	YES	NO
STORE	OVERALL_MANAGER	YES	YES	NO
STORE	PRODUCT_MANAGER	YES	YES	NO
STORE	RESOURCE	NO	YES	NO

Observe que store tem as atribuições CONNECT e RESOURCE, além das atribuições que criou anteriormente.

#### NOTA

CONNECT e RESOURCE são atribuições internas que foram concedidas a store quando você executou o script store\_schema.sql. Conforme será visto na próxima seção, as atribuições CONNECT e RESOURCE contêm vários privilégios.

## Verificando os privilégios de sistema concedidos a uma atribuição

Você pode verificar quais privilégios de sistema foram concedidos a uma atribuição consultando role\_sys\_privs. A Tabela 9-10 define as colunas de role\_sys\_privs. O exemplo a seguir recupera as linhas de role\_sys\_privs (supondo que você ainda esteja conectado como store):

```
SELECT *
FROM role_sys_privs
ORDER BY privilege;
```

ROLE	PRIVILEGE	ADM
-----	-----	---
RESOURCE	CREATE CLUSTER	NO
RESOURCE	CREATE INDEXTYPE	NO
RESOURCE	CREATE OPERATOR	NO
RESOURCE	CREATE PROCEDURE	NO
RESOURCE	CREATE SEQUENCE	NO
CONNECT	CREATE SESSION	NO
RESOURCE	CREATE TABLE	NO
RESOURCE	CREATE TRIGGER	NO
RESOURCE	CREATE TYPE	NO
HR_MANAGER	CREATE USER	NO

Note que a atribuição RESOURCE tem muitos privilégios atribuídos.

#### NOTA

A consulta anterior foi executada no Oracle Database 11g. Se você estiver usando uma versão diferente do software de banco de dados, seus resultados podem ser diferentes.

Tabela 9-10 Algumas colunas de role\_sys\_privs

Coluna	Tipo	Descrição
role	VARCHAR2 (30)	Nome da atribuição
privilege	VARCHAR2 (40)	Privilégio de sistema concedido à atribuição
admin_option	VARCHAR2 (3)	Se o privilégio foi concedido com a opção ADMIN (YES ou NO)

Verificando os privilégios de objeto concedidos a uma atribuição

Você pode verificar quais privilégios de objeto foram concedidos a uma atribuição consultando role\_tab\_privs. A Tabela 9-11 define as colunas de role\_tab\_privs. O exemplo a seguir consulta role\_tab\_privs onde a atribuição é HR\_MANAGER:

```
SELECT *
FROM role_tab_privs
WHERE role='HR_MANAGER'
ORDER BY table_name;
```

ROLE	OWNER	
-----		
TABLE_NAME	COLUMN_NAME	
-----		
PRIVILEGE		GRA
-----		
HR_MANAGER	STORE	
EMPLOYEES		
DELETE		NO
HR_MANAGER	STORE	
EMPLOYEES		
INSERT		NO
HR_MANAGER	STORE	
EMPLOYEES		
SELECT		NO
HR_MANAGER	STORE	
EMPLOYEES		
UPDATE		NO
HR_MANAGER	STORE	
SALARY_GRADES		
DELETE		NO
HR_MANAGER	STORE	
SALARY_GRADES		
INSERT		NO



```

HR_MANAGER          STORE
SALARY_GRADES
SELECT              NO

HR_MANAGER          STORE
SALARY_GRADES
UPDATE              NO

```

## Utilizando os privilégios concedidos a uma atribuição

Uma vez que o usuário tenha recebido um privilégio por meio de uma atribuição, ele pode utilizar esse privilégio para executar as tarefas autorizadas. Por exemplo, `steve` tem a atribuição `overall_manager`, que recebeu as atribuições `product_manager` e `hr_manager` roles. A atribuição `product_manager` recebeu o privilégio de objeto `SELECT` nas tabelas `products` e `product_types`. Portanto, `steve` é capaz de recuperar linhas dessas tabelas, como mostrado no exemplo a seguir:

```

CONNECT steve/button
SELECT p.name, pt.name
FROM store.products p, store.product_types pt
WHERE p.product_type_id = pt.product_type_id;

```

NAME	NAME
-----	-----
Modern Science	Book
Chemistry	Book
Supernova	Video
Tank War	Video
Z Files	Video
2412: The Return	Video
Space Force 9	DVD
From Another Planet	DVD
Classical Music	CD
Pop 3	CD
Creative Yell	CD

**Tabela 9-11** *Algumas colunas de `role_tab_privs`*

Coluna	Tipo	Descrição
<code>role</code>	<code>VARCHAR2 (30)</code>	Usuário para quem o privilégio foi concedido
<code>owner</code>	<code>VARCHAR2 (30)</code>	O usuário que possui o objeto
<code>table_name</code>	<code>VARCHAR2 (30)</code>	Nome do objeto no qual o privilégio foi concedido
<code>column_name</code>	<code>VARCHAR2 (30)</code>	Nome da coluna (se aplicável)
<code>privilege</code>	<code>VARCHAR2 (40)</code>	Privilégio no objeto
<code>grantable</code>	<code>VARCHAR2 (3)</code>	Se o privilégio foi concedido com a opção <code>GRANT</code> (YES ou NO)

## Atribuições padrão

Por padrão, quando uma atribuição é concedida a um usuário, ela é ativada para o usuário. Isso significa que, quando o usuário se conecta no banco de dados, a atribuição está automaticamente disponível para ele. Para melhorar a segurança, você pode desativar uma atribuição por padrão; quando o usuário se conectar, ele mesmo terá que ativar a atribuição antes de poder utilizá-la. Se a atribuição tem uma senha, o usuário precisará digitar essa senha antes que a atribuição seja ativada. Por exemplo, a atribuição `overall_manager` tem a senha `manager_password` e foi concedida a `steve`. No exemplo a seguir, você vai desativar `overall_manager` para que `steve` precise ativar essa atribuição e digitar a senha antes de poder utilizá-la. Você faz isso alterando a atribuição de modo que ela não seja mais uma atribuição padrão, usando a instrução `ALTER ROLE`. O exemplo a seguir se conecta como `system` e altera `steve` para que `overall_manager` não seja mais uma atribuição padrão:

```
CONNECT system/manager
ALTER USER steve DEFAULT ROLE ALL EXCEPT overall_manager;
```

Quando você se conectar como `steve`, precisará ativar `overall_manager` usando `SET ROLE`:

```
CONNECT steve/button
SET ROLE overall_manager IDENTIFIED BY manager_password;
```

Depois de configurar a atribuição, você poderá usar os privilégios concedidos a essa atribuição. Você pode configurar sua atribuição como “nenhuma” (isto é, nenhuma atribuição) usando a seguinte instrução:

```
SET ROLE NONE;
```

Você também pode configurar sua atribuição como “todas as atribuições”, exceto `overall_manager`, usando a seguinte instrução:

```
SET ROLE ALL EXCEPT overall_manager;
```

Designar senhas às atribuições e configurá-las para não serem ativadas por padrão para um usuário aumenta o nível de segurança.

## Revogando uma atribuição

`REVOKE` revoga uma atribuição. O exemplo a seguir se conecta como `store` e revoga a atribuição `overall_manager` de `steve`:

```
CONNECT store/store_password
REVOKE overall_manager FROM steve;
```

## Revogando privilégios de uma atribuição

Você revoga um privilégio de uma atribuição com `REVOKE`. O exemplo a seguir se conecta como `store` e revoga de `product_manager` todos os privilégios nas tabelas `products` e `product_types` (supondo que você ainda esteja conectado como `store`):

```
REVOKE ALL ON products FROM product_manager;
REVOKE ALL ON product_types FROM product_manager;
```

## Excluindo uma atribuição

Você exclui uma atribuição com `DROP ROLE`. O exemplo a seguir exclui as atribuições `overall_manager`, `product_manager` e `hr_manager` (supondo que você ainda esteja conectado como `store`):

```
DROP ROLE overall_manager;  
DROP ROLE product_manager;  
DROP ROLE hr_manager;
```

## AUDITORIA

O software de banco de dados Oracle contém recursos de auditoria que permitem monitorar as operações do banco de dados. A auditoria de algumas operações pode ser feita em um nível alto, como as tentativas mal-sucedidas de registrar-se no banco de dados, enquanto a de outras podem ser feitas em um nível detalhado, como quando um usuário recuperou linhas de uma tabela específica. Normalmente, o administrador do banco de dados será responsável por ativar a auditoria e monitorar a saída quanto a violações na segurança. Nesta seção, você vai ver alguns exemplos simples de auditoria, que é realizada com a instrução `AUDIT`.

### Privilégios necessários para fazer auditoria

Antes que um usuário possa executar instruções `AUDIT`, ele precisa ter certos privilégios:

- Para fazer auditoria de operações de alto nível, o usuário deve ter o privilégio `AUDIT SYSTEM`. Um exemplo de operação de alto nível é a execução de *qualquer* instrução `SELECT`, independentemente da tabela envolvida.
- Para monitorar operações em objetos de banco de dados específicos, o usuário deve ter o privilégio `AUDIT ANY` ou o objeto de banco de dados deve estar em seu esquema. Um exemplo de operação de objeto de banco de dados específica é a execução de uma instrução `SELECT` para uma tabela específica.

O exemplo a seguir se conecta no banco de dados como o usuário `system` e concede os privilégios `AUDIT SYSTEM` e `AUDIT ANY` ao usuário `store`:

```
CONNECT system/manager  
GRANT AUDIT SYSTEM TO store;  
GRANT AUDIT ANY TO store;
```

### Exemplos de auditoria

O exemplo a seguir se conecta no banco de dados como o usuário `store` e faz a auditoria da execução de instruções `CREATE TABLE`:

```
CONNECT store/store_password  
AUDIT CREATE TABLE;
```

Como resultado dessa instrução `AUDIT`, todas as instruções `CREATE TABLE` executadas passarão pela auditoria; por exemplo, a instrução a seguir cria uma tabela de teste simples:

```
CREATE TABLE test (  
  id INTEGER  
);
```

Você pode ver a trilha de auditoria das informações do usuário ao qual está conectado atualmente por meio da visão `USER_AUDIT_TRAIL`. O exemplo a seguir mostra o registro de auditoria gerado pela instrução `CREATE TABLE` anterior:

```
SELECT username, extended_timestamp, audit_option
FROM user_audit_trail
WHERE audit_option='CREATE TABLE';

USERNAME
-----
EXTENDED_TIMESTAMP
-----
AUDIT_OPTION
-----
STORE
20-MAY-07 04.13.43.453000 PM -07:00
CREATE TABLE
```

Você também pode fazer a auditoria da execução de instruções por parte de um usuário específico. O exemplo a seguir faz a auditoria de todas as instruções `SELECT` executadas pelo usuário `store`:

```
AUDIT SELECT TABLE BY store;
```

O exemplo a seguir faz a auditoria de todas as instruções `INSERT`, `UPDATE` e `DELETE` executadas pelos usuários `store` e `steve`:

```
AUDIT INSERT TABLE, UPDATE TABLE, DELETE TABLE BY store, steve;
```

Você também pode fazer a auditoria da execução de instruções para um objeto de banco de dados específico. O exemplo a seguir faz a auditoria de todas as instruções `SELECT` executadas para a tabela `products`:

```
AUDIT SELECT ON store.products;
```

O exemplo a seguir faz a auditoria de todas as instruções executadas para a tabela `employees`:

```
AUDIT ALL ON store.employees;
```

Você também pode usar as opções `WHENEVER SUCCESSFUL` e `WHENEVER NOT SUCCESSFUL` para indicar quando a auditoria deve ser realizada. `WHENEVER SUCCESSFUL` indica que a auditoria será feita quando a instrução tiver executado com sucesso. `WHENEVER NOT SUCCESSFUL` indica que a auditoria será feita quando a instrução não tiver executado com sucesso. O padrão é fazer ambas; isto é, auditoria independentemente do sucesso. Os exemplos a seguir usam a opção `WHENEVER NOT SUCCESSFUL`:

```
AUDIT UPDATE TABLE BY steve WHENEVER NOT SUCCESSFUL;
AUDIT INSERT TABLE WHENEVER NOT SUCCESSFUL;
```

O exemplo a seguir usa a opção `WHENEVER SUCCESSFUL` para fazer a auditoria da criação e da exclusão de um usuário:

```
AUDIT CREATE USER, DROP USER WHENEVER SUCCESSFUL;
```

O exemplo a seguir usa a opção `WHENEVER SUCCESSFUL` para fazer a auditoria da criação e da exclusão de um usuário pelo usuário `store`:

```
AUDIT CREATE USER, DROP USER BY store WHENEVER SUCCESSFUL;
```

Você também pode usar as opções `BY SESSION` e `BY ACCESS`. A opção `BY SESSION` faz com que apenas um registro de auditoria seja gravado quando o mesmo tipo de instrução é executado durante a mesma sessão de banco de dados do usuário; uma sessão de banco de dados começa quando o usuário se conecta nele e termina quando se desconecta. A opção `BY ACCESS` faz um registro de auditoria ser gravado sempre que o mesmo tipo de instrução é executado, independentemente da sessão de usuário. Os exemplos a seguir mostram o uso das opções `BY SESSION` e `BY ACCESS`:

```
AUDIT SELECT ON store.products BY SESSION;  
AUDIT DELETE ON store.employees BY ACCESS;  
AUDIT INSERT, UPDATE ON store.employees BY ACCESS;
```

## Visões de trilha de auditoria

Anteriormente, você viu o uso da visão `USER_AUDIT_TRAIL`. Esta e as demais visões de trilha de auditoria estão descritas na lista a seguir:

- **USER\_AUDIT\_OBJECT** exibe os registros de auditoria para todos os objetos acessíveis para o usuário atual.
- **USER\_AUDIT\_SESSION** exibe os registros de auditoria para conexões e desconexões do usuário atual.
- **USER\_AUDIT\_STATEMENT** exibe os registros de auditoria para instruções `GRANT`, `REVOKE`, `AUDIT`, `NOAUDIT` e `ALTER SYSTEM` executadas pelo usuário atual.
- **USER\_AUDIT\_TRAIL** exibe todas as entradas de trilha de auditoria relacionadas ao usuário atual.

Você pode usar estas visões para examinar o conteúdo da trilha de auditoria. Existem várias visões com nomes semelhantes que o administrador do banco de dados pode usar para examinar a trilha de auditoria; estas visões são chamadas `DBA_AUDIT_OBJECT`, `DBA_AUDIT_SESSION`, `DBA_AUDIT_STATEMENT`, `DBA_AUDIT_TRAIL`, entre outras. Elas permitem que o administrador do banco de dados veja registros de auditoria de todos os usuários. Para obter mais detalhes sobre essas visões, consulte o manual *Oracle Database Reference*, publicado pela Oracle Corporation.

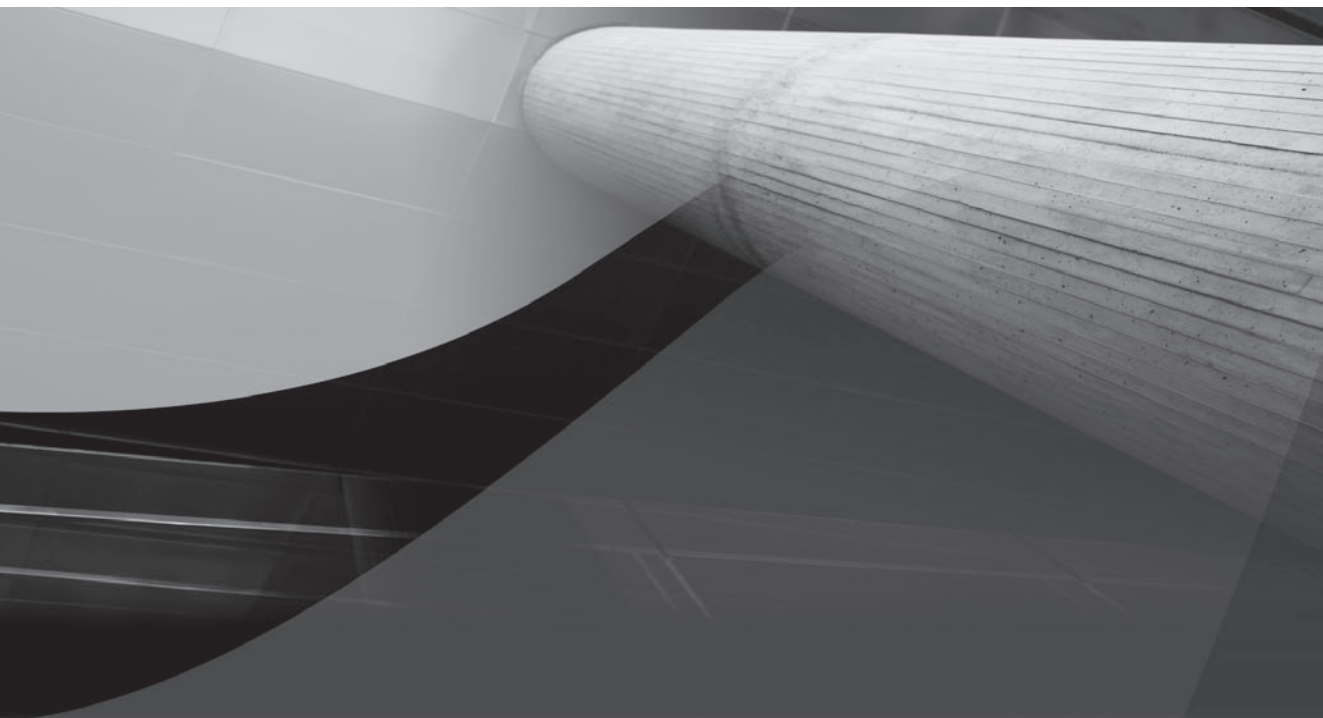
## RESUMO

Neste capítulo, você aprendeu que:

- Um usuário é criado com a instrução `CREATE USER`
- Os privilégios de sistema permitem realizar certas ações dentro do banco de dados, como executar instruções DDL

- Os privilégios de objeto permitem realizar certas ações em objetos de banco de dados, como executar instruções DML em tabelas
- Você pode evitar a digitação do nome do esquema criando um sinônimo para uma tabela
- Uma atribuição é um grupo de privilégios que você pode atribuir a um usuário ou a outra atribuição
- A auditoria da execução de instruções SQL pode ser realizada com a instrução `AUDIT`

O próximo capítulo aborda a criação de tabelas, índices, seqüências e visões.



# CAPÍTULO 10

Criando tabelas,  
seqüências, índices  
e visões

Neste capítulo, você vai aprender:

- Mais sobre tabelas
- Como criar e utilizar seqüências, as quais geram uma série de números
- Como criar e utilizar índices, os quais podem melhorar o desempenho das consultas
- Como utilizar visões, que são consultas predefinidas que permitem ocultar a complexidade dos usuários, dentre outras vantagens
- Sobre Flashback data archives, novidade do Oracle Database 11g, os quais armazenam as alterações feitas em uma tabela durante um período de tempo

Vamos começar examinando as tabelas.

## TABELAS

Nesta seção, você vai aprender mais sobre a criação de uma tabela. Você vai ver como modificar e excluir uma tabela, assim como o modo de recuperar informações sobre uma tabela a partir do dicionário de dados. O dicionário de dados contém informações sobre todos os itens do banco de dados, como tabelas, seqüências, índices etc.

### Criando uma tabela

Para criar uma tabela, use a instrução `CREATE TABLE`. A sintaxe simplificada da instrução `CREATE TABLE` é:

```
CREATE [GLOBAL TEMPORARY] TABLE nome_tabela (
    nome_coluna tipo [CONSTRAINT def_restrição DEFAULT exp_padrão]
    [, nome_coluna tipo [CONSTRAINT def_restrição DEFAULT exp_padrão] ...]
)
[ON COMMIT {DELETE | PRESERVE} ROWS]
TABLESPACE tablespace;
```

onde

- `GLOBAL TEMPORARY` significa que as linhas da tabela são temporárias (essas tabelas são conhecidas como tabelas temporárias). As linhas de uma tabela temporária são específicas para uma sessão de usuário e o tempo durante o qual persistem é definido na cláusula `ON COMMIT`.
- `nome_tabela` é o nome da tabela.
- `nome_coluna` é o nome de uma coluna.
- `tipo` é o tipo de uma coluna.
- `def_restrição` é uma restrição em uma coluna.
- `exp_padrão` é uma expressão para atribuir um valor padrão a uma coluna.



- `ON COMMIT` controla a duração das linhas em uma tabela temporária. `DELETE` significa que as linhas são excluídas no final de uma transação. `PRESERVE` significa que as linhas são mantidas até o fim de uma sessão de usuário, no ponto em que as linhas são excluídas. Se você omitir `ON COMMIT` para uma tabela temporária, o padrão `DELETE` será usado.
- `tablespace` é o `tablespace` da tabela. Se você omitir um `tablespace`, a tabela será armazenada no `tablespace` padrão do usuário.

**NOTA**

*A sintaxe completa de `CREATE TABLE` é bem mais rica do que a mostrada aqui. Para ver os detalhes completos, consulte o livro Oracle Database SQL Reference, publicado pela Oracle Corporation.*

O exemplo a seguir se conecta como o usuário `store` e cria uma tabela chamada `order_status2`:

```
CONNECT store/store_password
CREATE TABLE order_status2 (
  id INTEGER CONSTRAINT order_status2_pk PRIMARY KEY,
  status VARCHAR2(10),
  last_modified DATE DEFAULT SYSDATE
);
```

**NOTA**

*Se quiser acompanhar os exemplos deste capítulo, digite e execute as instruções SQL usando o `SQL*Plus`.*

O exemplo a seguir cria uma tabela temporária chamada `order_status_temp`, cujas linhas serão mantidas até o fim de uma sessão de usuário (`ON COMMIT PRESERVE ROWS`):

```
CREATE GLOBAL TEMPORARY TABLE order_status_temp (
  id INTEGER,
  status VARCHAR2(10),
  last_modified DATE DEFAULT SYSDATE
)
ON COMMIT PRESERVE ROWS;
```

O exemplo a seguir:

- Adiciona uma linha em `order_status_temp`
- Desconecta-se do banco de dados no fim da sessão, o que faz a linha em `order_status_temp` ser excluída
- Volta a se conectar como `store` e consulta `order_status_temp`, a qual mostra que não existem linhas nessa tabela

```
INSERT INTO order_status_temp (
  id, status
) VALUES (
  1, 'New'
);
```

```
1 row created.

DISCONNECT
CONNECT store/store_password
SELECT *
FROM order_status_temp;

no rows selected
```

## Obtendo informações sobre tabelas

Você pode obter informações sobre suas tabelas:

- Executando um comando `DESCRIBE` na tabela. Você já viu exemplos que utilizam o comando `DESCRIBE` em capítulos anteriores.
- Consultando a visão `user_tables`, que faz parte do dicionário de dados.

A Tabela 10-1 descreve algumas das colunas da visão `user_tables`.

### NOTA

*Você pode recuperar informações sobre todas as tabelas a que tem acesso consultando a visão `all_tables`.*

O exemplo a seguir recupera algumas das colunas de `user_tables` onde `table_name` é `order_status2` ou `order_status_temp`:

```
SELECT table_name, tablespace_name, temporary
FROM user_tables
WHERE table_name IN ('ORDER_STATUS2', 'ORDER_STATUS_TEMP');
```

TABLE_NAME	TABLESPACE_NAME	T
ORDER_STATUS2	USERS	N
ORDER_STATUS_TEMP		Y

Observe que a tabela `order_status_temp` é temporária, conforme indicado pelo `Y` na última coluna.

**Tabela 10-1** Algumas colunas da visão `user_tables`

Coluna	Tipo	Descrição
table_name	VARCHAR2 (30)	Nome da tabela.
tablespace_name	VARCHAR2 (30)	Nome do tablespace no qual a tabela está armazenada. Um tablespace é uma área usada pelo banco de dados para armazenar objetos, como tabelas, por exemplo.
temporary	VARCHAR2 (1)	Se a tabela é temporária. Isso é configurado como <code>Y</code> se é temporária ou como <code>N</code> , se não é.

## Obtendo informações sobre colunas nas tabelas

Você pode recuperar informações sobre as colunas de suas tabelas a partir da visão `user_tab_columns`. A Tabela 10-2 descreve algumas das colunas de `user_tab_columns`.



### NOTA

Você pode recuperar informações sobre todas as colunas das tabelas a que tem acesso consultando a visão `all_tab_columns`.

O exemplo a seguir recupera algumas das colunas de `user_tab_columns` para a tabela `products`:

```
COLUMN column_name FORMAT a15
COLUMN data_type FORMAT a10
SELECT column_name, data_type, data_length, data_precision, data_scale
FROM user_tab_columns
WHERE table_name = 'PRODUCTS';
```

COLUMN_NAME	DATA_TYPE	DATA_LENGTH	DATA_PRECISION	DATA_SCALE
PRODUCT_ID	NUMBER	22	38	0
PRODUCT_TYPE_ID	NUMBER	22	38	0
NAME	VARCHAR2	30		
DESCRIPTION	VARCHAR2	50		
PRICE	NUMBER	22	5	2

## Alterando uma tabela

Você altera uma tabela usando a instrução `ALTER TABLE`, que executa tarefas como:

- Adicionar, modificar ou excluir uma coluna
- Adicionar ou excluir uma restrição
- Ativar ou desativar uma restrição

Nas seções a seguir, você vai aprender a usar `ALTER TABLE` para executar cada uma dessas tarefas.

**Tabela 10-2** Algumas colunas da visão `user_tab_columns`

Coluna	Tipo	Descrição
<code>table_name</code>	<code>VARCHAR2(30)</code>	Nome da tabela
<code>column_name</code>	<code>VARCHAR2(30)</code>	Nome da coluna
<code>data_type</code>	<code>VARCHAR2(106)</code>	Tipo de dados da coluna
<code>data_length</code>	<code>NUMBER</code>	Comprimento dos dados
<code>data_precision</code>	<code>NUMBER</code>	Precisão de uma coluna numérica, se foi especificada uma precisão para a coluna
<code>data_scale</code>	<code>NUMBER</code>	Escala de uma coluna numérica

**Adicionando uma coluna**

O exemplo a seguir usa ALTER TABLE para adicionar uma coluna INTEGER chamada modified\_by na tabela order\_status2:

```
ALTER TABLE order_status2
ADD modified_by INTEGER;
```

O exemplo a seguir adiciona uma coluna chamada initially\_created em order\_status2:

```
ALTER TABLE order_status2
ADD initially_created DATE DEFAULT SYSDATE NOT NULL;
```

Você pode verificar a adição da nova coluna executando um comando DESCRIBE em order\_status2:

```
DESCRIBE order_status2
```

Name	Null?	Type
-----	-----	-----
ID	NOT NULL	NUMBER (38)
STATUS		VARCHAR2 (10)
LAST_MODIFIED		DATE
MODIFIED_BY		NUMBER (38)
INITIALLY_CREATED	NOT NULL	DATE

**Adicionando uma coluna virtual**

No Oracle Database 11g, é possível adicionar uma coluna virtual, uma coluna que se refere somente a outras colunas que já estão na tabela. Por exemplo, a instrução ALTER TABLE a seguir adiciona uma coluna virtual chamada average\_salary na tabela salary\_grades:

```
ALTER TABLE salary_grades
ADD (average_salary AS ((low_salary + high_salary)/2));
```

Observe que average\_salary é definida como a média dos valores de low\_salary e high\_salary. O comando DESCRIBE a seguir confirma a adição da coluna average\_salary na tabela salary\_grades:

```
DESCRIBE salary_grades
```

Name	Null?	Type
-----	-----	-----
SALARY_GRADE_ID	NOT NULL	NUMBER (38)
LOW_SALARY		NUMBER (6)
HIGH_SALARY		NUMBER (6)
AVERAGE_SALARY		NUMBER

A consulta a seguir recupera as linhas da tabela salary\_grades:

```
SELECT *
FROM salary_grades;
```

SALARY_GRADE_ID	LOW_SALARY	HIGH_SALARY	AVERAGE_SALARY
-----	-----	-----	-----
1	1	250000	125000.5
2	250001	500000	375000.5

3	500001	750000	625000.5
4	750001	999999	875000

### Modificando uma coluna

A lista a seguir mostra alguns aspectos que você pode modificar em uma coluna usando `ALTER TABLE`:

- Alterar o tamanho de uma coluna (caso o tipo de dados seja um cujo comprimento possa ser alterado, como `CHAR` ou `VARCHAR2`)
- Alterar a precisão de uma coluna numérica
- Alterar o tipo de dados de uma coluna
- Alterar o valor padrão de uma coluna

Você verá exemplos de como alterar esses aspectos da coluna nas seções a seguir.

### Alterando o tamanho de uma coluna

A instrução `ALTER TABLE` a seguir aumenta o comprimento máximo da coluna `order_status2.status` para 15 caracteres:

```
ALTER TABLE order_status2
MODIFY status VARCHAR2(15);
```

#### CUIDADO

*Você só pode diminuir o comprimento de uma coluna se não houver uma linha na tabela ou se todas as linhas contiverem valores nulos para essa coluna.*

### Alterando a precisão de uma coluna numérica

A instrução `ALTER TABLE` a seguir altera a precisão da coluna `order_status2.id` para 5:

```
ALTER TABLE order_status2
MODIFY id NUMBER(5);
```

#### CUIDADO

*Só é possível diminuir a precisão de uma coluna numérica se não houver uma linha na tabela ou se a coluna contiver valores nulos.*

### Alterando o tipo de dados de uma coluna

A instrução `ALTER TABLE` a seguir altera o tipo de dados da coluna `order_status2.status` para `CHAR`:

```
ALTER TABLE order_status2
MODIFY status CHAR(15);
```

Se a tabela estiver vazia ou se a coluna contiver valores nulos, você poderá alterar a coluna para qualquer tipo de dados (incluindo um tipo de dados mais curto); caso contrário, você só poderá alterar o tipo de dados de uma coluna para um que seja compatível. Por exemplo, você pode mudar um valor `VARCHAR2` para `CHAR` (e vice-versa), desde que não torne a coluna mais curta; você não pode mudar um valor `DATE` para `NUMBER`.

**Alterando o valor padrão de uma coluna**

A instrução ALTER TABLE a seguir altera o valor padrão da coluna order\_status2.last\_modified para SYSDATE - 1:

```
ALTER TABLE order_status2
MODIFY last_modified DEFAULT SYSDATE - 1;
```

O valor padrão só se aplica às novas linhas adicionadas na tabela. As linhas novas terão sua coluna last\_modified configurada com a data atual menos um dia.

**Excluindo uma coluna**

A instrução ALTER TABLE a seguir exclui a coluna order\_status2.initially\_created:

```
ALTER TABLE order_status2
DROP COLUMN initially_created;
```

**Adicionando uma restrição**

Em capítulos anteriores, você viu exemplos de tabelas com restrições PRIMARY KEY, FOREIGN KEY e NOT NULL. Essas restrições, junto com os outros tipos de restrições, estão resumidas na Tabela 10-3. Nas seções a seguir, você vai ver como adicionar algumas das restrições mostradas na Tabela 10-3.

**Tabela 10-3** Restrições e seus significados

Restrição	Tipo de restrição	Significado
CHECK	C	O valor de uma coluna ou de um grupo de colunas deve satisfazer determinada condição.
NOT NULL	C	A coluna não pode armazenar um valor nulo. Na verdade, isso é imposto como uma restrição CHECK.
PRIMARY KEY	P	A chave primária de uma tabela. Uma chave primária é constituída de uma ou mais colunas que identificam exclusivamente cada linha em uma tabela.
FOREIGN KEY	R	Uma chave estrangeira de uma tabela. Uma chave estrangeira referencia uma coluna em outra tabela ou uma coluna na mesma tabela (o que é conhecido como auto-referência).
UNIQUE	U	A coluna ou o grupo de colunas só pode armazenar valores exclusivos.
CHECK OPTION	V	As alterações nas linhas da tabela feitas por meio de uma visão devem passar primeiro por uma verificação. (Você vai aprender sobre isso posteriormente, na seção “Visões”.)
READ ONLY	O	A visão é somente leitura. (Você vai aprender sobre isso posteriormente, na seção “Visões”.)

### Adicionando uma restrição CHECK

A instrução `ALTER TABLE` a seguir adiciona uma restrição `CHECK` na tabela `order_status2`:

```
ALTER TABLE order_status2
ADD CONSTRAINT order_status2_status_ck
CHECK (status IN ('PLACED', 'PENDING', 'SHIPPED'));
```

Essa restrição garante que a coluna `status` seja sempre configurada como `PLACED`, `PENDING` ou `SHIPPED`. A instrução `INSERT` a seguir adiciona uma linha na tabela `order_status2` (`status` é configurada como `PENDING`):

```
INSERT INTO order_status2 (
    id, status, last_modified, modified_by
) VALUES (
    1, 'PENDING', '01-JAN-2005', 1
);
```

Se você tentar adicionar uma linha que não satisfaz a restrição `CHECK`, o banco de dados retornará o erro `ORA-02290`. Por exemplo, a instrução `INSERT` a seguir tenta adicionar uma linha cujo valor de `status` não está na lista:

```
INSERT INTO order_status2 (
    id, status, last_modified, modified_by
) VALUES (
    2, 'CLEARED', '01-JAN-2005', 2
);
INSERT INTO order_status2 (
    *
ERROR at line 1:
ORA-02290: check constraint (STORE.ORDER_STATUS2_STATUS_CK) violated
```

Como a restrição `CHECK` é violada, o banco de dados rejeita a nova linha. Você pode usar outros operadores de comparação com a restrição `CHECK`. O exemplo a seguir adiciona uma restrição `CHECK` que impõe que o valor de `id` é maior do que zero:

```
ALTER TABLE order_status2
ADD CONSTRAINT order_status2_id_ck CHECK (id > 0);
```

Ao se adicionar uma restrição, as linhas existentes na tabela devem satisfazer a restrição. Por exemplo, se a tabela `order_status2` contivesse linhas, então a coluna `id` das linhas precisaria ser maior do que zero.

#### NOTA

*Existem exceções à regra que exige que as linhas existentes devem satisfazer a restrição. Você pode desativar uma restrição quando a adiciona inicialmente e pode configurar uma restrição para ser aplicada somente aos novos dados, especificando `ENABLE NOVALIDATE`. Você vai aprender mais sobre isso posteriormente.*

### Adicionando uma restrição NOT NULL

A instrução `ALTER TABLE` a seguir adiciona uma restrição `NOT NULL` na coluna `status` da tabela `order_status2`:

```
ALTER TABLE order_status2
MODIFY status CONSTRAINT order_status2_status_nn NOT NULL;
```

Note que você usa `MODIFY` para adicionar uma restrição `NOT NULL`, em vez de usar `ADD CONSTRAINT`. O exemplo a seguir adiciona uma restrição `NOT NULL` na coluna `modified_by`:

```
ALTER TABLE order_status2
MODIFY modified_by CONSTRAINT order_status2_modified_by_nn NOT NULL;
```

O exemplo a seguir adiciona uma restrição `NOT NULL` na coluna `last_modified`:

```
ALTER TABLE order_status2
MODIFY last_modified NOT NULL;
```

Note que não fornecemos um nome para essa restrição. Neste caso, o banco de dados atribui automaticamente um nome não amigável à restrição, como `SYS_C003381`.

### DICA

*Sempre especifique um nome significativo para suas restrições. Desse modo, quando ocorrer um erro de restrição, você poderá identificar o problema facilmente.*

### Adicionando uma restrição FOREIGN KEY

Antes de você ver um exemplo de adição de uma restrição `FOREIGN KEY`, a instrução `ALTER TABLE` a seguir exclui a coluna `order_status2.modified_by`:

```
ALTER TABLE order_status2
DROP COLUMN modified_by;
```

A próxima instrução adiciona uma restrição `FOREIGN KEY` que referencia a coluna `employees.employee_id`:

```
ALTER TABLE order_status2
ADD CONSTRAINT order_status2_modified_by_fk
modified_by REFERENCES employees(employee_id);
```

Você usa a cláusula `ON DELETE CASCADE` com uma restrição `FOREIGN KEY` para especificar que, quando uma linha é excluída na tabela pai, todas as linhas correspondentes na tabela filho também são excluídas. O exemplo a seguir exclui a coluna `modified_by` e reescreve o exemplo anterior para incluir a cláusula `ON DELETE CASCADE`:

```
ALTER TABLE order_status2
DROP COLUMN modified_by;

ALTER TABLE order_status2
ADD CONSTRAINT order_status2_modified_by_fk
modified_by REFERENCES employees(employee_id) ON DELETE CASCADE;
```

Quando uma linha é excluída da tabela `employees`, todas as linhas correspondentes em `order_status2` também são excluídas. Você usa a cláusula `ON DELETE SET NULL` com uma restrição `FOREIGN KEY` para especificar que, quando uma linha na tabela pai é excluída, a coluna de chave estrangeira da linha (ou linhas) na tabela filho é configurada como nula. O exemplo a seguir exclui a coluna `modified_by` de `order_status2` e reescreve o exemplo anterior para incluir a cláusula `ON DELETE SET NULL`:

```
ALTER TABLE order_status2
DROP COLUMN modified_by;
```



```
ALTER TABLE order_status2
ADD CONSTRAINT order_status2_modified_by_fk
modified_by REFERENCES employees(employee_id) ON DELETE SET NULL;
```

Quando uma linha é excluída da tabela `employees`, a coluna `modified_by` de todas as linhas correspondentes em `order_status2` é configurada como nula. Para fazer a limpeza, antes de passar para a próxima seção, a instrução a seguir exclui a coluna `modified_by`:

```
ALTER TABLE order_status2
DROP COLUMN modified_by;
```

### **Adicionando uma restrição UNIQUE**

A instrução `ALTER TABLE` a seguir adiciona uma restrição `UNIQUE` na coluna `order_status2.status`:

```
ALTER TABLE order_status2
ADD CONSTRAINT order_status2_status_uq UNIQUE (status);
```

Todas as linhas existentes ou novas sempre devem ter um valor exclusivo na coluna `status`.

### **Excluindo uma restrição**

A cláusula `DROP CONSTRAINT` de `ALTER TABLE` exclui uma restrição. O exemplo a seguir exclui a restrição `order_status2_status_uq`:

```
ALTER TABLE order_status2
DROP CONSTRAINT order_status2_status_uq;
```

### **Desativando uma restrição**

Por padrão, uma restrição é ativada quando você a cria. Você pode desativar uma restrição adicionando `DISABLE` no final da cláusula `CONSTRAINT`. O exemplo a seguir adiciona uma restrição em `order_status2`, mas também a desativa:

```
ALTER TABLE order_status2
ADD CONSTRAINT order_status2_status_uq UNIQUE (status) DISABLE;
```

É possível desativar uma restrição existente usando a cláusula `DISABLE CONSTRAINT` de `ALTER TABLE`. O exemplo a seguir desativa a restrição `order_status2_status_nn`:

```
ALTER TABLE order_status2
DISABLE CONSTRAINT order_status2_status_nn;
```

É possível adicionar `CASCADE` após `DISABLE CONSTRAINT` para desativar todas as restrições que dependem da restrição especificada. Você usa `CASCADE` ao desativar uma restrição de chave primária ou exclusiva que faz parte de uma restrição de chave estrangeira de outra tabela.

### **Ativando uma restrição**

Você pode ativar uma restrição existente usando a cláusula `ENABLE CONSTRAINT` de `ALTER TABLE`. O exemplo a seguir ativa a restrição `order_status2_status_uq`:

```
ALTER TABLE order_status2
ENABLE CONSTRAINT order_status2_status_uq;
```

Para ativar uma restrição, todas as linhas da tabela devem satisfazê-la. Por exemplo, se a tabela `order_status2` contivesse linhas, a coluna `status` precisaria conter valores exclusivos. Você pode aplicar uma restrição somente a dados novos, especificando `ENABLE NOVALIDATE`; por exemplo:

```
ALTER TABLE order_status2
ENABLE NOVALIDATE CONSTRAINT order_status2_status_uq;
```

#### NOTA

*O padrão é `ENABLE VALIDATE`, que significa que as linhas existentes devem passar pela verificação da restrição.*

### Restrições adiadas

Uma restrição adiada é aquela imposta quando uma transação sofre commit; você usa a cláusula `DEFERRABLE` quando adiciona a restrição inicialmente. Uma vez que tenha adicionado uma restrição, você não pode alterá-la para `DEFERRABLE`; em vez disso, deve excluir e recriar a restrição.

Quando adiciona uma restrição `DEFERRABLE`, você pode marcá-la como `INITIALLY IMMEDIATE` ou `INITIALLY DEFERRED`. Marcar como `INITIALLY IMMEDIATE` significa que a restrição é verificada quando você adiciona, atualiza ou exclui linhas de uma tabela (isso é igual ao comportamento padrão de uma restrição). `INITIALLY DEFERRED` significa que a restrição é verificada somente quando uma transação é encerrada com commit. Vejamos um exemplo. A instrução a seguir exclui a restrição `order_status2_status_uq`:

```
ALTER TABLE order_status2
DROP CONSTRAINT order_status2_status_uq;
```

O exemplo a seguir adiciona a restrição `order_status2_status_uq`, configurando-a como `DEFERRABLE INITIALLY DEFERRED`:

```
ALTER TABLE order_status2
ADD CONSTRAINT order_status2_status_uq UNIQUE (status)
DEFERRABLE INITIALLY DEFERRED;
```

Se você adicionar linhas em `order_status2`, a restrição `order_status2_status_uq` não será imposta até que commit seja executado.

### Obtendo informações sobre restrições

Você pode recuperar informações sobre suas restrições consultando a visão `user_constraints`. A Tabela 10-4 descreve algumas das colunas de `user_constraints`.

#### NOTA

*Você pode recuperar informações sobre todas as restrições a que tem acesso consultando a visão `all_constraints`.*

O exemplo a seguir recupera algumas das colunas de `user_constraints` da tabela `order_status2`:

```
SELECT constraint_name, constraint_type, status, deferrable, deferred
FROM user_constraints
WHERE table_name = 'ORDER_STATUS2';
```

CONSTRAINT_NAME	C	STATUS	DEFERRABLE	DEFERRED
ORDER_STATUS2_PK	P	ENABLED	NOT DEFERRABLE	IMMEDIATE
ORDER_STATUS2_STATUS_CK	C	ENABLED	NOT DEFERRABLE	IMMEDIATE
ORDER_STATUS2_ID_CK	C	ENABLED	NOT DEFERRABLE	IMMEDIATE
ORDER_STATUS2_STATUS_NN	C	DISABLED	NOT DEFERRABLE	IMMEDIATE
ORDER_STATUS2_STATUS_UQ	U	ENABLED	DEFERRABLE	DEFERRED
SYS_C004807	C	ENABLED	NOT DEFERRABLE	IMMEDIATE

Note que todas as restrições, exceto uma, têm um nome útil. Uma restrição tem o nome gerado pelo banco de dados, SYS\_C004807 (esse nome é gerado automaticamente e será diferente em seu banco de dados). Essa restrição é uma das quais omitimos o nome quando a criamos anteriormente.

DICA

Sempre adicione um nome descritivo para suas restrições.

Obtendo informações sobre as restrições de uma coluna

Você pode recuperar informações sobre as restrições de uma coluna consultando a visão user\_cons\_columns. A Tabela 10-5 descreve algumas das colunas de user\_cons\_columns.

NOTA

Você pode recuperar informações sobre todas as restrições de coluna a que tem acesso consultando a visão all\_cons\_columns.

Tabela 10-4 Algumas colunas da visão user\_constraints

Coluna	Tipo	Descrição
owner	VARCHAR2 (30)	Proprietário da restrição.
constraint_name	VARCHAR2 (30)	Nome da restrição.
constraint_type	VARCHAR2 (1)	Tipo de restrição (P, R, C, U, V ou o). Consulte a Tabela 10-3 para ver os significados dos tipos de restrição.
table_name	VARCHAR2 (30)	Nome da tabela na qual a restrição é definida.
status	VARCHAR2 (8)	Status da restrição (ENABLED ou DISABLED).
deferrable	VARCHAR2 (14)	Se a restrição pode ser adiada (DEFERRABLE ou NOT DEFERRABLE).
deferred	VARCHAR2(9)	Se a restrição é imposta imediatamente ou é adiada (IMMEDIATE ou DEFERRED).

Tabela 10-5 Algumas colunas da visão `user_cons_columns`

Coluna	Tipo	Descrição
owner	VARCHAR2 (30)	Proprietário da restrição
constraint_name	VARCHAR2 (30)	Nome da restrição
table_name	VARCHAR2 (30)	Nome da tabela na qual a restrição é definida
column_name	VARCHAR2 (4000)	Nome da coluna na qual a restrição é definida

O exemplo a seguir recupera os valores de `constraint_name` e `column_name` de `user_cons_columns` para a tabela `order_status2`:

```
COLUMN column_name FORMAT a15
SELECT constraint_name, column_name
FROM user_cons_columns
WHERE table_name = 'ORDER_STATUS2'
ORDER BY constraint_name;

CONSTRAINT_NAME          COLUMN_NAME
-----
ORDER_STATUS2_ID_CK      ID
ORDER_STATUS2_PK        ID
ORDER_STATUS2_STATUS_CK  STATUS
ORDER_STATUS2_STATUS_NN  STATUS
ORDER_STATUS2_STATUS_UQ  STATUS
SYS_C004807              LAST_MODIFIED
```

A próxima consulta junta `user_constraints` e `user_cons_columns` para obter os valores de `column_name`, `constraint_name`, `constraint_type` e `status`:

```
SELECT ucc.column_name, ucc.constraint_name, uc.constraint_type, uc.status
FROM user_constraints uc, user_cons_columns ucc
WHERE uc.table_name = ucc.table_name
AND uc.constraint_name = ucc.constraint_name
AND ucc.table_name = 'ORDER_STATUS2'
ORDER BY ucc.constraint_name;

COLUMN_NAME      CONSTRAINT_NAME          C STATUS
-----
ID               ORDER_STATUS2_ID_CK      C ENABLED
ID               ORDER_STATUS2_PK        P ENABLED
STATUS           ORDER_STATUS2_STATUS_CK  C ENABLED
STATUS           ORDER_STATUS2_STATUS_NN  C DISABLED
STATUS           ORDER_STATUS2_STATUS_UQ  U ENABLED
LAST_MODIFIED    SYS_C004807              C ENABLED
```

## Mudando o nome de uma tabela

É possível mudar o nome de uma tabela com a instrução `RENAME`. O exemplo a seguir muda o nome de `order_status2` para `order_state`:

```
RENAME order_status2 TO order_state;
```

### NOTA

*Se você utilizou o nome da tabela em seus nomes de restrição, deve alterar os nomes de suas restrições.*

O exemplo a seguir muda o nome da tabela de volta ao original:

```
RENAME order_state TO order_status2;
```

## Adicionando um comentário em uma tabela

Um comentário pode ajudá-lo a lembrar para que a tabela ou coluna é usada. Você adiciona um comentário em uma tabela ou coluna com a instrução `COMMENT`. O exemplo a seguir adiciona um comentário na tabela `order_status2`:

```
COMMENT ON TABLE order_status2 IS
'order_status2 armazena o estado de um pedido';
```

O exemplo a seguir adiciona um comentário na coluna `order_status2.last_modified`:

```
COMMENT ON COLUMN order_status2.last_modified IS
'last_modified armazena a data e hora da última modificação do pedido';
```

## Recuperando comentários de tabela

Você pode recuperar os comentários de suas tabelas a partir da visão `user_tab_comments`, como mostrado a seguir:

```
SELECT *
FROM user_tab_comments
WHERE table_name = 'ORDER_STATUS2';
```

TABLE_NAME	Tabela_TYPE	COMMENTS
ORDER_STATUS2	TABLE	order_status2 armazena o estado de um pedido

## Recuperando comentários de coluna

Você pode recuperar os comentários de suas colunas a partir da visão `user_col_comments`, por exemplo:

```
SELECT *
FROM user_col_comments
WHERE table_name = 'ORDER_STATUS2';
```

TABLE_NAME	COLUMN_NAME	COMMENTS
ORDER_STATUS2	ID	

ORDER_STATUS2	STATUS
ORDER_STATUS2	LAST_MODIFIED

last\_modified armazena a data e hora da última modificação do pedido

## Truncando uma tabela

É possível truncar uma tabela usando a instrução `TRUNCATE`. Isso remove *todas* as linhas de uma tabela e zera a área de armazenamento de uma tabela. O exemplo a seguir trunca `order_status2`:

```
TRUNCATE TABLE order_status2;
```

### DICA

*Se você precisa remover todas as linhas de uma tabela, deve usar `TRUNCATE` em vez de `DELETE`. Isso porque `TRUNCATE` zera a área de armazenamento de uma tabela e a deixa pronta para receber novas linhas. Uma instrução `TRUNCATE` não exige um espaço de undo no banco de dados e você não precisa executar uma instrução `COMMIT` para tornar a exclusão permanente. O espaço de undo é uma área que o software de banco de dados utiliza para registrar as alterações feitas nele.*

## Excluindo uma tabela

Você exclui uma tabela com a instrução `DROP TABLE`. O exemplo a seguir exclui a tabela `order_status2`:

```
DROP TABLE order_status2;
```

Isso conclui a discussão sobre tabelas. Na próxima seção, você vai aprender sobre seqüências.

## SEQÜÊNCIAS

Uma *seqüência* é um item de banco de dados que gera uma série de números inteiros. Normalmente, você usa os números inteiros gerados por uma seqüência para preencher uma coluna de chave primária numérica. Nesta seção, você vai aprender a:

- Criar uma seqüência
- Recuperar informações de uma seqüência a partir do dicionário de dados
- Usar uma seqüência
- Modificar uma seqüência
- Excluir uma seqüência

## Criando uma seqüência

Você cria uma seqüência com a instrução `CREATE SEQUENCE`, que tem a seguinte sintaxe:

```
CREATE SEQUENCE nome_seqüência
[START WITH num_início]
[INCREMENT BY num_incremento]
[ { MAXVALUE num_máximo | NOMAXVALUE } ]
```

```
[ { MINVALUE num_mínimo | NOMINVALUE } ]
[ { CYCLE | NOCYCLE } ]
[ { CACHE num_cache | NOCACHE } ]
[ { ORDER | NOORDER } ];
```

onde

- *nome\_sequência* é o nome da sequência.
- *num\_início* é o número inteiro para iniciar a sequência. O número inicial padrão é 1.
- *num\_incremento* é o número inteiro para incrementar a sequência. O número de incremento padrão é 1. O valor absoluto de *num\_incremento* deve ser menor do que a diferença entre *num\_máximo* e *num\_mínimo*.
- *num\_máximo* é o número inteiro máximo da sequência; *num\_máximo* deve ser maior ou igual a *num\_início* e *num\_máximo* deve ser maior do que *num\_mínimo*.
- NOMAXVALUE especifica que o máximo é  $10^{27}$  para uma sequência crescente ou  $-1$  para uma sequência decrescente. NOMAXVALUE é o padrão.
- *num\_mínimo* é o número inteiro mínimo da sequência; *num\_mínimo* deve ser menor ou igual a *num\_início* e *num\_mínimo* deve ser menor do que *num\_máximo*.
- NOMINVALUE especifica o que o mínimo é 1 para uma sequência crescente ou  $-10^{26}$  para uma sequência decrescente. NOMINVALUE é o padrão.
- CYCLE significa que a sequência gera números inteiros mesmo depois de atingir seu valor máximo ou mínimo. Quando uma sequência crescente atinge seu valor máximo, o próximo valor gerado é o mínimo. Quando uma sequência decrescente atinge seu valor mínimo, o próximo valor gerado é o máximo.
- NOCYCLE significa que a sequência não pode gerar mais número inteiro algum após atingir seu valor máximo ou mínimo. NOCYCLE é o padrão.
- *num\_cache* é o número de valores inteiros a manter na memória. O número de valores inteiro padrão para colocar no cache é 20. O número mínimo de valores inteiros que podem ser colocados no cache é 2. O número máximo de valores inteiros que podem ser colocados no cache é determinado pela fórmula  $\text{CEIL}(\text{num\_máximo} - \text{num\_mínimo}) / \text{ABS}(\text{num\_incremento})$ .
- NOCACHE significa sem cache. Isso impede o banco de dados de alocar valores previamente para a sequência, o que evita lacunas numéricas na sequência, mas reduz o desempenho. As lacunas ocorrem porque os valores colocados no cache são perdidos quando o banco de dados é fechado. Se você omitir CACHE e NOCACHE, o banco de dados colocará 20 números de sequência no cache, por padrão.
- ORDER garante que os números inteiros sejam gerados na ordem da solicitação. Normalmente, você usa ORDER ao utilizar Real Application Clusters, que são configurados e gerenciados por administradores de banco de dados. Os Real Application Clusters são vários servidores de banco de dados que compartilham a mesma memória. Os Real Application Clusters podem melhorar o desempenho.
- NOORDER não garante que os números inteiros sejam gerados na ordem da solicitação. NOORDER é o padrão.

O exemplo a seguir se conecta como o usuário `store` e cria uma seqüência chamada `s_test` (colocaremos `s_` no início de seqüências):

```
CONNECT store/store_password
CREATE SEQUENCE s_test;
```

Como essa instrução `CREATE SEQUENCE` omite os parâmetros opcionais, os valores padrão são usados. Isso significa que `num_início` e `num_incremento` são configurados com o valor padrão, que é 1. O exemplo a seguir cria uma seqüência chamada `s_test2` e especifica valores para os parâmetros opcionais:

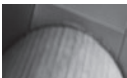
```
CREATE SEQUENCE s_test2
START WITH 10 INCREMENT BY 5
MINVALUE 10 MAXVALUE 20
CYCLE CACHE 2 ORDER;
```

O último exemplo cria uma seqüência chamada `s_test3` que começa em 10 e faz contagem regressiva até 1:

```
CREATE SEQUENCE s_test3
START WITH 10 INCREMENT BY -1
MINVALUE 1 MAXVALUE 10
CYCLE CACHE 5;
```

Recuperando informações sobre seqüências

Você pode recuperar informações sobre suas seqüências a partir da visão `user_sequences`. A Tabela 10-6 descreve as colunas de `user_sequences`.



**NOTA**  
*Você pode recuperar informações sobre todas as seqüências a que tem acesso consultando a visão `all_sequences`.*

Tabela 10-6 Algumas colunas da visão `user_sequences`

Coluna	Tipo	Descrição
sequence_name	VARCHAR2 (30)	Nome da seqüência
min_value	NUMBER	Valor mínimo
max_value	NUMBER	Valor máximo
increment_by	NUMBER	Número para incrementar ou decrementar a seqüência
cycle_flag	VARCHAR2 (1)	Se a seqüência é cíclica (Y ou N)
order_flag	VARCHAR2 (1)	Se a seqüência é ordenada (Y ou N)
cache_size	NUMBER	Número de valores da seqüência armazenados na memória
last_number	NUMBER	Último número gerado ou colocado em cache pela seqüência



O exemplo a seguir recupera os detalhes das seqüências de `user_sequences`:

```
COLUMN nome_sequence FORMAT a13
SELECT * FROM user_sequences
ORDER BY sequence_name;
```

SEQUENCE_NAME	MIN_VALUE	MAX_VALUE	INCREMENT_BY	C	O	CACHE_SIZE	LAST_NUMBER
S_TEST	1	1.0000E+27	1	N	N	20	1
S_TEST2	10	20	5	Y	Y	2	10
S_TEST3	1	10	-1	Y	N	5	10

Usando uma seqüência

Uma seqüência gera uma série de números. Uma seqüência contém duas pseudocolunas, chamadas `currval` e `nextval`, que você utiliza para obter o valor atual e o próximo valor da seqüência. Antes de recuperar o valor atual, você deve primeiro inicializar a seqüência recuperando o próximo valor. Quando você seleciona `s_test.nextval`, a seqüência é inicializada com 1. Por exemplo, a consulta a seguir recupera `s_test.nextval`; observe que a tabela `dual` é utilizada na cláusula `FROM`:

```
SELECT s_test.nextval
FROM dual;
```

NEXTVAL
1

O primeiro valor na seqüência `s_test` é 1. Uma vez inicializada a seqüência, você pode obter seu valor atual recuperando `currval`. Por exemplo:

```
SELECT s_test.currval
FROM dual;
```

CURRVAL
1

Quando você recupera `currval`, `nextval` permanece inalterada; `nextval` só muda quando você a recupera para obter o próximo valor. O exemplo a seguir recupera `s_test.nextval` e `s_test.currval`; observe que ambos os valores são 2:

```
SELECT s_test.nextval, s_test.currval
FROM dual;
```

NEXTVAL	CURRVAL
2	2

Recuperar `s_test.nextval` obtém o próximo valor da seqüência, que é 2; `s_test.currval` também é 2.

O exemplo a seguir inicializa `s_test2` recuperando `s_test2.nextval`; observe que o primeiro valor da sequência é 10:

```
SELECT s_test2.nextval
FROM dual;

NEXTVAL
-----
      10
```

O valor máximo de `s_test2` é 20 e a sequência foi criada com a opção `CYCLE`, significando que ela voltará a 10 quando atingir o máximo de 20:

```
SELECT s_test2.nextval
FROM dual;

NEXTVAL
-----
      15

SELECT s_test2.nextval
FROM dual;

NEXTVAL
-----
      20

SELECT s_test2.nextval
FROM dual;

NEXTVAL
-----
      10
```

A sequência `s_test3` começa em 10 e faz contagem regressiva até 1:

```
SELECT s_test3.nextval
FROM dual;

NEXTVAL
-----
      10

SELECT s_test3.nextval
FROM dual;

NEXTVAL
-----
       9

SELECT s_test3.nextval
FROM dual;
```

NEXTVAL

-----

8

## Preenchendo uma chave primária usando uma seqüência

As seqüências são úteis para preencher valores numéricos de coluna de chave primária. Vejamos um exemplo. A instrução a seguir recria a tabela `order_status2`:

```
CREATE TABLE order_status2 (
    id INTEGER CONSTRAINT order_status2_pk PRIMARY KEY,
    status VARCHAR2(10),
    last_modified DATE DEFAULT SYSDATE
);
```

Em seguida, a próxima instrução cria uma seqüência chamada `s_order_status2` (em breve essa seqüência será usada para preencher a coluna `order_status2.id`):

```
CREATE SEQUENCE s_order_status2 NOCACHE;
```

### DICA

*Ao usar uma seqüência para preencher uma coluna de chave primária, normalmente você deve usar `NOCACHE` para evitar lacunas na seqüência de números (as lacunas ocorrem porque os valores colocados na cache são perdidos quando o banco de dados é fechado). Entretanto, usar `NOCACHE` reduz o desempenho. Se você tiver certeza absoluta de que pode conviver com lacunas nos valores de chave primária, considere o uso de `CACHE`.*

As instruções `INSERT` a seguir adicionam linhas em `order_status2`; observe que o valor da coluna `id` é configurado usando `s_order_status2.nextval` (retorna 1 para a primeira instrução `INSERT` e 2 para a segunda):

```
INSERT INTO order_status2 (
    id, status, last_modified
) VALUES (
    s_order_status2.nextval, 'PLACED', '01-JAN-2006'
);
```

```
INSERT INTO order_status2 (
    id, status, last_modified
) VALUES (
    s_order_status2.nextval, 'PENDING', '01-FEB-2006'
);
```

A consulta a seguir recupera as linhas de `order_status2`; observe que a coluna `id` é configurada com os dois primeiros valores (1 e 2) da seqüência `s_order_status2`:

```
SELECT *
FROM order_status2;
```

ID	STATUS	LAST_MODI
1	PLACED	01-JAN-06
2	PENDING	01-FEB-06

## Modificando uma seqüência

É possível modificar uma seqüência usando a instrução `ALTER SEQUENCE`. Existem algumas limitações sobre o que você pode modificar em uma seqüência:

- Você não pode alterar o valor inicial de uma seqüência
- O valor mínimo não pode ser maior do que o valor atual da seqüência
- O valor máximo não pode ser menor do que o valor atual da seqüência

O exemplo a seguir modifica `s_test` para incrementar a seqüência de números por 2:

```
ALTER SEQUENCE s_test
INCREMENT BY 2;
```

Quando isso é feito, os novos valores gerados por `s_test` são incrementados por 2. Por exemplo, se `s_test.currval` é 2, `s_test.nextval` é 4. Isso é mostrado no exemplo a seguir:

```
SELECT s_test.currval
FROM dual;
```

```
      CURRVAL
-----
          2
```

```
SELECT s_test.nextval
FROM dual;
```

```
      NEXTVAL
-----
          4
```

## Excluindo uma seqüência

Você exclui uma seqüência usando `DROP SEQUENCE`. O exemplo a seguir exclui `s_test3`:

```
DROP SEQUENCE s_test3;
```

Isso conclui a discussão sobre seqüências. Na próxima seção, você vai aprender sobre índices.

## ÍNDICES

Ao procurar um assunto específico em um livro, você pode percorrer o livro inteiro ou usar o índice para encontrar o local. Um índice de uma tabela de banco de dados é semelhante a um índice de livro, exceto que os índices de banco de dados são usados para localizar linhas específicas em uma tabela. O inconveniente dos índices é que, quando uma linha é adicionada na tabela, é preciso gastar um tempo para atualizar o índice da nova linha. Geralmente, você deve criar um índice em uma coluna quando está recuperando um pequeno número de linhas de uma tabela que contém muitas linhas. Uma boa regra geral é:

*Crie um índice quando uma consulta recuperar <= 10% do total de linhas em uma tabela.*

Isso significa que a coluna do índice deve conter uma ampla gama de valores. Esses tipos de índices são chamados de índices de “árvore B”, um nome que vem de uma estrutura de dados em árvore utilizada na ciência da computação. Uma boa candidata para indexação de árvore B seria uma coluna contendo um valor exclusivo para cada linha (por exemplo, um número de CPF). Uma candidata ruim para uma indexação de árvore B seria uma coluna contendo apenas um pequeno intervalo de valores (por exemplo, N, S, E, W ou 1, 2, 3, 4, 5, 6). Um banco de dados Oracle cria automaticamente um índice de árvore B para a chave primária de uma tabela e para colunas incluídas em uma restrição exclusiva. Para colunas que contêm um pequeno intervalo de valores, você pode usar um índice de “bitmap”. Nesta seção, você vai aprender a:

- Criar um índice de árvore B
- Criar um índice baseado em função
- Recuperar informações sobre um índice a partir do dicionário de dados
- Modificar um índice
- Excluir um índice
- Criar um índice de bitmap

## Criando um índice de árvore B

É possível criar um índice de árvore B usando `CREATE INDEX`, que tem a seguinte sintaxe simplificada:

```
CREATE [UNIQUE] INDEX nome_índice ON  
nome_tabela(nome_coluna[, nome_coluna...])  
TABLESPACE tablespace;
```

onde

- `UNIQUE` significa que os valores nas colunas indexadas devem ser exclusivos.
- `nome_índice` é o nome do índice.
- `nome_tabela` é uma tabela do banco de dados.
- `nome_coluna` é a coluna indexada. Você pode criar um índice em várias colunas (tal índice é conhecido como *índice composto*).
- `tablespace` é o tablespace do índice. Se você não fornecer um tablespace, o índice será armazenado no tablespace padrão do usuário.

### DICA

*Por motivos de desempenho, normalmente você deve armazenar índices em um tablespace diferente do das tabelas. Por simplicidade, os exemplos deste capítulo utilizam o tablespace padrão. Em um banco de dados de produção, o administrador do banco de dados deve criar tablespaces separados para tabelas e índices.*

Agora, vamos examinar os processos para criar um índice de árvore B para a coluna `customers.last_name`. Suponha que a tabela `customers` contenha uma grande quantidade de linhas e que você recupere linhas regularmente usando o seguinte tipo de consulta:

```
SELECT customer_id, first_name, last_name
FROM customers
WHERE last_name = 'Brown';
```

Suponha também que a coluna `last_name` contém valores exclusivos, de modo que qualquer consulta utilizando essa coluna em uma cláusula `WHERE` retorne menos de 10% do número total de linhas na tabela. Isso significa, portanto, que a coluna `last_name` é uma boa candidata para indexação. A instrução `CREATE INDEX` a seguir cria um índice chamado `i_customers_last_name` na coluna `last_name` da tabela `customers` (colocaremos `i_` no início de nomes de índice):

```
CREATE INDEX i_customers_last_name ON customers(last_name);
```

Uma vez criado o índice, a consulta anterior demorará menos tempo para terminar. Você pode impor a exclusividade de valores de coluna usando um índice exclusivo. Por exemplo, a instrução a seguir cria um índice exclusivo chamado `i_customers_phone` na coluna `customers.phone`:

```
CREATE UNIQUE INDEX i_customers_phone ON customers(phone);
```

Você também pode criar um índice composto em várias colunas. Por exemplo, a instrução a seguir cria um índice composto chamado `i_employees_first_last_name` nas colunas `first_name` e `last_name` da tabela `employees`:

```
CREATE INDEX i_employees_first_last_name ON
employees(first_name, last_name);
```

## Criando um índice baseado em função

Na seção anterior, você viu o índice `i_customers_last_name`. Digamos que você execute a consulta a seguir:

```
SELECT first_name, last_name
FROM customers
WHERE UPPER(last_name) = 'BROWN';
```

Como essa consulta usa uma função — `UPPER()`, neste caso —, o índice `i_customers_last_name` não é utilizado. Se quiser que um índice seja baseado nos resultados de uma função, você deve criar um índice baseado em função, como:

```
CREATE INDEX i_func_customers_last_name
ON customers(UPPER(last_name));
```

Além disso, o administrador do banco de dados deve configurar o parâmetro de inicialização `QUERY_REWRITE_ENABLED` como `true` (o padrão é `false`) para tirar proveito de índices baseados em função. O exemplo a seguir configura `QUERY_REWRITE_ENABLED` como `true`:

```
CONNECT system/manager
ALTER SYSTEM SET QUERY_REWRITE_ENABLED=TRUE;
```

Recuperando informações sobre índices

Você pode recuperar informações sobre seus índices a partir da visão `user_indexes`. A Tabela 10-7 descreve algumas das colunas de `user_indexes`.

**NOTA**  
Você pode recuperar informações sobre todos os índices a que tem acesso consultando a visão `all_indexes`.

O exemplo a seguir se conecta como o usuário `store` e recupera algumas das colunas de `user_indexes` das tabelas `customers` e `employees`; observe que a lista de índices inclui `customers_pk`, que é um índice exclusivo criado automaticamente pelo banco de dados para a coluna de chave primária `customer_id` da tabela `customers`:

```
CONNECT store/store_password
SELECT index_name, table_name, uniqueness, status
FROM user_indexes
WHERE table_name IN ('CUSTOMERS', 'EMPLOYEES')
ORDER BY index_name;
```

INDEX_NAME	TABLE_NAME	UNIQUENES	STATUS
CUSTOMERS_PK	CUSTOMERS	UNIQUE	VALID
EMPLOYEES_PK	EMPLOYEES	UNIQUE	VALID
I_CUSTOMERS_LAST_NAME	CUSTOMERS	NONUNIQUE	VALID
I_CUSTOMERS_PHONE	CUSTOMERS	UNIQUE	VALID
I_EMPLOYEES_FIRST_LAST_NAME	EMPLOYEES	NONUNIQUE	VALID
I_FUNC_CUSTOMERS_LAST_NAME	CUSTOMERS	NONUNIQUE	VALID

Recuperando informações sobre índices em uma coluna

Você pode recuperar informações sobre os índices em uma coluna consultando a visão `user_ind_columns`. A Tabela 10-8 descreve algumas das colunas de `user_ind_columns`.

Tabela 10-7 Algumas colunas da visão `user_indexes`

Coluna	Tipo	Descrição
<code>index_name</code>	<code>VARCHAR2 (30)</code>	Nome do índice
<code>table_owner</code>	<code>VARCHAR2 (30)</code>	O usuário que possui a tabela
<code>table_name</code>	<code>VARCHAR2 (30)</code>	O nome da tabela em que o índice foi criado
<code>uniqueness</code>	<code>VARCHAR2 (9)</code>	Indica se o índice é exclusivo ( <code>UNIQUE</code> ou <code>NONUNIQUE</code> )
<code>status</code>	<code>VARCHAR2 (8)</code>	Indica se o índice é válido ( <code>VALID</code> ou <code>INVALID</code> )

**Tabela 10-8** Algumas colunas da visão `user_ind_columns`

Coluna	Tipo	Descrição
<code>index_name</code>	<code>VARCHAR2(30)</code>	Nome do índice
<code>table_name</code>	<code>VARCHAR2(30)</code>	Nome da tabela
<code>column_name</code>	<code>VARCHAR2(4000)</code>	Nome da coluna indexada



**NOTA**

Você pode recuperar informações sobre todos os índices a que tem acesso consultando a visão `all_ind_columns`.

A consulta a seguir recupera algumas das colunas de `user_ind_columns` das tabelas `customers` e `employees`:

```
COLUMN table_name FORMAT a15
COLUMN column_name FORMAT a15
SELECT index_name, table_name, column_name
FROM user_ind_columns
WHERE table_name IN ('CUSTOMERS', 'EMPLOYEES')
ORDER BY index_name;
```

INDEX_NAME	TABLE_NAME	COLUMN_NAME
-----	-----	-----
CUSTOMERS_PK	CUSTOMERS	CUSTOMER_ID
EMPLOYEES_PK	EMPLOYEES	EMPLOYEE_ID
I_CUSTOMERS_LAST_NAME	CUSTOMERS	LAST_NAME
I_CUSTOMERS_PHONE	CUSTOMERS	PHONE
I_EMPLOYEES_FIRST_LAST_NAME	EMPLOYEES	LAST_NAME
I_EMPLOYEES_FIRST_LAST_NAME	EMPLOYEES	FIRST_NAME
I_FUNC_CUSTOMERS_LAST_NAME	CUSTOMERS	SYS_NC00006\$

**Modificando um índice**

Você modifica um índice usando `ALTER INDEX`. O exemplo a seguir muda o nome do índice `i_customers_phone` para `i_customers_phone_number`:

```
ALTER INDEX i_customers_phone RENAME TO i_customers_phone_number;
```

**Excluindo um índice**

Você exclui um índice usando a instrução `DROP INDEX`. O exemplo a seguir exclui o índice `i_customers_phone_number`:

```
DROP INDEX i_customers_phone_number;
```

**Criando um índice de bitmap**

Os índices de bitmap são normalmente usados em ambientes de *data warehouse*, que são bancos de dados que contêm volumes de dados muito grandes. Normalmente, os dados de um *data warehouse* são lidos por muitas consultas, mas não são modificados por muitas transações concorrentes.



tes. Os data warehouses são normalmente usados pelas organizações para análise de inteligência de negócios, como o monitoramento de tendências de vendas.

Uma coluna referenciada em muitas consultas, mas que contém apenas uma pequena gama de valores, é uma candidata para um índice de bitmap, por exemplo:

- N, S, E, W
- 1, 2, 3, 4, 5, 6
- “Pedido feito”, “Pedido despachado”

Basicamente, um índice contém um ponteiro para uma linha em uma tabela que encerra determinado valor de chave de índice; o valor de chave é usado para obter o rowid da linha na tabela. (Conforme discutido no Capítulo 2, um rowid é usado internamente pelo banco de dados para armazenar a localização física da linha.) Em um índice de árvore B, uma lista de rowids é armazenada para cada chave correspondente às linhas com esse valor de chave. Em um índice de árvore B, o banco de dados armazena uma lista de valores de chave com cada rowid, o que permite ao banco de dados localizar uma linha em uma tabela.

Entretanto, em um índice de bitmap, é usado um bitmap para cada valor de chave; este permite que o banco de dados localize uma linha. Cada bit do bitmap corresponde a um rowid possível. Se o bit está ativo, isso significa que a linha com o rowid correspondente contém o valor de chave. Uma função de mapeamento converte a posição do bit em um rowid real.

Os índices de bitmap são normalmente usados em tabelas que contêm grandes volumes de dados e cujo conteúdo não é modificado com muita frequência. Além disso, um índice de bitmap só deve ser criado em colunas que contêm um pequeno número de valores distintos. Se o número de valores distintos de uma coluna é menor do que 1% do número de linhas na tabela ou se os valores de uma coluna são repetidos mais de 100 vezes, a coluna é uma candidata a ter um índice de bitmap. Por exemplo, se você tivesse uma tabela com 1 milhão de linhas, uma coluna com 10.000 valores distintos ou menos seria uma boa candidata para ter um índice de bitmap; além disso, as atualizações nas linhas da tabela devem ser raras e a coluna precisaria ser utilizada frequentemente na cláusula `WHERE` das consultas.

A instrução a seguir cria um índice de bitmap na coluna `status` da tabela `order_status`:

```
CREATE BITMAP INDEX i_order_status ON order_status(status);
```

#### NOTA

*Evidentemente, esse exemplo não corresponde ao mundo real, pois a tabela `order_status` não contém linhas suficientes.*

Para mais informações sobre índices de bitmap, consulte os livros *Oracle Database Performance Tuning Guide* e *Oracle Database Concepts*, ambos publicados pela Oracle Corporation. Esses livros também contêm informações sobre outros tipos exóticos de índices que você pode usar. Isso conclui a discussão sobre índices. Na próxima seção, você vai aprender sobre visões.

## VISÕES

Uma visão é uma consulta predefinida em uma ou mais tabelas (conhecidas como *tabelas de base*). A recuperação de informações de uma visão é feita da mesma maneira que a recuperação de uma tabela: basta incluir a visão na cláusula `FROM` de uma consulta. Em algumas visões, você também pode executar operações DML nas tabelas de base.

**NOTA**

*As visões não armazenam linhas. As linhas são sempre armazenadas em tabelas.*

Você já viu alguns exemplos de recuperação de informações de visões quando selecionou linhas do dicionário de dados, que é acessado por meio de visões — por exemplo, `user_tables`, `user_sequences` e `user_indexes` são visões.

As visões oferecem diversas vantagens, como:

- Você pode colocar uma consulta complexa em uma visão e conceder aos usuários acesso à visão. Isso permite ocultar a complexidade dos usuários.
- Você pode impedir os usuários de consultar diretamente as tabelas de base, concedendo a eles acesso apenas à visão.
- Você pode permitir que uma visão acesse apenas certas linhas nas tabelas de base. Isso permite ocultar linhas de um usuário final.

Nesta seção, você vai aprender a:

- Criar e usar uma visão
- Obter detalhes de uma visão, a partir do dicionário de dados
- Modificar uma visão
- Excluir uma visão

## Criando e usando uma visão

Você cria uma visão usando `CREATE VIEW`, que tem a seguinte sintaxe simplificada:

```
CREATE [OR REPLACE] VIEW [{FORCE | NOFORCE}] VIEW nome_visao
[(nome_apelido[, nome_apelido...])] AS subconsulta
[WITH {CHECK OPTION | READ ONLY} CONSTRAINT nome_restricao];
```

onde

- `OR REPLACE` significa que a visão substitui outra já existente.
- `FORCE` significa que a visão deve ser criada mesmo que as tabelas de base não existam.
- `NOFORCE` significa que a visão não é criada se as tabelas de base não existem. `NOFORCE` é o padrão.
- `nome_visao` é o nome da visão.
- `nome_apelido` é o nome de um apelido de uma expressão na subconsulta. Deve haver o mesmo número de apelidos do que de expressões na subconsulta.
- `subconsulta` é a subconsulta que recupera das tabelas de base. Se você tiver fornecido apelidos, pode usá-los na lista após a instrução `SELECT`.
- `WITH CHECK OPTION` significa que somente as linhas que seriam recuperadas pela subconsulta podem ser inseridas, atualizadas ou excluídas. Por padrão, as linhas não são verificadas.

- *nome\_restrição* é o nome da restrição `WITH CHECK OPTION` ou `WITH READ ONLY`.
- `WITH READ ONLY` significa que as linhas só podem ler nas tabelas de base.

Existem dois tipos básicos de visão:

- Visões simples, que contêm uma subconsulta que recupera de uma única tabela de base
- Visões complexas, que contêm uma subconsulta que:
  - Recupera de várias tabelas de base
  - Agrupa linhas usando uma cláusula `GROUP BY` ou `DISTINCT`
  - Contém uma chamada de função

Você vai aprender a criar e utilizar esses tipos de visões nas seções a seguir.

### Privilégio para visões

Para criar uma visão, o usuário precisa ter o privilégio `CREATE VIEW`. O exemplo a seguir se conecta como o usuário `system` e concede o privilégio `CREATE VIEW` ao usuário `store`:

```
CONNECT system/manager
GRANT CREATE VIEW TO store;
```

### Criando e usando visões simples

As visões simples acessam uma única tabela de base. O exemplo a seguir se conecta como o usuário `store` e cria uma visão chamada `cheap_products_view`, cuja subconsulta recupera produtos somente onde o preço é menor do que US\$15:

```
CONNECT store/store_password
CREATE VIEW cheap_products_view AS
SELECT *
FROM products
WHERE price < 15;
```

O exemplo a seguir cria uma visão chamada `employees_view` cuja subconsulta recupera todas as colunas da tabela `employees`, exceto `salary`:

```
CREATE VIEW employees_view AS
SELECT employee_id, manager_id, first_name, last_name, title
FROM employees;
```

### Executando uma consulta em uma visão

Uma vez que tenha criado uma visão, você pode usá-la para acessar a tabela de base. A consulta a seguir recupera linhas de `cheap_products_view`:

```
SELECT product_id, name, price
FROM cheap_products_view;
```

PRODUCT_ID	NAME	PRICE
4	Tank War	13.95
6	2412: The Return	14.95

7	Space Force 9	13.49
8	From Another Planet	12.99
9	Classical Music	10.99
11	Creative Yell	14.99
12	My Front Line	13.49

O exemplo a seguir recupera linhas de `employees_view`:

```
SELECT *
FROM employees_view;
```

EMPLOYEE_ID	MANAGER_ID	FIRST_NAME	LAST_NAME	TITLE
-----	-----	-----	-----	-----
1		James	Smith	CEO
2	1	Ron	Johnson	Sales Manager
3	2	Fred	Hobbs	Salesperson
4	2	Susan	Jones	Salesperson

### ***Executando uma instrução INSERT usando uma visão***

Você pode executar instruções DML usando `cheap_products_view`. O exemplo a seguir executa uma instrução `INSERT` utilizando `cheap_products_view` e depois recupera a linha:

```
INSERT INTO cheap_products_view (
    product_id, product_type_id, name, price
) VALUES (
    13, 1, 'Western Front', 13.50
);
```

1 row created.

```
SELECT product_id, name, price
FROM cheap_products_view
WHERE product_id = 13;
```

PRODUCT_ID	NAME	PRICE
-----	-----	-----
13	Western Front	13.5

### **NOTA**

*Você só pode executar instruções DML com visões simples. Visões complexas não suportam DML.*

Como `cheap_products_view` não usou `WITH CHECK OPTION`, você pode inserir, atualizar e excluir linhas que não podem ser recuperadas pela visão. O exemplo a seguir insere uma linha cujo preço é US\$16,50 (isso é maior do que US\$15 e, portanto, não pode ser recuperada pela visão):

```
INSERT INTO cheap_products_view (
    product_id, product_type_id, name, price
) VALUES (
    14, 1, 'Eastern Front', 16.50
);
```

1 row created.

```
SELECT *
FROM cheap_products_view
WHERE product_id = 14;
```

no rows selected

A visão `employees_view` contém uma subconsulta que seleciona cada coluna de `employees`, exceto `salary`. Quando você executar uma instrução `INSERT` usando `employees_view`, a coluna `salary` da tabela de base `employees` será configurada como nula; por exemplo:

```
INSERT INTO employees_view (
    employee_id, manager_id, first_name, last_name, title
) VALUES (
    5, 1, 'Jeff', 'Jones', 'CTO'
);
```

1 row created.

```
SELECT employee_id, first_name, last_name, salary
FROM employees
WHERE employee_id = 5;
```

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	SALARY
5	Jeff	Jones	

A coluna `salary` é nula.

### ***Criando uma visão com uma restrição CHECK OPTION***

Você pode especificar que as instruções DML em uma visão devem satisfazer a subconsulta, usando uma restrição `CHECK OPTION`. Por exemplo, a instrução a seguir cria uma visão chamada `cheap_products_view2` que tem uma restrição `CHECK OPTION`:

```
CREATE VIEW cheap_products_view2 AS
SELECT *
FROM products
WHERE price < 15
WITH CHECK OPTION CONSTRAINT cheap_products_view2_price;
```

O exemplo a seguir tenta inserir uma linha usando `cheap_products_view2` com o preço US\$19,50; observe que o banco de dados retorna um erro, pois a linha não pode ser recuperada pela visão:

```
INSERT INTO cheap_products_view2 (
    product_id, product_type_id, name, price
) VALUES (
    15, 1, 'Southern Front', 19.50
);
INSERT INTO cheap_products_view2 (
    *
ERROR at line 1:
ORA-01402: view WITH CHECK OPTION where-clause violation
```

**Criando uma visão com uma restrição READ ONLY**

Você pode definir uma visão somente para leitura adicionando a restrição `READ ONLY`. Por exemplo, a instrução a seguir cria uma visão chamada `cheap_products_view3` que tem uma restrição `READ ONLY`:

```
CREATE VIEW cheap_products_view3 AS
SELECT *
FROM products
WHERE price < 15
WITH READ ONLY CONSTRAINT cheap_products_view3_read_only;
```

O exemplo a seguir tenta inserir uma linha usando `cheap_products_view3`; observe que o banco de dados retorna um erro, pois a visão é somente leitura e não permite instruções DML:

```
INSERT INTO cheap_products_view3 (
    product_id, product_type_id, name, price
) VALUES (
    16, 1, 'Northern Front', 19.50
);
    product_id, product_type_id, name, price
    *
```

ERROR at line 2:  
ORA-42399: cannot perform a DML operation on a read-only view

**Obtendo informações sobre definições de visões**

Você pode recuperar informações sobre definições de visões usando o comando `DESCRIBE`. O exemplo a seguir usa `DESCRIBE` com `cheap_products_view3`:

```
DESCRIBE cheap_products_view3
```

Name	Null?	Type
PRODUCT_ID	NOT NULL	NUMBER (38)
PRODUCT_TYPE_ID		NUMBER (38)
NAME	NOT NULL	VARCHAR2 (30)
DESCRIPTION		VARCHAR2 (50)
PRICE		NUMBER (5,2)

Você também pode recuperar informações sobre suas visões a partir da visão `user_views`. A Tabela 10-9 descreve algumas das colunas de `user_views`.

**Tabela 10-9** Algumas colunas da visão `user_views`

Coluna	Tipo	Descrição
view_name	VARCHAR2 (30)	Nome da visão
text_length	NUMBER	Número de caracteres na subconsulta da visão
text	LONG	Texto da subconsulta da visão

**NOTA**

*Você pode recuperar informações sobre todas as visões a que tem acesso consultando all\_views.*

Para ver a definição da visão inteira armazenada na coluna text, use o comando SET LONG do SQL\*Plus, que configura o número de caracteres exibidos pelo SQL\*Plus ao recuperar colunas LONG. Por exemplo, o comando a seguir configura LONG como 200:

```
SET LONG 200
```

A consulta a seguir recupera as colunas view\_name, text\_length e text de user\_views:

```
SELECT view_name, text_length, text
FROM user_views
ORDER BY view_name;
```

VIEW_NAME	TEXT_LENGTH	TEXT
CHEAP_PRODUCTS_VIEW	97	SELECT "PRODUCT_ID", "PRODUCT_TYPE_ID", "NAME", "DESCRIPTION", "PRICE" FROM products WHERE price < 15
CHEAP_PRODUCTS_VIEW2	116	SELECT "PRODUCT_ID", "PRODUCT_TYPE_ID", "NAME", "DESCRIPTION", "PRICE" FROM products WHERE price < 15 WITH CHECK OPTION
CHEAP_PRODUCTS_VIEW3	113	SELECT "PRODUCT_ID", "PRODUCT_TYPE_ID", "NAME", "DESCRIPTION", "PRICE" FROM products WHERE price < 15 WITH READ ONLY
EMPLOYEES_VIEW	75	SELECT employee_id, manager_id, first_name, last_name, title FROM employees

### Recuperando informações sobre restrições de visões

Anteriormente, você viu que pode adicionar restrições CHECK OPTION e READ ONLY em uma visão; cheap\_products\_view2 continha uma restrição CHECK OPTION para garantir que o preço fosse menor do que US\$15; cheap\_products\_view3 continha uma restrição READ ONLY para impedir modificações nas linhas da tabela de base. Você recupera informações sobre restrições de visões a partir da visão user\_constraints; por exemplo:

```
SELECT constraint_name, constraint_type, status, deferrable, deferred
FROM user_constraints
WHERE table_name IN ('CHEAP_PRODUCTS_VIEW2', 'CHEAP_PRODUCTS_VIEW3')
ORDER BY constraint_name;
```

CONSTRAINT_NAME	C STATUS	DEFERRABLE	DEFERRED
CHEAP_PRODUCTS_VIEW2_PRICE	V ENABLED	NOT DEFERRABLE	IMMEDIATE
CHEAP_PRODUCTS_VIEW3_READ_ONLY	O ENABLED	NOT DEFERRABLE	IMMEDIATE

O valor de `constraint_type` de `CHEAP_PRODUCTS_VIEW2_PRICE` é `V`, o qual, como mostrado anteriormente na Tabela 10-3, corresponde a uma restrição `CHECK OPTION`. O valor de `constraint_type` de `CHEAP_PRODUCTS_VIEW3_READ_ONLY` é `O`, o qual corresponde a uma restrição `READ ONLY`.

### ***Criando e usando visões complexas***

As visões complexas contêm subconsultas que:

- Recuperam linhas de várias tabelas de base
- Agrupam linhas usando uma cláusula `GROUP BY` ou `DISTINCT`
- Contêm uma chamada de função

O exemplo a seguir cria uma visão chamada `products_and_types_view` cuja subconsulta realiza uma junção externa completa nas tabelas `products` e `product_types`, usando a sintaxe SQL/92:

```
CREATE VIEW products_and_types_view AS
SELECT p.product_id, p.name product_name, pt.name product_type_name,
p.price
FROM products p FULL OUTER JOIN product_types pt
USING (product_type_id)
ORDER BY p.product_id;
```

O exemplo a seguir consulta `products_and_types_view`:

```
SELECT *
FROM products_and_types_view;
```

PRODUCT_ID	PRODUCT_NAME	PRODUCT_TY	PRICE
1	Modern Science	Book	19.95
2	Chemistry	Book	30
3	Supernova	Video	25.99
4	Tank War	Video	13.95
5	Z Files	Video	49.99
6	2412: The Return	Video	14.95
7	Space Force 9	DVD	13.49
8	From Another Planet	DVD	12.99
9	Classical Music	CD	10.99
10	Pop 3	CD	15.99
11	Creative Yell	CD	14.99
12	My Front Line		13.49
13	Western Front	Book	13.5
14	Eastern Front	Book	16.5
		Magazine	

O próximo exemplo cria uma visão chamada `employee_salary_grades_view` cuja subconsulta utiliza uma junção interna para recuperar os níveis salariais dos funcionários:



```
CREATE VIEW employee_salary_grades_view AS
SELECT e.first_name, e.last_name, e.title, e.salary, sg.salary_grade_id
FROM employees e INNER JOIN salary_grades sg
ON e.salary BETWEEN sg.low_salary AND sg.high_salary
ORDER BY sg.salary_grade_id;
```

O exemplo a seguir consulta `employee_salary_grades_view`:

```
SELECT *
FROM employee_salary_grades_view;
```

FIRST_NAME	LAST_NAME	TITLE	SALARY	SALARY_GRADE_ID
Fred	Hobbs	Salesperson	150000	1
Susan	Jones	Salesperson	500000	2
Ron	Johnson	Sales Manager	600000	3
James	Smith	CEO	800000	4

O próximo exemplo cria uma visão chamada `product_average_view` cuja subconsulta usa:

- Uma cláusula `WHERE` para filtrar as linhas da tabela `products` naquelas cujo valor de `price` é menor do que US\$15.
- Uma cláusula `GROUP BY` para agrupar as linhas restantes pela coluna `product_type_id`.
- Uma cláusula `HAVING` para filtrar os grupos de linhas naqueles cujo preço médio é maior do que US\$13.

```
CREATE VIEW product_average_view AS
SELECT product_type_id, AVG(price) average_price
FROM products
WHERE price < 15
GROUP BY product_type_id
HAVING AVG(price) > 13
ORDER BY product_type_id;
```

O exemplo a seguir consulta `product_average_view`:

```
SELECT *
FROM product_average_view;
```

PRODUCT_TYPE_ID	AVERAGE_PRICE
1	13.5
2	14.45
3	13.24
	13.49

## Modificando uma visão

Você pode substituir uma visão completamente usando `CREATE OR REPLACE VIEW`. O exemplo a seguir usa `CREATE OR REPLACE VIEW` para substituir `product_average_view`:

```
CREATE OR REPLACE VIEW product_average_view AS
SELECT product_type_id, AVG(price) average_price
FROM products
```

```
WHERE price < 12
GROUP BY product_type_id
HAVING AVG(price) > 11
ORDER BY product_type_id;
```

Você pode alterar as restrições sobre uma visão usando `ALTER VIEW`. O exemplo a seguir usa `ALTER VIEW` para excluir a restrição `cheap_products_view2_price` de `cheap_products_view2`:

```
ALTER VIEW cheap_products_view2
DROP CONSTRAINT cheap_products_view2_price;
```

## Excluindo uma visão

Uma visão pode ser excluída usando `DROP VIEW`. O exemplo a seguir exclui `cheap_products_view2`:

```
DROP VIEW cheap_products_view2;
```

Isso conclui a discussão sobre visões. Na próxima seção, você vai aprender sobre arquivos de dados de flashback.

## ARQUIVOS DE DADOS DE FLASHBACK

Os arquivos de dados de flashback, novidade do Oracle Database 11g, armazenam as alterações feitas em uma tabela no decorrer de um período de tempo e fornecem uma trilha de auditoria completa. Depois de criar um arquivo de flashback e adicionar uma tabela nele, você pode:

- Ver as linhas como estavam em um timestamp específico
- Ver as linhas como estavam entre dois timestamps

Você cria um arquivo de flashback usando a instrução `CREATE FLASHBACK ARCHIVE`. O exemplo a seguir se conecta como o usuário `system` e cria um arquivo de flashback chamado `test_archive`:

```
CONNECT system/manager
CREATE FLASHBACK ARCHIVE test_archive
TABLESPACE example
QUOTA 1 M
RETENTION 1 DAY;
```

Observe que:

- O arquivo é criado no tablespace `example`; você pode ver a lista completa de tablespaces executando a consulta `SELECT tablespace_name FROM dba_tablespace`.
- `test_archive` tem uma cota de 1 megabyte, o que significa que ele pode armazenar até 1 megabyte de dados no tablespace `example`.
- Os dados de `test_archive` são mantidos por 1 dia, depois são apagados.

Você pode alterar uma tabela existente para armazenar dados no arquivo, por exemplo:

```
ALTER TABLE store.products FLASHBACK ARCHIVE test_archive;
```

Agora, todas as alterações subseqüentes feitas na tabela `store.products` serão gravadas no arquivo. A instrução `INSERT` a seguir adiciona uma linha na tabela `store.products`:

```
INSERT INTO store.products (
    product_id, product_type_id, name, description, price
) VALUES (
    15, 1, 'Using Linux', 'How to Use Linux', 39.99
);
```

A consulta a seguir recupera essa linha:

```
SELECT product_id, name, price
FROM store.products
WHERE product_id = 15;
```

PRODUCT_ID	NAME	PRICE
15	Using Linux	39.99

Você pode ver as linhas como estavam 5 minutos atrás usando a consulta a seguir:

```
SELECT product_id, name, price
FROM store.products
AS OF TIMESTAMP
(SYSTIMESTAMP - INTERVAL '5' MINUTE);
```

PRODUCT_ID	NAME	PRICE
1	Modern Science	19.95
2	Chemistry	30
3	Supernova	25.99
4	Tank War	13.95
5	Z Files	49.99
6	2412: The Return	14.95
7	Space Force 9	13.49
8	From Another Planet	12.99
9	Classical Music	10.99
10	Pop 3	15.99
11	Creative Yell	14.99
12	My Front Line	13.49
13	Western Front	13.5
14	Eastern Front	16.5

Note que a nova linha está ausente. Isso porque ela foi adicionada na tabela após a data e hora especificadas na consulta (supondo que a instrução `INSERT` anterior tenha sido executada há menos de 5 minutos). Você também pode ver as linhas como estavam em um timestamp específico usando a consulta a seguir (se executar essa consulta, precisará alterar o timestamp para uma data e hora anterior àquelas em que executou a instrução `INSERT` anterior):

```
SELECT product_id, name, price
FROM store.products
AS OF TIMESTAMP
TO_TIMESTAMP('2007-08-12 13:05:00', 'YYYY-MM-DD HH24:MI:SS');
```

A nova linha ficará ausente dos resultados mais uma vez, pois foi adicionada na tabela após a data e hora especificadas na consulta. Você pode ver as linhas como estavam entre dois timestamps usando a consulta a seguir (é preciso alterar os timestamps):

```
SELECT product_id, name, price
FROM store.products VERSIONS BETWEEN TIMESTAMP
TO_TIMESTAMP('2007-08-12 12:00:00', 'YYYY-MM-DD HH24:MI:SS')
AND TO_TIMESTAMP('2007-08-12 12:59:59', 'YYYY-MM-DD HH24:MI:SS');
```

Você pode ver as linhas como estavam entre um timestamp e a hora atual usando a consulta a seguir (é preciso alterar o timestamp):

```
SELECT product_id, name, price
FROM store.products VERSIONS BETWEEN TIMESTAMP
TO_TIMESTAMP('2007-08-12 13:45:52', 'YYYY-MM-DD HH24:MI:SS')
AND MAXVALUE;
```

Você pode interromper o arquivamento dos dados de uma tabela no repositório usando ALTER TABLE; por exemplo:

```
ALTER TABLE store.products NO FLASHBACK ARCHIVE;
```

Quando cria uma tabela, você pode especificar um arquivo de flashback para ela, por exemplo:

```
CREATE TABLE store.test_table (
    id INTEGER,
    name VARCHAR2(10)
) FLASHBACK ARCHIVE test_archive;
```

Você pode ver os detalhes de um arquivo de flashback usando as seguintes visões:

- user\_flashback\_archive e dba\_flashback\_archive, que exibem informações gerais sobre os arquivos de flashback
- user\_flashback\_archive\_ts e dba\_flashback\_archive\_ts, que exibem informações sobre os tablespaces que contêm os arquivos de flashback
- user\_flashback\_archive\_tables e dba\_flashback\_archive\_tables, que exibem informações sobre as tabelas habilitadas para arquivamento de flashback

Você pode alterar um arquivo de flashback; por exemplo, a instrução a seguir altera o período de retenção de dados para 2 anos:

```
ALTER FLASHBACK ARCHIVE test_archive
MODIFY RETENTION 2 YEAR;
```

Você pode apagar os dados de um arquivo de flashback antes de um determinado timestamp; por exemplo, a instrução a seguir apaga dados mais antigos do que 1 dia:

```
ALTER FLASHBACK ARCHIVE test_archive
PURGE BEFORE TIMESTAMP(SYSTIMESTAMP - INTERVAL '1' DAY);
```

Você pode apagar todos os dados de um arquivo de flashback, por exemplo:

```
ALTER FLASHBACK ARCHIVE test_archive PURGE ALL;
```

Você pode excluir um arquivo de flashback, por exemplo:

```
DROP FLASHBACK ARCHIVE test_archive;
```

#### NOTA

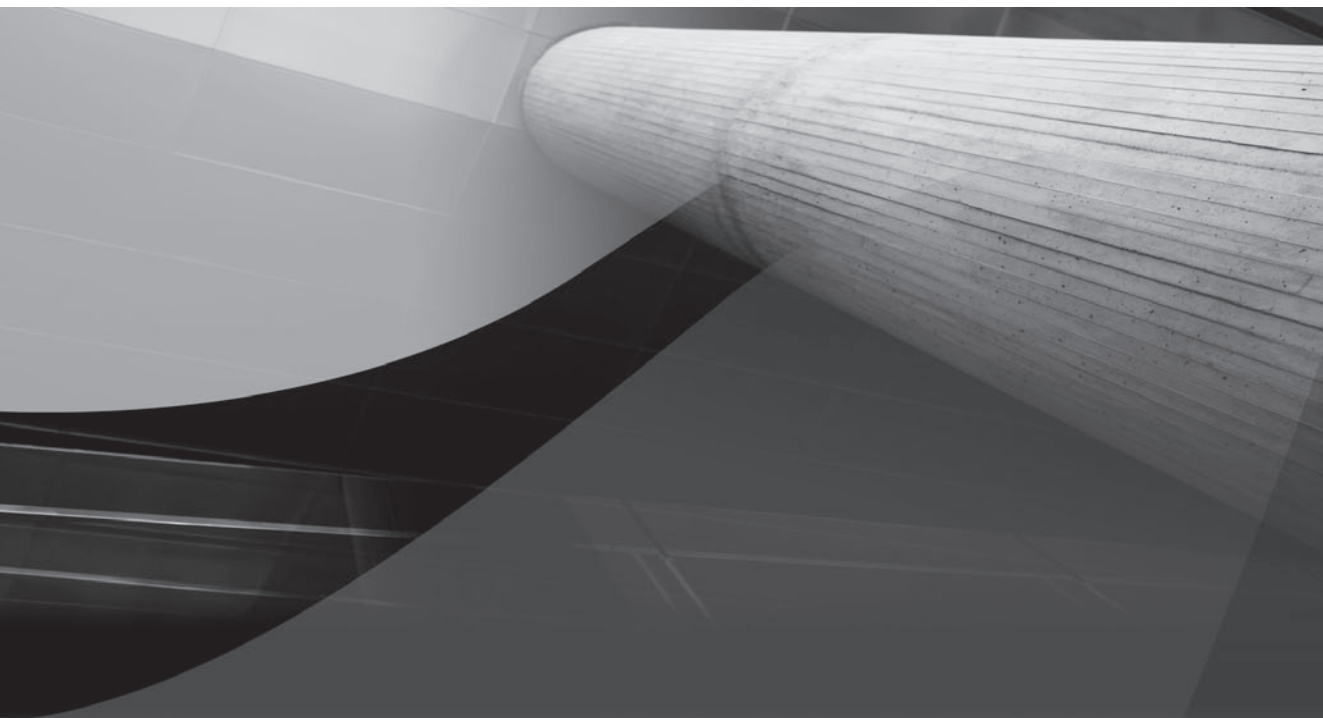
*Execute novamente `store_schema.sql` para recriar as tabelas da loja, a fim de que suas consultas correspondam às do restante do livro.*

## RESUMO

Neste capítulo, você aprendeu que:

- Uma tabela é criada com a instrução `CREATE TABLE`
- Uma sequência gera uma série de números inteiros
- Um índice de banco de dados pode acelerar o acesso às linhas
- Uma visão é uma consulta predefinida em uma ou mais tabelas de base
- Um arquivo de dados de flashback armazena as alterações feitas em uma tabela durante um período de tempo

No próximo capítulo, você vai aprender sobre programação PL/SQL.



# CAPÍTULO 11

Introdução à  
programação PL/SQL

A Oracle adicionou uma linguagem de programação procedural, conhecida como PL/SQL (Procedural Language/SQL), no Oracle Database 6. O PL/SQL permite escrever programas que contêm instruções SQL. Neste capítulo, você irá aprender sobre os seguintes tópicos PL/SQL:

- Estrutura de bloco
- Variáveis e tipos
- Lógica condicional
- Loops
- Cursores, que permitem à PL/SQL ler os resultados retornados por uma consulta
- Procedures
- Funções
- Pacotes, que são usados para agrupar procedures e funções em uma unidade
- Triggers, que são blocos de código executados quando certo evento ocorre no banco de dados
- Aprimoramentos feitos no PL/SQL pelo Oracle Database 11g

Você pode usar PL/SQL para adicionar lógica de negócio em um aplicativo de banco de dados. Essa lógica de negócio centralizada pode ser executada por qualquer programa que possa acessar o banco de dados, incluindo o SQL\*Plus, programas em Java, programas em C# e muito mais.



#### NOTA

*Para saber como acessar um banco de dados por meio de Java, consulte o livro Oracle9i JDBC Programming (Oracle Press, 2002). Para aprender a acessar um banco de dados por meio de C#, consulte o livro Mastering C# Database Programming (Sybex, 2003).*

## ESTRUTURA DE BLOCO

Os programas em PL/SQL são divididos em estruturas conhecidas como *blocos*, com cada bloco contendo instruções PL/SQL e SQL. Um bloco PL/SQL tem a seguinte estrutura:

```
[DECLARE
    instruções_de_declaração
]
BEGIN
    instruções_executáveis
[EXCEPTION
    instruções_de_tratamento_de_exceção
]
END;
/
```

onde

- *instruções\_de\_declaração* declaram as variáveis usadas no restante do bloco PL/SQL. Os blocos DECLARE são opcionais.
- *instruções\_executáveis* são as instruções que serão executadas, as quais podem incluir loops, lógica condicional etc.
- *instruções\_de\_tratamento\_de\_exceção* são instruções que tratam de todos os erros de execução que possam ocorrer quando o bloco é executado. Os blocos EXCEPTION são opcionais.

Toda instrução é terminada por um ponto-e-vírgula (;) e um bloco PL/SQL é terminado com o caractere de barra normal (/). Antes de entrarmos nos detalhes do PL/SQL, você verá um exemplo simples para ter uma noção geral da linguagem. O exemplo a seguir (contido no script `area_example.sql` no diretório `SQL`) calcula a largura de um retângulo, dada sua área e altura:

```
SET SERVEROUTPUT ON
```

```
DECLARE
  v_width INTEGER;
  v_height INTEGER := 2;
  v_area INTEGER := 6;
BEGIN
  -- configura a largura igual à área dividida pela altura
  v_width := v_area / v_height;
  DBMS_OUTPUT.PUT_LINE('v_width = ' || v_width);
EXCEPTION
  WHEN ZERO_DIVIDE THEN
    DBMS_OUTPUT.PUT_LINE('Division by zero');
END;
/
```

O comando `SET SERVEROUTPUT ON` ativa a saída do servidor para que você possa ver as linhas produzidas por `DBMS_OUTPUT.PUT_LINE()` na tela quando executar o script no SQL\*Plus. Depois desse comando inicial vem o bloco PL/SQL em si, que é dividido nos blocos DECLARE, BEGIN e EXCEPTION.

O bloco DECLARE contém declarações de três variáveis INTEGER, chamadas `v_width`, `v_height` e `v_area` (colocaremos `v_` no início dos nomes das variáveis). As variáveis `v_height` e `v_area` são inicializadas com 2 e 6, respectivamente.

Em seguida vem o bloco BEGIN, que contém três linhas. A primeira é um comentário que contém o texto “configura a largura igual à área dividida pela altura”. A segunda linha configura `v_width` como `v_area` dividida por `v_height`; isso significa que `v_width` é configurada com 3 (= 6 / 2). A terceira linha chama `DBMS_OUTPUT.PUT_LINE()` para exibir o valor de `v_width` na tela. `DBMS_OUTPUT` é um pacote de código que acompanha o banco de dados Oracle; dentre outros itens, `DBMS_OUTPUT` contém procedures que permitem exibir valores na tela.

Em seguida, o bloco EXCEPTION trata das tentativas de dividir um número por zero. Ele faz isso “capturando” a exceção `ZERO_DIVIDE`; no exemplo, nenhuma tentativa de dividir por zero é feita realmente, mas se você alterar `v_height` para 0 e executar o script, verá a exceção.

No final do script, o caractere de barra normal (/) marca o fim do bloco PL/SQL.



A listagem a seguir mostra a execução do script `area_example.sql` no SQL\*Plus:

```
SQL> @ C:\SQL\area_example.sql
v_width = 3
```

### NOTA

Se seu script `area_example.sql` está em um diretório que não é `C:\SQL`, use seu próprio diretório no comando anterior.

## VARIÁVEIS E TIPOS

As variáveis são declaradas dentro de um bloco `DECLARE`. Como foi visto no exemplo anterior, uma declaração de variável tem um nome e um tipo. Por exemplo, a variável `v_width` foi declarada como

```
v_width INTEGER;
```

### NOTA

Os tipos PL/SQL são semelhantes aos tipos de coluna de banco de dados. Você pode ver todos os tipos no apêndice.

O exemplo a seguir mostra mais declarações de variável (essas variáveis poderiam ser usadas para armazenar os valores de coluna da tabela `products`):

```
v_product_id      INTEGER;
v_product_type_id INTEGER;
v_name            VARCHAR2(30);
v_description      VARCHAR2(50);
v_price           NUMBER(5, 2);
```

Você também pode especificar o tipo de uma variável usando a palavra-chave `%TYPE`, que diz ao PL/SQL para que use o mesmo tipo da coluna especificada em uma tabela. O exemplo a seguir usa `%TYPE` para declarar uma variável do mesmo tipo da coluna `price` da tabela `products`, que é `NUMBER(5, 2)`:

```
v_product_price products.price%TYPE;
```

## LÓGICA CONDICIONAL

Você usa as palavras-chave `IF`, `THEN`, `ELSE`, `ELSIF` e `END IF` para executar lógica condicional:

```
IF condição1 THEN
    instruções1
ELSIF condição2 THEN
    instruções2
ELSE
    instruções3
END IF;
```

onde

- `condição1` e `condição2` são expressões booleanas avaliadas como verdadeiras ou falsas.
- `instruções1`, `instruções2` e `instruções3` são instruções PL/SQL.

A lógica condicional flui como segue:

- Se *condição1* é verdadeira, então *instruções1* são executadas.
- Se *condição1* é falsa, mas *condição2* é verdadeira, então *instruções2* são executadas.
- Se nem *condição1* nem *condição2* é verdadeira, então *instruções3* são executadas.

Você também pode incorporar uma instrução IF dentro de outra instrução IF, como mostrado:

```
IF v_count > 0 THEN
    v_message := 'v_count is positive';
    IF v_area > 0 THEN
        v_message := 'v_count and v_area are positive';
    END IF
ELSIF v_count = 0 THEN
    v_message := 'v_count is zero';
ELSE
    v_message := 'v_count is negative';
END IF;
```

Nesse exemplo, se *v\_count* é maior do que 0, então *v\_message* é configurada como 'v\_count is positive'. Se *v\_count* e *v\_area* são maiores do que 0, então *v\_message* é configurada como 'v\_count and v\_area are positive'. O restante da lógica é óbvio.

## LOOPS

Você usa um loop para executar instruções zero ou mais vezes. Existem três tipos de loops na PL/SQL:

- Os **loops simples** são executados até que você termine o loop explicitamente.
- Os **loops WHILE** são executados até que ocorra uma condição especificada.
- Os **loops FOR** são executados um número predeterminado de vezes.

Você vai aprender sobre esses loops nas seções a seguir.

### Loops simples

Um loop simples é executado até que você o termine explicitamente. A sintaxe de um loop simples é:

```
LOOP
    instruções
END LOOP;
```

Para terminar o loop, use uma instrução EXIT ou EXIT WHEN. A instrução EXIT termina um loop imediatamente; a instrução EXIT WHEN termina um loop quando ocorre uma condição especificada.

O exemplo a seguir mostra um loop simples. Uma variável chamada *v\_counter* é inicializada como 0 antes do início do loop. O loop soma 1 em *v\_counter* e termina quando *v\_counter* é igual a 5, usando uma instrução EXIT WHEN.

```
v_counter := 0;
LOOP
    v_counter := v_counter + 1;
    EXIT WHEN v_counter = 5;
END LOOP;
```

**NOTA**

A instrução `EXIT WHEN` pode aparecer em qualquer lugar no código do loop.

No Oracle Database 11g, você também pode terminar a iteração atual de um loop usando a instrução `CONTINUE` ou `CONTINUE WHEN`. A instrução `CONTINUE` termina a iteração atual do loop incondicionalmente e continua na próxima iteração; a instrução `CONTINUE WHEN` termina a iteração atual do loop quando ocorre uma condição especificada e, então, continua na próxima iteração. O exemplo a seguir mostra o uso da instrução `CONTINUE`:

```
v_counter := 0;
LOOP
  -- após a instrução CONTINUE ser executada, o controle retorna para cá
  v_counter := v_counter + 1;
  IF v_counter = 3 THEN
    CONTINUE; -- termina a iteração atual incondicionalmente
  END IF;
  EXIT WHEN v_counter = 5;
END LOOP;
```

O exemplo a seguir mostra o uso da instrução `CONTINUE WHEN`:

```
v_counter := 0;
LOOP
  -- após a instrução CONTINUE WHEN ser executada, o controle retorna para cá
  v_counter := v_counter + 1;
  CONTINUE WHEN v_counter = 3; -- termina a iteração atual quando v_counter
  = 3
  EXIT WHEN v_counter = 5;
END LOOP;
```

**NOTA**

Uma instrução `CONTINUE` ou `CONTINUE WHEN` não pode ultrapassar o limite de uma *procedure*, *função* ou *método*.

## Loops WHILE

Um loop `WHILE` é executado até que ocorra uma condição especificada. A sintaxe de um loop `WHILE` é:

```
WHILE condição LOOP
  instruções
END LOOP;
```

O exemplo a seguir mostra um loop `WHILE` que é executado enquanto a variável `v_counter` é menor do que 6:

```
v_counter := 0;
WHILE v_counter < 6 LOOP
  v_counter := v_counter + 1;
END LOOP;
```

## Loops FOR

Um loop `FOR` é executado um número predeterminado de vezes; você determina o número de vezes especificando os *limites inferior* e *superior* de uma variável de loop. Então, a variável de loop é incrementada (ou decrementada) em cada passagem do loop. A sintaxe de um loop `FOR` é:

```
FOR variável_loop IN [REVERSE] limite_inferior..limite_superior LOOP
    instruções
END LOOP;
```

onde

- *variável\_loop* é a variável de loop. Você pode usar uma variável que já exista como variável de loop ou simplesmente fazer o loop criar uma nova variável (isso ocorre se a variável especificada não existe). O valor da variável de loop é aumentado (ou diminuído se você usa a palavra-chave *REVERSE*) por 1 em cada passagem do loop.
- *REVERSE* significa que o valor da variável de loop deve ser decrementado em cada passagem do loop. A variável de loop é inicializada com o limite superior e é decrementada por 1 até atingir o limite inferior. Você deve especificar o limite inferior antes do limite superior.
- *limite\_inferior* é o limite inferior do loop. A variável de loop é inicializada com esse limite inferior, desde que a palavra-chave *REVERSE* não seja usada.
- *limite\_superior* é o limite superior do loop. Se a palavra-chave *REVERSE* for usada, a variável de loop será inicializada com esse limite superior.

O exemplo a seguir mostra um loop *FOR*. Note que a variável *v\_counter2* não é declarada explicitamente — portanto, o loop *FOR* cria automaticamente uma nova variável *INTEGER*, chamada *v\_counter2*:

```
FOR v_counter2 IN 1..5 LOOP
    DBMS_OUTPUT.PUT_LINE(v_counter2);
END LOOP;
```

O exemplo a seguir usa *REVERSE*:

```
FOR v_counter2 IN REVERSE 1..5 LOOP
    DBMS_OUTPUT.PUT_LINE(v_counter2);
END LOOP;
```

Nesse exemplo, *v\_counter2* começa em 5, é decrementada por 1 em cada passagem do loop e termina em 1.

## CURSORES

Você usa um *cursor* para buscar as linhas retornadas por uma consulta. Você recupera as linhas no cursor usando uma consulta e, então, busca as linhas do cursor, uma por vez. Normalmente, você usa os cinco passos a seguir ao utilizar um cursor:

1. Declarar variáveis para armazenar os valores de coluna de uma linha.
2. Declarar o cursor, o qual contém uma consulta.
3. Abrir o cursor.
4. Buscar uma linha do cursor por vez e armazenar os valores de coluna nas variáveis declaradas no passo 1. Então, você faria algo com essas variáveis, como exibi-las na tela, as usaria em um cálculo etc.
5. Fechar o cursor.

Você vai aprender os detalhes desses cinco passos nas seções a seguir e vai ver um exemplo simples que obtém as colunas `product_id`, `name` e `price` da tabela `products`.

## Passo 1: Declarar as variáveis para armazenar os valores de coluna

O primeiro passo é declarar as variáveis que serão usadas para armazenar os valores de coluna. Essas variáveis devem ser compatíveis com os tipos de coluna.



### DICA

*Anteriormente, você viu que %TYPE pode ser usado para obter o tipo de uma coluna. Se você usar %TYPE ao declarar suas variáveis, elas terão o tipo correto automaticamente.*

O exemplo a seguir declara três variáveis para armazenar as colunas `product_id`, `name` e `price` da tabela `products`; observe que `%TYPE` é usado para configurar o tipo das variáveis automaticamente com o mesmo tipo das colunas:

```
DECLARE
  v_product_id products.product_id%TYPE;
  v_name products.name%TYPE;
  v_price products.price%TYPE;
```

## Passo 2: Declarar o cursor

O passo 2 é declarar o cursor. Uma declaração de cursor consiste em um nome atribuído a ele e a consulta que você deseja executar. A declaração de cursor, assim como todas as outras declarações, é colocada na seção de declaração. A sintaxe para declarar um cursor é:

```
CURSOR nome_cursor IS
  instrução_SELECT;
```

onde

- `nome_cursor` é o nome do cursor
- `instrução_SELECT` é a consulta

O exemplo a seguir declara um cursor chamado `v_product_cursor` cuja consulta recupera as colunas `product_id`, `name` e `price` da tabela `products`:

```
CURSOR v_product_cursor IS
  SELECT product_id, name, price
  FROM products
  ORDER BY product_id;
```

A consulta não é executada até que você abra o cursor.

## Passo 3: Abrir o cursor

O passo 3 é abrir o cursor. Você abre um cursor usando a instrução `OPEN`, que deve ser colocada na seção executável do bloco.

O exemplo a seguir abre `v_product_cursor`, que executa a consulta:

```
OPEN v_product_cursor;
```

## Passo 4: Buscar as linhas do cursor

O passo 4 é buscar as linhas do cursor, o que você faz usando a instrução `FETCH`. A instrução `FETCH` lê os valores de coluna nas variáveis declaradas no passo 1. A instrução `FETCH` usa a seguinte sintaxe:

```
FETCH nome_cursor
INTO variável[, variável...];
```

onde

- `nome_cursor` é o nome do cursor.
- `variável` é a variável em que um valor de coluna do cursor é armazenado. Você precisa fornecer variáveis correspondentes para cada valor de coluna.

O exemplo de instrução `FETCH` a seguir recupera uma linha de `v_product_cursor` e armazena os valores de coluna nas variáveis `v_product_id`, `v_name` e `v_price`, criadas anteriormente no passo 1:

```
FETCH v_product_cursor
INTO v_product_id, v_name, v_price;
```

Como um cursor pode conter muitas linhas, você precisa fazer um loop para lê-las. Para descobrir quando o deve terminar o loop, você pode usar a variável booleana `v_product_cursor%NOTFOUND`. Essa variável é verdadeira quando não existem mais linhas para ler em `v_product_cursor`. O exemplo a seguir mostra um loop:

```
LOOP
  -- busca as linhas do cursor
  FETCH v_product_cursor
  INTO v_product_id, v_name, v_price;

  -- sai do loop quando não há mais linhas, conforme indicado pela
  -- variável booleana v_product_cursor%NOTFOUND (= verdadeira quando
  -- não há mais linhas)
  EXIT WHEN v_product_cursor%NOTFOUND;

  -- usa DBMS_OUTPUT.PUT_LINE() para exibir as variáveis
  DBMS_OUTPUT.PUT_LINE(
    'v_product_id = ' || v_product_id || ', v_name = ' || v_name ||
    ', v_price = ' || v_price
  );
END LOOP;
```

Note que `DBMS_OUTPUT.PUT_LINE()` foi usada para exibir as variáveis `v_product_id`, `v_name` e `v_price` que foram lidas de cada linha. Em um aplicativo real, você poderia usar `v_price` em um cálculo complexo.

## Passo 5: Fechar o cursor

O passo 5 é fechar o cursor usando a instrução `CLOSE`. O fechamento de um cursor libera recursos do sistema. O exemplo a seguir fecha `v_product_cursor`:

```
CLOSE v_product_cursor;
```

A seção a seguir mostra um script completo que contém todos os cinco passos.

## Exemplo completo: product\_cursor.sql

O script `product_cursor.sql` a seguir está contido no diretório SQL:

```
-- product_cursor.sql exibe as colunas product_id, name
-- e price da tabela products usando um cursor

SET SERVEROUTPUT ON

DECLARE
    -- passo 1: declarar as variáveis
    v_product_id products.product_id%TYPE;
    v_name products.name%TYPE;
    v_price products.price%TYPE;

    -- passo 2: declarar o cursor
    CURSOR v_product_cursor IS
        SELECT product_id, name, price
        FROM products
        ORDER BY product_id;
BEGIN
    -- passo 3: abrir o cursor
    OPEN v_product_cursor;

    LOOP
        -- passo 4: buscar as linhas do cursor
        FETCH v_product_cursor
        INTO v_product_id, v_name, v_price;

        -- sai do loop quando não existem mais linhas, conforme indicado pela
        -- variável booleana v_product_cursor%NOTFOUND (= verdadeira quando
        -- não existem mais linhas)
        EXIT WHEN v_product_cursor%NOTFOUND;

        -- usa DBMS_OUTPUT.PUT_LINE() para exibir as variáveis
        DBMS_OUTPUT.PUT_LINE(
            'v_product_id = ' || v_product_id || ', v_name = ' || v_name ||
            ', v_price = ' || v_price
        );
    END LOOP;

    -- passo 5: fechar o cursor
    CLOSE v_product_cursor;
END;
```

Para executar esse script, siga estes passos:

1. Conecte-se no banco de dados como `store` com a senha `store_password`.
2. Execute o script `product_cursor.sql` usando o SQL\*Plus:

```
SQL> @ C:\SQL\product_cursor.sql
```

**NOTA**

*Se seu script `product_cursor.sql` estiver em um diretório diferente de `C:\SQL`, use seu próprio diretório no comando anterior.*

A saída de `product_cursor.sql` é:

```
v_product_id = 1, v_name = Modern Science, v_price = 19.95
v_product_id = 2, v_name = Chemistry, v_price = 30
v_product_id = 3, v_name = Supernova, v_price = 25.99
v_product_id = 4, v_name = Tank War, v_price = 13.95
v_product_id = 5, v_name = Z Files, v_price = 49.99
v_product_id = 6, v_name = 2412: The Return, v_price = 14.95
v_product_id = 7, v_name = Space Force 9, v_price = 13.49
v_product_id = 8, v_name = From Another Planet, v_price = 12.99
v_product_id = 9, v_name = Classical Music, v_price = 10.99
v_product_id = 10, v_name = Pop 3, v_price = 15.99
v_product_id = 11, v_name = Creative Yell, v_price = 14.99
v_product_id = 12, v_name = My Front Line, v_price = 13.49
```

## Cursores e loops FOR

Você pode usar um loop FOR para acessar as linhas de um cursor. Quando faz isso, você não precisa abrir e fechar o cursor explicitamente—o loop FOR faz isso automaticamente para você. O script `product_cursor2.sql` a seguir usa um loop FOR para acessar as linhas de `v_product_cursor`; observe que esse script contém menos código do que `product_cursor.sql`:

```
-- product_cursor2.sql exibe as colunas product_id, name
-- e price da tabela products usando um cursor
-- e um loop FOR
```

```
SET SERVEROUTPUT ON
```

```
DECLARE
```

```
    CURSOR v_product_cursor IS
        SELECT product_id, name, price
        FROM products
        ORDER BY product_id;
```

```
BEGIN
```

```
    FOR v_product IN v_product_cursor LOOP
        DBMS_OUTPUT.PUT_LINE(
            'product_id = ' || v_product.product_id ||
            ', name = ' || v_product.name ||
            ', price = ' || v_product.price
        );
```

```
    END LOOP;
```

```
END;
```

```
/
```

Para executar o script `product_cursor2.sql`, execute um comando semelhante ao seguinte:

```
SQL> @ "C:\SQL\product_cursor2.sql"
```



A saída desse script é:

```
product_id = 1, name = Modern Science, price = 19.95
product_id = 2, name = Chemistry, price = 30
product_id = 3, name = Supernova, price = 25.99
product_id = 4, name = Tank War, price = 13.95
product_id = 5, name = Z Files, price = 49.99
product_id = 6, name = 2412: The Return, price = 14.95
product_id = 7, name = Space Force 9, price = 13.49
product_id = 8, name = From Another Planet, price = 12.99
product_id = 9, name = Classical Music, price = 10.99
product_id = 10, name = Pop 3, price = 15.99
product_id = 11, name = Creative Yell, price = 14.99
product_id = 12, name = My Front Line, price = 13.49
```

## Instrução OPEN-FOR

Você também pode usar a instrução OPEN-FOR com um cursor, a qual proporciona ainda mais flexibilidade no processamento de cursores, pois é possível atribuir uma consulta diferente ao cursor. Isso está mostrado no script `product_cursor3.sql` a seguir:

```
-- product_cursor3.sql exibe as colunas product_id, name
-- e price da tabela products usando uma variável de
-- cursor e a instrução OPEN-FOR

SET SERVEROUTPUT ON

DECLARE
    -- declara um tipo REF CURSOR chamado t_product_cursor
    TYPE t_product_cursor IS
    REF CURSOR RETURN products%ROWTYPE;

    -- declara um objeto t_product_cursor chamado v_product_cursor
    v_product_cursor t_product_cursor;

    -- declara um objeto para armazenar as colunas da tabela products
    -- chamado v_product (de tipo products%ROWTYPE)
    v_product products%ROWTYPE;
BEGIN
    -- atribui uma consulta a v_product_cursor e o abre usando OPEN-FOR
    OPEN v_product_cursor FOR
    SELECT * FROM products WHERE product_id < 5;

    -- usa um loop para buscar as linhas de v_product_cursor em v_product
    LOOP
        FETCH v_product_cursor INTO v_product;
        EXIT WHEN v_product_cursor%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(
            'product_id = ' || v_product.product_id ||
            ', name = ' || v_product.name ||
            ', price = ' || v_product.price
        );
    END LOOP;
```

```

END LOOP;

-- fecha v_product_cursor
CLOSE v_product_cursor;
END;
/

```

No bloco DECLARE, a seguinte instrução declara um tipo REF CURSOR chamado t\_product\_cursor (colocaremos t\_ no início de nomes de tipo):

```

TYPE t_product_cursor IS
REF CURSOR RETURN products%ROWTYPE;

```

Um tipo REF CURSOR é um ponteiro para um cursor e é semelhante a um ponteiro da linguagem de programação C++. A instrução anterior declara um tipo definido pelo usuário chamado t\_product\_cursor e retorna uma linha contendo as várias colunas da tabela products (isso é indicado usando %ROWTYPE). Esse tipo definido pelo usuário pode ser usado para declarar um objeto, como mostrado na instrução a seguir, que declara um objeto chamado v\_product\_cursor:

```

v_product_cursor t_product_cursor;

```

A instrução a seguir declara um objeto para armazenar as colunas da tabela products chamado v\_product (de tipo products%ROWTYPE):

```

v_product products%ROWTYPE;

```

No bloco BEGIN, uma consulta é atribuída a v\_product\_cursor e o cursor é aberto pela seguinte instrução OPEN-FOR:

```

OPEN v_product_cursor FOR
SELECT * FROM products WHERE product_id < 5;

```

Após essa instrução ser executada, v\_product\_cursor será carregado com as quatro primeiras linhas da tabela products. A consulta atribuída a v\_product\_cursor pode ser qualquer instrução SELECT válida; isso significa que você pode reutilizar o cursor e atribuir outra consulta a ele posteriormente no código PL/SQL. Depois, o loop a seguir busca as linhas de v\_product\_cursor em v\_product e exibe os detalhes da linha:

```

LOOP
  FETCH v_product_cursor INTO v_product;
  EXIT WHEN v_product_cursor%NOTFOUND;
  DBMS_OUTPUT.PUT_LINE(
    'product_id = ' || v_product.product_id ||
    ', name = ' || v_product.name ||
    ', price = ' || v_product.price
  );
END LOOP;

```

Após o loop, v\_product\_cursor é fechado por meio da seguinte instrução:

```

CLOSE v_product_cursor;

```

A saída desse script é igual à saída de product\_cursor2.sql.

## Cursores irrestritos

Todos os cursores da seção anterior têm um tipo de retorno específico; esses cursores são conhecidos como cursores restritos. O tipo de retorno de um cursor restrito deve corresponder às colunas da consulta executada pelo cursor. Um cursor irrestrito não tem tipo de retorno e, portanto, pode executar qualquer consulta.

O uso de um cursor irrestrito está mostrado no script `unconstrained_cursor.sql` a seguir; observe que, no código, `v_cursor` é usado para executar duas consultas diferentes:

```
-- Este script mostra o uso de cursores irrestritos

SET SERVEROUTPUT ON

DECLARE
    -- declara um tipo REF CURSOR chamado t_cursor (que não tem nenhum tipo de
    -- retorno e, portanto, pode executar qualquer consulta)
    TYPE t_cursor IS REF CURSOR;

    -- declara um objeto t_cursor chamado v_cursor
    v_cursor t_cursor;

    -- declara um objeto para armazenar as colunas da tabela products,
    -- chamado v_product (de tipo products%ROWTYPE)
    v_product products%ROWTYPE;

    -- declara um objeto para armazenar as colunas da tabela customers,
    -- chamado v_customer (de tipo customers%ROWTYPE)
    v_customer customers%ROWTYPE;
BEGIN
    -- atribui uma consulta a v_cursor e abre-o usando OPEN-FOR
    OPEN v_cursor FOR
    SELECT * FROM products WHERE product_id < 5;

    -- usa um loop para buscar as linhas de v_cursor em v_product
    LOOP
        FETCH v_cursor INTO v_product;
        EXIT WHEN v_cursor%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(
            'product_id = ' || v_product.product_id ||
            ', name = ' || v_product.name ||
            ', price = ' || v_product.price
        );
    END LOOP;

    -- atribui uma nova consulta a v_cursor e abre-o usando OPEN-FOR
    OPEN v_cursor FOR
    SELECT * FROM customers WHERE customer_id < 3;

    -- usa um loop para buscar as linhas de v_cursor em v_product
    LOOP
        FETCH v_cursor INTO v_customer;
        EXIT WHEN v_cursor%NOTFOUND;
```

```

DBMS_OUTPUT.PUT_LINE(
    'customer_id = ' || v_customer.customer_id ||
    ', first_name = ' || v_customer.first_name ||
    ', last_name = ' || v_customer.last_name
);
END LOOP;

-- fecha v_cursor
CLOSE v_cursor;
END;
/

```

Para executar o script `unconstrained_cursor.sql`, execute um comando semelhante ao seguinte:

```
SQL> @ "C:\SQL\unconstrained_cursor.sql"
```

A saída desse script é:

```

product_id = 1, name = Modern Science, price = 19.95
product_id = 2, name = Chemistry, price = 30
product_id = 3, name = Supernova, price = 25.99
product_id = 4, name = Tank War, price = 13.95
customer_id = 1, first_name = John, last_name = Brown
customer_id = 2, first_name = Cynthia, last_name = Green

```

Você vai aprender mais sobre variáveis REF CURSOR posteriormente neste capítulo e mais sobre tipos definidos pelo usuário no próximo capítulo.

## EXCEÇÕES

As exceções são usadas para tratar de erros em tempo de execução no código PL/SQL. Anteriormente, você viu o seguinte exemplo de código PL/SQL, que contém um bloco `EXCEPTION`:

```

DECLARE
    v_width INTEGER;
    v_height INTEGER := 2;
    v_area INTEGER := 6;
BEGIN
    -- configura a largura igual à área dividida pela altura
    v_width := v_area / v_height;
    DBMS_OUTPUT.PUT_LINE('v_width = ' || v_width);
EXCEPTION
    WHEN ZERO_DIVIDE THEN
        DBMS_OUTPUT.PUT_LINE('Division by zero');
END;
/

```

O bloco `EXCEPTION` desse exemplo trata de uma tentativa de dividir um número por zero. Na terminologia da PL/SQL, o bloco `EXCEPTION` *captura* uma exceção `ZERO_DIVIDE` *lançada* no bloco `BEGIN` (embora no exemplo de código, `ZERO_DIVIDE` nunca seja realmente lançada). A exceção `ZERO_DIVIDE` e as outras exceções comuns estão mostradas na Tabela 11-1.

**Tabela 11-1** Exceções predefinidas

Exceção	Erro	Descrição
ACCESS_INTO_NULL	ORA-06530	Foi feita uma tentativa de designar valores aos atributos de um objeto não inicializado. (Você vai aprender sobre objetos no Capítulo 12.)
CASE_NOT_FOUND	ORA-06592	Nenhuma das cláusulas WHEN de uma instrução CASE foi selecionada e não há nenhuma cláusula ELSE padrão.
COLLECTION_IS_NULL	ORA-06531	Foi feita uma tentativa de chamar um método de coleção (não sendo o método EXISTS) em uma tabela aninhada ou em um varray não inicializados ou uma tentativa de atribuir valores aos elementos de uma tabela aninhada ou em um varray não inicializados. (Você vai aprender sobre coleções no Capítulo 13.)
CURSOR_ALREADY_OPEN	ORA-06511	Foi feita uma tentativa de abrir um cursor já aberto. O cursor deve ser fechado antes que possa ser reaberto.
DUP_VAL_ON_INDEX	ORA-00001	Foi feita uma tentativa de armazenar valores duplicados em uma coluna restrita por um índice exclusivo.
INVALID_CURSOR	ORA-01001	Foi feita uma tentativa de executar uma operação de cursor inválida, como fechar um cursor não aberto.
INVALID_NUMBER	ORA-01722	Uma tentativa de converter uma string de caracteres em um número falhou porque a string não representa um número válido. Nota: nas instruções PL/SQL, é lançado VALUE_ERROR, em vez de INVALID_NUMBER.
LOGIN_DENIED	ORA-01017	Foi feita uma tentativa de conectar um banco de dados usando um nome de usuário ou uma senha inválida.
NO_DATA_FOUND	ORA-01403	Uma instrução SELECT INTO não retornou linhas ou foi feita uma tentativa de acessar um elemento excluído em uma tabela aninhada ou um elemento não inicializado em uma tabela de "index by".
NOT_LOGGED_ON	ORA-01012	Foi feita uma tentativa de acessar um item de banco de dados sem estar conectado no banco de dados.
PROGRAM_ERROR	ORA-06501	O PL/SQL teve um problema interno.
ROWTYPE_MISMATCH	ORA-06504	A variável de cursor do host e a variável de cursor da PL/SQL envolvidas em uma atribuição têm tipos de retorno incompatíveis. Por exemplo, quando uma variável de cursor de host aberto é passada para uma procedure armazenada ou para uma função, os tipos de retorno dos parâmetros reais e formais devem ser compatíveis.

(continua)

Tabela 11-1 Exceções predefinidas (continuação)

Exceção	Erro	Descrição
SELF_IS_NULL	ORA-30625	Foi feita uma tentativa de chamar um método MEMBER em um objeto nulo. Isto é, o parâmetro interno SELF (que é sempre o primeiro parâmetro passado para um método MEMBER) é nulo.
STORAGE_ERROR	ORA-06500	O módulo PL/SQL ficou sem memória ou a memória se corrompeu.
SUBSCRIPT_BEYOND_COUNT	ORA-06533	Foi feita uma tentativa de referenciar um elemento de tabela aninhada ou varray usando um número de índice maior do que o número de elementos da coleção.
SUBSCRIPT_OUTSIDE_LIMIT	ORA-06532	Foi feita uma tentativa de referenciar um elemento de tabela aninhada ou varray usando um número de índice que está fora do intervalo válido (-1, por exemplo).
SYS_INVALID_ROWID	ORA-01410	A conversão de uma string de caracteres em um rowid universal falhou porque a string não representa um rowid válido.
TIMEOUT_ON_RESOURCE	ORA-00051	O tempo limite foi atingido enquanto o banco de dados estava esperando um recurso.
TOO_MANY_ROWS	ORA-01422	Uma instrução SELECT INTO retornou mais de uma linha.
VALUE_ERROR	ORA-06502	Ocorreu um erro de aritmética, conversão, truncamento ou restrição de tamanho. Por exemplo, ao se selecionar um valor de coluna em uma variável de caractere, se o valor é maior do que o comprimento declarado da variável, o PL/SQL cancela a atribuição e lança VALUE_ERROR. Nota: nas instruções PL/SQL, VALUE_ERROR é lançado se a conversão de uma string de caracteres em um número falha. Nas instruções SQL, é lançado INVALID_NUMBER, em vez de VALUE_ERROR.
ZERO_DIVIDE	ORA-01476	Foi feita uma tentativa de dividir um número por zero.

As seções a seguir mostram exemplos que lançam algumas das exceções mostradas na Tabela 11-1.

Exceção ZERO\_DIVIDE

A exceção ZERO\_DIVIDE é lançada quando é feita uma tentativa de dividir um número por zero. O exemplo a seguir tenta dividir 1 por 0 no bloco BEGIN e, portanto, lança a exceção ZERO\_DIVIDE:

```
BEGIN
  DBMS_OUTPUT.PUT_LINE(1 / 0);
EXCEPTION
  WHEN ZERO_DIVIDE THEN
```

```
DBMS_OUTPUT.PUT_LINE('Division by zero');
END;
/
```

Division by zero

Quando uma exceção é lançada, o controle do programa passa para o bloco `EXCEPTION` e a cláusula `WHEN` é examinada para encontrar uma exceção correspondente; então, o código dentro da cláusula correspondente é executado. No exemplo anterior, a exceção `ZERO_DIVIDE` é lançada no bloco `BEGIN` e o controle do programa passa então para o bloco `EXCEPTION`; uma exceção correspondente é encontrada na cláusula `WHEN` e o código de dentro da cláusula é executado. Se nenhuma exceção correspondente é encontrada, a exceção é propagada para o bloco externo. Por exemplo, se o bloco `EXCEPTION` fosse omitido do código anterior, a exceção seria propagada até o `SQL*Plus`:

```
BEGIN
  DBMS_OUTPUT.PUT_LINE(1 / 0);
END;
BEGIN
  *
ERROR at line 1:
ORA-01476: divisor is equal to zero
ORA-06512: at line 2
```

Como você pode ver, o `SQL*Plus` exibe um erro padrão que mostra os números de linha, os códigos de erro do Oracle e uma descrição simples.

## Exceção DUP\_VAL\_ON\_INDEX

A exceção `DUP_VAL_ON_INDEX` é lançada quando é feita uma tentativa de armazenar valores duplicados em uma coluna restrita por um índice exclusivo. O exemplo a seguir tenta inserir uma linha na tabela `customers` com um valor de `customer_id` igual a 1; isso faz `DUP_VAL_ON_INDEX` ser lançada, pois a tabela `customers` já contém uma linha com um valor de `customer_id` igual a 1:

```
BEGIN
  INSERT INTO customers (
    customer_id, first_name, last_name
  ) VALUES (
    1, 'Greg', 'Green'
  );
EXCEPTION
  WHEN DUP_VAL_ON_INDEX THEN
    DBMS_OUTPUT.PUT_LINE('Duplicate value on an index');
END;
/
```

Duplicate value on an index

## Exceção INVALID\_NUMBER

A exceção `INVALID_NUMBER` é lançada quando é feita uma tentativa de converter uma string de caracteres inválida em um número. O exemplo a seguir tenta converter a string `123X` em um número

que é usado em uma instrução `INSERT`, o que faz `INVALID_NUMBER` ser lançada, pois `123X` não é um número válido:

```
BEGIN
    INSERT INTO customers (
        customer_id, first_name, last_name
    ) VALUES (
        '123X', 'Greg', 'Green'
    );
EXCEPTION
    WHEN INVALID_NUMBER THEN
        DBMS_OUTPUT.PUT_LINE('Conversion of string to number failed');
END;
/

Conversion of string to number failed
```

## Exceção OTHERS

Você pode usar a exceção `OTHERS` para tratar de todas as exceções, como mostrado:

```
BEGIN
    DBMS_OUTPUT.PUT_LINE(1 / 0);
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('An exception occurred');
END;
/

An exception occurred
```

Como `OTHERS` corresponde a todas as exceções, você deve listá-la depois de todas as exceções específicas em seu bloco `EXCEPTION`. Se você tentar listar `OTHERS` em outro lugar qualquer, o banco de dados retornará o erro `PLS-00370`; por exemplo:

```
SQL> BEGIN
2     DBMS_OUTPUT.PUT_LINE(1 / 0);
3 EXCEPTION
4     WHEN OTHERS THEN
5         DBMS_OUTPUT.PUT_LINE('An exception occurred');
6     WHEN ZERO_DIVIDE THEN
7         DBMS_OUTPUT.PUT_LINE('Division by zero');
8 END;
9 /
    WHEN OTHERS THEN
    *
ERROR at line 4:
ORA-06550: line 4, column 3:
PLS-00370: OTHERS handler must be last among the exception
    handlers of a block
ORA-06550: line 0, column 0:
PL/SQL: Compilation unit analysis terminated
```



## PROCEDURES

Uma procedure contém um grupo de instruções SQL e PL/SQL. As procedures permitem centralizar sua lógica do negócio no banco de dados e podem ser usadas por qualquer programa que acesse o banco de dados. Nesta seção, você vai aprender a:

- Criar uma procedure
- Chamar uma procedure
- Obter informações sobre procedures
- Excluir uma procedure
- Ver erros em uma procedure

### Criando uma procedure

Para criar uma procedure, use a instrução `CREATE PROCEDURE`. A sintaxe simplificada da instrução `CREATE PROCEDURE` é:

```
CREATE [OR REPLACE] PROCEDURE nome_procedure
[(nome_parâmetro [IN | OUT | IN OUT] tipo [,...])]
{IS | AS}
BEGIN
    corpo_procedure
END nome_procedure;
```

onde

- `OR REPLACE` significa que a procedure deve substituir uma procedure existente.
- *nome\_procedure* é o nome da procedure.
- *nome\_parâmetro* é o nome de um parâmetro passado para a procedure. Você pode passar vários parâmetros para uma procedure.
- `IN | OUT | IN OUT` é o *modo* do parâmetro. Você pode escolher um dos seguintes modos para cada parâmetro:
  - `IN`, que é o modo padrão para um parâmetro. Um parâmetro `IN` deve ser configurado com um valor quando a procedure é executada. O valor de um parâmetro `IN` não pode ser alterado no corpo da procedure.
  - `OUT`, que significa que o parâmetro é configurado com um valor no corpo da procedure.
  - `IN OUT`, que significa que o parâmetro pode ter um valor quando a procedure é executada e o valor pode ser alterado no corpo.
- *tipo* é o tipo do parâmetro.
- *corpo\_procedure* contém o código da procedure.

O exemplo a seguir cria uma procedure chamada `update_product_price()` — essa procedure (e o outro código PL/SQL mostrado no restante deste capítulo) foi criada quando você

executou o script `store_schema.sql`. A procedure `update_product_price()` multiplica o preço de um produto por um fator; a identificação do produto e o fator são passados como parâmetros para a procedure. Se o produto existe, a procedure multiplica seu preço pelo fator e confirma a alteração.

```
CREATE PROCEDURE update_product_price(
    p_product_id IN products.product_id%TYPE,
    p_factor IN NUMBER
) AS
    v_product_count INTEGER;
BEGIN
    -- conta o número de produtos com o valor de
    -- product_id fornecido (será 1 se o produto existe)
    SELECT COUNT(*)
    INTO v_product_count
    FROM products
    WHERE product_id = p_product_id;

    -- se o produto existe (v_product_count = 1) então
    -- atualiza seu preço
    IF v_product_count = 1 THEN
        UPDATE products
        SET price = price * p_factor
        WHERE product_id = p_product_id;
        COMMIT;
    END IF;
EXCEPTION
    WHEN OTHERS THEN
        ROLLBACK;
END update_product_price;
/
```

A procedure aceita dois parâmetros, chamados `p_product_id` e `p_factor` (colocaremos `p_` no início de nomes de parâmetro). Esses dois parâmetros usam o modo `IN`, o que significa que seus valores devem ser configurados quando a procedure for executada e que os valores de parâmetro não podem ser alterados no corpo da procedure.

A seção de declaração contém uma variável `INTEGER` chamada `v_product_count`:

```
v_product_count INTEGER;
```

O corpo da procedure começa depois de `BEGIN`. A instrução `SELECT` no corpo obtém o número de linhas da tabela `products` cujo valor de `product_id` é igual a `p_product_id`:

```
SELECT COUNT(*)
INTO v_product_count
FROM products
WHERE product_id = p_product_id;
```

#### NOTA

`COUNT(*)` retorna o número de linhas encontradas.

Se o produto for encontrado, `v_product_count` será configurado como 1; caso contrário, `v_product_count` será configurado como 0. Se `v_product_count` é 1, a coluna `price` é multiplicada por `p_factor` usando a instrução `UPDATE` e a alteração é confirmada:

```
IF v_product_count = 1 THEN
    UPDATE products
    SET price = price * p_factor
    WHERE product_id = p_product_id;
    COMMIT;
END IF;
```

O bloco `EXCEPTION` executa uma instrução `ROLLBACK` se uma exceção é lançada:

```
EXCEPTION
    WHEN OTHERS THEN
        ROLLBACK;
```

Por fim, a palavra-chave `END` é usada para marcar o fim da procedure:

```
END update_product_price;
/
```

#### NOTA

*A repetição do nome da procedure depois da palavra-chave `END` não é obrigatória, mas é considerada uma boa prática de programação colocá-lo.*

## Chamando uma procedure

Você executa (ou *chama*) uma procedure usando a instrução `CALL`. O exemplo que você verá nesta seção multiplicará o preço do produto nº 1 por 1,5 usando a procedure mostrada na seção anterior. Primeiramente, a consulta a seguir recupera o preço do produto nº 1 para que você possa compará-lo com o preço modificado posteriormente:

```
SELECT price
FROM products
WHERE product_id = 1;

PRICE
-----
19.95
```

A instrução a seguir chama `update_product_price()`, passando os valores de parâmetro 1 (o `product_id`) e 1,5 (o fator pelo qual o preço do produto é multiplicado):

```
CALL update_product_price(1, 1.5);
```

Call completed.

Essa instrução mostra o uso da *notação posicional* para indicar os valores a serem passados para a procedure ou função. Na notação posicional, a posição dos parâmetros é utilizada para designar os valores passados para a procedure. No exemplo, o primeiro valor na chamada é 1 e isso é passado para o primeiro parâmetro na procedure (`p_product_id`); o segundo valor na chamada é 1.5 e isso é passado para o segundo parâmetro (`p_factor`). No Oracle Database 11g, além da notação posicional, você também pode usar notação nomeada e mista. Esses tipos de notação serão apresentados em breve.

A próxima consulta recupera os detalhes do produto nº 1 novamente; observe que o preço foi multiplicado por 1,5:

```
SELECT price
FROM products
WHERE product_id = 1;

PRICE
-----
29.93
```

No Oracle Database 11g, você pode passar parâmetros usando notação nomeada e mista. Na *notação nomeada*, você inclui o nome do parâmetro ao chamar uma procedure. Por exemplo, a instrução a seguir chama `update_product_price()` usando notação nomeada; observe que os valores dos parâmetros `p_factor` e `p_product_id` são indicados usando `=>`:

```
CALL update_product_price(p_factor => 1.3, p_product_id => 2);
```

**DICA**

*A notação nomeada torna seu código mais fácil de ler e manter, pois os parâmetros são mostrados explicitamente.*

Na *notação mista*, você usa notação posicional e nomeada; a notação posicional é usada para o primeiro conjunto de parâmetros e a notação nomeada para o último conjunto de parâmetros. A notação mista é útil quando você tem procedures e funções que têm parâmetros obrigatórios e opcionais; você usa notação posicional para os parâmetros obrigatórios e notação nomeada para os parâmetros opcionais. O exemplo a seguir usa notação mista; observe que a notação posicional vem antes da notação nomeada na especificação dos valores de parâmetro:

```
CALL update_product_price(3, p_factor => 1.7);
```

**Obtendo informações sobre procedures**

Você pode obter informações sobre suas procedures a partir da visão `user_procedures`. A Tabela 11-2 descreve algumas das colunas de `user_procedures`.

**Tabela 11-2** Algumas colunas da visão `user_procedures`

Coluna	Tipo	Descrição
OBJECT_NAME	VARCHAR2 (30)	O nome do objeto, que pode ser um nome de procedure, função ou pacote
PROCEDURE_NAME	VARCHAR2 (30)	O nome da procedure
AGGREGATE	VARCHAR2 (3)	Se a procedure é uma função agregada (YES ou NO)
IMPLTYPEOWNER	VARCHAR2 (30)	O proprietário do tipo (se houver)
IMPLTYPENAME	VARCHAR2 (30)	O nome do tipo (se houver)
PARALLEL	VARCHAR2 (3)	Se a procedure é ativada para consultas paralelas (YES ou NO)



**NOTA**

*Você pode obter informações sobre todas as procedures a que tem acesso usando all\_procedures.*

O exemplo a seguir recupera as colunas object\_name, aggregate e parallel de user\_procedures para update\_product\_price():

```
SELECT object_name, aggregate, parallel
FROM user_procedures
WHERE object_name = 'UPDATE_PRODUCT_PRICE';
```

OBJECT_NAME	AGG	PAR
-----	----	----
UPDATE_PRODUCT_PRICE	NO	NO

**Excluindo uma procedure**

Você exclui uma procedure usando DROP PROCEDURE. Por exemplo, a instrução a seguir exclui update\_product\_price():

```
DROP PROCEDURE update_product_price;
```

**Vendo erros em uma procedure**

Se o banco de dados relata um erro quando uma procedure é criada, você pode ver os erros usando o comando SHOW ERRORS. Por exemplo, a instrução CREATE PROCEDURE a seguir tenta criar uma procedure que tem um erro de sintaxe na linha 6 (o parâmetro deveria ser p\_dob e não p\_dobs):

```
SQL> CREATE PROCEDURE update_customer_dob (
2   p_customer_id INTEGER, p_dob DATE
3 ) AS
4 BEGIN
5   UPDATE customers
6   SET dob = p_dobs
7   WHERE customer_id = p_customer_id;
8 END update_customer_dob;
9 /
```

Warning: Procedure created with compilation errors.

Como você pode ver, há um erro de compilação. Para ver os erros, você usa SHOW ERRORS:

```
SQL> SHOW ERRORS
Errors for PROCEDURE UPDATE_CUSTOMER_DOB:

LINE/COL ERROR
-----
5/3 PL/SQL: SQL Statement ignored
6/13 PL/SQL: ORA-00904: invalid column name
```

A linha 5 foi ignorada porque um nome de coluna inválido foi referenciado na linha 6. Você pode corrigir o erro executando um comando EDIT para editar a instrução CREATE PROCEDURE, alterando p\_dobs para p\_dob e executando a instrução novamente, digitando /.

## FUNÇÕES

Uma *função* é semelhante a uma *procedure*, exceto que uma função deve retornar um valor. Juntas, as *procedures* armazenadas e as funções às vezes são referidas como *subprogramas armazenados*, pois são pequenos programas.

Nesta seção, você vai aprender a:

- Criar uma função
- Chamar uma função
- Obter informações sobre funções
- Excluir uma função

### Criando uma função

Você cria uma função usando a instrução `CREATE FUNCTION`. A sintaxe simplificada da instrução `CREATE FUNCTION` é:

```
CREATE [OR REPLACE] FUNCTION nome_função
[(nome_parâmetro [IN | OUT | IN OUT] tipo [...])]
RETURN tipo
{IS | AS}
BEGIN
    corpo_função
END nome_função;
```

onde

- `OR REPLACE` significa que a *procedure* deve substituir uma função existente.
- *nome\_função* é o nome da função.
- *nome\_parâmetro* é o nome de um parâmetro passado para a função. Você pode passar vários parâmetros para uma função.
- `IN | OUT | IN OUT` é o modo do parâmetro.
- *tipo* é o tipo do parâmetro.
- *corpo\_função* contém o código da função. Ao contrário de uma *procedure*, o corpo de uma função deve retornar um valor do tipo especificado na cláusula `RETURN`.

O exemplo a seguir cria uma função chamada `circle_area()`, a qual retorna a área de um círculo. O raio do círculo é passado como um parâmetro chamado `p_radius` para `circle_area()`; observe que `circle_area()` retorna um valor `NUMBER`:

```
CREATE FUNCTION circle_area (
    p_radius IN NUMBER
) RETURN NUMBER AS
    v_pi NUMBER := 3.1415926;
    v_area NUMBER;
BEGIN
    -- a área do círculo é pi multiplicado pelo raio ao quadrado
```

```

    v_area := v_pi * POWER(p_radius, 2);
    RETURN v_area;
END circle_area;
/

```

O próximo exemplo cria uma função chamada `average_product_price()`, a qual retorna o preço médio dos produtos cujo valor de `product_type_id` é igual ao valor do parâmetro:

```

CREATE FUNCTION average_product_price (
    p_product_type_id IN INTEGER
) RETURN NUMBER AS
    v_average_product_price NUMBER;
BEGIN
    SELECT AVG(price)
    INTO v_average_product_price
    FROM products
    WHERE product_type_id = p_product_type_id;
    RETURN v_average_product_price;
END average_product_price;
/

```

## Chamando uma função

Você chama suas próprias funções como chamaria qualquer uma das funções internas do banco de dados; você viu como chamar funções internas no Capítulo 4. (Apenas para refrescar sua memória, é possível chamar uma função usando uma instrução `SELECT` que usa a tabela `dual` na cláusula `FROM`.) O exemplo a seguir chama `circle_area()`, passando um raio de 2 metros para a função, usando notação posicional:

```

SELECT circle_area(2)
FROM dual;

CIRCLE_AREA(2)
-----
      12.5663704

```

No Oracle Database 11g, você também pode usar notação nomeada e mista ao chamar funções. Por exemplo, a consulta a seguir usa notação nomeada ao chamar `circle_area()`:

```

SELECT circle_area(p_radius => 4)
FROM dual;

CIRCLE_AREA(P_RADIUS=>4)
-----
      50.2654816

```

O exemplo a seguir chama `average_product_price()`, passando o valor de parâmetro 1 para a função, para obter o preço médio dos produtos cujo valor de `product_type_id` é 1:

```

SELECT average_product_price(1)
FROM dual;

AVERAGE_PRODUCT_PRICE(1)
-----
      29.965

```

## Obtendo informações sobre funções

Você pode obter informações sobre suas funções a partir da visão `user_procedures`; esta visão foi abordada anteriormente, na seção “Obtendo informações sobre procedures”. O exemplo a seguir recupera as colunas `object_name`, `aggregate` e `parallel` de `user_procedures` para as funções `circle_area()` e `average_product_price()`:

```
SELECT object_name, aggregate, parallel
FROM user_procedures
WHERE object_name IN ('CIRCLE_AREA', 'AVERAGE_PRODUCT_PRICE');
```

OBJECT_NAME	AGG	PAR
AVERAGE_PRODUCT_PRICE	NO	NO
CIRCLE_AREA	NO	NO

## Excluindo uma função

Você exclui uma função usando `DROP FUNCTION`. Por exemplo, a instrução a seguir exclui `circle_area()`:

```
DROP FUNCTION circle_area;
```

## PACOTES (PACKAGES)

Nesta seção, você vai aprender a agrupar procedures e funções em *pacotes*. Os pacotes permitem encapsular funcionalidade relacionada em uma unidade independente. Modularizando seu código PL/SQL por meio de pacotes, é possível construir suas próprias bibliotecas de código que outros programadores podem reutilizar. O banco de dados Oracle vem com uma biblioteca de pacotes pré-configurados, os quais permitem acessar arquivos externos, gerenciar o banco de dados, gerar código HTML e muito mais; para ver todos os pacotes, você deve consultar o manual *Oracle Database PL/SQL Packages and Types Reference* da Oracle Corporation.

Normalmente, os pacotes são constituídos de dois componentes: uma *especificação* e um *corpo*. A especificação do pacote lista as procedures, funções, tipos e objetos disponíveis. Você pode tornar os itens listados na especificação disponíveis para todos os usuários do banco de dados e nos referimos a esses itens como sendo *públicos* (embora somente os usuários a quem você tenha concedido privilégios para acessar seu pacote possam utilizá-lo). A especificação não contém o código que constitui as procedures e funções; o código está contido no corpo do pacote.

Todos os itens do corpo que não estão listados na especificação são *privados* do pacote. Os itens privados só podem ser usados dentro do corpo do pacote. Usando uma combinação de itens público e privados, é possível construir um pacote cuja complexidade fica oculta do mundo exterior. Esse é um dos principais objetivos de toda programação: ocultar a complexidade de seus usuários.

## Criando uma especificação de pacote

Você cria uma especificação de pacote usando a instrução `CREATE PACKAGE`. A sintaxe simplificada da instrução `CREATE PACKAGE` é:

```
CREATE [OR REPLACE] PACKAGE nome_pacote
{IS | AS}
    especificação_pacote
END nome_pacote;
```



onde

- *nome\_pacote* é o nome do pacote.
- *especificação\_pacote* lista as procedures, funções, tipos e objetos públicos disponíveis para os usuários de seu pacote.

O exemplo a seguir cria uma especificação para um pacote chamado `product_package`:

```
CREATE PACKAGE product_package AS
  TYPE t_ref_cursor IS REF CURSOR;
  FUNCTION get_products_ref_cursor RETURN t_ref_cursor;
  PROCEDURE update_product_price (
    p_product_id IN products.product_id%TYPE,
    p_factor IN NUMBER
  );
END product_package;
/
```

O tipo `t_ref_cursor` é um tipo `REF CURSOR` da PL/SQL. Um `REF CURSOR` é semelhante a um ponteiro na linguagem de programação C++ e aponta para um cursor; conforme foi visto anteriormente, um cursor permite ler as linhas retornadas por uma consulta. A função `get_products_ref_cursor()` retorna um tipo `t_ref_cursor` e, conforme você verá na próxima seção, ele aponta para um cursor que contém as linhas recuperadas da tabela `products`. A procedure `update_product_price()` multiplica o preço de um produto e confirma a alteração.

## Criando o corpo de um pacote

Você cria um corpo de pacote usando a instrução `CREATE PACKAGE BODY`. A sintaxe simplificada da instrução `CREATE PACKAGE BODY` é:

```
CREATE [OR REPLACE] PACKAGE BODY nome_pacote
{IS | AS}
  corpo_pacote
END nome_pacote;
```

onde

- *nome\_pacote* é o nome do pacote, o qual deve corresponder ao nome do pacote na especificação.
- *corpo\_pacote* contém o código das procedures e funções.

O exemplo a seguir cria o corpo do pacote `product_package`:

```
CREATE PACKAGE BODY product_package AS
  FUNCTION get_products_ref_cursor
  RETURN t_ref_cursor IS
    v_products_ref_cursor t_ref_cursor;
  BEGIN
    -- obtém o REF CURSOR
    OPEN v_products_ref_cursor FOR
      SELECT product_id, name, price
      FROM products;
    -- retorna o REF CURSOR
```

```

    RETURN v_products_ref_cursor;
END get_products_ref_cursor;

PROCEDURE update_product_price (
    p_product_id IN products.product_id%TYPE,
    p_factor IN NUMBER
) AS
    v_product_count INTEGER;
BEGIN
    -- conta o número de produtos com o valor de
    -- product_id fornecido (será 1 se o produto existe)
    SELECT COUNT(*)
    INTO v_product_count
    FROM products
    WHERE product_id = p_product_id;

    -- se o produto existe (v_product_count = 1) então
    -- atualiza seu preço
    IF v_product_count = 1 THEN
        UPDATE products
        SET price = price * p_factor
        WHERE product_id = p_product_id;
        COMMIT;
    END IF;
EXCEPTION
    WHEN OTHERS THEN
        ROLLBACK;
END update_product_price;
END product_package;
/

```

A função `get_products_ref_cursor()` abre o cursor e recupera as colunas `product_id`, `name` e `price` da tabela `products`. A referência a esse cursor (o `REF CURSOR`) é armazenada em `v_products_ref_cursor` e retornada pela função.

A procedure `update_product_price()` multiplica o preço de um produto e confirma a alteração. Essa procedure é idêntica àquela mostrada anteriormente na seção “Criando uma procedure”; portanto, não discutiremos novamente como ela funciona.

## Chamando funções e procedures em um pacote

Ao chamar funções e procedures em um pacote, você deve incluir o nome do pacote na chamada. O exemplo a seguir chama `product_package.get_products_ref_cursor()`, que retorna uma referência para um cursor contendo os valores de `product_id`, `name` e `price` dos produtos:

```

SELECT product_package.get_products_ref_cursor
FROM dual;

GET_PRODUCTS_REF_CUR
-----
CURSOR STATEMENT : 1

CURSOR STATEMENT : 1

```

PRODUCT_ID	NAME	PRICE
1	Modern Science	19.95
2	Chemistry	30
3	Supernova	25.99
4	Tank War	13.95
5	Z Files	49.99
6	2412: The Return	14.95
7	Space Force 9	13.49
8	From Another Planet	12.99
9	Classical Music	10.99
10	Pop 3	15.99
11	Creative Yell	14.99
12	My Front Line	13.49

O próximo exemplo chama `product_package.update_product_price()` para multiplicar o preço do produto nº 3 por 1.25:

```
CALL product_package.update_product_price(3, 1.25);
```

A próxima consulta recupera os detalhes do produto nº 3; observe que o preço aumentou:

```
SELECT price
FROM products
WHERE product_id = 3;
```

```
PRICE
-----
32.49
```

## Obtendo informações sobre funções e procedures em um pacote

Você pode obter informações sobre suas funções e procedures em um pacote a partir da visão `user_procedures`; esta visão foi abordada na seção “Obtendo informações sobre procedures”. O exemplo a seguir recupera as colunas `object_name` e `procedure_name` de `user_procedures` para `product_package`:

```
SELECT object_name, procedure_name
FROM user_procedures
WHERE object_name = 'PRODUCT_PACKAGE';
```

OBJECT_NAME	PROCEDURE_NAME
PRODUCT_PACKAGE	GET_PRODUCTS_REF_CURSOR
PRODUCT_PACKAGE	UPDATE_PRODUCT_PRICE

## Excluindo um pacote

Você exclui um pacote usando `DROP PACKAGE`. Por exemplo, a instrução a seguir exclui `product_package`:

```
DROP PACKAGE product_package;
```

## TRIGGERS

Um *trigger* é uma procedure executada (ou *disparada*) automaticamente pelo banco de dados, quando uma instrução DML (INSERT, UPDATE ou DELETE) especificada é executada em determinada tabela do banco de dados. Os triggers são úteis para fazer coisas como a auditoria avançada das alterações feitas nos valores de coluna em uma tabela.

### Quando um trigger é disparado

Um trigger pode disparar antes ou depois de uma instrução DML ser executada. Além disso, como uma instrução DML pode afetar mais de uma linha, o código do trigger pode ser executado uma vez para cada linha afetada (um *trigger em nível de linha*) ou apenas uma vez para todas as linhas (um *trigger em nível de instrução*). Por exemplo, se você criasse um trigger em nível de linha que fosse disparado para uma instrução UPDATE em uma tabela e executasse uma instrução UPDATE que modificasse dez linhas dessa tabela, esse trigger seria executado dez vezes. Se, entretanto, seu trigger fosse um trigger em nível de instrução, ele seria disparado uma vez para a instrução UPDATE inteira, independentemente do número de linhas afetadas.

Há outra diferença entre um trigger em nível de linha e um trigger em nível de instrução: um trigger em nível de linha tem acesso aos valores de coluna antigos e novos quando é disparado como resultado de uma instrução UPDATE nessa coluna. O disparo de um trigger em nível de linha também pode ser limitado com uma *condição* de trigger; por exemplo, você poderia definir uma condição que limitasse o trigger a disparar somente quando um valor de coluna fosse menor do que um valor especificado.

### Configuração do trigger de exemplo

Conforme foi mencionado, os triggers são úteis para fazer a auditoria avançada das alterações feitas em valores de coluna. Na próxima seção, você verá um trigger que registra quando o preço de um produto é diminuído em mais de 25%; quando isso ocorrer, o trigger adicionará uma linha na tabela `product_price_audit`. A tabela `product_price_audit` é criada pela instrução a seguir no script `store_schema.sql`:

```
CREATE TABLE product_price_audit (
    product_id INTEGER
    CONSTRAINT price_audit_fk_products
    REFERENCES products(product_id),
    old_price NUMBER(5, 2),
    new_price NUMBER(5, 2)
);
```

Como você pode ver, a coluna `product_id` da tabela `product_price_audit` é uma chave estrangeira para a coluna `product_id` da tabela `products`. A coluna `old_price` será usada para armazenar o preço antigo de um produto antes da alteração e a coluna `new_price` será usada para armazenar o novo preço, após a alteração.

### Criando um trigger

A instrução `CREATE TRIGGER` cria um trigger. A sintaxe simplificada da instrução `CREATE TRIGGER` é:

```
CREATE [OR REPLACE] TRIGGER nome_trigger
{BEFORE | AFTER | INSTEAD OF | FOR} evento_trigger
ON nome_tabela
[FOR EACH ROW]
[{FORWARD | REVERSE} CROSSSESSION]
```

```

[{FOLLOWS | PRECEDES} esquema.outro_trigger}
[{ENABLE | DISABLE}]
[WHEN condição_trigger]
BEGIN
    corpo_trigger
END nome_trigger;

```

onde

- **OR REPLACE** significa que o trigger deve substituir um trigger existente, se estiver presente.
- *nome\_trigger* é o nome do trigger.
- **BEFORE** significa que o trigger é disparado antes que o evento de disparo seja executado. **AFTER** significa que o trigger é disparado depois que o evento de disparo é executado. **INSTEAD OF** significa que o trigger é disparado em vez do evento de disparo. **FOR**, que é novidade do Oracle Database 11g, permite criar um trigger composto, consistindo em até quatro seções no corpo do trigger.
- *evento\_trigger* é o evento que faz o trigger disparar.
- *nome\_tabela* é a tabela a que o trigger faz referência.
- **FOR EACH ROW** significa que o trigger é um trigger em nível de linha; ou seja, o código contido dentro de *corpo\_trigger* é executado para cada linha quando o trigger dispara. Se você omitir **FOR EACH ROW**, o trigger será um trigger em nível de instrução, o que significa que o código dentro de *corpo\_trigger* será executado uma vez quando o trigger disparar.
- **{FORWARD | REVERSE} CROSS EDITION** é novo no Oracle Database 11g e normalmente será usado pelos administradores de banco de dados ou administradores de aplicativo. Um trigger cross edition **FORWARD** se destina a ser disparado quando uma instrução DML faz uma alteração no banco de dados enquanto um aplicativo online que está acessando o banco de dados no momento *está recebendo um patch ou está sendo atualizado* (**FORWARD** é o padrão); o código no corpo do trigger deve ser projetado para tratar as alterações DML quando o patch ou a atualização do aplicativo tiver terminado. Um trigger **CROSS EDITION REVERSE** é semelhante, exceto que se destina a ser disparado e a tratar de alterações DML feitas *depois que o aplicativo online tiver recebido o patch ou tiver sido atualizado*.
- **{FOLLOWS | PRECEDES} esquema.outro\_trigger** é novo no Oracle Database 11g e especifica se o disparo do trigger vem depois ou antes do disparo de outro trigger especificado em *esquema.outro\_trigger*. Você pode criar uma série de triggers que disparam em uma ordem específica.
- **{ENABLE | DISABLE}** é novo no Oracle Database 11g e indica se o trigger é inicialmente ativado ou desativado ao ser criado (o padrão é **ENABLE**). Você ativa um trigger desativado usando a instrução **ALTER TRIGGER nome\_trigger ENABLE** ou ativando todos os triggers de uma tabela, utilizando **ALTER TABLE nome\_tabela ENABLE ALL TRIGGERS**.
- *condição\_trigger* é uma condição booleana que limita quando um trigger realmente executa seu código.
- *corpo\_trigger* contém o código do trigger.

O exemplo de trigger que você verá nesta seção dispara antes da atualização da coluna `price` na tabela `products`; portanto, chamaremos o trigger de `before_product_price_update`. Além disso, como queremos usar os valores da coluna `price` antes e depois que uma instrução `UPDATE` os modifique, devemos utilizar um trigger em nível de linha. Por fim, queremos fazer uma auditoria da alteração de preço quando o novo preço for diminuído em mais de 25% em relação ao preço antigo, portanto, precisaremos especificar uma condição de trigger para comparar o preço novo com o antigo. A instrução a seguir cria o trigger `before_product_price_update`:

```
CREATE TRIGGER before_product_price_update
BEFORE UPDATE OF price
ON products
FOR EACH ROW WHEN (new.price < old.price * 0.75)
BEGIN
    dbms_output.put_line('product_id = ' || :old.product_id);
    dbms_output.put_line('Old price = ' || :old.price);
    dbms_output.put_line('New price = ' || :new.price);
    dbms_output.put_line('The price reduction is more than 25%');

    -- insere linha na tabela product_price_audit
    INSERT INTO product_price_audit (
        product_id, old_price, new_price
    ) VALUES (
        :old.product_id, :old.price, :new.price
    );
END before_product_price_update;
/
```

Existem cinco detalhes que você deve observar a respeito dessa instrução:

- `BEFORE UPDATE OF price` significa que o trigger dispara antes de uma atualização da coluna `price`.
- `FOR EACH ROW` significa que esse é um trigger em nível de linha; ou seja, o código do trigger contido dentro das palavras-chave `BEGIN` e `END` é executado uma vez para cada linha modificada pela atualização.
- A condição do trigger é `(new.price < old.price * 0.75)`, o que significa que o trigger dispara somente quando o novo preço é menor do que 75% em relação ao preço antigo (isto é, quando o preço é reduzido em mais de 25%).
- Os valores de coluna novos e antigos são acessados usando-se os apelidos `:old` e `:new` no trigger.
- O código do trigger exibe o valores de `product_id`, os preços novos e antigos e uma mensagem dizendo que a redução de preço é maior do que 25%. Então, o código adiciona uma linha na tabela `product_price_audit`, contendo o valor de `product_id` e os preços antigos e novos.

## Disparando um trigger

Para ver a saída do trigger, você precisa executar o comando `SET SERVEROUTPUT ON`:

```
SET SERVEROUTPUT ON
```

Para disparar o trigger `before_product_price_update`, você precisa reduzir o preço de um produto em mais de 25%. Execute a instrução `UPDATE` a seguir a fim de reduzir o preço dos pro-

utos nº 5 e 10 em 30% (para tanto, a coluna `price` é multiplicada por .7). A instrução `UPDATE` a seguir faz o trigger `before_product_price_update` disparar:

```
UPDATE products
SET price = price *.7
WHERE product_id IN (5, 10);

product_id = 10
Old price = 15.99
New price = 11.19
The price reduction is more than 25%
product_id = 5
Old price = 49.99
New price = 34.99
The price reduction is more than 25%

2 rows updated.
```

Como você pode ver, o trigger disparou para os produtos nº 10 e 5. Você pode ver se o trigger adicionou de fato as duas linhas exigidas, contendo os valores de `product_id` e os preços antigos e novos, na tabela `product_price_audit` usando a consulta a seguir:

```
SELECT *
FROM product_price_audit
ORDER BY product_id;
```

PRODUCT_ID	OLD_PRICE	NEW_PRICE
5	49.99	34.99
10	15.99	11.19

Obtendo informações sobre triggers

Você pode obter informações sobre seus triggers a partir da visão `user_triggers`. A Tabela 11-3 descreve algumas das colunas de `user_triggers`.

Tabela 11-3 Algumas colunas da visão `user_triggers`

Coluna	Tipo	Descrição
TRIGGER_NAME	VARCHAR2 (30)	Nome do trigger.
TRIGGER_TYPE	VARCHAR2 (16)	Tipo do trigger.
TRIGGERING_EVENT	VARCHAR2 (227)	Evento que faz o triggerdisparar.
TABLE_OWNER	VARCHAR2 (30)	Usuário que possui a tabela a que o trigger referencia.
BASE_OBJECT_TYPE	VARCHAR2 (16)	Tipo de objeto referenciado pelo trigger.
TABLE_NAME	VARCHAR2 (30)	Nome da tabela referenciada pelo trigger.
COLUMN_NAME	VARCHAR2 (4000)	Nome da coluna referenciada pelo trigger.

(continua)

Tabela 11-3 Algumas colunas da visão `user_triggers` (Cotinuação)

Coluna	Tipo	Descrição
REFERENCING_NAMES	VARCHAR2 (128)	Nome dos apelidos antigo e novo.
WHEN_CLAUSE	VARCHAR2 (4000)	Condição do trigger que limita quando ele executa seu código.
STATUS	VARCHAR2 (8)	Se o trigger está ativado ou desativado (ENABLED ou DISABLED).
DESCRIPTION	VARCHAR2 (4000)	Descrição do trigger.
ACTION_TYPE	VARCHAR2 (11)	Tipo de ação do trigger (CALL ou PL/SQL).
TRIGGER_BODY	LONG	Código contido no corpo do trigger. (O tipo LONG permite o armazenamento de grandes volumes de texto. Você vai aprender sobre o tipo LONG no Capítulo 14.)

NOTA

Você pode obter informações sobre todos os triggers a que tem acesso usando `all_triggers`.

O exemplo a seguir recupera os detalhes do trigger `before_product_price_update` de `user_triggers` (a saída foi formatada para clareza na leitura):

```
SELECT trigger_name, trigger_type, triggering_event, table_owner
       base_object_type, table_name, referencing_names, when_clause, status,
       description, action_type, trigger_body
FROM user_triggers
WHERE trigger_name = 'BEFORE_PRODUCT_PRICE_UPDATE';

TRIGGER_NAME                                TRIGGER_TYPE
-----
BEFORE_PRODUCT_PRICE_UPDATE                BEFORE EACH ROW

TRIGGERING_EVENT
-----
UPDATE

TABLE_OWNER                                BASE_OBJECT_TYPE TABLE_NAME
-----
STORE                                    Tabela            PRODUCTS

REFERENCING_NAMES
-----
REFERENCING NEW AS NEW OLD AS OLD

WHEN_CLAUSE
-----
new.price < old.price * 0.75
```



```

STATUS
-----
ENABLED

DESCRIPTION
-----
before_product_price_update
BEFORE UPDATE OF
  price
ON
  products
FOR EACH ROW

ACTION_TYPE
-----
PL/SQL

TRIGGER_BODY
-----
BEGIN
  dbms_output.put_line('product_id = ' || :old.product_id);
  dbms_output...

```

**NOTA**

Você pode ver todo o código do trigger usando o comando `SET LONG` do *SQL\*Plus*, por exemplo, `SET LONG 1000`.

**Desativando e ativando um trigger**

Você pode interromper o disparo de um trigger desativando-o com a instrução `ALTER TRIGGER`. O exemplo a seguir desativa o trigger `before_product_price_update`:

```
ALTER TRIGGER before_product_price_update DISABLE;
```

O exemplo a seguir ativa o trigger `before_product_price_update`:

```
ALTER TRIGGER before_product_price_update ENABLE;
```

**Excluindo um trigger**

Você exclui um trigger usando `DROP TRIGGER`. O exemplo a seguir exclui o trigger `before_product_price_update`:

```
DROP TRIGGER before_product_price_update;
```

**NOVOS RECURSOS PL/SQL NO ORACLE DATABASE 11g**

Nesta seção, você verá alguns recursos novos PL/SQL introduzidos no Oracle Database 11g, especificamente:

- O tipo `SIMPLE_INTEGER`
- Suporte à seqüências em PL/SQL
- Geração de código de máquina nativo PL/SQL

## Tipo SIMPLE\_INTEGER

O tipo `SIMPLE_INTEGER` é um subtipo de `BINARY_INTEGER`; ele pode armazenar o mesmo intervalo que `BINARY_INTEGER`, mas não pode armazenar um valor `NULL`. O intervalo de valores que `SIMPLE_INTEGER` pode armazenar é de  $-2^{31}$  ( $-2.147.483.648$ ) a  $2^{31}$  ( $2.147.483.648$ ).

Ao se usar valores `SIMPLE_INTEGER`, o estouro aritmético é truncado; portanto, os cálculos não lançam um erro ao ocorrer um estouro. Como os erros de estouro são ignorados, os valores armazenados em um tipo `SIMPLE_INTEGER` podem mudar automaticamente de positivos para negativos e de negativos para positivos, como, por exemplo:

$$2^{30} + 2^{30} = 0x40000000 + 0x40000000 = 0x80000000 = -2^{31}$$

$$-2^{31} + -2^{31} = 0x80000000 + 0x80000000 = 0x00000000 = 0$$

No primeiro exemplo, dois valores positivos são somados e é produzido um total negativo. No segundo exemplo, dois valores negativos são somados e zero é produzido.

Como o estouro é ignorado e truncado ao se usar valores `SIMPLE_INTEGER` em cálculos, o tipo `SIMPLE_INTEGER` oferece desempenho muito melhor do que `BINARY_INTEGER` quando o administrador configura o banco de dados para compilar PL/SQL em código de máquina nativo. Por causa dessa vantagem, você deve usar `SIMPLE_INTEGER` em seu código PL/SQL quando não precisar armazenar um valor `NULL` e não se preocupar com truncamento de estouro ocorrendo em seus cálculos; caso contrário, você deverá usar `BINARY_INTEGER`.

A procedure `get_area()` a seguir mostra o uso do tipo `SIMPLE_INTEGER`; `get_area()` calcula e exibe a área de um retângulo:

```
CREATE PROCEDURE get_area
AS
    v_width SIMPLE_INTEGER := 10;
    v_height SIMPLE_INTEGER := 2;
    v_area SIMPLE_INTEGER := v_width * v_height;
BEGIN
    DBMS_OUTPUT.PUT_LINE('v_area = ' || v_area);
END get_area;
/
```

### NOTA

*Você encontrará este e outros exemplos desta seção em um script chamado `plsql_11g_examples.sql` no diretório `SQL`. Você poderá executar esse script se estiver usando o Oracle Database 11g.*

O exemplo a seguir mostra a execução de `get_area()`:

```
SET SERVEROUTPUT ON
CALL get_area();
v_area = 20
```

Conforme o esperado, a área calculada é igual a 20.

## Seqüências em PL/SQL

No capítulo anterior, você viu como criar e utilizar seqüências de números em SQL. No Oracle Database 11g, você também pode usar seqüências em código PL/SQL.

Como lembrete, uma seqüência gera uma série de números. Quando você cria uma seqüência em SQL, pode especificar seu valor inicial e um incremento para a série de números subsequentes.

Você usa a pseudocoluna `currval` para obter o valor atual da sequência e `nextval` para gerar o próximo número. Antes de acessar `currval`, primeiro você precisa utilizar `nextval` para gerar um número inicial. A instrução a seguir cria uma tabela chamada `new_products`; essa tabela será usada em breve:

```
CREATE TABLE new_products (
    product_id INTEGER CONSTRAINT new_products_pk PRIMARY KEY,
    name VARCHAR2(30) NOT NULL,
    price NUMBER(5, 2)
);
```

A próxima instrução cria uma sequência chamada `s_product_id`:

```
CREATE SEQUENCE s_product_id;
```

A instrução a seguir cria uma procedure chamada `add_new_products`, a qual utiliza `s_product_id` para configurar a coluna `product_id` em uma linha adicionada na tabela `new_products`; observe o uso das pseudocolunas `nextval` e `currval` no código PL/SQL (este é um novo recurso do Oracle Database 11g):

```
CREATE PROCEDURE add_new_products
AS
    v_product_id BINARY_INTEGER;
BEGIN
    -- usa nextval para gerar o número inicial da sequência
    v_product_id := s_product_id.nextval;
    DBMS_OUTPUT.PUT_LINE('v_product_id = ' || v_product_id);

    -- adiciona uma linha em new_products
    INSERT INTO new_products
    VALUES (v_product_id, 'Plasma Physics book', 49.95);

    DBMS_OUTPUT.PUT_LINE('s_product_id.currval = ' || s_product_id.currval);

    -- usa nextval para gerar o próximo número da sequência
    v_product_id := s_product_id.nextval;
    DBMS_OUTPUT.PUT_LINE('v_product_id = ' || v_product_id);

    -- adiciona outra linha em new_products
    INSERT INTO new_products
    VALUES (v_product_id, 'Quantum Physics book', 69.95);

    DBMS_OUTPUT.PUT_LINE('s_product_id.currval = ' || s_product_id.currval);
END add_new_products;
/
```

O exemplo a seguir executa `add_new_products()` e mostra o conteúdo da tabela `new_products`:

```
SET SERVEROUTPUT ON
CALL add_new_products();
v_product_id = 1
s_product_id.currval = 1
```

```
v_product_id = 2
s_product_id.currval = 2
```

```
SELECT * FROM new_products;
```

PRODUCT_ID	NAME	PRICE
1	Plasma Physics book	49.95
2	Quantum Physics book	69.95

Conforme o esperado, duas linhas foram adicionadas na tabela.

## Geração de código de máquina nativo PL/SQL

Por padrão, cada unidade de programa PL/SQL é compilada em código legível pela máquina, na forma intermediária. Esse código legível pela máquina é armazenado no banco de dados e interpretado sempre que o código é executado. Com a compilação nativa PL/SQL, o PL/SQL é transformado em código nativo e armazenado em bibliotecas compartilhadas. O código nativo é executado muito mais rapidamente do que o código intermediário, pois não precisa ser interpretado antes da execução.

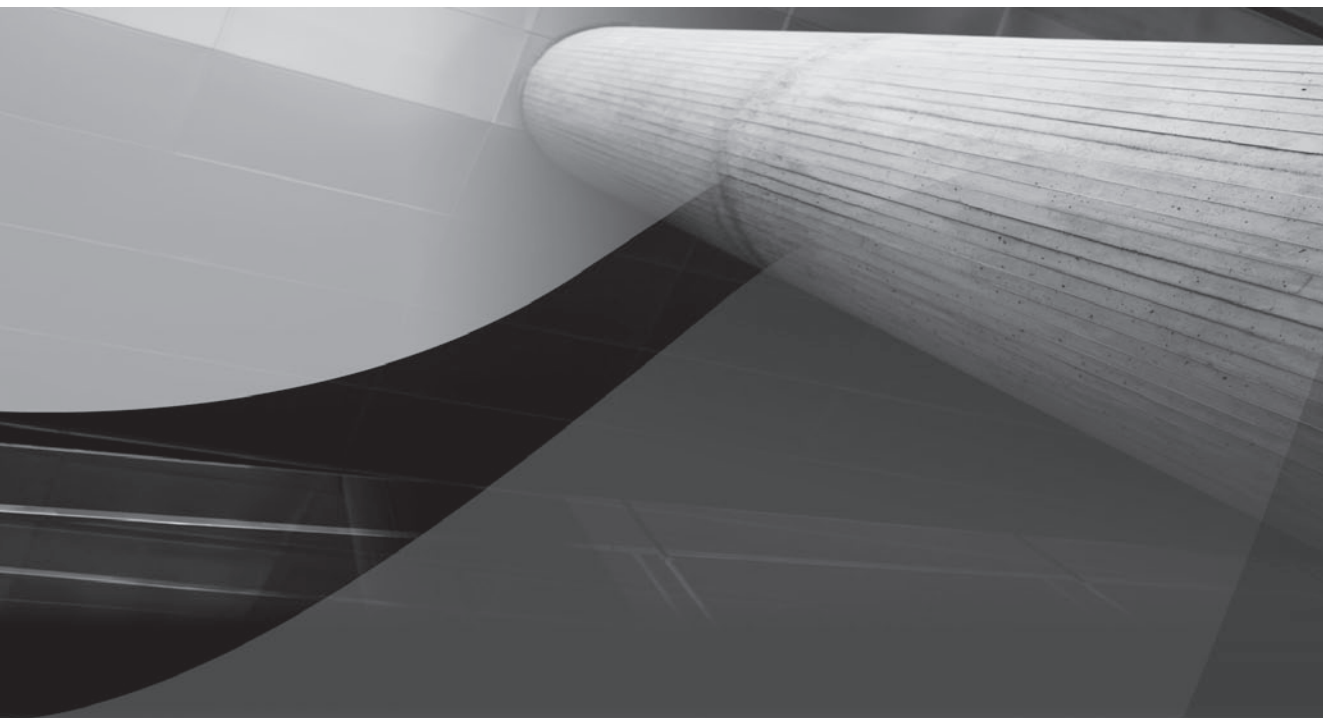
Em certas versões do banco de dados anteriores ao Oracle Database 11g, você pode compilar código PL/SQL em código C e, então, compilar o código C em código de máquina; esse é um processo muito trabalhoso e complexo. No Oracle Database 11g, o compilador de PL/SQL pode gerar código de máquina nativo diretamente. A configuração do banco de dados para gerar código de máquina nativo deve ser feita somente por um administrador experiente (assim, sua abordagem está fora dos objetivos deste livro). Você pode ler sobre geração de código de máquina nativo PL/SQL no manual *PL/SQL User's Guide and Reference* da Oracle Corporation.

## RESUMO

Neste capítulo, você aprendeu que:

- Os programas PL/SQL são divididos em blocos que contêm instruções PL/SQL e SQL.
- Um loop, como um loop `WHILE` ou `FOR`, executa instruções várias vezes.
- Um cursor permite que o PL/SQL leia as linhas retornadas por uma consulta.
- Exceções são usadas para tratar erros em tempo de execução que ocorrem em seu código PL/SQL.
- Uma procedure contém um grupo de instruções. As procedures permitem centralizar a lógica de negócio no banco de dados e podem ser executadas por qualquer programa que acesse o banco de dados.
- Uma função é semelhante a uma procedure, exceto que deve retornar um valor.
- Você pode agrupar procedures e funções em pacotes, os quais encapsulam funcionalidades relacionadas em uma unidade independente.
- Um trigger é uma procedure executada automaticamente pelo banco de dados, quando uma instrução `INSERT`, `UPDATE` ou `DELETE` específica é executada. Os triggers são úteis para fazer coisas como a auditoria avançada das alterações feitas nos valores de coluna em uma tabela.

No próximo capítulo, você vai aprender sobre objetos de banco de dados.



# CAPÍTULO 12

Objetos de banco  
de dados

Neste capítulo, você vai aprender:

- Sobre objetos do banco de dados
- A criar tipos de objeto contendo atributos e métodos
- A usar tipos de objeto para definir objetos de coluna e tabelas de objeto
- A criar e manipular objetos em SQL e em PL/SQL
- Como um tipo pode herdar de outro tipo e criar hierarquias de tipos
- A definir seus próprios construtores para definir os atributos de um objeto
- Como sobrescrever um método de um tipo com um método de outro tipo

## INTRODUÇÃO AOS OBJETOS

As linguagens de programação orientadas a objetos, como Java, C++ e C#, permitem definir classes que atuam como modelos a partir dos quais você pode criar objetos. As classes definem atributos e métodos; os atributos são usados para armazenar o estado de um objeto e os métodos são usados para modelar os comportamentos de um objeto.

Com a versão 8 do Oracle Database, os objetos se tornaram disponíveis dentro do banco de dados e as características dos objetos foram aprimoradas nas versões posteriores do produto. A disponibilidade de objetos no banco de dados foi um avanço importante, pois eles permitem que você defina suas próprias classes, conhecidas como *tipos de objeto*. Assim como as classes em Java e C#, os tipos de objeto de banco de dados podem conter atributos e métodos. Às vezes, os tipos de objeto também são conhecidos como tipos definidos pelo usuário.

Um exemplo simples de tipo de objeto seria um que representasse um produto. Esse tipo de objeto poderia conter atributos para o nome, descrição e preço do produto e, no caso de um produto perecível, o número de dias que pode ficar na prateleira antes de precisar ser descartado. Esse tipo de objeto produto também poderia conter um método que retornaria a data de vencimento do produto, com base na vida de prateleira do produto e na data atual. Outro exemplo de tipo de objeto é aquele que representa uma pessoa, o qual poderia armazenar atributos para o nome, sobrenome, data de nascimento e endereço da pessoa. O próprio endereço da pessoa poderia ser representado por um tipo de objeto e poderia armazenar informações como a rua, cidade, estado e CEP. Neste capítulo, você verá exemplos de tipos de objeto que representam um produto, uma pessoa e um endereço. Você também verá como criar tabelas a partir desses tipos de objeto, como preencher essas tabelas com objetos reais e como manipular esses objetos em SQL e PL/SQL.

Existe um script SQL\*Plus chamado `object_schema.sql` no diretório SQL, o qual cria um usuário chamado `object_user`, com a senha `object_password`. Esse script também cria os tipos e tabelas, executa as várias instruções `INSERT` e cria o código PL/SQL mostrado na primeira parte deste capítulo. Você deve executar esse script enquanto estiver conectado como um usuário com os privilégios necessários para criar um novo usuário com os privilégios `CONNECT`, `RESOURCE` e `CREATE PUBLIC SYNONYM`; você pode usar o usuário `system` para executar os scripts. Depois que o script terminar, você estará conectado como `object_user`.

## CRIANDO TIPOS DE OBJETO

Você cria um tipo de objeto usando a instrução `CREATE TYPE`. O exemplo a seguir usa a instrução `CREATE TYPE` para criar um tipo de objeto chamado `t_address`. Esse tipo de objeto é usado para representar um endereço e contém quatro atributos, `street`, `city`, `state` e `zip`:

```
CREATE TYPE t_address AS OBJECT (  
    street VARCHAR2(15),  
    city   VARCHAR2(15),  
    state  CHAR(2),  
    zip    VARCHAR2(5)  
);  
/
```

O exemplo mostra que cada atributo é definido usando um tipo de banco de dados. Por exemplo, `street` é definido como `VARCHAR2(15)`. Conforme você verá em breve, o tipo de um atributo pode ser um tipo de objeto.

O exemplo a seguir cria um tipo de objeto chamado `t_person`; observe que `t_person` tem um atributo chamado `address`, que é de tipo `t_address`:

```
CREATE TYPE t_person AS OBJECT (  
    id            INTEGER,  
    first_name    VARCHAR2(10),  
    last_name     VARCHAR2(10),  
    dob           DATE,  
    phone         VARCHAR2(12),  
    address       t_address  
);  
/
```

O exemplo a seguir cria um tipo de objeto chamado `t_product` que será usado para representar produtos; observe que esse tipo declara uma função chamada `get_sell_by_date()`, usando a cláusula `MEMBER FUNCTION`:

```
CREATE TYPE t_product AS OBJECT (  
    id            INTEGER,  
    name          VARCHAR2(15),  
    description    VARCHAR2(22),  
    price         NUMBER(5, 2),  
    days_valid     INTEGER,  
  
    -- get_sell_by_date() retorna a data até a qual  
    -- o produto precisa ser vendido  
    MEMBER FUNCTION get_sell_by_date RETURN DATE  
);  
/
```

Como `t_product` contém uma declaração de método, um *corpo* também deve ser criado para `t_product`. O corpo contém o código do método e é criado com a instrução `CREATE TYPE BODY`. O exemplo a seguir cria o corpo de `t_product`; observe que o corpo contém o código da função `get_sell_by_date()`.

```
CREATE TYPE BODY t_product AS
-- get_sell_by_date() retorna a data até a qual
-- o produto precisa ser vendido
MEMBER FUNCTION get_sell_by_date RETURN DATE IS
    v_sell_by_date DATE;
BEGIN
    -- calcula a data de vencimento somando o atributo days_valid
    -- à data atual (SYSDATE)
    SELECT days_valid + SYSDATE
    INTO v_sell_by_date
    FROM dual;

    -- retorna a data de vencimento
    RETURN v_sell_by_date;
END;
/
```

Como você pode ver, `get_sell_by_date()` calcula e retorna a data até a qual o produto precisa ser vendido; a função faz isso somando o atributo `days_valid` à data atual retornada pela função `SYSDATE()` do banco de dados.

Você também pode criar um sinônimo público para um tipo, o que permite que todos os usuários vejam o tipo e o utilizem para definir colunas em suas próprias tabelas. O exemplo a seguir cria um sinônimo público chamado `t_pub_product` para `t_product`:

```
CREATE PUBLIC SYNONYM t_pub_product FOR t_product;
```

## USANDO DESCRIBE PARA OBTER INFORMAÇÕES SOBRE TIPOS DE OBJETO

Você pode usar o comando `DESCRIBE` para obter informações sobre um tipo de objeto. Os exemplos a seguir mostram os tipos `t_address`, `t_person` e `t_product`:

```
DESCRIBE t_address
```

Name	Null?	Type
STREET		VARCHAR2 (15)
CITY		VARCHAR2 (15)
STATE		CHAR (2)
ZIP		VARCHAR2 (5)

```
DESCRIBE t_person
```

Name	Null?	Type
ID		NUMBER (38)
FIRST_NAME		VARCHAR2 (10)
LAST_NAME		VARCHAR2 (10)
DOB		DATE
PHONE		VARCHAR2 (12)
ADDRESS		T_ADDRESS



```
DESCRIBE t_product
```

Name	Null?	Type
-----	-----	-----
ID		NUMBER (38)
NAME		VARCHAR2 (10)
DESCRIPTION		VARCHAR2 (22)
PRICE		NUMBER (5,2)
DAYS_VALID		INTEGER
METHOD		
-----		
MEMBER FUNCTION GET_SELL_BY_DATE RETURNS DATE		

Você pode definir a profundidade com a qual DESCRIBE mostrará informações de tipos incorporados, usando SET DESCRIBE DEPTH. O exemplo a seguir define a profundidade como 2 e então descreve t\_person novamente; observe que são exibidos os atributos de address, que é um objeto incorporado de tipo t\_address:

```
SET DESCRIBE DEPTH 2
```

```
DESCRIBE t_person
```

Name	Null?	Type
-----	-----	-----
ID		NUMBER (38)
FIRST_NAME		VARCHAR2 (10)
LAST_NAME		VARCHAR2 (10)
DOB		DATE
PHONE		VARCHAR2 (12)
ADDRESS		T_ADDRESS
STREET		VARCHAR2 (15)
CITY		VARCHAR2 (15)
STATE		CHAR (2)
ZIP		VARCHAR2 (5)

## USANDO TIPOS DE OBJETO EM TABELAS DE BANCO DE DADOS

Agora que você já aprendeu a criar tipos de objeto, vejamos como utilizar esses tipos em tabelas de banco de dados. Você pode usar um tipo de objeto para definir uma coluna individual em uma tabela e os objetos armazenados nessa coluna são conhecidos como *objetos de coluna*. Você também pode usar um tipo de objeto para definir uma linha inteira em uma tabela, que será conhecida como *tabela de objeto*. Por fim, você pode usar uma *referência de objeto* para acessar uma linha individual em uma tabela de objeto; uma referência de objeto é semelhante a um ponteiro em C++. Você verá exemplos de objetos de coluna, tabelas de objeto e referências de objeto nesta seção.

### Objetos de coluna

O exemplo a seguir cria uma tabela chamada products que contém uma coluna chamada product de tipo t\_product; a tabela também contém uma coluna chamada quantity\_in\_stock, que é usada para armazenar o número desses produtos atualmente no estoque:

```
CREATE TABLE products (
    product          t_product,
    quantity_in_stock INTEGER
);
```

Ao adicionar uma linha nessa tabela, você deve usar um *construtor* para fornecer os valores de atributo do novo objeto `t_product`; como lembrete, o tipo `t_product` foi criado com a seguinte instrução:

```
CREATE TYPE t_product AS OBJECT (
    id            INTEGER,
    name          VARCHAR2(10),
    description   VARCHAR2(22),
    price         NUMBER(5, 2),
    days_valid    INTEGER,

    -- declara a função membro get_sell_by_date(),
    -- get_sell_by_date() retorna a data até a qual
    -- o produto precisa ser vendido
    MEMBER FUNCTION get_sell_by_date RETURN DATE
);
/
```

Um construtor é um método gerado automaticamente para o tipo de objeto e tem o mesmo nome deste; o construtor aceita os parâmetros usados para configurar os atributos do novo objeto. O construtor do tipo `t_product` é chamado `t_product` e aceita cinco parâmetros, um para definir cada um dos atributos; por exemplo, `t_product(1, pasta, 20 oz bag of pasta, 3.95, 10)` cria um novo objeto `t_product` e define seu valor de `id` como 1, `name` como `pasta`, `description` como `20 oz bag of pasta`, `price` como 3.95 e `days_valid` como 10.

As instruções `INSERT` a seguir adicionam duas linhas na tabela `products`; observe o uso do construtor `t_product` para fornecer os valores de atributo dos objetos de coluna `product`:

```
INSERT INTO products (
    product,
    quantity_in_stock
) VALUES (
    t_product(1, 'pasta', '20 oz bag of pasta', 3.95, 10),
    50
);

INSERT INTO products (
    product,
    quantity_in_stock
) VALUES (
    t_product(2, 'sardines', '12 oz box of sardines', 2.99, 5),
    25
);
```

A consulta a seguir recupera essas linhas da tabela `products`; observe que os atributos dos objetos de coluna `product` são exibidos dentro de um construtor para `t_product`:

```
SELECT *
FROM products;
```

```

PRODUCT(ID, NAME, DESCRIPTION, PRICE, DAYS_VALID)
-----
QUANTITY_IN_STOCK
-----
T_PRODUCT(1, 'pasta', '20 oz bag of pasta', 3.95, 10)
          50
T_PRODUCT(2, 'sardines', '12 oz box of sardines', 2.99, 5)
          25

```

Você também pode recuperar um objeto de coluna individual de uma tabela. Para tanto, você precisa fornecer um apelido de tabela, por meio do qual seleciona o objeto. A consulta a seguir recupera o produto nº 1 da tabela `products`; observe o uso do apelido `p` para a tabela `products`, por meio do qual o atributo `id` do objeto `product` é especificado na cláusula `WHERE`:

```

SELECT p.product
FROM products p
WHERE p.product.id = 1;

```

```

PRODUCT(ID, NAME, DESCRIPTION, PRICE, DAYS_VALID)
-----
T_PRODUCT(1, 'pasta', '20 oz bag of pasta', 3.95, 10)

```

A próxima consulta inclui explicitamente os atributos `id`, `name`, `price` e `days_valid` do objeto `product` na instrução `SELECT`, além de `quantity_in_stock`:

```

SELECT p.product.id, p.product.name,
       p.product.price, p.product.days_valid, p.quantity_in_stock
FROM products p
WHERE p.product.id = 1;

```

```

PRODUCT.ID PRODUCT.NA PRODUCT.PRICE PRODUCT.DAYS_VALID QUANTITY_IN_STOCK
-----
          1 pasta                3.95                10                50

```

O tipo de objeto `t_product` contém uma função chamada `get_sell_by_date()`, a qual calcula e retorna a data até a qual o produto deve ser vendido. A função faz isso somando o atributo `days_valid` à data atual, a qual é obtida do banco de dados com a função `SYSDATE()`. Você pode chamar a função `get_sell_by_date()` usando um apelido de tabela, como mostrado na consulta a seguir, que utiliza o apelido `p` para a tabela `products`:

```

SELECT p.product.get_sell_by_date()
FROM products p;

```

```

P.PRODUCT
-----
19-JUN-07
13-JUN-07

```

Se você executar essa consulta, suas datas serão diferentes, pois são calculadas usando `SYSDATE()`, que retorna a data e hora atuais.

A instrução `UPDATE` a seguir modifica a descrição do produto nº 1; observe que o apelido `p` é usado novamente:

```
UPDATE products p
SET p.product.description = '30 oz bag of pasta'
WHERE p.product.id = 1;
```

1 row updated.

A instrução `DELETE` a seguir remove o produto nº 2:

```
DELETE FROM products p
WHERE p.product.id = 2;
```

1 row deleted.

`ROLLBACK;`

### NOTA

*Se você executar as instruções `UPDATE` e `DELETE`, execute a instrução `ROLLBACK` para que seus dados de exemplo correspondam àqueles mostrados no restante deste capítulo.*

## Tabelas de objeto

Você pode usar um tipo de objeto para definir uma tabela inteira; tal tabela é conhecida como tabela de objeto. O exemplo a seguir cria uma tabela de objeto chamada `object_products`, a qual armazena objetos de tipo `t_product`; observe o uso da palavra-chave `OF` para identificar a tabela como uma tabela de objeto de tipo `t_product`:

```
CREATE TABLE object_products OF t_product;
```

Ao inserir uma linha em uma tabela de objeto, você pode escolher se vai usar um construtor para fornecer valores de atributo ou se vai fornecer os valores da mesma maneira como forneceria valores de coluna em uma tabela relacional. A instrução `INSERT` a seguir adiciona uma linha na tabela `object_products` usando o construtor de `t_product`:

```
INSERT INTO object_products VALUES (
    t_product(1, 'pasta', '20 oz bag of pasta', 3.95, 10)
);
```

A próxima instrução `INSERT` omite o construtor de `t_product`; observe que os valores de atributo de `t_product` são fornecidos da mesma maneira que colunas seriam inseridas em uma tabela relacional:

```
INSERT INTO object_products (
    id, name, description, price, days_valid
) VALUES (
    2, 'sardines', '12 oz box of sardines', 2.99, 5
);
```

A consulta a seguir recupera essas linhas da tabela `object_products`:

```
SELECT *
FROM object_products;
```

ID	NAME	DESCRIPTION	PRICE	DAYS_VALID
1	pasta	20 oz bag of pasta	3.95	10
2	sardines	12 oz box of sardines	2.99	5

Você também pode especificar atributos de objeto individuais em uma consulta, por exemplo, fazendo isto:

```
SELECT id, name, price
FROM object_products op
WHERE id = 1;
```

ID	NAME	PRICE
1	pasta	3.95

ou isto:

```
SELECT op.id, op.name, op.price
FROM object_products op
WHERE op.id = 1;
```

ID	NAME	PRICE
1	pasta	3.95

É possível usar a função interna `VALUE()` do banco de dados Oracle para selecionar uma linha de uma tabela de objeto. `VALUE()` trata a linha como um objeto real e retorna os atributos do objeto dentro de um construtor para o tipo de objeto. `VALUE()` aceita um parâmetro contendo um apelido de tabela, como mostrado na consulta a seguir:

```
SELECT VALUE(op)
FROM object_products op;

VALUE(OP) (ID, NAME, DESCRIPTION, PRICE, DAYS_VALID)
-----
T_PRODUCT(1, 'pasta', '20 oz bag of pasta', 3.95, 10)
T_PRODUCT(2, 'sardines', '12 oz box of sardines', 2.99, 5)
```

Você também pode adicionar um atributo de objeto após `VALUE()`:

```
SELECT VALUE(op).id, VALUE(op).name, VALUE(op).price
FROM object_products op;
```

VALUE(OP).ID	VALUE(OP).NAME	VALUE(OP).PRICE
1	pasta	3.95
2	sardines	2.99

A instrução `UPDATE` a seguir modifica a descrição do produto nº 1:

```
UPDATE object_products
SET description = '25 oz bag of pasta'
WHERE id = 1;
```

1 row updated.

A instrução DELETE a seguir remove o produto nº 2:

```
DELETE FROM object_products
WHERE id = 2;
```

1 row deleted.

```
ROLLBACK;
```

Vejamos uma tabela de objeto mais complexa. A instrução CREATE TABLE a seguir cria uma tabela de objeto chamada object\_customers, a qual armazena objetos de tipo t\_person:

```
CREATE TABLE object_customers OF t_person;
```

O tipo t\_person contém um objeto t\_address incorporado; t\_person foi criado com a seguinte instrução:

```
CREATE TYPE t_person AS OBJECT (
  id          INTEGER,
  first_name  VARCHAR2(10),
  last_name   VARCHAR2(10),
  dob         DATE,
  phone       VARCHAR2(12),
  address     t_address
);
/
```

As instruções INSERT a seguir adicionam duas linhas em object\_customers. A primeira instrução INSERT usa construtores de t\_person e t\_address, enquanto a segunda instrução INSERT omite o construtor de t\_person:

```
INSERT INTO object_customers VALUES (
  t_person(1, 'John', 'Brown', '01-FEB-1955', '800-555-1211',
    t_address('2 State Street', 'Beantown', 'MA', '12345')
)
);

INSERT INTO object_customers (
  id, first_name, last_name, dob, phone,
  address
) VALUES (
  2, 'Cynthia', 'Green', '05-FEB-1968', '800-555-1212',
  t_address('3 Free Street', 'Middle Town', 'CA', '12345')
);
```

A consulta a seguir recupera essas linhas da tabela object\_customers; observe que os atributos do objeto de coluna incorporado address são exibidos dentro do construtor de t\_address:

```
SELECT *
FROM object_customers;
```

```

      ID FIRST_NAME LAST_NAME DOB      PHONE
-----
ADDRESS(STREET, CITY, STATE, ZIP)
-----
      1 John      Brown      01-FEB-55 800-555-1211
T_ADDRESS('2 State Street', 'Beantown', 'MA', '12345')

      2 Cynthia   Green      05-FEB-68 800-555-1212
T_ADDRESS('3 Free Street', 'Middle Town', 'CA', '12345')

```

A próxima consulta recupera o cliente nº 1 de `object_customers`; observe o uso do apelido de tabela `oc`, por meio do qual o atributo `id` é especificado na cláusula `WHERE`:

```

SELECT *
FROM object_customers oc
WHERE oc.id = 1;

```

```

      ID FIRST_NAME LAST_NAME DOB      PHONE
-----
ADDRESS(STREET, CITY, STATE, ZIP)
-----
      1 John      Brown      01-FEB-55 800-555-1211
T_ADDRESS('2 State Street', 'Beantown', 'MA', '12345')

```

Na consulta a seguir, um cliente é recuperado com base no atributo `state` do objeto de coluna `address`:

```

SELECT *
FROM object_customers oc
WHERE oc.address.state = 'MA';

```

```

      ID FIRST_NAME LAST_NAME DOB      PHONE
-----
ADDRESS(STREET, CITY, STATE, ZIP)
-----
      1 John      Brown      01-FEB-55 800-555-1211
T_ADDRESS('2 State Street', 'Beantown', 'MA', '12345')

```

Na próxima consulta os atributos `id`, `first_name` e `last_name` do cliente nº 1 são incluídos explicitamente na instrução `SELECT`, junto com os atributos do objeto de coluna incorporado `address`:

```

SELECT oc.id, oc.first_name, oc.last_name,
       oc.address.street, oc.address.city, oc.address.state, oc.address.zip
FROM object_customers oc
WHERE oc.id = 1;

```

```

      ID FIRST_NAME LAST_NAME ADDRESS.STREET ADDRESS.CITY  AD  ADDRE
-----
      1 John      Brown      2 State Street  Beantown      MA 12345

```

## Identificadores de objeto e referências de objeto

Cada objeto em uma tabela de objeto tem um *identificador de objeto* (OID) exclusivo e você pode recuperar o OID de um objeto usando a função `REF()`. Por exemplo, a consulta a seguir recupera o OID do cliente nº 1 na tabela `object_customers`:

```
SELECT REF(oc)
FROM object_customers oc
WHERE oc.id = 1;

REF(OC)
-----
0000280209D66AB93F991647649D78D08B267EE44858C7B9989D9D40689FB4DA92820
AFFE2010003280000
```

A longa string de números e letras é o OID, o qual identifica a localização do objeto no banco de dados. Você pode armazenar um OID em uma referência de objeto e, posteriormente, acessar o objeto a que ela se refere. Uma referência de objeto, que é semelhante a um ponteiro em C++, aponta para um objeto armazenado em uma tabela de objeto usando o OID. Você pode usar referências de objeto para modelar relações entre tabelas de objeto e, conforme verá posteriormente, pode usar referências de objeto em PL/SQL para acessar objetos.

Para definir uma referência de objeto, use o tipo `REF`; a instrução a seguir cria uma tabela chamada `purchases` que contém duas colunas de referência de objeto, chamadas `customer_ref` e `product_ref`:

```
CREATE TABLE purchases (
    id            INTEGER PRIMARY KEY,
    customer_ref  REF t_person SCOPE IS object_customers,
    product_ref   REF t_product SCOPE IS object_products
);
```

A cláusula `SCOPE IS` restringe uma referência de objeto a apontar para objetos de uma tabela específica. Por exemplo, a coluna `customer_ref` está restrita a apontar somente para objetos da tabela `object_customers`; da mesma forma, a coluna `product_ref` está restrita a apontar somente para objetos da tabela `object_products`.

Conforme mencionamos anteriormente, em uma tabela de objeto cada objeto tem um identificador de objeto exclusivo (OID) que você pode armazenar em uma referência de objeto; você pode recuperar um OID usando a função `REF()` e armazená-lo em uma referência de objeto. Por exemplo, a instrução `INSERT` a seguir adiciona uma linha na tabela `purchases`; observe que a função `REF()` é usada nas consultas para obter os identificadores de objeto do cliente nº 1 e do produto nº 1 das tabelas `object_customers` e `object_products`:

```
INSERT INTO purchases (
    id,
    customer_ref,
    product_ref
) VALUES (
    1,
    (SELECT REF(oc) FROM object_customers oc WHERE oc.id = 1),
    (SELECT REF(op) FROM object_products op WHERE op.id = 1)
);
```

Esse exemplo registra que o cliente nº 1 comprou o produto nº 1.



A consulta a seguir seleciona a linha da tabela `purchases`; observe que as colunas `customer_ref` e `product_ref` contêm referências para os objetos nas tabelas `object_customers` e `object_products`:

```
SELECT *
FROM purchases;

          ID
-----
CUSTOMER_REF
-----
PRODUCT_REF
-----
          1
0000220208D66AB93F991647649D78D08B267EE44858C7B9989D9D40689FB4DA92820
AFFE2
0000220208662E2AB4256711D6A1B50010A4E7AE8A662E2AB2256711D6A1B50010A4E
7AE8A
```

Você pode recuperar os objetos armazenados em uma referência de objeto usando a função `DEREF()`, que aceita uma referência de objeto como parâmetro e retorna o objeto. Por exemplo, a consulta a seguir usa `DEREF()` para recuperar o cliente nº 1 e o produto nº 1 por meio das colunas `customer_ref` e `product_ref` da tabela `purchases`:

```
SELECT DEREF(customer_ref), DEREF(product_ref)
FROM purchases;

DEREF(CUSTOMER_REF) (ID, FIRST_NAME, LAST_NAME, DOB, PHONE,
ADDRESS (STREET, CITY,
-----
DEREF(PRODUCT_REF) (ID, NAME, DESCRIPTION, PRICE, DAYS_VALID)
-----
T_PERSON(1, 'John', 'Brown', '01-FEB-55', '800-555-1211',
T_ADDRESS('2 State Street', 'Beantown', 'MA', '12345'))
T_PRODUCT(1, 'pasta', '20 oz bag of pasta', 3.95, 10)
```

A próxima consulta recupera os atributos `first_name` e `address.street` do cliente, além do atributo `name` do produto:

```
SELECT DEREF(customer_ref).first_name,
DEREF(customer_ref).address.street, DEREF(product_ref).name
FROM purchases;

DEREF(CUST DEREF(CUSTOMER_ DEREF(PROD
-----
John          2 State Street  pasta
```

A instrução `UPDATE` a seguir modifica a coluna `product_ref` para apontar para o produto nº 2:

```
UPDATE purchases SET product_ref = (
SELECT REF(op) FROM object_products op WHERE op.id = 2
) WHERE id = 1;

1 row updated.
```

A consulta a seguir verifica essa alteração:

```
SELECT Deref(customer_ref), Deref(product_ref)
FROM purchases;
Deref(CUSTOMER_REF) (ID, FIRST_NAME, LAST_NAME, DOB, PHONE,
ADDRESS(STREET, CITY,
-----
Deref(PRODUCT_REF) (ID, NAME, DESCRIPTION, PRICE, DAYS_VALID)
-----
T_PERSON(1, 'John', 'Brown', '01-FEB-55', '800-555-1211',
T_ADDRESS('2 State Street', 'Beantown', 'MA', '12345'))
T_PRODUCT(2, 'sardines', '12 oz box of sardines', 2.99, 5)
```

## Comparando valores de objeto

Você pode comparar o valor de dois objetos na cláusula WHERE de uma consulta usando o operador de igualdade (=). Por exemplo, a consulta a seguir recupera o cliente nº 1 da tabela object\_customers:

```
SELECT oc.id, oc.first_name, oc.last_name, oc.dob
FROM object_customers oc
WHERE VALUE(oc) =
    t_person(1, 'John', 'Brown', '01-FEB-1955', '800-555-1211',
    t_address('2 State Street', 'Beantown', 'MA', '12345')
);
```

ID	FIRST_NAME	LAST_NAME	DOB
1	John	Brown	01-FEB-55

A próxima consulta recupera o produto nº 1 da tabela object\_products:

```
SELECT op.id, op.name, op.price, op.days_valid
FROM object_products op
WHERE VALUE(op) = t_product(1, 'pasta', '20 oz bag of pasta', 3.95, 10);
```

ID	NAME	PRICE	DAYS_VALID
1	pasta	3.95	10

Você também pode usar os operadores <> e IN na cláusula WHERE:

```
SELECT oc.id, oc.first_name, oc.last_name, oc.dob
FROM object_customers oc
WHERE VALUE(oc) <>
    t_person(1, 'John', 'Brown', '01-FEB-1955', '800-555-1211',
    t_address('2 State Street', 'Beantown', 'MA', '12345')
);
```

ID	FIRST_NAME	LAST_NAME	DOB
2	Cynthia	Green	05-FEB-68

```
SELECT op.id, op.name, op.price, op.days_valid
FROM object_products op
WHERE VALUE(op) IN t_product(1, 'pasta', '20 oz bag of pasta', 3.95, 10);
```

ID	NAME	PRICE	DAYS_VALID
1	pasta	3.95	10

Se quiser usar um operador como <, >, <=, >=, LIKE ou BETWEEN, você precisa fornecer uma função de mapeamento para o tipo. Uma função de mapeamento deve retornar um único valor de um dos tipos internos, que o banco de dados pode então utilizar para comparar dois objetos. O valor retornado pela função de mapeamento será diferente para cada tipo de objeto e você precisa descobrir qual atributo ou concatenação de atributos representa melhor o valor de um objeto. Por exemplo, com o tipo `t_product`, seria retornado o atributo `price`; com o tipo `t_person`, seria retornada uma concatenação dos atributos `last_name` e `first_name`.

As instruções a seguir criam um tipo chamado `t_person2` que contém uma função de mapeamento chamada `get_string()`; observe que `get_string()` retorna uma string `VARCHAR2` contendo uma concatenação dos atributos `last_name` e `first_name`:

```
CREATE TYPE t_person2 AS OBJECT (
  id          INTEGER,
  first_name  VARCHAR2(10),
  last_name   VARCHAR2(10),
  dob         DATE,
  phone       VARCHAR2(12),
  address     t_address,

  -- declara a função de mapeamento get_string(),
  -- que retorna uma string VARCHAR2
  MAP MEMBER FUNCTION get_string RETURN VARCHAR2
);
/

CREATE TYPE BODY t_person2 AS
  -- define a função de mapeamento get_string()
  MAP MEMBER FUNCTION get_string RETURN VARCHAR2 IS
  BEGIN
    -- retorna uma string concatenada contendo os
    -- atributos last_name e first_name
    RETURN last_name || ' ' || first_name;
  END get_string;
END;
/
```

Conforme será visto em breve, o banco de dados chamará `get_string()` automaticamente ao comparar objetos `t_person2`. As instruções a seguir criam uma tabela chamada `object_customers2` e adicionam linhas nesta tabela:

```
CREATE TABLE object_customers2 OF t_person2;

INSERT INTO object_customers2 VALUES (
```

```
t_person2(1, 'John', 'Brown', '01-FEB-1955', '800-555-1211',
t_address('2 State Street', 'Beantown', 'MA', '12345')
)
);

INSERT INTO object_customers2 VALUES (
t_person2(2, 'Cynthia', 'Green', '05-FEB-1968', '800-555-1212',
t_address('3 Free Street', 'Middle Town', 'CA', '12345')
)
);
```

A consulta a seguir usa > na cláusula WHERE:

```
SELECT oc2.id, oc2.first_name, oc2.last_name, oc2.dob
FROM object_customers2 oc2
WHERE VALUE(oc2) >
t_person2(1, 'John', 'Brown', '01-FEB-1955', '800-555-1211',
t_address('2 State Street', 'Beantown', 'MA', '12345')
);
```

ID	FIRST_NAME	LAST_NAME	DOB
2	Cynthia	Green	05-FEB-68

Quando a consulta é executada, o banco de dados chama `get_string()` automaticamente para comparar os objetos da tabela `object_customers2` com o objeto que está após > na cláusula WHERE. A função `get_string()` retorna uma concatenação dos atributos `last_name` e `first_name` dos objetos e como `Green Cynthia` é maior do que `Brown John`, ela é retornada pela consulta.

## USANDO OBJETOS EM PL/SQL

É possível criar e manipular objetos em PL/SQL. Nesta seção, você aprenderá a usar o pacote `product_package`, criado quando o script `object_schema.sql` é executado. `product_package` contém os seguintes métodos:

- Uma função chamada `get_products()` que retorna um `REF CURSOR` que aponta para os objetos da tabela `object_products`
- Uma procedure chamada `display_product()` que exibe os atributos de um único objeto da tabela `object_products`
- Uma procedure chamada `insert_product()` que adiciona um objeto na tabela `object_products`
- Uma procedure chamada `update_product_price()` que atualiza o atributo `price` de um objeto da tabela `object_products`
- Uma função chamada `get_product()` que retorna um único objeto da tabela `object_products`
- Uma procedure chamada `update_product()` que atualiza todo os atributos de um objeto da tabela `object_products`

- Uma função chamada `get_product_ref()` que retorna uma referência para um único objeto da tabela `object_products`
- Uma procedure chamada `delete_product()` que exclui um único objeto da tabela `object_products`

O script `object_schema.sql` contém a seguinte especificação de pacote:

```
CREATE PACKAGE product_package AS
  TYPE t_ref_cursor IS REF CURSOR;
  FUNCTION get_products RETURN t_ref_cursor;
  PROCEDURE display_product(
    p_id IN object_products.id%TYPE
  );
  PROCEDURE insert_product(
    p_id          IN object_products.id%TYPE,
    p_name        IN object_products.name%TYPE,
    p_description IN object_products.description%TYPE,
    p_price       IN object_products.price%TYPE,
    p_days_valid  IN object_products.days_valid%TYPE
  );
  PROCEDURE update_product_price(
    p_id      IN object_products.id%TYPE,
    p_factor  IN NUMBER
  );
  FUNCTION get_product(
    p_id IN object_products.id%TYPE
  ) RETURN t_product;
  PROCEDURE update_product(
    p_product t_product
  );
  FUNCTION get_product_ref(
    p_id IN object_products.id%TYPE
  ) RETURN REF t_product;
  PROCEDURE delete_product(
    p_id IN object_products.id%TYPE
  );
END product_package;
/
```

Os métodos do corpo de `product_package` serão abordados nas seções a seguir.

## A função `get_products()`

A função `get_products()` retorna um `REF CURSOR` que aponta para os objetos da tabela `object_products`; `get_products()` é definida como segue no corpo do `product_package`:

```
FUNCTION get_products
  RETURN t_ref_cursor IS
  -- declara um objeto t_ref_cursor
  v_products_ref_cursor t_ref_cursor;
BEGIN
```

```

-- obtém o REF CURSOR
OPEN v_products_ref_cursor FOR
  SELECT VALUE(op)
  FROM object_products op
  ORDER BY op.id;

-- retorna o REF CURSOR
RETURN v_products_ref_cursor;
END get_products;

```

A consulta a seguir chama `product_package.get_products()` para recuperar os produtos de `object_products`:

```

SELECT product_package.get_products
FROM dual;

GET_PRODUCTS
-----
CURSOR STATEMENT : 1

CURSOR STATEMENT : 1

VALUE(OP) (ID, NAME, DESCRIPTION, PRICE, DAYS_VALID)
-----
T_PRODUCT(1, 'pasta', '20 oz bag of pasta', 3.95, 10)
T_PRODUCT(2, 'sardines', '12 oz box of sardines', 2.99, 5)

```

## A procedure `display_product()`

A procedure `display_product()` exibe os atributos de um único objeto da tabela `object_products`; `display_product()` é definida como segue no corpo de `product_package`:

```

PROCEDURE display_product(
  p_id IN object_products.id%TYPE
) AS
  -- declara um objeto t_product chamado v_product
  v_product t_product;
BEGIN
  -- tenta obter o produto e armazená-lo em v_product
  SELECT VALUE(op)
  INTO v_product
  FROM object_products op
  WHERE id = p_id;

  -- exibe os atributos de v_product
  DBMS_OUTPUT.PUT_LINE('v_product.id=' ||
    v_product.id);
  DBMS_OUTPUT.PUT_LINE('v_product.name=' ||
    v_product.name);
  DBMS_OUTPUT.PUT_LINE('v_product.description=' ||
    v_product.description);
  DBMS_OUTPUT.PUT_LINE('v_product.price=' ||

```

```

        v_product.price);
DBMS_OUTPUT.PUT_LINE('v_product.days_valid=' ||
        v_product.days_valid);

-- chama v_product.get_sell_by_date() e exibe a data
DBMS_OUTPUT.PUT_LINE('Sell by date=' ||
        v_product.get_sell_by_date());
END display_product;

```

O exemplo a seguir chama `product_package.display_product(1)` para recuperar o produto nº 1 da tabela `object_products`:

```

SET SERVEROUTPUT ON
CALL product_package.display_product(1);
v_product.id=1
v_product.name=pasta
v_product.description=20 oz bag of pasta
v_product.price=3.95
v_product.days_valid=10
Sell by date=25-JUN-07

```

## A procedure `insert_product()`

A procedure `insert_product()` adiciona um objeto na tabela `object_products`; `insert_product()` é definida como segue no corpo de `product_package`:

```

PROCEDURE insert_product (
    p_id          IN object_products.id%TYPE,
    p_name        IN object_products.name%TYPE,
    p_description IN object_products.description%TYPE,
    p_price       IN object_products.price%TYPE,
    p_days_valid  IN object_products.days_valid%TYPE
) AS
    -- cria um objeto t_product chamado v_product
    v_product t_product :=
        t_product(
            p_id, p_name, p_description, p_price, p_days_valid
        );
BEGIN
    -- adiciona v_product na tabela object_products
    INSERT INTO object_products VALUES (v_product);
    COMMIT;
EXCEPTION
    WHEN OTHERS THEN
        ROLLBACK;
END insert_product;

```

O exemplo a seguir chama `product_package.insert_product()` para adicionar um novo objeto na tabela `object_products`:

```

CALL product_package.insert_product(3, 'salsa',
    '15 oz jar of salsa', 1.50, 20);

```

## A procedure update\_product\_price()

A procedure `update_product_price()` atualiza o atributo `price` de um objeto da tabela `object_products`; `update_product_price()` é definida como segue no corpo de `product_package`:

```
PROCEDURE update_product_price(
    p_id      IN object_products.id%TYPE,
    p_factor  IN NUMBER
) AS
    -- declara um objeto t_product chamado v_product
    v_product t_product;
BEGIN
    -- tenta selecionar o produto para atualizá-lo e
    -- armazená-lo em v_product
    SELECT VALUE(op)
    INTO v_product
    FROM object_products op
    WHERE id = p_id
    FOR UPDATE;

    -- exibe o preço atual de v_product
    DBMS_OUTPUT.PUT_LINE('v_product.price=' ||
        v_product.price);

    -- multiplica v_product.price por p_factor
    v_product.price := v_product.price * p_factor;
    DBMS_OUTPUT.PUT_LINE('New v_product.price=' ||
        v_product.price);

    -- atualiza o produto na tabela object_products
    UPDATE object_products op
    SET op = v_product
    WHERE id = p_id;
    COMMIT;
EXCEPTION
    WHEN OTHERS THEN
        ROLLBACK;
END update_product_price;
```

O exemplo a seguir chama `product_package.update_product_price()` para atualizar o preço do produto nº 3 na tabela `object_products`:

```
CALL product_package.update_product_price(3, 2.4);
v_product.price=1.5
New v_product.price=3.6
```

## A função get\_product()

A função `get_product()` retorna um único objeto da tabela `object_products`; `get_product()` é definida como segue no corpo de `product_package`:



```

FUNCTION get_product (
  p_id IN object_products.id%TYPE
)
RETURN t_product IS
  -- declara um objeto t_product chamado v_product
  v_product t_product;
BEGIN
  -- obtém o produto e o armazena em v_product
  SELECT VALUE(op)
  INTO v_product
  FROM object_products op
  WHERE op.id = p_id;

  -- retorna v_product
  RETURN v_product;
END get_product;

```

A consulta a seguir chama `product_package.get_product()` para obter o produto nº 3 da tabela `object_products`:

```

SELECT product_package.get_product(3)
FROM dual;

PRODUCT_PACKAGE.GET_PRODUCT(3) (ID, NAME, DESCRIPTION)
-----
T_PRODUCT(3, 'salsa', '15 oz jar of salsa', 3.6, 20)

```

## A procedure `update_product()`

A procedure `update_product()` atualiza todos os atributos de um objeto da tabela `object_products`; `update_product()` é definida como segue no corpo de `product_package`:

```

PROCEDURE update_product (
  p_product IN t_product
) AS
BEGIN
  -- atualiza o produto na tabela object_products
  UPDATE object_products op
  SET op = p_product
  WHERE id = p_product.id;
  COMMIT;
EXCEPTION
  WHEN OTHERS THEN
    ROLLBACK;
END update_product;

```

O exemplo a seguir chama `product_package.update_product()` para atualizar o produto nº 3 da tabela `object_products`:

```

CALL product_package.update_product(t_product(3, 'salsa',
  '25 oz jar of salsa', 2.70, 15));

```

## A função `get_product_ref()`

A função `get_product_ref()` retorna uma referência para um único objeto da tabela `object_products`; `get_product_ref()` é definida como segue no corpo de `product_package`:

```
FUNCTION get_product_ref(
    p_id IN object_products.id%TYPE
)
RETURN REF t_product IS
    -- declara uma referência para um t_product
    v_product_ref REF t_product;
BEGIN
    -- obtém a REF do produto e
    -- a armazena em v_product_ref
    SELECT REF(op)
    INTO v_product_ref
    FROM object_products op
    WHERE op.id = p_id;

    -- retorna v_product_ref
    RETURN v_product_ref;
END get_product_ref;
```

A consulta a seguir chama `product_package.get_product_ref()` para obter a referência para o produto nº 3 da tabela `object_products`:

```
SELECT product_package.get_product_ref(3)
FROM dual;

PRODUCT_PACKAGE.GET_PRODUCT_REF(3)
-----
000028020956DBE8BEFDEF4D5BA8C806A7B31B49DF916CDB2CAC1B46E9808BA181F9F2760F01
00033D0002
```

O exemplo a seguir chama `product_package.get_product_ref()` novamente, desta vez usando `DEREF()` para obter o produto real:

```
SELECT DEREF(product_package.get_product_ref(3))
FROM dual;

DEREF(PRODUCT_PACKAGE.GET_PRODUCT_REF(3)) (ID, NAME,
-----
T_PRODUCT(3, 'salsa', '25 oz jar of salsa', 2.7, 15)
```

## A procedure `delete_product()`

A procedure `delete_product()` exclui um único objeto da tabela `object_products`; `delete_product()` é definida como segue no corpo de `product_package`:

```
PROCEDURE delete_product(
    p_id IN object_products.id%TYPE
) AS
BEGIN
```

```

-- exclui o produto
DELETE FROM object_products op
WHERE op.id = p_id;
COMMIT;
EXCEPTION
  WHEN OTHERS THEN
    ROLLBACK;
END delete_product;

```

O exemplo a seguir chama `product_package.delete_product()` para excluir o produto nº 3 da tabela `object_products`:

```
CALL product_package.delete_product(3);
```

Agora que você já viu todos os métodos de `product_package`, é hora de conhecer as procedures `product_lifecycle()` e `product_lifecycle2()`, que chamam os vários métodos do pacote. As duas procedures são criadas quando você executa o script `object_schema.sql`.

## A procedure `product_lifecycle()`

A procedure `product_lifecycle()` é definida como segue:

```

CREATE PROCEDURE product_lifecycle AS
  -- declara objeto
  v_product t_product;
BEGIN
  -- insere um novo produto
  product_package.insert_product(4, 'beef',
    '25 lb pack of beef', 32, 10);

  -- exibe o produto
  product_package.display_product(4);

  -- obtém o novo produto e o armazena em v_product
  SELECT product_package.get_product(4)
  INTO v_product
  FROM dual;

  -- altera alguns atributos de v_product
  v_product.description := '20 lb pack of beef';
  v_product.price := 36;
  v_product.days_valid := 8;

  -- atualiza o produto
  product_package.update_product(v_product);

  -- exibe o produto
  product_package.display_product(4);

  -- exclui o produto
  product_package.delete_product(4);
END product_lifecycle;
/

```

O exemplo a seguir chama `product_lifecycle()`:

```
CALL product_lifecycle();
v_product.id=4
v_product.name=beef
v_product.description=25 lb pack of beef
v_product.price=32
v_product.days_valid=10
Sell by date=27-JUN-07
v_product.id=4
v_product.name=beef
v_product.description=20 lb pack of beef
v_product.price=36
v_product.days_valid=8
Sell by date=25-JUN-07
```

## A procedure `product_lifecycle2()`

A procedure `product_lifecycle2()` usa uma referência de objeto para acessar um produto; ela é definida como segue:

```
CREATE PROCEDURE product_lifecycle2 AS
  -- declara objeto
  v_product t_product;

  -- declara referência de objeto
  v_product_ref REF t_product;
BEGIN
  -- insere um novo produto
  product_package.insert_product(4, 'beef',
    '25 lb pack of beef', 32, 10);

  -- exibe o produto
  product_package.display_product(4);

  -- obtém a nova referência do produto e a armazena em v_product_ref
  SELECT product_package.get_product_ref(4)
  INTO v_product_ref
  FROM dual;

  -- retira a referência de v_product_ref usando a consulta a seguir
  SELECT Deref(v_product_ref)
  INTO v_product
  FROM dual;

  -- altera alguns atributos de v_product
  v_product.description := '20 lb pack of beef';
  v_product.price := 36;
  v_product.days_valid := 8;

  -- atualiza o produto
  product_package.update_product(v_product);
  -- exibe o produto
  product_package.display_product(4);
```

```
-- exclui o produto
product_package.delete_product(4);
END product_lifecycle2;
/
```

Observe que, para retirar a referência de `v_product_ref`, você precisa usar a consulta a seguir:

```
SELECT Deref(v_product_ref)
  INTO v_product
 FROM dual;
```

Você precisa utilizar essa consulta porque não pode usar `Deref()` diretamente em código PL/SQL. Por exemplo, a instrução a seguir não seria compilada em PL/SQL:

```
v_product := Deref(v_product_ref);
```

O exemplo a seguir chama `product_lifecycle2()`:

```
CALL product_lifecycle2();
v_product.id=4
v_product.name=beef
v_product.description=25 lb pack of beef
v_product.price=32
v_product.days_valid=10
Sell by date=27-JUN-07
v_product.id=4
v_product.name=beef
v_product.description=20 lb pack of beef
v_product.price=36
v_product.days_valid=8
Sell by date=25-JUN-07
```

## HERANÇA DE TIPO

O Oracle Database 9i introduziu a *herança* de tipo de objeto, que permite definir hierarquias de tipos de objeto. Por exemplo, talvez você queira definir um tipo de objeto “usuário\_executivo” e fazer esse tipo herdar os atributos existentes de `t_person`. O tipo “usuário\_executivo” poderia estender `t_person` com atributos para armazenar o nome do cargo da pessoa e o nome da empresa onde trabalha. Para que `t_person` possa ser herdado, sua definição precisa incluir a cláusula `NOT FINAL`:

```
CREATE TYPE t_person AS OBJECT (
  id          INTEGER,
  first_name  VARCHAR2(10),
  last_name   VARCHAR2(10),
  dob         DATE,
  phone       VARCHAR2(12),
  address     t_address,
  MEMBER FUNCTION display_details RETURN VARCHAR2
) NOT FINAL;
/
```

A cláusula `NOT FINAL` indica que `t_person` pode ser herdado na definição de outro tipo. (O padrão ao se definir tipos é `FINAL`, significando que o tipo de objeto não pode ser herdado.)

A instrução a seguir cria o corpo de `t_person`; observe que a função `display_details()` retorna um valor `VARCHAR2` contendo os atributos `id` e `name` da pessoa:

```
CREATE TYPE BODY t_person AS
  MEMBER FUNCTION display_details RETURN VARCHAR2 IS
  BEGIN
    RETURN 'id=' || id || ', name=' || first_name || ' ' || last_name;
  END;
END;
/
```

### NOTA

*Fornecemos um script SQL\*Plus chamado `object_schema2.sql`, o qual cria todos os itens mostrados nesta e nas próximas seções. Você pode executar o script se estiver usando Oracle Database 9i ou superior. Depois que o script terminar, você estará conectado como `object_user2`.*

Para fazer o novo tipo herdar atributos e métodos de um tipo existente, use a palavra-chave `UNDER` ao definir o novo tipo. Nosso tipo “usuário\_executivo”, que chamaremos de `t_business_person`, usa a palavra-chave `UNDER` para herdar os atributos de `t_person`:

```
CREATE TYPE t_business_person UNDER t_person (
  title VARCHAR2(20),
  company VARCHAR2(20)
);
/
```

Nesse exemplo, `t_person` é conhecido como *supertipo* e `t_business_person` é conhecido como *subtipo*. Você pode usar `t_business_person` ao definir objetos de coluna ou tabelas de objeto. Por exemplo, a instrução a seguir cria uma tabela de objeto chamada `object_business_customers`:

```
CREATE TABLE object_business_customers OF t_business_person;
```

A instrução `INSERT` a seguir adiciona um objeto em `object_business_customers`; observe que os dois atributos `title` e `company` adicionais são fornecidos no final do construtor de `t_business_person`:

```
INSERT INTO object_business_customers VALUES (
  t_business_person(1, 'John', 'Brown', '01-FEB-1955', '800-555-1211',
    t_address('2 State Street', 'Beantown', 'MA', '12345'),
    'Manager', 'XYZ Corp'
  )
);
```

A consulta a seguir recupera esse objeto:

```
SELECT *
FROM object_business_customers
WHERE id = 1;
```

```

      ID FIRST_NAME LAST_NAME  DOB      PHONE
-----
ADDRESS (STREET, CITY, STATE, ZIP)
-----
TITLE                                COMPANY
-----
      1 John      Brown      01-FEB-55 800-555-1211
T_ADDRESS('2 State Street', 'Beantown', 'MA', '12345')
Manager                                XYZ Corp

```

A consulta a seguir chama a função `display_details()` para esse objeto:

```

SELECT o.display_details()
FROM object_business_customers o
WHERE id = 1;

O.DISPLAY_DETAILS()
-----
id=1, name=John Brown

```

Quando você chama um método, o banco de dados procura esse método primeiro no subtipo; se o método não é encontrado, o supertipo é pesquisado. Se você tiver uma hierarquia de tipos, o banco de dados procurará o método para cima na hierarquia; se o método não for encontrado, o banco de dados relatará um erro.

## USANDO UM OBJETO DE SUBTIPO NO LUGAR DE UM OBJETO DE SUPERTIPO

Nesta seção, você verá como usar um objeto de subtipo no lugar de um objeto de supertipo; isso proporciona excelente flexibilidade ao armazenar e manipular tipos relacionados. Nos exemplos, você verá como utilizar um objeto `t_business_person` (um objeto de subtipo) no lugar de um objeto `t_person` (um objeto de supertipo).

### Exemplos em SQL

A instrução a seguir cria uma tabela chamada `object_customers` de tipo `t_person`:

```
CREATE TABLE object_customers OF t_person;
```

A instrução `INSERT` a seguir adiciona um objeto `t_person` nessa tabela (o nome é Jason Bond):

```

INSERT INTO object_customers VALUES (
    t_person(1, 'Jason', 'Bond', '03-APR-1965', '800-555-1212',
    t_address('21 New Street', 'Anytown', 'CA', '12345')
    )
);

```

Nada há de incomum na instrução anterior: `INSERT` simplesmente adiciona um objeto `t_person` na tabela `object_customers`. Agora, como a tabela `object_customers` armazena objetos de tipo `t_person` e `t_person` é um supertipo de `t_business_person`, você pode armazenar um

objeto `t_business_person` em `object_customers`; a instrução `INSERT` a seguir mostra isso, adicionando um cliente chamado Steve Edwards:

```
INSERT INTO object_customers VALUES (
    t_business_person(2, 'Steve', 'Edwards', '03-MAR-1955', '800-555-1212',
        t_address('1 Market Street', 'Anytown', 'VA', '12345'),
        'Manager', 'XYZ Corp'
    )
);
```

Agora a tabela `object_customers` contém dois objetos: o objeto `t_person` adicionado anteriormente (Jason Bond) e o novo objeto `t_business_person` (Steve Edwards). A consulta a seguir recupera esses dois objetos; observe que os atributos `title` e `company` de Steve Edwards estão ausentes na saída:

```
SELECT *
FROM object_customers o;
```

	ID	FIRST_NAME	LAST_NAME	DOB	PHONE
-----					
ADDRESS(STREET, CITY, STATE, ZIP)					
-----					
	1	Jason	Bond	03-APR-65	800-555-1212
T_ADDRESS('21 New Street', 'Anytown', 'CA', '12345')					
	2	Steve	Edwards	03-MAR-55	800-555-1212
T_ADDRESS('1 Market Street', 'Anytown', 'VA', '12345')					

Você pode obter o conjunto de atributos completo de Steve Edwards usando `VALUE()` na consulta, como mostrado no exemplo a seguir. Observe os tipos diferentes dos objetos de Jason Bond (um objeto `t_person`) e Steve Edwards (um objeto `t_business_person`) e note que os atributos `title` e `company` de Steve Edwards agora aparecem na saída:

```
SELECT VALUE(o)
FROM object_customers o;
```

VALUE(O) (ID, FIRST_NAME, LAST_NAME, DOB, PHONE, ADDRESS(STREET, CITY, STATE, ZIP)
-----
T_PERSON(1, 'Jason', 'Bond', '03-APR-65', '800-555-1212', T_ADDRESS('21 New Street', 'Anytown', 'CA', '12345'))
T_BUSINESS_PERSON(2, 'Steve', 'Edwards', '03-MAR-55', '800-555-1212', T_ADDRESS('1 Market Street', 'Anytown', 'VA', '12345'), 'Manager', 'XYZ Corp')

Exemplos em PL/SQL

Você também pode manipular objetos de subtipo e de supertipo em PL/SQL. Por exemplo, a procedure a seguir, chamada `subtypes_and_supertypes()`, manipula objetos `t_business_person` e `t_person`:

```
CREATE PROCEDURE subtypes_and_supertypes AS
    -- cria objetos
```



```

v_business_person t_business_person :=
  t_business_person(
    1, 'John', 'Brown',
    '01-FEB-1955', '800-555-1211',
    t_address('2 State Street', 'Beantown', 'MA', '12345'),
    'Manager', 'XYZ Corp'
  );
v_person t_person :=
  t_person(1, 'John', 'Brown', '01-FEB-1955', '800-555-1211',
    t_address('2 State Street', 'Beantown', 'MA', '12345'));
v_business_person2 t_business_person;
v_person2 t_person;
BEGIN
  -- atribui v_business_person a v_person2
  v_person2 := v_business_person;
  DBMS_OUTPUT.PUT_LINE('v_person2.id = ' || v_person2.id);
  DBMS_OUTPUT.PUT_LINE('v_person2.first_name = ' ||
    v_person2.first_name);
  DBMS_OUTPUT.PUT_LINE('v_person2.last_name = ' ||
    v_person2.last_name);

  -- as linhas a seguir não serão compiladas, pois v_person2
  -- é de tipo t_person e t_person não conhece os
  -- atributos title e company adicionais
  -- DBMS_OUTPUT.PUT_LINE('v_person2.title = ' ||
  --   v_person2.title);
  -- DBMS_OUTPUT.PUT_LINE('v_person2.company = ' ||
  --   v_person2.company);

  -- a linha a seguir não será compilada, pois você não pode
  -- atribuir diretamente um objeto t_person a um objeto
  -- t_business_person
  -- v_business_person2 := v_person;
END subtypes_and_supertypes;
/

```

O exemplo a seguir mostra o resultado da chamada de `subtypes_and_supertypes()`:

```

SET SERVEROUTPUT ON
CALL subtypes_and_supertypes();
v_person2.id = 1
v_person2.first_name = John
v_person2.last_name = Brown

```

## Objetos NOT SUBSTITUTABLE

Se quiser impedir o uso de um objeto de subtipo no lugar de um objeto de supertipo, marque uma tabela de objeto ou uma coluna de objeto como “insubstituível”. Por exemplo, a instrução a seguir cria uma tabela chamada `object_customers2`:

```

CREATE TABLE object_customers_not_subs OF t_person
NOT SUBSTITUTABLE AT ALL LEVELS;

```

A cláusula `NOT SUBSTITUTABLE AT ALL LEVELS` indica que nenhum objeto de um tipo que não seja `t_person` pode ser inserido na tabela. Se for feita uma tentativa de adicionar um objeto de tipo `t_business_person` nessa tabela, um erro será retornado:

```
SQL> INSERT INTO object_customers_not_subs VALUES (
2   t_business_person(1, 'Steve', 'Edwards', '03-MAR-1955', '800-555-1212',
3   t_address('1 Market Street', 'Anytown', 'VA', '12345'),
4   'Manager', 'XYZ Corp'
5   )
6 );
t_business_person(1, 'Steve', 'Edwards', '03-MAR-1955', '800-555-1212',
*
ERROR at line 2:
ORA-00932: inconsistent datatypes: expected OBJECT_USER2.T_PERSON got
OBJECT_USER2.T_BUSINESS_PERSON
```

Você também pode marcar uma coluna de objeto como insubstituível. Por exemplo, a instrução a seguir cria uma tabela com uma coluna de objeto chamada `product` que só pode armazenar objetos de tipo `t_product`:

```
CREATE TABLE products (
  product          t_product,
  quantity_in_stock INTEGER
)
COLUMN product NOT SUBSTITUTABLE AT ALL LEVELS;
```

Todas as tentativas de adicionar um objeto que não seja de tipo `t_product` na coluna `product` resultarão em um erro.

## OUTRAS FUNÇÕES DE OBJETO ÚTEIS

Nas seções anteriores deste capítulo, você aprendeu a usar as das funções `REF()`, `DEREF()` e `VALUE()`. Nesta seção, você verá as seguintes funções adicionais que podem ser usadas com objetos:

- `IS OF()` verifica se um objeto é de um tipo ou subtipo específico
- `TREAT()` realiza uma verificação em tempo de execução para ver se o tipo de um objeto pode ser tratado como um supertipo
- `SYS_TYPEID()` retorna a identificação do tipo de um objeto

### IS OF()

Você usa `IS OF()` para verificar se um objeto é de um tipo ou subtipo específico. Por exemplo, a consulta a seguir usa `IS OF()` para verificar se os objetos da tabela `object_business_customers` são de tipo `t_business_person` — como eles são, uma linha é retornada pela consulta:

```
SELECT VALUE(o)
FROM object_business_customers o
WHERE VALUE(o) IS OF (t_business_person);

VALUE(O) (ID, FIRST_NAME, LAST_NAME, DOB, PHONE,
ADDRESS (STREET, CITY, STATE, ZIP
-----
```

```
T_BUSINESS_PERSON(1, 'John', 'Brown', '01-FEB-55', '800-555-1211',
  T_ADDRESS('2 State Street', 'Beantown', 'MA', '12345'),
  'Manager', 'XYZ Corp')
```

Você também pode usar `IS OF()` para verificar se um objeto é de um subtipo do tipo especificado. Por exemplo, os objetos da tabela `object_business_customers` são de tipo `t_business_person`, que é um subtipo de `t_person`; portanto, a consulta a seguir retorna o mesmo resultado mostrado no exemplo anterior:

```
SELECT VALUE(o)
FROM object_business_customers o
WHERE VALUE(o) IS OF (t_person);

VALUE(O) (ID, FIRST_NAME, LAST_NAME, DOB, PHONE,
  ADDRESS(STREET, CITY, STATE, ZIP)
-----
T_BUSINESS_PERSON(1, 'John', 'Brown', '01-FEB-55', '800-555-1211',
  T_ADDRESS('2 State Street', 'Beantown', 'MA', '12345'),
  'Manager', 'XYZ Corp')
```

Você pode incluir mais de um tipo em `IS OF()`, por exemplo:

```
SELECT VALUE(o)
FROM object_business_customers o
WHERE VALUE(o) IS OF (t_business_person, t_person);

VALUE(O) (ID, FIRST_NAME, LAST_NAME, DOB, PHONE,
  ADDRESS(STREET, CITY, STATE, ZIP)
-----
T_BUSINESS_PERSON(1, 'John', 'Brown', '01-FEB-55', '800-555-1211',
  T_ADDRESS('2 State Street', 'Beantown', 'MA', '12345'),
  'Manager', 'XYZ Corp')
```

Na seção “Usando um objeto de subtipo no lugar de um objeto de supertipo”, você viu a adição de um objeto `t_person` (Jason Bond) e de um objeto `t_business_person` (Steve Edwards) na tabela `object_customers`. Como lembrete, a consulta a seguir mostra esses objetos:

```
SELECT VALUE(o)
FROM object_customers o;

VALUE(O) (ID, FIRST_NAME, LAST_NAME, DOB, PHONE,
  ADDRESS(STREET, CITY, STATE, ZIP)
-----
T_PERSON(1, 'Jason', 'Bond', '03-APR-65', '800-555-1212',
  T_ADDRESS('21 New Street', 'Anytown', 'CA', '12345'))

T_BUSINESS_PERSON(2, 'Steve', 'Edwards', '03-MAR-55', '800-555-1212',
  T_ADDRESS('1 Market Street', 'Anytown', 'VA', '12345'),
  'Manager', 'XYZ Corp')
```

Como o tipo `t_business_person` é um subtipo de `t_person`, `IS OF (t_person)` retorna `true` quando um objeto `t_business_person` ou um objeto `t_person` é verificado. Isso está

ilustrado na consulta a seguir, que recupera Jason Bond e Steve Edwards usando IS OF (t\_person):

```
SELECT VALUE(o)
FROM object_customers o
WHERE VALUE(o) IS OF (t_person);

VALUE(O) (ID, FIRST_NAME, LAST_NAME, DOB, PHONE,
ADDRESS(STREET, CITY, STATE, ZIP
-----
T_PERSON(1, 'Jason', 'Bond', '03-APR-65', '800-555-1212',
T_ADDRESS('21 New Street', 'Anytown', 'CA', '12345'))

T_BUSINESS_PERSON(2, 'Steve', 'Edwards', '03-MAR-55', '800-555-1212',
T_ADDRESS('1 Market Street', 'Anytown', 'VA', '12345'),
'Manager', 'XYZ Corp')
```

Você também pode usar a palavra-chave ONLY em conjunto com IS OF () para verificar a existência de objetos somente de um tipo específico: IS OF () retorna false para objetos de outro tipo na hierarquia. Por exemplo, IS OF (ONLY t\_person) retorna true somente para objetos de tipo t\_person e retorna false para objetos de tipo t\_business\_person. Desse modo, você pode usar IS OF (ONLY t\_person) para restringir o retorno de uma consulta na tabela object\_customers ao objeto Jason Bond, como mostrado a seguir:

```
SELECT VALUE(o)
FROM object_customers o
WHERE VALUE(o) IS OF (ONLY t_person);

VALUE(O) (ID, FIRST_NAME, LAST_NAME, DOB, PHONE,
ADDRESS(STREET, CITY, STATE, ZIP
-----
T_PERSON(1, 'Jason', 'Bond', '03-APR-65', '800-555-1212',
T_ADDRESS('21 New Street', 'Anytown', 'CA', '12345'))
```

Da mesma forma, IS OF (ONLY t\_business\_person) retorna true somente para objetos de tipo t\_business\_person e retorna false para objetos de tipo t\_person. Por exemplo, a consulta a seguir recupera somente o objeto t\_business\_person e, portanto, Steve Edwards é retornado:

```
SELECT VALUE(o)
FROM object_customers o
WHERE VALUE(o) IS OF (ONLY t_business_person);

VALUE(O) (ID, FIRST_NAME, LAST_NAME, DOB, PHONE,
ADDRESS(STREET, CITY, STATE, ZIP
-----
T_BUSINESS_PERSON(2, 'Steve', 'Edwards', '03-MAR-55', '800-555-1212',
T_ADDRESS('1 Market Street', 'Anytown', 'VA', '12345'),
'Manager', 'XYZ Corp')
```

Você pode incluir vários tipos após ONLY. Por exemplo, IS OF (ONLY t\_person, t\_business\_person) retorna true somente para objetos t\_person e t\_business\_person. A consulta a seguir mostra isso retornando, conforme o esperado, tanto Jason Bond como Steve Edwards:

```

SELECT VALUE(o)
FROM object_customers o
WHERE VALUE(o) IS OF (ONLY t_person, t_business_person);

VALUE(O) (ID, FIRST_NAME, LAST_NAME, DOB, PHONE,
ADDRESS(STREET, CITY, STATE, ZIP
-----
T_PERSON(1, 'Jason', 'Bond', '03-APR-65', '800-555-1212',
T_ADDRESS('21 New Street', 'Anytown', 'CA', '12345'))

T_BUSINESS_PERSON(2, 'Steve', 'Edwards', '03-MAR-55', '800-555-1212',
T_ADDRESS('1 Market Street', 'Anytown', 'VA', '12345'),
'Manager', 'XYZ Corp')

```

Você também pode usar `IS OF()` em PL/SQL. Por exemplo, a procedure a seguir, chamada `check_types()`, cria objetos `t_business_person` e `t_person` e utiliza `IS OF()` para verificar seus tipos:

```

CREATE PROCEDURE check_types AS
-- cria objetos
v_business_person t_business_person :=
  t_business_person(
    1, 'John', 'Brown',
    '01-FEB-1955', '800-555-1211',
    t_address('2 State Street', 'Beantown', 'MA', '12345'),
    'Manager', 'XYZ Corp'
  );
v_person t_person :=
  t_person(1, 'John', 'Brown', '01-FEB-1955', '800-555-1211',
    t_address('2 State Street', 'Beantown', 'MA', '12345'));
BEGIN
-- verifica os tipos dos objetos
IF v_business_person IS OF (t_business_person) THEN
  DBMS_OUTPUT.PUT_LINE('v_business_person is of type ' ||
    't_business_person');
END IF;
IF v_person IS OF (t_person) THEN
  DBMS_OUTPUT.PUT_LINE('v_person is of type t_person');
END IF;
IF v_business_person IS OF (t_person) THEN
  DBMS_OUTPUT.PUT_LINE('v_business_person is of type t_person');
END IF;
IF v_business_person IS OF (t_business_person, t_person) THEN
  DBMS_OUTPUT.PUT_LINE('v_business_person is of ' ||
    'type t_business_person or t_person');
END IF;
IF v_business_person IS OF (ONLY t_business_person) THEN
  DBMS_OUTPUT.PUT_LINE('v_business_person is of only ' ||
    'type t_business_person');
END IF;
IF v_business_person IS OF (ONLY t_person) THEN
  DBMS_OUTPUT.PUT_LINE('v_business_person is of only ' ||

```

```

        'type t_person');
    ELSE
        DBMS_OUTPUT.PUT_LINE('v_business_person is not of only ' ||
        'type t_person');
    END IF;
END check_types;
/

```

O exemplo a seguir mostra o resultado da chamada de `check_types()`:

```

SET SERVEROUTPUT ON
CALL check_types();
v_business_person is of type t_business_person
v_person is of type t_person
v_business_person is of type t_person
v_business_person is of type t_business_person or t_person
v_business_person is of only type t_business_person
v_business_person is not of only type t_person

```

## TREAT()

Você usa `TREAT()` para realizar uma verificação em tempo de execução a fim de ver se um objeto de um subtipo pode ser tratado como um objeto de um supertipo. Se assim for, `TREAT()` retornará um objeto; caso contrário, `TREAT()` retornará nulo. Por exemplo, como `t_business_person` é um subtipo de `t_person`, um objeto `t_business_person` pode ser tratado como um objeto `t_person`. Você viu isso na seção “Usando um objeto de subtipo no lugar de um objeto de supertipo”, onde um objeto `t_business_person` (Steve Edwards) foi inserido na tabela `object_customers`, que normalmente contém objetos `t_person`. A consulta a seguir usa `TREAT()` para verificar se Steve Edwards pode ser tratado como um objeto `t_person`:

```

SELECT NVL2(TREAT(VALUE(o) AS t_person), 'yes', 'no')
FROM object_customers o
WHERE first_name = 'Steve' AND last_name = 'Edwards';

NVL
---
yes

```

`NVL2()` retorna `yes`, pois `TREAT(VALUE(o) AS t_person)` retorna um objeto (isto é, não um valor nulo). Isso significa que Steve Edwards pode ser tratado como um objeto `t_person`. A próxima consulta verifica se Jason Bond (um objeto `t_person`) pode ser tratado como um objeto `t_business_person` — ele não pode, portanto, `TREAT()` retorna nulo e `NVL2()` retorna `no`:

```

SELECT NVL2(TREAT(VALUE(o) AS t_business_person), 'yes', 'no')
FROM object_customers o
WHERE first_name = 'Jason' AND last_name = 'Bond';

NVL
---
no

```

Como `TREAT()` retorna nulo para o objeto inteiro, todos os atributos individuais do objeto também são nulos. Por exemplo, a consulta a seguir tenta acessar o atributo `first_name` por meio de Jason Bond e é retornado nulo (conforme o esperado):

```
SELECT
  NVL2(TREAT(VALUE(o) AS t_business_person).first_name, 'not null', 'null')
FROM object_customers o
WHERE first_name = 'Jason' AND last_name = 'Bond';

NVL2
----
null
```

A próxima consulta usa `TREAT()` para verificar se Jason Bond pode ser tratado como um objeto `t_person` — ele é um objeto `t_person` e, portanto, `yes` é retornado:

```
SELECT NVL2(TREAT(VALUE(o) AS t_person).first_name, 'yes', 'no')
FROM object_customers o
WHERE first_name = 'Jason' AND last_name = 'Bond';

NVL
---
yes
```

Você também pode recuperar um objeto usando `TREAT()`. Por exemplo, a consulta a seguir recupera Steve Edwards:

```
SELECT TREAT(VALUE(o) AS t_business_person)
FROM object_customers o
WHERE first_name = 'Steve' AND last_name = 'Edwards';

TREAT(VALUE(O)AST_BUSINESS_PERSON)(ID, FIRST_NAME, LAST_NAME, DOB, PHONE,
ADDRESS
-----
T_BUSINESS_PERSON(2, 'Steve', 'Edwards', '03-MAR-55', '800-555-1212',
T_ADDRESS('1 Market Street', 'Anytown', 'VA', '12345'),
'Manager', 'XYZ Corp')
```

Se você tentar essa consulta com Jason Bond, será retornado nulo, conforme o esperado. Portanto, nada aparece na saída da consulta a seguir:

```
SELECT TREAT(VALUE(o) AS t_business_person)
FROM object_customers o
WHERE first_name = 'Jason' AND last_name = 'Bond';

TREAT(VALUE(O)AST_BUSINESS_PERSON)(ID, FIRST_NAME, LAST_NAME, DOB, PHONE,
ADDRESS
-----
```

Veja o uso de `TREAT()` com a tabela `object_business_customers`, que contém o objeto `t_business_person` John Brown:

```
SELECT VALUE(o)
FROM object_business_customers o;
```

```

VALUE(O) (ID, FIRST_NAME, LAST_NAME, DOB, PHONE,
ADDRESS(STREET, CITY, STATE, ZIP
-----
T_BUSINESS_PERSON(1, 'John', 'Brown', '01-FEB-55', '800-555-1211',
T_ADDRESS('2 State Street', 'Beantown', 'MA', '12345'),
'Manager', 'XYZ Corp')

```

A consulta a seguir usa `TREAT()` para verificar se John Brown pode ser tratado como um objeto `t_person`. Ele pode, pois `t_business_person` é um subtipo de `t_person`; portanto, `yes` é retornado pela consulta:

```

SELECT NVL2(TREAT(VALUE(o) AS t_person), 'yes', 'no')
FROM object_business_customers o
WHERE first_name = 'John' AND last_name = 'Brown';

NVL
---
yes

```

O exemplo a seguir mostra o objeto retornado por `TREAT()` ao se consultar a tabela `object_business_customers`; observe que você ainda obtém os atributos `title` e `company` de John Brown:

```

SELECT TREAT(VALUE(o) AS t_person)
FROM object_business_customers o;

TREAT(VALUE(O)AST_PERSON) (ID, FIRST_NAME, LAST_NAME, DOB, PHONE,
ADDRESS(STREET,
-----
T_BUSINESS_PERSON(1, 'John', 'Brown', '01-FEB-55', '800-555-1211',
T_ADDRESS('2 State Street', 'Beantown', 'MA', '12345'),
'Manager', 'XYZ Corp')

```

Você também pode usar `TREAT()` em PL/SQL. Por exemplo, a procedure a seguir, chamada `treat_example()`, ilustra o uso de `TREAT()` (você deve estudar os comentários presentes no código para entender como `TREAT()` funciona em PL/SQL):

```

CREATE PROCEDURE treat_example AS
-- cria objetos
v_business_person t_business_person :=
t_business_person(
1, 'John', 'Brown',
'01-FEB-1955', '800-555-1211',
t_address('2 State Street', 'Beantown', 'MA', '12345'),
'Manager', 'XYZ Corp'
);
v_person t_person :=
t_person(1, 'John', 'Brown', '01-FEB-1955', '800-555-1211',
t_address('2 State Street', 'Beantown', 'MA', '12345'));
v_business_person2 t_business_person;
v_person2 t_person;

```



```

BEGIN
  -- atribui v_business_person a v_person2
  v_person2 := v_business_person;
  DBMS_OUTPUT.PUT_LINE('v_person2.id = ' || v_person2.id);
  DBMS_OUTPUT.PUT_LINE('v_person2.first_name = ' ||
    v_person2.first_name);
  DBMS_OUTPUT.PUT_LINE('v_person2.last_name = ' ||
    v_person2.last_name);

  -- as linhas a seguir não serão compiladas, pois v_person2
  -- é de tipo t_person e t_person não conhece os
  -- atributos title e company adicionais
  -- DBMS_OUTPUT.PUT_LINE('v_person2.title = ' ||
  --   v_person2.title);
  -- DBMS_OUTPUT.PUT_LINE('v_person2.company = ' ||
  --   v_person2.company);

  -- usa TREAT ao atribuir v_business_person a v_person2
  DBMS_OUTPUT.PUT_LINE('Using TREAT');
  v_person2 := TREAT(v_business_person AS t_person);
  DBMS_OUTPUT.PUT_LINE('v_person2.id = ' || v_person2.id);
  DBMS_OUTPUT.PUT_LINE('v_person2.first_name = ' ||
    v_person2.first_name);
  DBMS_OUTPUT.PUT_LINE('v_person2.last_name = ' ||
    v_person2.last_name);

  -- as linhas a seguir ainda não serão compiladas, pois v_person2
  -- é de tipo t_person e t_person não conhece os
  -- atributos title e company adicionais
  -- DBMS_OUTPUT.PUT_LINE('v_person2.title = ' ||
  --   v_person2.title);
  -- DBMS_OUTPUT.PUT_LINE('v_person2.company = ' ||
  --   v_person2.company);

  -- as linhas a seguir serão compiladas, pois TREAT é usado
  DBMS_OUTPUT.PUT_LINE('v_person2.title = ' ||
    TREAT(v_person2 AS t_business_person).title);
  DBMS_OUTPUT.PUT_LINE('v_person2.company = ' ||
    TREAT(v_person2 AS t_business_person).company);

  -- a linha a seguir não será compilada, pois você não pode
  -- atribuir diretamente um objeto t_person a um objeto
  -- t_business_person
  -- v_business_person2 := v_person;

  -- a linha a seguir lança um erro de runtime, pois você não pode
  -- atribuir um objeto de supertipo (v_person) a um objeto de subtipo
  -- (v_business_person2)
  -- v_business_person2 := TREAT(v_person AS t_business_person);
END treat_example;
/

```

O exemplo a seguir mostra o resultado da chamada de `treat_example()`:

```
SET SERVEROUTPUT ON
CALL treat_example();
v_person2.id = 1
v_person2.first_name = John
v_person2.last_name = Brown
Using TREAT
v_person2.id = 1
v_person2.first_name = John
v_person2.last_name = Brown
v_person2.title = Manager
v_person2.company = XYZ Corp
```

### SYS\_TYPEID()

Você usa `SYS_TYPEID()` para obter a ID do tipo de um objeto. Por exemplo, a consulta a seguir usa `SYS_TYPEID()` para obter a ID do tipo de objeto na tabela `object_business_customers`:

```
SELECT first_name, last_name, SYS_TYPEID(VALUE(o))
FROM object_business_customers o;
```

FIRST_NAME	LAST_NAME	SY
John	Brown	02

Você pode obter os detalhes sobre os tipos definidos pelo usuário por meio da visão `user_types`. A consulta a seguir recupera os detalhes do tipo com `typeid` igual a '02' (o valor de ID retornado por `SYS_TYPEID()` anteriormente) e o valor de `type_name` de `T_BUSINESS_PERSON`:

```
SELECT typecode, attributes, methods, supertype_name
FROM user_types
WHERE typeid = '02'
AND type_name = 'T_BUSINESS_PERSON';
```

TYPECODE	ATTRIBUTES	METHODS	SUPERTYPE_NAME
OBJECT	8	1	T_PERSON

A partir da saída dessa consulta, você pode ver que o supertipo de `t_business_person` é `t_person`. Além disso, `t_business_person` tem oito atributos e um método.

### TIPOS DE OBJETO NOT INSTANTIABLE

Você pode marcar um tipo de objeto como `NOT INSTANTIABLE`, o que impede a criação de objetos desse tipo. Talvez queira marcar um tipo de objeto como `NOT INSTANTIABLE` quando usar o tipo somente como um supertipo abstrato e nunca criar objetos desse tipo. Por exemplo, você poderia criar um tipo abstrato `t_vehicle` e utilizá-lo como um supertipo para um subtipo `t_car` e para um subtipo `t_motorcycle`; então, criaria objetos `t_car` e `t_motorcycle`, mas nunca objetos `t_vehicle`.

A instrução a seguir cria um tipo chamado `t_vehicle` que é marcado como `NOT INSTANTIABLE`:

```
CREATE TYPE t_vehicle AS OBJECT (
    id      INTEGER,
    make    VARCHAR2(15),
    model   VARCHAR2(15)
) NOT FINAL NOT INSTANTIABLE;
/
```

### NOTA

O tipo `t_vehicle` também é marcado como `NOT FINAL`, pois um tipo `NOT INSTANTIABLE` não pode ser `FINAL`. Se fosse `FINAL`, você não poderia utilizá-lo como supertipo, que é o próprio objetivo de sua criação.

O exemplo a seguir cria um subtipo chamado `t_car` sob o supertipo `t_vehicle`; observe que `t_car` tem um atributo adicional chamado `convertible`, que será usado para registrar se o carro é conversível (Y para sim, N para não):

```
CREATE TYPE t_car UNDER t_vehicle (
    convertible CHAR(1)
);
/
```

O exemplo a seguir cria um subtipo chamado `t_motorcycle` sob o supertipo `t_vehicle`; observe que `t_motorcycle` tem um atributo adicional chamado `sidecar`, que será usado para registrar se a motocicleta tem sidecar (Y para sim, N para não):

```
CREATE TYPE t_motorcycle UNDER t_vehicle (
    sidecar CHAR(1)
);
/
```

O exemplo a seguir cria tabelas chamadas `vehicles`, `cars` e `motorcycles`, que são tabelas de objeto dos tipos `t_vehicle`, `t_car` e `t_motorcycle`, respectivamente:

```
CREATE TABLE vehicles OF t_vehicle;
CREATE TABLE cars OF t_car;
CREATE TABLE motorcycles OF t_motorcycle;
```

Como `t_vehicle` é `NOT INSTANTIABLE`, você não pode adicionar um objeto na tabela `vehicles`. Se tentar fazer isso, o banco de dados retornará um erro:

```
SQL> INSERT INTO vehicles VALUES (
2     t_vehicle(1, 'Toyota', 'MR2', '01-FEB-1955')
3 );
t_vehicle(1, 'Toyota', 'MR2', '01-FEB-1955')
*
ERROR at line 2:
ORA-22826: cannot construct an instance of a non instantiable type
```

Os exemplos a seguir adicionam objetos nas tabelas cars e motorcycles:

```
INSERT INTO cars VALUES (
    t_car(1, 'Toyota', 'MR2', 'Y')
);

INSERT INTO motorcycles VALUES (
    t_motorcycle(1, 'Harley-Davidson', 'V-Rod', 'N')
);
```

As consultas a seguir recuperam os objetos das tabelas cars e motorcycles:

```
SELECT *
FROM cars;
```

ID	MAKE	MODEL	C
1	Toyota	MR2	Y

```
SELECT *
FROM motorcycles;
```

ID	MAKE	MODEL	S
1	Harley-Davidson	V-Rod	N

## CONSTRUTORES DEFINIDOS PELO USUÁRIO

Assim como em outras linguagens orientadas a objetos, como Java e C#, você pode definir seus próprios construtores em PL/SQL para inicializar um novo objeto. Você pode definir seu próprio construtor para fazer coisas como configurar programaticamente os atributos de um novo objeto com valores padrão.

O exemplo a seguir cria um tipo chamado `t_person2` que declara dois métodos construtores com diferentes números de parâmetros:

```
CREATE TYPE t_person2 AS OBJECT (
    id          INTEGER,
    first_name  VARCHAR2(10),
    last_name   VARCHAR2(10),
    dob         DATE,
    phone       VARCHAR2(12),
    CONSTRUCTOR FUNCTION t_person2(
        p_id          INTEGER,
        p_first_name  VARCHAR2,
        p_last_name   VARCHAR2
    ) RETURN SELF AS RESULT,
    CONSTRUCTOR FUNCTION t_person2(
        p_id          INTEGER,
        p_first_name  VARCHAR2,
        p_last_name   VARCHAR2,
        p_dob         DATE
    ) RETURN SELF AS RESULT
);
/
```

Observe o seguinte a respeito das declarações de construtor:

- As palavras-chave `CONSTRUCTOR FUNCTION` são usadas para identificar os construtores.
- As palavras-chave `RETURN SELF AS RESULT` indicam que o objeto processado atualmente é retornado por cada construtor; `SELF` representa o objeto atual que está sendo processado. Isso significa que o construtor retorna o novo objeto que cria.
- O primeiro construtor aceita três parâmetros (`p_id`, `p_first_name` e `p_last_name`) e o segundo aceita quatro parâmetros (`p_id`, `p_first_name`, `p_last_name` e `p_dob`).

As declarações de construtor não contêm as definições de código dos construtores; as definições estão contidas no corpo do tipo, que é criado pela instrução a seguir:

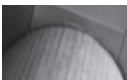
```
CREATE TYPE BODY t_person2 AS
  CONSTRUCTOR FUNCTION t_person2(
    p_id          INTEGER,
    p_first_name  VARCHAR2,
    p_last_name   VARCHAR2
  ) RETURN SELF AS RESULT IS
  BEGIN
    SELF.id := p_id;
    SELF.first_name := p_first_name;
    SELF.last_name := p_last_name;
    SELF.dob := SYSDATE;
    SELF.phone := '555-1212';
    RETURN;
  END;
  CONSTRUCTOR FUNCTION t_person2(
    p_id          INTEGER,
    p_first_name  VARCHAR2,
    p_last_name   VARCHAR2,
    p_dob         DATE
  ) RETURN SELF AS RESULT IS
  BEGIN
    SELF.id := p_id;
    SELF.first_name := p_first_name;
    SELF.last_name := p_last_name;
    SELF.dob := p_dob;
    SELF.phone := '555-1213';
    RETURN;
  END;
END;
/
```

Observe o seguinte:

- Os construtores utilizam `SELF` para referenciar o novo objeto que está sendo criado. Por exemplo, `SELF.id := p_id` configura o atributo `id` do novo objeto com o valor do parâmetro `p_id` passado no construtor.

- O primeiro construtor configura os atributos `id`, `first_name` e `last_name` com os valores de parâmetro `p_id`, `p_first_name` e `p_last_name` passados no construtor; o atributo `dob` é configurado com a data/horário atual retornada por `SYSDATE()` e o atributo `phone` é configurado como 555-1212.
- O segundo construtor configura os atributos `id`, `first_name`, `last_name` e `dob` com os valores de parâmetro `p_id`, `p_first_name`, `p_last_name` e `p_dob` passados no construtor; o atributo `phone` restante é configurado como 555-1213.

Embora não seja mostrado, o banco de dados fornece automaticamente um construtor padrão que aceita cinco parâmetros e configura cada atributo com o valor de parâmetro apropriado passado no construtor. Você verá um exemplo disso em breve.



**NOTA**

*Os construtores mostram um exemplo de sobrecarga de método, por meio da qual os métodos de mesmo nome, mas parâmetros diferentes, são definidos no mesmo tipo. Um método pode ser sobrecarregado fornecendo-se números de parâmetros, tipos de parâmetros ou ordem de parâmetros diferentes.*

O exemplo a seguir descreve `t_person2`; observe as definições do construtor na saída:

```
DESCRIBE t_person2
```

Name	Null?	Type
-----	-----	-----
ID		NUMBER (38)
FIRST_NAME		VARCHAR2 (10)
LAST_NAME		VARCHAR2 (10)
DOB		DATE
PHONE		VARCHAR2 (12)

METHOD

```
-----
FINAL CONSTRUCTOR FUNCTION T_PERSON2 RETURNS SELF AS RESULT
Argument Name      Type              In/Out Default?
-----
```

P_ID	NUMBER	IN
P_FIRST_NAME	VARCHAR2	IN
P_LAST_NAME	VARCHAR2	IN

METHOD

```
-----
FINAL CONSTRUCTOR FUNCTION T_PERSON2 RETURNS SELF AS RESULT
Argument Name      Type              In/Out Default?
-----
```

P_ID	NUMBER	IN
P_FIRST_NAME	VARCHAR2	IN
P_LAST_NAME	VARCHAR2	IN
P_DOB	DATE	IN

A instrução a seguir cria uma tabela de tipo `t_person2`:

```
CREATE TABLE object_customers2 OF t_person2;
```

A instrução `INSERT` a seguir adiciona um objeto na tabela; observe que são passados três parâmetros para o construtor `t_person2`:

```
INSERT INTO object_customers2 VALUES (
    t_person2(1, 'Jeff', 'Jones')
);
```

Como são passados três parâmetros para `t_person2`, essa instrução `INSERT` utiliza o primeiro construtor. Esse construtor configura os atributos `id`, `first_name` e `last_name` do novo objeto como 1, Jeff e Jones; os atributos `dob` e `phone` restantes são configurados com o resultado retornado por `SYSDATE()` e pela literal 555-1212. A consulta a seguir recupera o novo objeto:

```
SELECT *
FROM object_customers2
WHERE id = 1;
```

ID	FIRST_NAME	LAST_NAME	DOB	PHONE
1	Jeff	Jones	17-JUN-07	555-1212

A próxima instrução `INSERT` adiciona outro objeto na tabela; observe que são passados quatro parâmetros para o construtor `t_person2`:

```
INSERT INTO object_customers2 VALUES (
    t_person2(2, 'Gregory', 'Smith', '03-APR-1965')
);
```

Como são passados quatro parâmetros para `t_person2`, essa instrução `INSERT` utiliza o segundo construtor. Esse construtor configura os atributos `id`, `first_name`, `last_name` e `dob` do objeto como 2, Gregory, Smith e 03-APR-1965, respectivamente; o atributo `phone` restante é configurado como 555-1213. A consulta a seguir recupera o novo objeto:

```
SELECT *
FROM object_customers2
WHERE id = 2;
```

ID	FIRST_NAME	LAST_NAME	DOB	PHONE
2	Gregory	Smith	03-APR-65	555-1213

A próxima instrução `INSERT` adiciona outro objeto na tabela; observe que são passados cinco parâmetros para o construtor `t_person2`:

```
INSERT INTO object_customers2 VALUES (
    t_person2(3, 'Jeremy', 'Hill', '05-JUN-1975', '555-1214')
);
```

Como são passados cinco parâmetros para `t_person2`, essa instrução `INSERT` utiliza o construtor padrão. Esse construtor configura os atributos `id`, `first_name`, `last_name`, `dob` e `phone`

como 3, Jeremy, Hill, 05-JUN-1975 e 555-1214, respectivamente. A consulta a seguir recupera o novo objeto:

```
SELECT *
FROM object_customers2
WHERE id = 3;
```

ID	FIRST_NAME	LAST_NAME	DOB	PHONE
3	Jeremy	Hill	05-JUN-75	555-1214

## SOBRESCREVENDO MÉTODOS

Quando você cria um subtipo sob um supertipo, pode sobrescrever um método do supertipo com um método do subtipo. Isso proporciona uma maneira muito flexível de definir métodos em uma hierarquia de tipos.

As instruções a seguir criam um supertipo chamado `t_person3`; observe que a função `display_details()` retorna um `VARCHAR2` contendo os valores de atributo do objeto:

```
CREATE TYPE t_person3 AS OBJECT (
    id          INTEGER,
    first_name  VARCHAR2(10),
    last_name   VARCHAR2(10),
    MEMBER FUNCTION display_details RETURN VARCHAR2
) NOT FINAL;
/
CREATE TYPE BODY t_person3 AS
    MEMBER FUNCTION display_details RETURN VARCHAR2 IS
    BEGIN
        RETURN 'id=' || id ||
            ', name=' || first_name || ' ' || last_name;
    END;
END;
/
```

O próximo conjunto de instruções cria um subtipo chamado `t_business_person3` sob `t_person3`. Observe que a função `display_details()` é sobrescrita com o uso da palavra-chave `OVERRIDING` e que a função retorna um `VARCHAR2` contendo os valores de atributo originais e estendidos do objeto:

```
CREATE TYPE t_business_person3 UNDER t_person3 (
    title       VARCHAR2(20),
    company     VARCHAR2(20),
    OVERRIDING MEMBER FUNCTION display_details RETURN VARCHAR2
);
/
CREATE TYPE BODY t_business_person3 AS
    OVERRIDING MEMBER FUNCTION display_details RETURN VARCHAR2 IS
    BEGIN
        RETURN 'id=' || id ||
            ', name=' || first_name || ' ' || last_name ||
            ', title=' || title || ', company=' || company;
```



```
END;
END;
/
```

O uso da palavra-chave `OVERRIDING` indica que `display_details()` em `t_business_person3` sobreescreve `display_details()` em `t_person3`; portanto, quando `display_details()` em `t_business_person3` é chamada, ela chama `display_details()` em `t_business_person3` e não `display_details()` em `t_person3`.

### NOTA

*Na próxima seção deste capítulo, você verá como pode chamar diretamente um método de um supertipo a partir de um subtipo. Isso evita que tenha que recriar no subtipo um código que já está no supertipo. Você faz essa chamada direta usando um novo recurso do Oracle Database 11g, chamado invocação generalizada.*

As instruções a seguir criam uma tabela chamada `object_business_customers3` e adicionam um objeto nessa tabela:

```
CREATE TABLE object_business_customers3 OF t_business_person3;

INSERT INTO object_business_customers3 VALUES (
    t_business_person3(1, 'John', 'Brown', 'Manager', 'XYZ Corp')
);
```

O exemplo a seguir chama `display_details()` usando `object_business_customers3`:

```
SELECT o.display_details()
FROM object_business_customers3 o
WHERE id = 1;

O.DISPLAY_DETAILS()
-----
id=1, name=John Brown, title=Manager, company=XYZ Corp
```

Como a função `display_details()` definida em `t_business_person3` é chamada, o `VARCHAR2` retornado pela função contém os atributos `id`, `first_name` e `last_name`, junto com os atributos `title` e `company`.

## INVOCÇÃO GENERALIZADA

Como foi visto na seção anterior, é possível sobrescrever um método do supertipo com um método do subtipo. A *invocção generalizada* é um novo recurso do Oracle Database 11g e permite que você chame um método de um supertipo a partir de um subtipo. A invocção generalizada evita que seja necessário recriar no subtipo um código que já está no supertipo.

### NOTA

*Fornecemos um script SQL\*Plus chamado `object_schema3.sql`, que cria todos os itens mostrados no restante deste capítulo. Você só poderá executar o script `object_schema3.sql` se estiver usando o Oracle Database 11g. Depois que o script terminar, você estará conectado como `object_user3`.*

As instruções a seguir criam um supertipo chamado `t_person`; observe que a função `display_details()` retorna um `VARCHAR2` contendo os valores de atributo:

```
CREATE TYPE t_person AS OBJECT (
    id          INTEGER,
    first_name  VARCHAR2(10),
    last_name   VARCHAR2(10),
    MEMBER FUNCTION display_details RETURN VARCHAR2
) NOT FINAL;
/
CREATE TYPE BODY t_person AS
    MEMBER FUNCTION display_details RETURN VARCHAR2 IS
    BEGIN
        RETURN 'id=' || id ||
            ', name=' || first_name || ' ' || last_name;
    END;
END;
/
```

O próximo conjunto de instruções cria um subtipo chamado `t_business_person` sob `t_person`; observe que a função `display_details()` é sobrescrita com o uso da palavra-chave `OVERRIDING`:

```
CREATE TYPE t_business_person UNDER t_person (
    title VARCHAR2(20),
    company VARCHAR2(20),
    OVERRIDING MEMBER FUNCTION display_details RETURN VARCHAR2
);
/

CREATE TYPE BODY t_business_person AS
    OVERRIDING MEMBER FUNCTION display_details RETURN VARCHAR2 IS
    BEGIN
        -- usa invocação generalizada para chamar display_details() em t_person
        RETURN (SELF AS t_person).display_details ||
            ', title=' || title || ', company=' || company;
    END;
END;
/
```

`display_details()` em `t_business_person` sobrescreve `display_details()` em `t_person`. A linha a seguir em `display_details()` usa invocação generalizada para chamar um método de um supertipo a partir de um subtipo:

```
RETURN (SELF AS t_person).display_details ||
    ', title=' || title || ', company=' || company;
```

O que `(SELF AS t_person).display_details` faz é tratar um objeto do tipo atual (que é `t_business_person`) como um objeto de tipo `t_person` e, então, chamar `display_details()` em `t_person`. Portanto, quando `display_details()` em `t_business_person` é chamada, ela primeiro chama `display_details()` em `t_person` (que exibe os valores de atributo `id`, `first_name` e `last_name`) e, então, exibe os valores de atributo `title` and `company`. Isso significa que não foi necessário recriar o código já existente de `t_person.display_details()` em `t_busi-`

`ness_person.display_details()`. Se você tiver métodos mais complexos em seus tipos, esse recurso poderá economizar muito trabalho e tornar seu código mais fácil de manter.

As instruções a seguir criam uma tabela chamada `object_business_customers` e adicionam um objeto nessa tabela:

```
CREATE TABLE object_business_customers OF t_business_person;

INSERT INTO object_business_customers VALUES (
    t_business_person(1, 'John', 'Brown', 'Manager', 'XYZ Corp')
);
```

A consulta a seguir chama `display_details()` usando `object_business_customers`:

```
SELECT o.display_details()
FROM object_business_customers o;

O.DISPLAY_DETAILS()
-----
id=1, name=John Brown, dob=01-FEB-55, title=Manager, company=XYZ Corp
```

Como você pode ver, são exibidos os atributos `id`, `name` e a data de nascimento (`dob`) (que vêm de `display_details()` em `t_person`), seguidos dos atributos `title` e `company` (que vêm de `display_details()` em `t_business_person`).

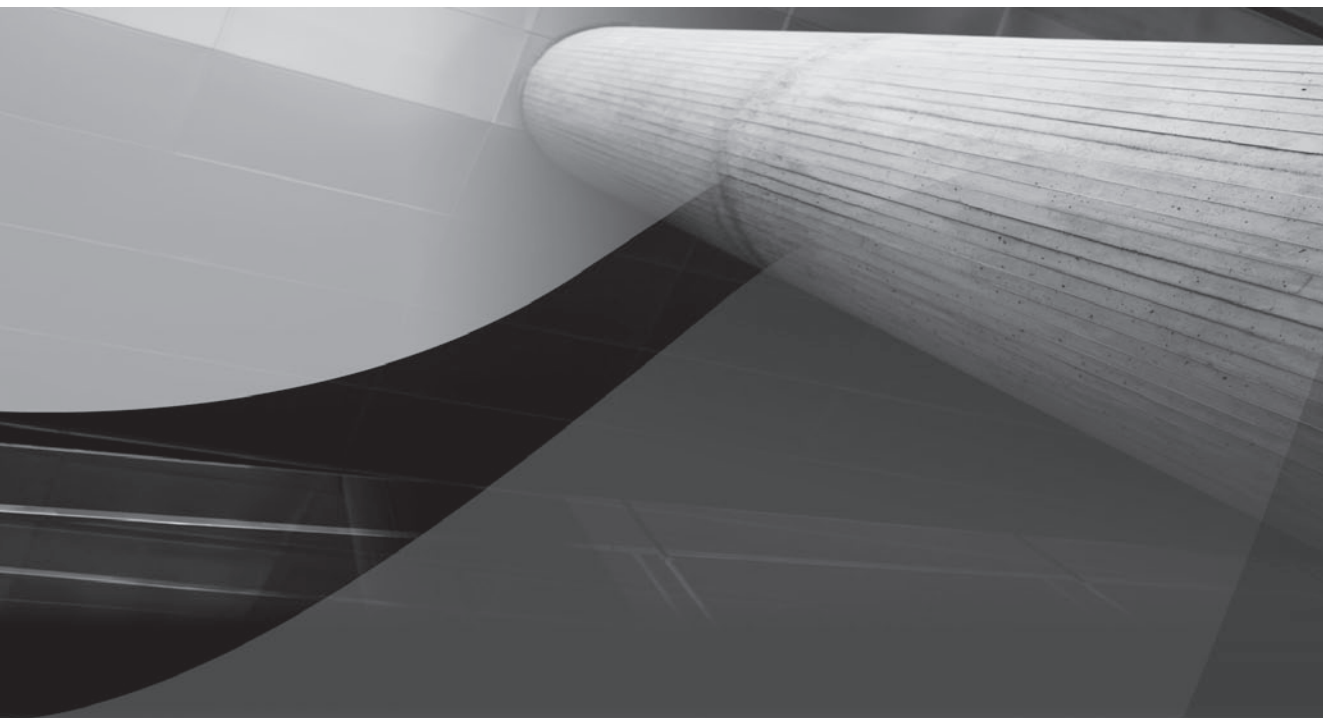
## RESUMO

Neste capítulo, você aprendeu que:

- O banco de dados Oracle permite criar tipos de objeto. Um tipo de objeto é como uma classe em Java, C++ e C# e pode conter atributos e métodos. Você cria um tipo de objeto usando a instrução `CREATE TYPE`.
- Você pode usar um tipo de objeto para definir um objeto de coluna ou uma tabela de objeto.
- Você pode usar uma referência de objeto para acessar uma linha individual em uma tabela de objeto. Uma referência de objeto é semelhante a um ponteiro em C++.
- Você pode criar e manipular objetos em SQL e em PL/SQL.
- Com a versão 9i do Oracle Database, você pode usar herança de tipo de objeto. Isso permite definir hierarquias de tipos de banco de dados.
- Você pode usar um objeto de subtipo no lugar de um objeto de supertipo, o que proporciona excelente flexibilidade ao armazenar e manipular tipos relacionados. Se quiser impedir o uso de um objeto de subtipo no lugar do objeto de supertipo, você pode marcar uma tabela de objeto ou uma coluna de objeto como `NOT SUBSTITUTABLE`.
- Você pode usar várias funções úteis com objetos, como `REF()`, `DEREF()`, `VALUE()`, `IS OF()`, `SYS_TYPEID()` e `TREAT()`.

- Você pode marcar um tipo de objeto como `NOT INSTANTIABLE`, o que impede a criação de objetos desse tipo. Você desejará marcar um tipo de objeto como `NOT INSTANTIABLE` quando usar esse tipo como um supertipo abstrato e nunca criar realmente objetos desse tipo.
- Você pode definir seus próprios construtores para fazer coisas como configurar programaticamente um padrão para atributos de um objeto.
- Você pode sobrescrever um método de um supertipo com um método de um subtipo, proporcionando uma maneira muito flexível de definir métodos em uma hierarquia de tipos.
- Você pode usar o novo recurso de invocação generalizada do Oracle Database 11g para chamar métodos do supertipo a partir de um subtipo. Isso evita muito trabalho e torna seu código mais fácil de manter.

No próximo capítulo, você vai aprender sobre coleções.



# CAPÍTULO 13

Coleções

Neste capítulo, você vai aprender:

- Sobre coleções
- A criar tipos de coleção
- A usar tipos de coleção para definir colunas em tabelas
- A criar e manipular dados de coleção em SQL e em PL/SQL
- Como uma coleção pode conter coleções incorporadas (uma coleção “de vários níveis”)
- A examinar as melhorias nas coleções introduzidas no Oracle Database 10g

## INTRODUÇÃO ÀS COLEÇÕES

O Oracle Database 8 introduziu dois novos tipos de banco de dados, conhecidos como *coleções*, que permitem armazenar conjuntos de elementos. O Oracle Database 9i ampliou esses recursos para incluir coleções de vários níveis, que permitem incorporar uma coleção dentro de outra. O Oracle Database 10g ampliou ainda mais as coleções para incluir arrays associativos e muito mais.

Existem três tipos de coleções:

- **Varrays** Um varray é semelhante a um array em Java, C++ e C#. Um varray armazena um conjunto de elementos ordenados e cada elemento tem um índice que registra sua posição no array. Os elementos de um varray só podem ser modificados como um todo e não individualmente; isso significa que, mesmo que você queira modificar apenas um elemento, deve fornecer todos os elementos do varray. Um varray tem um tamanho máximo que você configura ao criá-lo, mas é possível alterá-lo posteriormente.
- **Tabelas aninhadas** Uma tabela aninhada é um tabela incorporada dentro de outra. Você pode inserir, atualizar e excluir elementos individuais em uma tabela aninhada; isso as torna mais flexíveis do que um varray, cujos elementos só podem ser modificados como um todo. Uma tabela aninhada não tem um tamanho máximo e você pode armazenar um número arbitrário de elementos nela.
- **Arrays associativos (anteriormente conhecidos como tabelas “index-by”)** Um array associativo é semelhante a uma tabela de hashing em Java. Introduzido no Oracle Database 10g, um array associativo é um conjunto de pares de chave e valor. Você pode obter o valor do array usando a chave (que pode ser uma string) ou um número inteiro que especifica a posição do valor no array. Um array associativo só pode ser usado em PL/SQL e não pode ser armazenado no banco de dados.

Você pode estar se perguntando por que deveria utilizar coleções. Afinal, usar duas tabelas com uma chave estrangeira já permite modelar relações entre dados. A resposta é que as coleções seguem o estilo orientado a objetos da programação moderna; além disso, os dados armazenados na coleção podem ser acessados mais rapidamente pelo banco de dados do que se duas tabelas relacionais fossem utilizadas para armazená-los.

Fornecemos um script SQL\*Plus chamado `collection_schema.sql` no diretório SQL. O script cria um usuário chamado `collection_user` com a senha `collection_password` e cria os tipos de coleção, as tabelas e o código PL/SQL utilizados na primeira parte deste capítulo. Você deve executar esse script enquanto estiver conectado como um usuário com os privilégios necessá-

rios para criar um novo usuário com os privilégios `CONNECT` e `RESOURCE`; para executar os scripts, você pode se conectar como o usuário `system` no banco de dados. Depois que o script terminar, você estará conectado como `collection_user`.

## CRIANDO TIPOS DE COLEÇÃO

Nesta seção, você verá como criar um tipo de varray e um tipo de tabela aninhada.

### Criando um tipo de varray

Um varray armazena um conjunto de elementos ordenados, todos do mesmo tipo, e o tipo pode ser interno do banco de dados ou um tipo de objeto definido pelo usuário. Cada elemento tem um índice correspondente à sua posição no array e você só pode modificar elementos no varray como um todo.

Você cria um tipo de varray com a instrução `CREATE TYPE`, na qual especifica o tamanho máximo e o tipo de elementos armazenados no varray. O exemplo a seguir cria um tipo chamado `t_varray_address` que pode armazenar até três strings `VARCHAR2`:

```
CREATE TYPE t_varray_address AS VARRAY(3) OF VARCHAR2(50);
/
```

Cada string `VARCHAR2` será usada para representar um endereço diferente para um cliente da nossa loja de exemplo. No Oracle Database 10g e em versões superiores, você pode alterar o número máximo de elementos de um varray usando a instrução `ALTER TYPE`. Por exemplo, a instrução a seguir altera o número máximo de elementos para dez:

```
ALTER TYPE t_varray_address MODIFY LIMIT 10 CASCADE;
```

A opção `CASCADE` propaga a alteração para todos os objetos dependentes no banco de dados.

### Criando um tipo de tabela aninhada

Uma tabela aninhada armazena um conjunto desordenado de qualquer número de elementos. É possível inserir, atualizar e excluir elementos individuais em uma tabela aninhada. Como ela não tem tamanho máximo, você pode armazenar um número arbitrário de elementos.

Nesta seção, você verá um tipo de tabela aninhada que armazena tipos de objeto `t_address`. Você aprendeu sobre `t_address` no capítulo anterior; ele é usado para representar um endereço e é definido como segue:

```
CREATE TYPE t_address AS OBJECT (
    street VARCHAR2(15),
    city   VARCHAR2(15),
    state  CHAR(2),
    zip    VARCHAR2(5)
);
/
```

Você cria um tipo de tabela aninhada com a instrução `CREATE TYPE`. O exemplo a seguir cria um tipo chamado `t_nested_table_address` que armazena objetos `t_address`:

```
CREATE TYPE t_nested_table_address AS TABLE OF t_address;
/
```

Note que você não especifica o tamanho máximo de uma tabela aninhada. Isso porque esta pode armazenar qualquer número de elementos.

## USANDO UM TIPO DE COLEÇÃO PARA DEFINIR UMA COLUNA EM UMA TABELA

Uma vez que você tenha criado um tipo de coleção, pode usá-lo para definir uma coluna em uma tabela. Você vai ver como utilizar o tipo de varray e o tipo de tabela aninhada criados na seção anterior para definir uma coluna em uma tabela.

### Usando um tipo de varray para definir uma coluna em uma tabela

A instrução a seguir cria uma tabela chamada `customers_with_varray`, que utiliza `t_varray_address` para definir uma coluna chamada `addresses`:

```
CREATE TABLE customers_with_varray (
    id            INTEGER PRIMARY KEY,
    first_name    VARCHAR2(10),
    last_name     VARCHAR2(10),
    addresses     t_varray_address
);
```

Os elementos de um varray são armazenados diretamente dentro da tabela, quando o tamanho do varray é de 4KB ou menos; caso contrário, o varray é armazenado fora da tabela. Quando um varray é armazenado com a tabela, o acesso aos seus elementos é mais rápido do que o acesso aos elementos de uma tabela aninhada.

### Usando um tipo de tabela aninhada para definir uma coluna em uma tabela

A instrução a seguir cria uma tabela chamada `customers_with_nested_table`, que usa `t_nested_table_address` para definir uma coluna chamada `addresses`:

```
CREATE TABLE customers_with_nested_table (
    id            INTEGER PRIMARY KEY,
    first_name    VARCHAR2(10),
    last_name     VARCHAR2(10),
    addresses     t_nested_table_address
)
NESTED TABLE
    addresses
STORE AS
    nested_addresses;
```

A cláusula `NESTED TABLE` identifica o nome da coluna da tabela aninhada (`addresses`, no exemplo) e a cláusula `STORE AS` especifica o nome da tabela aninhada (`nested_addresses`, no exemplo) onde os elementos são armazenados. Você não pode acessar a tabela aninhada independentemente da tabela na qual ela está incorporada.



## OBTENDO INFORMAÇÕES SOBRE COLEÇÕES

Conforme verá nesta seção, você pode usar o comando `DESCRIBE` e algumas visões de usuário para obter informações sobre suas coleções.

### Obtendo informações sobre um varray

O exemplo a seguir descreve `t_varray_address`:

```
DESCRIBE t_varray_address
t_varray_address VARRAY(3) OF VARCHAR2(50)
```

O próximo exemplo descreve a tabela `customers_with_varray`, cuja coluna `addresses` é do tipo `t_varray_address`:

```
DESCRIBE customers_with_varray
Name                               Null?    Type
-----
ID                                 NOT NULL NUMBER(38)
FIRST_NAME                        VARCHAR2(10)
LAST_NAME                         VARCHAR2(10)
ADDRESSES                         T_VARRAY_ADDRESS
```

Você também pode obter informações sobre seus varrays a partir da visão `user_varrays`. A Tabela 13-1 descreve as colunas de `user_varrays`.

**Tabela 13-1** Colunas da visão `user_varrays`

Coluna	Tipo	Descrição
parent_table_name	VARCHAR2(30)	Nome da tabela que contém o varray.
parent_table_column	VARCHAR2(4000)	Nome da coluna na tabela pai que contém o varray.
type_owner	VARCHAR2(30)	Usuário que possui o tipo de varray.
type_name	VARCHAR2(30)	Nome do tipo de varray.
lob_name	VARCHAR2(30)	Nome do large object (LOB), quando o varray é armazenado em um LOB. Você vai aprender sobre LOBs no próximo capítulo.
storage_spec	VARCHAR2(30)	Especificação de armazenamento do varray.
return_type	VARCHAR2(20)	Tipo de retorno da coluna.
element_substitutable	VARCHAR2(25)	Se o elemento do varray pode ser substituído por um subtipo ou não (Y/N).



**NOTA**

Obtenha informações sobre todos os varrays a que tem acesso usando a visão `all_varrays`.

O exemplo a seguir recupera os detalhes de `t_varray_address` a partir de `user_varrays`:

```
SELECT parent_table_name, parent_table_column, type_name
FROM user_varrays
WHERE type_name = 'T_VARRAY_ADDRESS';

PARENT_TABLE_NAME
-----
PARENT_TABLE_COLUMN
-----
TYPE_NAME
-----
CUSTOMERS_WITH_VARRAY
ADDRESSES
T_VARRAY_ADDRESS
```

**Obtendo informações sobre uma tabela aninhada**

Você também pode usar `DESCRIBE` com uma tabela aninhada, como mostra o exemplo a seguir, que descreve `t_nested_table_address`:

```
DESCRIBE t_nested_table_address
t_nested_table_address TABLE OF T_ADDRESS
Name                                     Null?      Type
-----
STREET                                  VARCHA2(15)
CITY                                    VARCHA2(15)
STATE                                   CHAR(2)
ZIP                                     VARCHA2(5)
```

O próximo exemplo descreve a tabela `customers_with_nested_table`, cuja coluna `addresses` é de tipo `t_nested_table_address`:

```
DESCRIBE customers_with_nested_table
Name                                     Null?      Type
-----
ID                                     NOT NULL  NUMBER(38)
FIRST_NAME                             VARCHA2(10)
LAST_NAME                              VARCHA2(10)
ADDRESSES                              T_NESTED_TABLE_ADDRESS
```

Se você configurar a profundidade como 2 e descrever `customers_with_nested_table`, poderá ver os atributos que constituem `t_nested_table_address`:

```
SET DESCRIBE DEPTH 2
DESCRIBE customers_with_nested_table
Name                                     Null?      Type
-----
ID                                     NOT NULL  NUMBER(38)
FIRST_NAME                             VARCHA2(10)
```

LAST_NAME	VARCHAR2 (10)
ADDRESSES	T_NESTED_TABLE_ADDRESS
STREET	VARCHAR2 (15)
CITY	VARCHAR2 (15)
STATE	CHAR (2)
ZIP	VARCHAR2 (5)

Você também pode obter informações sobre suas tabelas aninhadas a partir da visão `user_nested_tables`. A Tabela 13-2 descreve as colunas de `user_nested_tables`.



**NOTA**

Você pode obter informações sobre todas as tabelas aninhadas a que tem acesso usando a visão `all_nested_tables`.

O exemplo a seguir recupera os detalhes da tabela `nested_addresses` a partir de `user_nested_tables`:



```
SELECT table_name, table_type_name, parent_table_name, parent_table_column
FROM user_nested_tables
WHERE table_name = 'NESTED_ADDRESSES';
```

TABLE_NAME	TABLE_TYPE_NAME
-----	-----
PARENT_TABLE_NAME	
-----	
PARENT_TABLE_COLUMN	
-----	-----
NESTED_ADDRESSES	T_NESTED_TABLE_ADDRESS
CUSTOMERS_WITH_NESTED_Tabela	
ADDRESSES	

**Tabela 13-2** Colunas da visão `user_nested_tables`

Coluna	Tipo	Descrição
table_name	VARCHAR2 (30)	Nome da tabela aninhada
table_type_owner	VARCHAR2 (30)	Usuário que possui o tipo de tabela aninhada
table_type_name	VARCHAR2 (30)	Nome do tipo de tabela aninhada
parent_table_name	VARCHAR2 (30)	Nome da tabela pai que contém a tabela aninhada
parent_table_column	VARCHAR2 (4000)	Nome da coluna na tabela pai que contém a tabela aninhada
storage_spec	VARCHAR2 (30)	Especificação de armazenamento da tabela aninhada
return_type	VARCHAR2 (20)	Tipo de retorno da coluna
element_substitutable	VARCHAR2 (25)	Se o elemento da tabela aninhada pode ser substituído por um subtipo ou não (Y/N).

## PREENCHENDO UMA COLEÇÃO COM ELEMENTOS

Nesta seção, você vai aprender a preencher um varray e uma tabela aninhada com elementos usando instruções INSERT. Não é necessário executar as instruções INSERT mostradas nesta seção: elas são executadas quando você executa o script `collection_schema.sql`.

### Preenchendo um varray com elementos

As instruções INSERT a seguir adicionam linhas na tabela `customers_with_varray`; observe o uso do construtor `t_varray_address` para especificar as strings dos elementos do varray:

```
INSERT INTO customers_with_varray VALUES (
    1, 'Steve', 'Brown',
    t_varray_address(
        '2 State Street, Beantown, MA, 12345',
        '4 Hill Street, Lost Town, CA, 54321'
    )
);

INSERT INTO customers_with_varray VALUES (
    2, 'John', 'Smith',
    t_varray_address(
        '1 High Street, Newtown, CA, 12347',
        '3 New Street, Anytown, MI, 54323',
        '7 Market Street, Main Town, MA, 54323'
    )
);
```

Como você pode ver, a primeira linha tem dois endereços e a segunda tem três. Qualquer número de endereços pode ser armazenado, até o limite máximo do varray.

### Preenchendo uma tabela aninhada com elementos

As instruções INSERT a seguir adicionam linhas em `customers_with_nested_table`; observe o uso dos construtores `t_nested_table_address` e `t_address` para especificar os elementos da tabela aninhada:

```
INSERT INTO customers_with_nested_table VALUES (
    1, 'Steve', 'Brown',
    t_nested_table_address(
        t_address('2 State Street', 'Beantown', 'MA', '12345'),
        t_address('4 Hill Street', 'Lost Town', 'CA', '54321')
    )
);

INSERT INTO customers_with_nested_table VALUES (
    2, 'John', 'Smith',
    t_nested_table_address(
```

```

    t_address('1 High Street', 'Newtown', 'CA', '12347'),
    t_address('3 New Street', 'Anytown', 'MI', '54323'),
    t_address('7 Market Street', 'Main Town', 'MA', '54323')
  )
);

```

Como você pode ver, a primeira linha tem dois endereços e a segunda tem três. Qualquer número de endereços pode ser armazenado em uma tabela aninhada.

## RECUPERANDO ELEMENTOS DE COLEÇÕES

Nesta seção, você vai aprender a recuperar elementos de um varray e de uma tabela aninhada usando consultas. A saída das consultas foi formatada para tornar os resultados mais legíveis.

### Recuperando elementos de um varray

A consulta a seguir recupera o cliente nº 1 da tabela `customers_with_varray`; uma linha é retornada e ela contém os dois endereços armazenados no varray:

```

SELECT *
FROM customers_with_varray
WHERE id = 1;

      ID FIRST_NAME LAST_NAME
-----
ADDRESSES
-----
      1 Steve      Brown
T_VARRAY_ADDRESS('2 State Street, Beantown, MA, 12345',
'4 Hill Street, Lost Town, CA, 54321')

```

A próxima consulta especifica os nomes de coluna:

```

SELECT id, first_name, last_name, addresses
FROM customers_with_varray
WHERE id = 1;

      ID FIRST_NAME LAST_NAME
-----
ADDRESSES
-----
      1 Steve      Brown
T_VARRAY_ADDRESS('2 State Street, Beantown, MA, 12345',
'4 Hill Street, Lost Town, CA, 54321')

```

Todos esses exemplos retornam os endereços do varray como uma única linha. Na seção “Usando TABLE() para tratar uma coleção como uma série de linhas”, você verá como pode tratar os dados armazenados em uma coleção como uma série de linhas.

## Recuperando elementos de uma tabela aninhada

A consulta a seguir recupera o cliente nº 1 de `customers_with_nested_table`; uma linha é retornada e ela contém os dois endereços armazenados na tabela aninhada:

```
SELECT *
FROM customers_with_nested_table
WHERE id = 1;

      ID FIRST_NAME LAST_NAME
-----
ADDRESSES (STREET, CITY, STATE, ZIP)
-----
      1 Steve      Brown
T_NESTED_TABLE_ADDRESS(
  T_ADDRESS('2 State Street', 'Beantown', 'MA', '12345'),
  T_ADDRESS('4 Hill Street', 'Lost Town', 'CA', '54321'))
```

A próxima consulta especifica os nomes de coluna:

```
SELECT id, first_name, last_name, addresses
FROM customers_with_nested_table
WHERE id = 1;

      ID FIRST_NAME LAST_NAME
-----
ADDRESSES (STREET, CITY, STATE, ZIP)
-----
      1 Steve      Brown
T_NESTED_TABLE_ADDRESS(
  T_ADDRESS('2 State Street', 'Beantown', 'MA', '12345'),
  T_ADDRESS('4 Hill Street', 'Lost Town', 'CA', '54321'))
```

A próxima consulta obtém apenas a tabela aninhada `addresses`. Como nos exemplos anteriores, uma linha é retornada e ela contém os dois endereços armazenados na tabela aninhada:

```
SELECT addresses
FROM customers_with_nested_table
WHERE id = 1;

ADDRESSES (STREET, CITY, STATE, ZIP)
-----
T_NESTED_TABLE_ADDRESS(
  T_ADDRESS('2 State Street', 'Beantown', 'MA', '12345'),
  T_ADDRESS('4 Hill Street', 'Lost Town', 'CA', '54321'))
```

## USANDO TABLE() PARA TRATAR UMA COLEÇÃO COMO UMA SÉRIE DE LINHAS

As consultas anteriores que você viu neste capítulo retornavam o conteúdo de uma coleção como uma única linha. Às vezes você quer tratar os dados armazenados em uma coleção como uma série

de linhas; você poderia, por exemplo, estar trabalhando com um aplicativo legado que só pode usar linhas. Para tratar uma coleção como uma série de linhas, use a função `TABLE()`. Nesta seção, você verá como utilizar `TABLE()` com um varray e com uma tabela aninhada.

## Usando `TABLE()` com um varray

A consulta a seguir usa `TABLE()` para recuperar os dois endereços do cliente nº 1 da tabela `customers_with_varray`; são retornadas duas linhas separadas:

```
SELECT a.*
FROM customers_with_varray c, TABLE(c.addresses) a
WHERE id = 1;

COLUMN_VALUE
-----
2 State Street, Beantown, MA, 12345
4 Hill Street, Lost Town, CA, 54321
```

Observe como o software do banco de dados Oracle adiciona automaticamente o nome de coluna `COLUMN_VALUE` nas linhas retornadas pela consulta. `COLUMN_VALUE` é um apelido de pseudocoluna e é adicionado automaticamente quando uma coleção contém dados de um dos tipos de dados internos, como `VARCHAR2`, `CHAR`, `NUMBER` ou `DATE`. Como o exemplo de varray contém dados `VARCHAR2`, o apelido `COLUMN_VALUE` é adicionado. Se o varray contivesse dados de um tipo de objeto definido pelo usuário, então `TABLE()` retornaria objetos desse tipo e `COLUMN_VALUE` não apareceria. Você verá um exemplo disso na próxima seção.

Você também pode incorporar uma instrução `SELECT` inteira dentro de `TABLE()`. Por exemplo, a consulta a seguir reescreve o exemplo anterior, colocando uma instrução `SELECT` dentro de `TABLE()`:

```
SELECT *
FROM TABLE(
  -- obtém os endereços do cliente nº 1
  SELECT addresses
  FROM customers_with_varray
  WHERE id = 1
);

COLUMN_VALUE
-----
2 State Street, Beantown, MA, 12345
4 Hill Street, Lost Town, CA, 54321
```

A consulta a seguir mostra outro exemplo que usa `TABLE()` para obter os endereços:

```
SELECT c.id, c.first_name, c.last_name, a.*
FROM customers_with_varray c, TABLE(c.addresses) a
WHERE id = 1;

ID FIRST_NAME LAST_NAME
-----
1 Steve Brown
```

```
2 State Street, Beantown, MA, 12345
      1 Steve      Brown
4 Hill Street, Lost Town, CA, 54321
```

## Usando TABLE() com uma tabela aninhada

A consulta a seguir usa TABLE() para recuperar os dois endereços do cliente nº 1 de customers\_with\_nested\_table; observe que são retornadas duas linhas separadas:

```
SELECT a.*
FROM customers_with_nested_table c, TABLE(c.addresses) a
WHERE id = 1;
```

STREET	CITY	ST ZIP
2 State Street	Beantown	MA 12345
4 Hill Street	Lost Town	CA 54321

A próxima consulta obtém os atributos street e state dos endereços:

```
SELECT a.street, a.state
FROM customers_with_nested_table c, TABLE(c.addresses) a
WHERE id = 1;
```

STREET	ST
2 State Street	MA
4 Hill Street	CA

A consulta a seguir mostra outro exemplo que usa TABLE() para obter os endereços:

```
SELECT c.id, c.first_name, c.last_name, a.*
FROM customers_with_nested_table c, TABLE(c.addresses) a
WHERE c.id = 1;
```

ID	FIRST_NAME	LAST_NAME	STREET	CITY	ST ZIP
1	Steve	Brown	2 State Street	Beantown	MA 12345
1	Steve	Brown	4 Hill Street	Lost Town	CA 54321

Você vai ver um importante uso de TABLE() na seção “Modificando elementos de uma tabela aninhada”.

## MODIFICANDO ELEMENTOS DE COLEÇÕES

Nesta seção, você verá como modificar os elementos de um varray e de uma tabela aninhada. Sinta-se livre para executar as instruções UPDATE, INSERT e DELETE mostradas nesta seção.

### Modificando elementos de um varray

Os elementos de um varray só podem ser modificados como um todo, o que significa que, mesmo que você queira modificar apenas um elemento, deverá fornecer todos os elementos do varray. A instrução UPDATE a seguir modifica os endereços do cliente nº 2 na tabela customers\_with\_varray:



```

UPDATE customers_with_varray
SET addresses = t_varray_address(
    '6 Any Street, Lost Town, GA, 33347',
    '3 New Street, Anytown, MI, 54323',
    '7 Market Street, Main Town, MA, 54323'
)
WHERE id = 2;

1 row updated.

```

## Modificando elementos de uma tabela aninhada

Ao contrário de um varray, os elementos de uma tabela aninhada podem ser modificados individualmente. É possível inserir, atualizar e excluir elementos individuais em uma tabela aninhada; você verá como realizar todas essas três modificações nesta seção.

A instrução `INSERT` a seguir adiciona um endereço no cliente nº 2 em `customer_with_nested_table`; observe que a função `TABLE()` é usada para obter os endereços como uma série de linhas:

```

INSERT INTO TABLE(
    -- obtém os endereços do cliente nº 2
    SELECT addresses
    FROM customers_with_nested_table
    WHERE id = 2
) VALUES (
    t_address('5 Main Street', 'Uptown', 'NY', '55512')
);

1 row created.

```

A instrução `UPDATE` a seguir altera o endereço '1 High Street' do cliente nº 2 para '9 Any Street'; observe o uso do apelido `addr` nas cláusulas `VALUE` ao se especificar os endereços:

```

UPDATE TABLE(
    -- obtém os endereços do cliente nº 2
    SELECT addresses
    FROM customers_with_nested_table
    WHERE id = 2
) addr
SET VALUE(addr) =
    t_address('9 Any Street', 'Lost Town', 'VA', '74321')
WHERE VALUE(addr) =
    t_address('1 High Street', 'Newtown', 'CA', '12347');

1 row updated.

```

A instrução `DELETE` a seguir remove o endereço '3 New Street...' do cliente nº 2:

```

DELETE FROM TABLE(
    -- obtém os endereços do cliente nº 2
    SELECT addresses
    FROM customers_with_nested_table

```

```

WHERE id = 2
) addr
WHERE VALUE(addr) =
  t_address('3 New Street', 'Anytown', 'MI', '54323');

```

1 row deleted.

## USANDO UM MÉTODO DE MAPEAMENTO PARA COMPARAR O CONTEÚDO DE TABELAS ANINHADAS

Você pode comparar o conteúdo de uma tabela aninhada com o conteúdo de outra. Duas tabelas aninhadas são iguais somente se:

- Elas são do mesmo tipo
- Elas têm o mesmo número de linhas
- Todos os seus elementos contêm os mesmos valores

Se os elementos da tabela aninhada são de um tipo interno do banco de dados, como `NUMBER`, `VARCHAR2` etc., o banco de dados comparará o conteúdo das tabelas aninhadas automaticamente. Contudo, se os elementos são de um tipo de objeto definido pelo usuário, você precisará fornecer uma função de mapeamento que contenha código para comparar os objetos (as funções de mapeamento foram mostradas na seção “Comparando valores de objeto” do capítulo anterior).

As instruções a seguir criam um tipo chamado `t_address2` que contém uma função de mapeamento chamada `get_string()`; observe que `get_string()` retorna uma string `VARCHAR2` contendo os valores dos atributos `zip`, `state`, `city` e `street`:

```

CREATE TYPE t_address2 AS OBJECT (
    street VARCHAR2(15),
    city   VARCHAR2(15),
    state  CHAR(2),
    zip    VARCHAR2(5),

    -- declara a função de mapeamento get_string(),
    -- que retorna uma string VARCHAR2
    MAP MEMBER FUNCTION get_string RETURN VARCHAR2
);
/

CREATE TYPE BODY t_address2 AS
    -- define a função de mapeamento get_string()
    MAP MEMBER FUNCTION get_string RETURN VARCHAR2 IS
    BEGIN
        -- retorna uma string concatenada contendo os
        -- atributos zip, state, city e street
        RETURN zip || ' ' || state || ' ' || city || ' ' || street;
    END get_string;
END;
/

```

Conforme você verá em breve, o banco de dados chamará `get_string()` automaticamente ao comparar objetos `t_address2`. As instruções a seguir criam um tipo de tabela aninhada e uma tabela e adicionam uma linha na tabela:

```
CREATE TYPE t_nested_table_address2 AS TABLE OF t_address2;
/

CREATE TABLE customers_with_nested_table2 (
    id            INTEGER PRIMARY KEY,
    first_name    VARCHAR2(10),
    last_name     VARCHAR2(10),
    addresses     t_nested_table_address2
)
NESTED TABLE
    addresses
STORE AS
    nested_addresses2;

INSERT INTO customers_with_nested_table2 VALUES (
    1, 'Steve', 'Brown',
    t_nested_table_address2(
        t_address2('2 State Street', 'Beantown', 'MA', '12345'),
        t_address2('4 Hill Street', 'Lost Town', 'CA', '54321')
    )
);
```

A consulta a seguir inclui uma tabela aninhada na cláusula `WHERE`; observe que os endereços após `o =` na cláusula `WHERE` são iguais àqueles da instrução `INSERT` anterior:

```
SELECT cn.id, cn.first_name, cn.last_name
FROM customers_with_nested_table2 cn
WHERE cn.addresses =
    t_nested_table_address2(
        t_address2('2 State Street', 'Beantown', 'MA', '12345'),
        t_address2('4 Hill Street', 'Lost Town', 'CA', '54321')
    );
```

```
      ID FIRST_NAME LAST_NAME
-----
      1 Steve      Brown
```

Quando a consulta é executada, o banco de dados chama `get_string()` automaticamente para comparar os objetos `t_address2` de `cn.addresses` com os objetos `t_address2` após `o =` na cláusula `WHERE`. A função `get_string()` retorna uma string `VARCHAR2` contendo os atributos `zip`, `state`, `city` e `street` dos objetos e, quando as strings são iguais para cada objeto, as tabelas aninhadas também são iguais.

A próxima consulta não retorna uma linha, pois o único endereço após `o =` na cláusula `WHERE` corresponde a apenas um dos endereços em `cn.addresses` (lembre-se: duas tabelas aninhadas só são iguais se são do mesmo tipo, *têm o mesmo número de linhas* e seus elementos contêm os mesmos valores):

```
SELECT cn.id, cn.first_name, cn.last_name
FROM customers_with_nested_table2 cn
```

```
WHERE cn.addresses =  
      t_nested_table_address2(  
        t_address2('4 Hill Street', 'Lost Town', 'CA', '54321')  
      );
```

no rows selected

No Oracle Database 10g e em versões superiores, você pode usar o operador `SUBMULTISET` para verificar se o conteúdo de uma tabela aninhada é um subconjunto de outra tabela aninhada. A consulta a seguir reescreve o exemplo anterior e retorna uma linha:

```
SELECT cn.id, cn.first_name, cn.last_name  
FROM customers_with_nested_table2 cn  
WHERE  
      t_nested_table_address2(  
        t_address2('4 Hill Street', 'Lost Town', 'CA', '54321')  
      )  
      SUBMULTISET OF cn.addresses;
```

ID	FIRST_NAME	LAST_NAME
1	Steve	Brown

Como o endereço na primeira parte da cláusula `WHERE` é um subconjunto dos endereços em `cn.addresses`, uma correspondência é encontrada e uma linha é retornada.

A consulta a seguir mostra outro exemplo; desta vez, os endereços em `cn.addresses` são um subconjunto dos endereços após `OF` na cláusula `WHERE`:

```
SELECT cn.id, cn.first_name, cn.last_name  
FROM customers_with_nested_table2 cn  
WHERE  
      cn.addresses SUBMULTISET OF  
      t_nested_table_address2(  
        t_address2('2 State Street', 'Beantown', 'MA', '12345'),  
        t_address2('4 Hill Street', 'Lost Town', 'CA', '54321'),  
        t_address2('6 State Street', 'Beantown', 'MA', '12345')  
      );
```

ID	FIRST_NAME	LAST_NAME
1	Steve	Brown

Você vai aprender mais sobre o operador `SUBMULTISET` na seção “Operador `SUBMULTISET`”. Além disso, na seção “Operadores igual e diferente”, você verá como utilizar os operadores ANSI implementados no Oracle Database 10g para comparar tabelas aninhadas.

## NOTA

Não existe um mecanismo direto para comparar o conteúdo de `varrays`.

## USANDO CAST() PARA CONVERTER COLEÇÕES DE UM TIPO PARA OUTRO

É possível usar `CAST()` para converter uma coleção de um tipo para outro. Nesta seção, você verá como usar `CAST()` para converter um varray em uma tabela aninhada e vice-versa.

### Usando `CAST()` para converter um varray em uma tabela aninhada

As instruções a seguir criam e preenchem uma tabela chamada `customers_with_varray2` que contém uma coluna `addresses` de tipo `t_varray_address2`:

```
CREATE TYPE t_varray_address2 AS VARRAY(3) OF t_address;
/

CREATE TABLE customers_with_varray2 (
    id            INTEGER PRIMARY KEY,
    first_name    VARCHAR2(10),
    last_name     VARCHAR2(10),
    addresses     t_varray_address2
);

INSERT INTO customers_with_varray2 VALUES (
    1, 'Jason', 'Bond',
    t_varray_address2(
        t_address('9 Newton Drive', 'Sometown', 'WY', '22123'),
        t_address('6 Spring Street', 'New City', 'CA', '77712')
    )
);
```

A consulta a seguir usa `CAST()` para retornar os endereços do varray do cliente nº 1 como uma tabela aninhada; observe que os endereços aparecem em um construtor do tipo `T_NESTED_TABLE_ADDRESS`, indicando a conversão dos elementos para esse tipo:

```
SELECT CAST(cv.addresses AS t_nested_table_address)
FROM customers_with_varray2 cv
WHERE cv.id = 1;

CAST(CV.ADDRESSESAS T_NESTED_TABLE_ADDRESS)(STREET, CITY, STATE, ZIP)
-----
T_NESTED_TABLE_ADDRESS(
    T_ADDRESS('9 Newton Drive', 'Sometown', 'WY', '22123'),
    T_ADDRESS('6 Spring Street', 'New City', 'CA', '77712'))
```

### Usando `CAST()` para converter uma tabela aninhada em um varray

A consulta a seguir usa `CAST()` para retornar os endereços do cliente nº 1 em `customers_with_nested_table` como um varray; observe que os endereços aparecem em um construtor de `T_VARRAY_ADDRESS2`:

```
SELECT CAST(cn.addresses AS t_varray_address2)
FROM customers_with_nested_table cn
```

```
WHERE cn.id = 1;
```

```
CAST(CN.ADDRESSESAST_VARRAY_ADDRESS2) (STREET, CITY, STATE, ZIP)
```

```
-----  
T_VARRAY_ADDRESS2(
```

```
  T_ADDRESS('2 State Street', 'Beantown', 'MA', '12345'),
```

```
  T_ADDRESS('4 Hill Street', 'Lost Town', 'CA', '54321'))
```

## USANDO COLEÇÕES EM PL/SQL

É possível usar coleções em PL/SQL. Nesta seção, você vai aprender a executar as seguintes tarefas em PL/SQL:

- Manipular um varray
- Manipular uma tabela aninhada
- Usar os métodos de coleção PL/SQL para acessar e manipular coleções

Todos os pacotes examinados nesta seção são criados quando você executa o script `collection_schema.sql`. Se você executou qualquer uma das instruções `INSERT`, `UPDATE` ou `DELETE` mostradas nas seções anteriores deste capítulo, execute o script `collection_schema.sql` novamente para que sua saída corresponda a esta seção.

### Manipulando um varray

Nesta seção, você verá um pacote chamado `varray_package`, que contém os seguintes itens:

- Um tipo `REF CURSOR` chamado `t_ref_cursor`
- Uma função chamada `get_customers()`, a qual retorna um objeto `t_ref_cursor` que aponta para as linhas da tabela `customers_with_varray`
- Uma procedure chamada `insert_customer()`, a qual adiciona uma linha na tabela `customers_with_varray`

O script `collection_schema.sql` contém a seguinte especificação de pacote e corpo de `varray_package`:

```
CREATE PACKAGE varray_package AS
  TYPE t_ref_cursor IS REF CURSOR;
  FUNCTION get_customers RETURN t_ref_cursor;
  PROCEDURE insert_customer(
    p_id          IN customers_with_varray.id%TYPE,
    p_first_name  IN customers_with_varray.first_name%TYPE,
    p_last_name   IN customers_with_varray.last_name%TYPE,
    p_addresses   IN customers_with_varray.addresses%TYPE
  );
END varray_package;
/
```

```

CREATE PACKAGE BODY varray_package AS
  -- a função get_customers() retorna um REF CURSOR
  -- que aponta para as linhas de customers_with_varray
  FUNCTION get_customers
  RETURN t_ref_cursor IS
    -- declara o objeto REF CURSOR
    v_customers_ref_cursor t_ref_cursor;
  BEGIN
    -- obtém o REF CURSOR
    OPEN v_customers_ref_cursor FOR
      SELECT *
      FROM customers_with_varray;
    -- retorna o REF CURSOR
    RETURN customers_ref_cursor;
  END get_customers;

  -- a procedure insert_customer() adiciona uma linha em
  -- customers_with_varray
  PROCEDURE insert_customer(
    p_id          IN customers_with_varray.id%TYPE,
    p_first_name  IN customers_with_varray.first_name%TYPE,
    p_last_name   IN customers_with_varray.last_name%TYPE,
    p_addresses   IN customers_with_varray.addresses%TYPE
  ) IS
  BEGIN
    INSERT INTO customers_with_varray
    VALUES (p_id, p_first_name, p_last_name, p_addresses);
    COMMIT;
  EXCEPTION
    WHEN OTHERS THEN
      ROLLBACK;
  END insert_customer;
END varray_package;
/

```

O exemplo a seguir chama `insert_customer()` para adicionar uma nova linha na tabela `customers_with_varray`:

```

CALL varray_package.insert_customer(
  3, 'James', 'Red',
  t_varray_address(
    '10 Main Street, Green Town, CA, 22212',
    '20 State Street, Blue Town, FL, 22213'
  )
);

```

Call completed.

O próximo exemplo chama `get_products()` para recuperar as linhas de `customers_with_varray`:

```
SELECT varray_package.get_customers
FROM dual;
```

```
GET_CUSTOMERS
```

```
-----
CURSOR STATEMENT : 1
```

```
CURSOR STATEMENT : 1
```

```
      ID FIRST_NAME LAST_NAME
```

```
-----
ADDRESSES
```

```
-----
      1 Steve      Brown
T_VARRAY_ADDRESS('2 State Street, Beantown, MA, 12345',
'4 Hill Street, Lost Town, CA, 54321')

      2 John      Smith
T_VARRAY_ADDRESS('1 High Street, Newtown, CA, 12347',
'3 New Street, Anytown, MI, 54323',
'7 Market Street, Main Town, MA, 54323')

      3 James      Red
T_VARRAY_ADDRESS('10 Main Street, Green Town, CA, 22212',
'20 State Street, Blue Town, FL, 22213')
```

## Manipulando uma tabela aninhada

Nesta seção, você verá um pacote chamado `nested_table_package`, que contém os seguintes itens:

- Um tipo `REF CURSOR` chamado `t_ref_cursor`
- Uma função chamada `get_customers()`, a qual retorna um objeto `t_ref_cursor` que aponta para as linhas de `customers_with_nested_table`
- Uma procedure chamada `insert_customer()`, a qual adiciona uma linha em `customers_with_nested_table`

O script `collection_schema.sql` contém a seguinte especificação de pacote e corpo de `nested_table_package`:

```
CREATE PACKAGE nested_table_package AS
TYPE t_ref_cursor IS REF CURSOR;
FUNCTION get_customers RETURN t_ref_cursor;
PROCEDURE insert_customer(
    p_id          IN customers_with_nested_table.id%TYPE,
    p_first_name  IN customers_with_nested_table.first_name%TYPE,
    p_last_name   IN customers_with_nested_table.last_name%TYPE,
    p_addresses   IN customers_with_nested_table.addresses%TYPE
);
```



```

END nested_table_package;
/

CREATE PACKAGE BODY nested_table_package AS
  -- a função get_customers() retorna um REF CURSOR
  -- que aponta para as linhas de customers_with_nested_table
  FUNCTION get_customers
  RETURN t_ref_cursor IS
    -- declara o objeto REF CURSOR
    v_customers_ref_cursor t_ref_cursor;
  BEGIN
    -- obtém o REF CURSOR
    OPEN v_customers_ref_cursor FOR
      SELECT *
      FROM customers_with_nested_table;
    -- retorna o REF CURSOR
    RETURN customers_ref_cursor;
  END get_customers;

  -- a procedure insert_customer() adiciona uma linha em
  -- customers_with_nested_table
  PROCEDURE insert_customer(
    p_id          IN customers_with_nested_table.id%TYPE,
    p_first_name  IN customers_with_nested_table.first_name%TYPE,
    p_last_name   IN customers_with_nested_table.last_name%TYPE,
    p_addresses   IN customers_with_nested_table.addresses%TYPE
  ) IS
  BEGIN
    INSERT INTO customers_with_nested_table
    VALUES (p_id, p_first_name, p_last_name, p_addresses);
    COMMIT;
  EXCEPTION
    WHEN OTHERS THEN
      ROLLBACK;
  END insert_customer;
END nested_table_package;
/

```

O exemplo a seguir chama `insert_customer()` para adicionar uma nova linha em `customers_with_nested_table`:

```

CALL nested_table_package.insert_customer(
  3, 'James', 'Red',
  t_nested_table_address(
    t_address('10 Main Street', 'Green Town', 'CA', '22212'),
    t_address('20 State Street', 'Blue Town', 'FL', '22213')
  )
);

```

Call completed.

O próximo exemplo chama `get_customers()` para recuperar as linhas de `customers_with_nested_table`:

```
SELECT nested_table_package.get_customers
FROM dual;

GET_CUSTOMERS
-----
CURSOR STATEMENT : 1

CURSOR STATEMENT : 1

      ID FIRST_NAME LAST_NAME
-----
ADDRESSES(STREET, CITY, STATE, ZIP)
-----
      1 Steve      Brown
T_NESTED_TABLE_ADDRESS(
  T_ADDRESS('2 State Street', 'Beantown', 'MA', '12345'),
  T_ADDRESS('4 Hill Street', 'Lost Town', 'CA', '54321'))

      2 John      Smith
T_NESTED_TABLE_ADDRESS(
  T_ADDRESS('1 High Street', 'Newtown', 'CA', '12347'),
  T_ADDRESS('3 New Street', 'Anytown', 'MI', '54323'),
  T_ADDRESS('7 Market Street', 'Main Town', 'MA', '54323'))

      3 James      Red
T_NESTED_TABLE_ADDRESS(
  T_ADDRESS('10 Main Street', 'Green Town', 'CA', '22212'),
  T_ADDRESS('20 State Street', 'Blue Town', 'FL', '22213'))
```

Métodos de coleção PL/SQL

Nesta seção, você verá os métodos PL/SQL que podem ser usados com coleções. A Tabela 13-3 resume os métodos de coleção. Esses métodos só podem ser usados em PL/SQL. As seções a seguir usam um pacote chamado `collection_method_examples`; o exemplo ilustra o uso dos métodos mostrados na tabela anterior. O pacote é criado pelo script `collection_schema.sql` e nas seções a seguir você verá os métodos individuais definidos nesse pacote.

COUNT()

COUNT retorna o número de elementos em uma coleção. Como uma tabela aninhada pode ter elementos individuais vazios, COUNT retorna o número de elementos não vazios de uma tabela aninhada. Por exemplo, digamos que a tabela aninhada `v_nested_table` tenha seus elementos definidos como mostrado a seguir.

Índice do elemento	Vazio/Não vazio
1	Vazio
2	Não vazio
3	Vazio
4	Não vazio

**Tabela 13-3** *Métodos de coleção PL/SQL*

Método	Descrição
COUNT	Retorna o número de elementos em uma coleção. Como uma tabela aninhada pode ter elementos individuais vazios, COUNT retorna o número de elementos não vazios de uma tabela aninhada.
DELETE DELETE ( <i>n</i> ) DELETE ( <i>n</i> , <i>m</i> )	Remove elementos de uma coleção. Existem três formas de DELETE: <ul style="list-style-type: none"> <li>■ DELETE remove todos os elementos.</li> <li>■ DELETE (<i>n</i>) remove o elemento <i>n</i>.</li> <li>■ DELETE (<i>n</i>, <i>m</i>) remove os elementos <i>n</i> até <i>m</i>.</li> </ul> Como varrays sempre têm subscritos consecutivos, você não pode excluir elementos individuais de um varray (exceto do final, usando TRIM).
EXISTS ( <i>n</i> )	Retorna verdadeiro se o elemento <i>n</i> de uma coleção existe: EXISTS retorna verdadeiro para elementos não vazios e falso para elementos vazios de tabelas aninhadas ou elementos fora do intervalo de uma coleção.
EXTEND EXTEND ( <i>n</i> ) EXTEND ( <i>n</i> , <i>m</i> )	Adiciona elementos no final de uma coleção. Existem três formas de EXTEND: <ul style="list-style-type: none"> <li>■ EXTEND adiciona um elemento, o qual é configurado como nulo.</li> <li>■ EXTEND (<i>n</i>) adiciona <i>n</i> elementos, os quais são configurados como nulos.</li> <li>■ EXTEND (<i>n</i>, <i>m</i>) adiciona <i>n</i> elementos, os quais são configurados como uma cópia do elemento <i>m</i>.</li> </ul>
FIRST	Retorna o índice do primeiro elemento de uma coleção. Se a coleção está completamente vazia, FIRST retorna nulo. Como uma tabela aninhada pode ter elementos individuais vazios, FIRST retorna o índice mais baixo de um elemento não vazio de uma tabela aninhada.
LAST	Retorna o índice do último elemento de uma coleção. Se a coleção está completamente vazia, LAST retorna nulo. Como uma tabela aninhada pode ter elementos individuais vazios, LAST retorna o índice mais alto de um elemento não vazio de uma tabela aninhada.
LIMIT	Para tabelas aninhadas, que não têm nenhum tamanho declarado, LIMIT retorna nulo. Para varrays, LIMIT retorna o número máximo de elementos que o varray pode conter. Você especifica o limite na definição de tipo. O limite é alterado ao se usar TRIM e EXTEND ou quando você utiliza ALTER TYPE para alterá-lo.
NEXT ( <i>n</i> )	Retorna o índice do elemento após <i>n</i> . Como uma tabela aninhada pode ter elementos individuais vazios, NEXT retorna o índice de um elemento não vazio após <i>n</i> . Se não existe nenhum elemento após <i>n</i> , NEXT retorna nulo.
PRIOR ( <i>n</i> )	Retorna o índice do elemento antes de <i>n</i> . Como uma tabela aninhada pode ter elementos individuais vazios, PRIOR retorna o índice de um elemento não vazio antes de <i>n</i> . Se não existe nenhum elemento antes de <i>n</i> , PRIOR retorna nulo.
TRIM TRIM ( <i>n</i> )	Remove elementos do final de uma coleção. Existem duas formas de TRIM: <ul style="list-style-type: none"> <li>■ TRIM remove um elemento do final.</li> <li>■ TRIM (<i>n</i>) remove <i>n</i> elementos do final.</li> </ul>

Dada essa configuração, `v_nested_table.COUNT` retorna 2, o número de elementos não vazios. `COUNT` é usado nos métodos `get_addresses()` e `display_addresses()` do pacote `collection_method_examples`. A função `get_addresses()` retorna os endereços do cliente especificado de `customers_with_nested_table`, cujo valor de `id` é passado para a função:

```
FUNCTION get_addresses(
    p_id customers_with_nested_table.id%TYPE
) RETURN t_nested_table_address IS
    -- declara objeto chamado v_addresses para armazenar a
    -- tabela aninhada de endereços
    v_addresses t_nested_table_address;
BEGIN
    -- recupera a tabela aninhada de endereços em v_addresses
    SELECT addresses
    INTO v_addresses
    FROM customers_with_nested_table
    WHERE id = p_id;

    -- exibe o número de endereços usando v_addresses.COUNT
    DBMS_OUTPUT.PUT_LINE(
        'Number of addresses = ' || v_addresses.COUNT
    );

    -- retorna v_addresses
    RETURN v_addresses;
END get_addresses;
```

O exemplo a seguir ativa a saída do servidor e chama `get_addresses()` para o cliente nº 1:

```
SET SERVEROUTPUT ON
SELECT collection_method_examples.get_addresses(1) addresses
FROM dual;

ADDRESSES(STREET, CITY, STATE, ZIP)
-----
T_NESTED_TABLE_ADDRESS(
  T_ADDRESS('2 State Street', 'Beantown', 'MA', '12345'),
  T_ADDRESS('4 Hill Street', 'Lost Town', 'CA', '54321'))

Number of addresses = 2
```

A procedure `display_addresses()` a seguir aceita um parâmetro chamado `p_addresses`, o qual contém uma tabela aninhada de endereços; a procedure exibe o número de endereços em `p_addresses` usando `COUNT` e, então, exibe esses endereços utilizando um loop:

```
PROCEDURE display_addresses(
    p_addresses t_nested_table_address
) IS
    v_count INTEGER;
```

```

BEGIN
  -- exibe o número de endereços em p_addresses
  DBMS_OUTPUT.PUT_LINE(
    'Current number of addresses = ' || p_addresses.COUNT
  );

  -- exibe os endereços em p_addresses usando um loop
  FOR v_count IN 1..p_addresses.COUNT LOOP
    DBMS_OUTPUT.PUT_LINE('Address #' || v_count || ':');
    DBMS_OUTPUT.PUT(p_addresses(v_count).street || ', ');
    DBMS_OUTPUT.PUT(p_addresses(v_count).city || ', ');
    DBMS_OUTPUT.PUT(p_addresses(v_count).state || ', ');
    DBMS_OUTPUT.PUT_LINE(p_addresses(v_count).zip);
  END LOOP;
END display_addresses;

```

Você vai ver o uso de `display_addresses()` em breve.

## DELETE()

DELETE remove elementos de uma coleção. Existem três formas de DELETE:

- DELETE remove todos os elementos.
- DELETE(*n*) remove o elemento *n*.
- DELETE(*n*, *m*) remove os elementos *n* até *m*.

Por exemplo, digamos que você tenha uma tabela aninhada chamada `v_nested_table` com sete elementos; então, `v_nested_table.DELETE(2, 5)` remove os elementos 2 até 5.

A procedure `delete_address()` a seguir obtém os endereços do cliente nº 1 e depois usa DELETE para remover o endereço cujo índice é especificado pelo parâmetro `p_address_num`:

```

PROCEDURE delete_address(
  p_address_num INTEGER
) IS
  v_addresses t_nested_table_address;
BEGIN
  v_addresses := get_addresses(1);
  display_addresses(v_addresses);
  DBMS_OUTPUT.PUT_LINE('Deleting address #' || p_address_num);

  -- exclui o endereço especificado por p_address_num
  v_addresses.DELETE(p_address_num);

  display_addresses(v_addresses);
END delete_address;

```

O exemplo a seguir chama `delete_address(2)` para remover o endereço nº 2 do cliente nº 1:

```

CALL collection_method_examples.delete_address(2);
Number of addresses = 2
Current number of addresses = 2

```

```
Address #1:
2 State Street, Beantown, MA, 12345
Address #2:
4 Hill Street, Lost Town, CA, 54321
Deleting endereço nº 2
Current number of addresses = 1
Address #1:
2 State Street, Beantown, MA, 12345
```

### EXISTS()

EXISTS(*n*) retorna verdadeiro se o elemento *n* de uma coleção existe; EXISTS retorna verdadeiro para elementos não vazios e retorna falso para elementos vazios de tabelas aninhadas ou elementos fora do intervalo de uma coleção. Por exemplo, digamos que você tenha uma tabela aninhada chamada `v_nested_table` com seus elementos definidos como mostrado a seguir.

Índice do elemento	Vazio/Não vazio
1	Vazio
2	Não vazio
3	Vazio
4	Não vazio

Dada essa configuração, `v_nested_table.EXISTS(2)` retorna verdadeiro (pois o elemento nº 2 não está vazio) e `v_nested_table.EXISTS(3)` retorna falso (pois o elemento nº 3 está vazio).

A procedure `exist_addresses()` a seguir obtém os endereços do cliente nº 1, usa DELETE para remover o endereço nº 1 e depois usa EXISTS para verificar se os endereços nº 1 e 2 existem (o nº 1 não existe, porque foi excluído; o nº 2 existe):

```
PROCEDURE exist_addresses IS
    v_addresses t_nested_table_address;
BEGIN
    v_addresses := get_addresses(1);
    DBMS_OUTPUT.PUT_LINE('Deleting address #1');
    v_addresses.DELETE(1);

    -- usa EXISTS para verificar se os endereços existem
    IF v_addresses.EXISTS(1) THEN
        DBMS_OUTPUT.PUT_LINE('Address #1 does exist');
    ELSE
        DBMS_OUTPUT.PUT_LINE('Address #1 does not exist');
    END IF;
    IF v_addresses.EXISTS(2) THEN
        DBMS_OUTPUT.PUT_LINE('Address #2 does exist');
    END IF;
END exist_addresses;
```

O exemplo a seguir chama `exist_addresses()`:

```
CALL collection_method_examples.exist_addresses();
Number of addresses = 2
```

```

Deleting address #1
Address #1 does not exist
Address #2 does exist

```

### EXTEND()

EXTEND adiciona elementos no final de uma coleção. Existem três formas de EXTEND:

- EXTEND adiciona um elemento, o qual é configurado como nulo.
- EXTEND(*n*) adiciona *n* elementos, os quais são configurados como nulos.
- EXTEND(*n*, *m*) adiciona *n* elementos, os quais são configurados como uma cópia do elemento *m*.

Por exemplo, digamos que você tenha uma coleção chamada `v_nested_table` com sete elementos; então, `v_nested_table.EXTEND(2, 5)` adiciona o elemento nº 5 duas vezes no final da coleção.

A procedure `extend_addresses()` a seguir obtém os endereços do cliente nº 1 em `v_addresses` e, depois, usa EXTEND para copiar o endereço nº 1 duas vezes no final de `v_addresses`:

```

PROCEDURE extend_addresses IS
    v_addresses t_nested_table_address;
BEGIN
    v_addresses := get_addresses(1);
    display_addresses(v_addresses);
    DBMS_OUTPUT.PUT_LINE('Extending addresses');

    -- copia o endereço nº 1 duas vezes no final de v_addresses
    v_addresses.EXTEND(2, 1);

    display_addresses(v_addresses);
END extend_addresses;

```

O exemplo a seguir chama `extend_addresses()`:

```

CALL collection_method_examples.extend_addresses();
Number of addresses = 2
Current number of addresses = 2
Address #1:
2 State Street, Beantown, MA, 12345
Address #2:
4 Hill Street, Lost Town, CA, 54321
Extending addresses
Current number of addresses = 4
Address #1:
2 State Street, Beantown, MA, 12345
Address #2:
4 Hill Street, Lost Town, CA, 54321
Address #3:
2 State Street, Beantown, MA, 12345
Address #4:
2 State Street, Beantown, MA, 12345

```

**FIRST()**

Você usa `FIRST` para obter o índice do primeiro elemento de uma coleção. Se a coleção está completamente vazia, `FIRST` retorna nulo. Como uma tabela aninhada pode ter elementos individuais vazios, `FIRST` retorna o índice mais baixo de um elemento não vazio de uma tabela aninhada. Por exemplo, digamos que você tenha uma tabela aninhada chamada `v_nested_table` cujos elementos são definidos como mostrado a seguir.

Índice do elemento	Vazio/Não vazio
1	Vazio
2	Não vazio
3	Vazio
4	Não vazio

Dada essa configuração, `v_nested_table.FIRST` retorna 2, o índice mais baixo contendo um elemento não vazio.

A procedure `first_address()` a seguir obtém os endereços do cliente nº 1 em `v_addresses` e, então, usa `FIRST` para exibir o índice do primeiro endereço de `v_addresses`. Depois, a procedure exclui o endereço nº 1 usando `DELETE` e exibe o novo índice retornado por `FIRST`:

```
PROCEDURE first_address IS
    v_addresses t_nested_table_address;
BEGIN
    v_addresses := get_addresses(1);

    -- exibe o PRIMEIRO endereço
    DBMS_OUTPUT.PUT_LINE('First address = ' || v_addresses.FIRST);
    DBMS_OUTPUT.PUT_LINE('Deleting address #1');
    v_addresses.DELETE(1);

    -- exibe o PRIMEIRO endereço novamente
    DBMS_OUTPUT.PUT_LINE('First address = ' || v_addresses.FIRST);
END first_address;
```

O exemplo a seguir chama `first_address()`:

```
CALL collection_method_examples.first_address();
Number of addresses = 2
First address = 1
Deleting address #1
First address = 2
```

**LAST()**

`LAST` retorna o índice do último elemento de uma coleção. Se a coleção está completamente vazia, `LAST` retorna nulo. Como uma tabela aninhada pode ter elementos individuais vazios, `LAST` retorna o índice mais alto de um elemento não vazio de uma tabela aninhada. Por exemplo, digamos que você tenha uma tabela aninhada chamada `v_nested_table` cujos elementos são definidos como mostrado a seguir.



Índice do elemento	Vazio/Não vazio
1	Não vazio
2	Vazio
3	Vazio
4	Não vazio

Dada essa configuração, `v_nested_table.LAST` retorna 4, o índice mais alto contendo um elemento não vazio.

A procedure `last_address()` a seguir obtém os endereços do cliente nº 1 em `v_addresses` e depois usa `LAST` para exibir o índice do último endereço em `v_addresses`. Então, a procedure exclui o endereço nº 2 usando `DELETE` e exibe o novo índice retornado por `LAST`:

```
PROCEDURE last_address IS
    v_addresses t_nested_table_address;
BEGIN
    v_addresses := get_addresses(1);

    -- exibe o ÚLTIMO endereço
    DBMS_OUTPUT.PUT_LINE('Last address = ' || v_addresses.LAST);
    DBMS_OUTPUT.PUT_LINE('Deleting address #2');
    v_addresses.DELETE(2);

    -- exibe o ÚLTIMO endereço novamente
    DBMS_OUTPUT.PUT_LINE('Last address = ' || v_addresses.LAST);
END last_address;
```

O exemplo a seguir chama `last_address()`:

```
CALL collection_method_examples.last_address();
Number of addresses = 2
Last address = 2
Deleting address #2
Last address = 1
```

### NEXT()

`NEXT(n)` retorna o índice do elemento após `n`. Como uma tabela aninhada pode ter elementos individuais vazios, `NEXT` retorna o índice de um elemento não vazio após `n`. Se não existe um elemento após `n`, `NEXT` retorna nulo. Por exemplo, digamos que você tenha uma tabela aninhada chamada `v_nested_table` cujos elementos são definidos como mostrado a seguir.

Índice do elemento	Vazio/Não vazio
1	Não vazio
2	Vazio
3	Vazio
4	Não vazio

Dada essa configuração, `v_nested_table.NEXT(1)` retorna 4, o índice contendo o próximo elemento não vazio; `v_nested_table.NEXT(4)` retorna nulo.

A procedure `next_address()` a seguir obtém os endereços do cliente nº 1 em `v_addresses` e, então, usa `NEXT(1)` para obter o índice do endereço após o endereço nº 1 de `v_addresses`. Depois, a procedure usa `NEXT(2)` para tentar obter o índice do endereço após o endereço nº 2 (não há, pois o cliente nº 1 só tem dois endereços; portanto, nulo é retornado):

```
PROCEDURE next_address IS
    v_addresses t_nested_table_address;
BEGIN
    v_addresses := get_addresses(1);

    -- usa NEXT(1) para obter o índice do endereço
    -- após o endereço nº 1
    DBMS_OUTPUT.PUT_LINE(
        'v_addresses.NEXT(1) = ' || v_addresses.NEXT(1)
    );

    -- usa NEXT(2) para tentar obter o índice do
    -- endereço após o endereço nº 2 (não há nenhum;
    -- portanto, nulo é retornado)
    DBMS_OUTPUT.PUT_LINE(
        'v_addresses.NEXT(2) = ' || v_addresses.NEXT(2)
    );
END next_address;
```

O exemplo a seguir chama `next_address()`; `v_addresses.NEXT(2)` é nulo e, assim, nenhuma saída é mostrada após o = para esse elemento:

```
CALL collection_method_examples.next_address();
Number of addresses = 2
v_addresses.NEXT(1) = 2
v_addresses.NEXT(2) =
```

### **PRIOR()**

`PRIOR(n)` retorna o índice do elemento antes de *n*. Como uma tabela aninhada pode ter elementos individuais vazios, `PRIOR` retorna o índice de um elemento não vazio antes de *n*. Se não existe um elemento antes de *n*, `PRIOR` retorna nulo. Por exemplo, digamos que você tenha uma tabela aninhada chamada `v_nested_table` cujos elementos são definidos como mostrado na tabela a seguir.

Índice do elemento	Vazio/Não vazio
1	Não vazio
2	Vazio
3	Vazio
4	Não vazio

Dada essa configuração, `v_nested_table.PRIOR(4)` retorna 1, o índice contendo o elemento não vazio anterior; `v_nested_table.PRIOR(1)` retorna nulo.

A procedure `prior_address()` a seguir obtém os endereços do cliente nº 1 em `v_addresses` e depois usa `PRIOR(2)` para obter o índice do endereço antes do endereço nº 2 de `v_addresses`; então, a procedure usa `PRIOR(1)` para tentar obter o índice do endereço antes do endereço nº 1 (não há; portanto, nulo é retornado):

```
PROCEDURE prior_address IS
    v_addresses t_nested_table_address;
BEGIN
    v_addresses := get_addresses(1);

    -- usa PRIOR(2) para obter o índice do endereço
    -- antes do endereço nº 2
    DBMS_OUTPUT.PUT_LINE(
        'v_addresses.PRIOR(2) = ' || v_addresses.PRIOR(2)
    );

    -- usa PRIOR(1) para tentar obter o índice do
    -- endereço antes do endereço nº 1 (não há nenhum;
    -- portanto, nulo é retornado)
    DBMS_OUTPUT.PUT_LINE(
        'v_addresses.PRIOR(1) = ' || v_addresses.PRIOR(1)
    );
END prior_address;
```

O exemplo a seguir chama `prior_address()`; `v_addresses.PRIOR(1)` é nulo e, assim, nenhuma saída é mostrada após o = para esse elemento:

```
CALL collection_method_examples.prior_address();
Number of addresses = 2
v_addresses.PRIOR(2) = 1
v_addresses.PRIOR(1) =
```

### TRIM()

TRIM remove elementos do final de uma coleção. Existem duas formas de TRIM:

- TRIM remove um elemento do final
- TRIM(*n*) remove *n* elementos do final

Por exemplo, digamos que você tenha uma tabela aninhada chamada `v_nested_table`; então, `v_nested_table.TRIM(2)` remove dois elementos do final.

A procedure `trim_addresses()` a seguir obtém os endereços do cliente nº 1, copia o endereço nº 1 no final de `v_addresses` três vezes usando `EXTEND(3, 1)`, e depois remove dois endereços do final de `v_addresses`, usando `TRIM(2)`:

```
PROCEDURE trim_addresses IS
    v_addresses t_nested_table_address;
BEGIN
    v_addresses := get_addresses(1);
    display_addresses(v_addresses);
    DBMS_OUTPUT.PUT_LINE('Extending addresses');
```

```

v_addresses.EXTEND(3, 1);
display_addresses(v_addresses);
DBMS_OUTPUT.PUT_LINE('Trimming 2 addresses from end');

-- remove 2 endereços do final de v_addresses
-- usando TRIM(2)
v_addresses.TRIM(2);

display_addresses(v_addresses);
END trim_addresses;

```

O exemplo a seguir chama `trim_addresses()`:

```

CALL collection_method_examples.trim_addresses();
Number of addresses = 2
Current number of addresses = 2
Address #1:
2 State Street, Beantown, MA, 12345
Address #2:
4 Hill Street, Lost Town, CA, 54321
Extending addresses
Current number of addresses = 5
Address #1:
2 State Street, Beantown, MA, 12345
Address #2:
4 Hill Street, Lost Town, CA, 54321
Address #3:
2 State Street, Beantown, MA, 12345
Address #4:
2 State Street, Beantown, MA, 12345
Address #5:
2 State Street, Beantown, MA, 12345
Trimming 2 addresses from end
Current number of addresses = 3
Address #1:
2 State Street, Beantown, MA, 12345
Address #2:
4 Hill Street, Lost Town, CA, 54321
Address #3:
2 State Street, Beantown, MA, 12345

```

## COLEÇÕES DE MÚLTIPLOS NÍVEIS

A partir da versão 9i do Oracle Database, você pode criar uma coleção no banco de dados cujos elementos também são uma coleção. Essas “coleções de coleções” são conhecidas como *coleções de múltiplos níveis*. A lista a seguir mostra as coleções de múltiplos níveis válidas:

- Uma tabela aninhada de tabelas aninhadas
- Uma tabela aninhada de varrays
- Um varray de varrays
- Um varray de tabelas aninhadas

Fornecemos um script SQL\*Plus chamado `collection_schema2.sql` no diretório SQL. Esse script cria um usuário chamado `collection_user2`, com a senha `collection_password`, junto com os tipos e a tabela mostrados nesta seção. Você poderá executar esse script se estiver usando Oracle Database 9i ou superior. Depois que o script terminar, você estará conectado como `collection_user2`.

Digamos que você quisesse armazenar um conjunto de números de telefone para cada endereço de um cliente. O exemplo a seguir cria um tipo de varray com três strings `VARCHAR2`, chamado `t_varray_phone`, para representar números de telefone:

```
CREATE TYPE t_varray_phone AS VARRAY(3) OF VARCHAR2(14);
/
```

Em seguida, o próximo exemplo cria um tipo de objeto chamado `t_address` que contém um atributo chamado `phone_numbers`. Esse atributo é de tipo `t_varray_phone`:

```
CREATE TYPE t_address AS OBJECT (
    street      VARCHAR2(15),
    city        VARCHAR2(15),
    state       CHAR(2),
    zip         VARCHAR2(5),
    phone_numbers t_varray_phone
);
/
```

O exemplo a seguir cria um tipo de tabela aninhada objetos de `t_address`:

```
CREATE TYPE t_nested_table_address AS TABLE OF t_address;
/
```

O exemplo a seguir cria uma tabela chamada `customers_with_nested_table`, a qual contém uma coluna chamada `addresses` de tipo `t_nested_table_address`:

```
CREATE TABLE customers_with_nested_table (
    id          INTEGER PRIMARY KEY,
    first_name  VARCHAR2(10),
    last_name   VARCHAR2(10),
    addresses   t_nested_table_address
)
NESTED TABLE
    addresses
STORE AS
    nested_addresses;
```

Assim, `customers_with_nested_table` contém uma tabela aninhada cujos elementos contém um endereço com um varray de números de telefone.

A instrução `INSERT` a seguir adiciona uma linha em `customers_with_nested_table`; observe a estrutura e o conteúdo da instrução `INSERT`, que contém elementos da tabela aninhada de endereços, cada um dos quais tendo um varray incorporado de números de telefone:

```
INSERT INTO customers_with_nested_table VALUES (
    1, 'Steve', 'Brown',
    t_nested_table_address(
        t_address('2 State Street', 'Beantown', 'MA', '12345',
```

```
        t_varray_phone(
          '(800)-555-1211',
          '(800)-555-1212',
          '(800)-555-1213'
        )
      ),
      t_address('4 Hill Street', 'Lost Town', 'CA', '54321',
        t_varray_phone(
          '(800)-555-1211',
          '(800)-555-1212'
        )
      )
    )
  );
```

Você pode ver que o primeiro endereço tem três números de telefone, enquanto o segundo endereço tem dois. A consulta a seguir recupera a linha de `customers_with_nested_table`:

```
SELECT *
FROM customers_with_nested_table;

      ID FIRST_NAME LAST_NAME
-----
ADDRESSES(STREET, CITY, STATE, ZIP, PHONE_NUMBERS)
-----
          1 Steve      Brown
T_NESTED_TABLE_ADDRESS(
  T_ADDRESS('2 State Street', 'Beantown', 'MA', '12345',
    T_VARRAY_PHONE('(800)-555-1211', '(800)-555-1212', '(800)-555-1213')),
  T_ADDRESS('4 Hill Street', 'Lost Town', 'CA', '54321',
    T_VARRAY_PHONE('(800)-555-1211', '(800)-555-1212')))
```

Você pode usar `TABLE()` para tratar os dados armazenados nas coleções como uma série de linhas, como mostrado na consulta a seguir:

```
SELECT cn.first_name, cn.last_name, a.street, a.city, a.state, p.*
FROM customers_with_nested_table cn,
TABLE(cn.addresses) a, TABLE(a.phone_numbers) p;

FIRST_NAME LAST_NAME STREET CITY ST COLUMN_VALUE
-----
Steve      Brown      2 State Street Beantown MA (800)-555-1211
Steve      Brown      2 State Street Beantown MA (800)-555-1212
Steve      Brown      2 State Street Beantown MA (800)-555-1213
Steve      Brown      4 Hill Street Lost Town CA (800)-555-1211
Steve      Brown      4 Hill Street Lost Town CA (800)-555-1212
```

A instrução `UPDATE` a seguir mostra como atualizar os números de telefone do endereço 2 State Street; observe que `TABLE()` é usada para obter os endereços como uma série de linhas e que um `varray` contendo os novos números de telefone é fornecido na cláusula `SET`:

```

UPDATE TABLE(
  -- obtém os endereços do cliente n° 1
  SELECT cn.addresses
  FROM customers_with_nested_table cn
  WHERE cn.id = 1
) addr$
SET addr$.phone_numbers =
  t_varray_phone(
    '(800)-555-1214',
    '(800)-555-1215'
  )
WHERE addr$.street = '2 State Street';

```

1 row updated.

A consulta a seguir verifica a alteração:

```

SELECT cn.first_name, cn.last_name, a.street, a.city, a.state, p.*
FROM customers_with_nested_table cn,
     TABLE(cn.addresses) a, TABLE(a.phone_numbers) p;

```

FIRST_NAME	LAST_NAME	STREET	CITY	ST	COLUMN_VALUE
Steve	Brown	2 State Street	Beantown	MA	(800)-555-1214
Steve	Brown	2 State Street	Beantown	MA	(800)-555-1215
Steve	Brown	4 Hill Street	Lost Town	CA	(800)-555-1211
Steve	Brown	4 Hill Street	Lost Town	CA	(800)-555-1212

O suporte para tipos de coleção de múltiplos níveis é uma extensão muito poderosa do software de banco de dados Oracle; considere utilizá-lo nos seus próximos projetos de banco de dados.

## APRIMORAMENTOS FEITOS NAS COLEÇÕES PELO ORACLE DATABASE 10g

Nesta seção, você vai aprender sobre os seguintes aprimoramentos feitos nas coleções pelo Oracle Database 10g:

- Suporte para arrays associativos
- Capacidade de mudar o tamanho ou a precisão de um tipo de elemento
- Capacidade de aumentar o número de elementos em um varray
- Capacidade de usar colunas de varray em tabelas temporárias
- Capacidade de usar um tablespace diferente para tabela de armazenamento de uma tabela aninhada
- Suporte ANSI para tabelas aninhadas

As diversas instruções que criam itens mostradas nesta seção estão contidas no script `collection_schema3.sql`. Esse script cria um usuário chamado `collection_user3` com a senha

collection\_password e cria os tipos de coleção, tabelas e código PL/SQL. Você poderá executar esse script se estiver usando Oracle Database 10g ou superior. Depois que o script terminar, você estará conectado como collection\_user3.

## Arrays associativos

Um array associativo é um conjunto de pares de chave e valor. É possível obter o valor do array usando a chave (que pode ser uma string) ou um número inteiro que especifique a posição do valor no array. O exemplo de procedure a seguir, chamada customers\_associative\_array(), ilustra o uso de arrays associativos:

```
CREATE PROCEDURE customers_associative_array AS
  -- define um tipo de array associativo chamado t_assoc_array;
  -- o valor armazenado em cada elemento do array é do tipo NUMBER
  -- e a chave de índice para acessar cada elemento é uma string VARCHAR2
  TYPE t_assoc_array IS TABLE OF NUMBER INDEX BY VARCHAR2(15);

  -- declara um objeto chamado v_customer_array de tipo t_assoc_array;
  -- v_customer_array será usado para armazenar a idade dos clientes
  v_customer_array t_assoc_array;
BEGIN
  -- atribui os valores a v_customer_array; a chave VARCHAR2 é o
  -- nome do cliente e o valor NUMBER é sua idade
  v_customer_array('Jason') := 32;
  v_customer_array('Steve') := 28;
  v_customer_array('Fred') := 43;
  v_customer_array('Cynthia') := 27;

  -- exibe os valores armazenados em v_customer_array
  DBMS_OUTPUT.PUT_LINE(
    'v_customer_array[''Jason''] = ' || v_customer_array('Jason')
  );
  DBMS_OUTPUT.PUT_LINE(
    'v_customer_array[''Steve''] = ' || v_customer_array('Steve')
  );
  DBMS_OUTPUT.PUT_LINE(
    'v_customer_array[''Fred''] = ' || v_customer_array('Fred')
  );
  DBMS_OUTPUT.PUT_LINE(
    'v_customer_array[''Cynthia''] = ' || v_customer_array('Cynthia')
  );
END customers_associative_array;
/
```

O exemplo a seguir ativa a saída do servidor e chama customers\_associative\_array():

```
SET SERVEROUTPUT ON
CALL customers_associative_array();
v_customer_array['Jason'] = 32
v_customer_array['Steve'] = 28
v_customer_array['Fred'] = 43
v_customer_array['Cynthia'] = 27
```



## Alterando o tamanho de um tipo de elemento

É possível mudar o tamanho de um tipo de elemento em uma coleção, quando o tipo de elemento é um dos tipos de caractere, numérico ou raw (o tipo raw é usado para armazenar dados binários — você vai aprender sobre isso no próximo capítulo). Anteriormente neste capítulo, você viu a instrução a seguir, que cria um tipo de varray chamado `t_varray_address`:

```
CREATE TYPE t_varray_address AS VARRAY(2) OF VARCHAR2(50);
/
```

O exemplo a seguir muda o tamanho dos elementos `VARCHAR2` em `t_varray_address` para 60:

```
ALTER TYPE t_varray_address
MODIFY ELEMENT TYPE VARCHAR2(60) CASCADE;
```

Type altered.

A opção `CASCADE` propaga a alteração para todos os objetos dependentes no banco de dados, o qual, no exemplo, é a tabela `customers_with_varray`, que contém uma coluna chamada `addresses` de tipo `t_varray_address`. Você também pode usar a opção `INVALIDATE` para invalidar todos os objetos dependentes e recompilar o código PL/SQL imediatamente para o tipo.

## Aumentando o número de elementos em um varray

Você pode aumentar o número de elementos em um varray. O exemplo a seguir aumenta o número de elementos em `t_varray_address` para 5:

```
ALTER TYPE t_varray_address
MODIFY LIMIT 5 CASCADE;
```

Type altered.

## Usando varrays em tabelas temporárias

Você pode usar varrays em tabelas temporárias, que são tabelas cujas linhas são temporárias e específicas para uma sessão de usuário (as tabelas temporárias foram abordadas na seção “Criando uma tabela”, no Capítulo 10). O exemplo a seguir cria uma tabela temporária chamada `cust_with_varray_temp_table` que contém um varray chamado `addresses` de tipo `t_varray_address`:

```
CREATE GLOBAL TEMPORARY TABLE cust_with_varray_temp_table (
    id          INTEGER PRIMARY KEY,
    first_name  VARCHAR2(10),
    last_name   VARCHAR2(10),
    addresses   t_varray_address
);
```

## Usando um tablespace diferente para a tabela de armazenamento de uma tabela aninhada

Por padrão, a tabela de armazenamento de uma tabela aninhada é criada no mesmo tablespace da tabela pai (um tablespace é uma área utilizada pelo banco de dados para armazenar objetos, como tabelas — consulte a seção “Criando uma tabela”, no Capítulo 10, para ver os detalhes).

No Oracle Database 10g e superiores, você pode especificar um tablespace diferente para a tabela de armazenamento de uma tabela aninhada. O exemplo a seguir cria uma tabela chamada `cust_with_nested_table` que contém uma tabela aninhada chamada `addresses` de tipo `t_nested_table_address`; observe que o tablespace da tabela de armazenamento `nested_addresses2` é `users`:

```
CREATE TABLE cust_with_nested_table (
    id          INTEGER PRIMARY KEY,
    first_name  VARCHAR2(10),
    last_name   VARCHAR2(10),
    addresses   t_nested_table_address
)
NESTED TABLE
    addresses
STORE AS
    nested_addresses2 TABLESPACE users;
```

Você deve ter um tablespace chamado `users` para que esse exemplo funcione e, por isso, o exemplo foi deixado como comentário no script `collection_schema3.sql`. É possível ver todos os tablespaces a que tem acesso executando a consulta a seguir:

```
SELECT tablespace_name
FROM user_tablespaces;

TABLESPACE_NAME
-----
SYSTEM
SYSAUX
UNDOTBS1
TEMP
USERS
EXAMPLE
```

Se quiser executar a instrução `CREATE TABLE` anterior, edite o exemplo no script `collection_schema3.sql` para referenciar um de seus tablespaces e depois copiar a instrução no SQL\*Plus e executá-la.

## Suporte ANSI para tabelas aninhadas

A especificação ANSI (American National Standards Institute) inclui vários operadores que podem ser usados com tabelas aninhadas. Você vai aprender sobre esses operadores nas seções a seguir.

### Operadores igual e diferente

Os operadores igual (=) e diferente (<>) comparam duas tabelas aninhadas, as quais são consideradas iguais quando satisfazem todas as condições a seguir:

- As tabelas são do mesmo tipo
- As tabelas têm a mesma cardinalidade, isto é, elas contêm o mesmo número de elementos
- Todos os elementos da tabela têm o mesmo valor

A procedure `equal_example()` a seguir ilustra o uso dos operadores igual e diferente:

```

CREATE PROCEDURE equal_example AS
  -- declara um tipo chamado t_nested_table
  TYPE t_nested_table IS TABLE OF VARCHAR2(10);

  -- cria objetos t_nested_table chamados v_customer_nested_table1,
  -- v_customer_nested_table2 e v_customer_nested_table3;
  -- esses objetos são usados para armazenar os nomes dos clientes
  v_customer_nested_table1 t_nested_table :=
    t_nested_table('Fred', 'George', 'Susan');
  v_customer_nested_table2 t_nested_table :=
    t_nested_table('Fred', 'George', 'Susan');
  v_customer_nested_table3 t_nested_table :=
    t_nested_table('John', 'George', 'Susan');

  v_result BOOLEAN;
BEGIN
  -- usa o operador = para comparar v_customer_nested_table1 com
  -- v_customer_nested_table2 (elas contêm os mesmos nomes; portanto,
  -- v_result é definido como verdadeiro)
  v_result := v_customer_nested_table1 = v_customer_nested_table2;
  IF v_result THEN
    DBMS_OUTPUT.PUT_LINE(
      'v_customer_nested_table1 equal to v_customer_nested_table2'
    );
  END IF;

  -- usa o operador <> para comparar v_customer_nested_table1 com
  -- v_customer_nested_table3 (elas não são iguais, pois os
  -- nomes 'Fred' e 'John' são diferentes e v_result é configurado
  -- como verdadeiro)
  v_result := v_customer_nested_table1 <> v_customer_nested_table3;
  IF v_result THEN
    DBMS_OUTPUT.PUT_LINE(
      'v_customer_nested_table1 not equal to v_customer_nested_table3'
    );
  END IF;
END equal_example;
/

```

O exemplo a seguir chama equal\_example():

```

CALL equal_example();
v_customer_nested_table1 equal to v_customer_nested_table2
v_customer_nested_table1 not equal to v_customer_nested_table3

```

### Operadores IN e NOT IN

O operador IN verifica se os elementos de uma tabela aninhada aparecem em outra tabela aninhada. Da mesma forma, NOT IN verifica se os elementos de uma tabela aninhada não aparecem em outra tabela aninhada. A procedure in\_example() a seguir ilustra o uso de IN e NOT IN:

```

CREATE PROCEDURE in_example AS
  TYPE t_nested_table IS TABLE OF VARCHAR2(10);

```

```

v_customer_nested_table1 t_nested_table :=
  t_nested_table('Fred', 'George', 'Susan');
v_customer_nested_table2 t_nested_table :=
  t_nested_table('John', 'George', 'Susan');
v_customer_nested_table3 t_nested_table :=
  t_nested_table('Fred', 'George', 'Susan');
v_result BOOLEAN;
BEGIN
  -- usa o operador IN para verificar se os elementos de v_customer_
  nested_table3
  -- estão em v_customer_nested_table1 (eles estão; portanto, v_result é
  -- configurado como verdadeiro)
  v_result := v_customer_nested_table3 IN
    (v_customer_nested_table1);
  IF v_result THEN
    DBMS_OUTPUT.PUT_LINE(
      'v_customer_nested_table3 in v_customer_nested_table1'
    );
  END IF;

  -- usa o operador NOT IN para verificar se os elementos de
  -- v_customer_nested_table3 não estão em v_customer_nested_table2
  -- (eles não estão; portanto, v_result é configurado como verdadeiro)
  v_result := v_customer_nested_table3 NOT IN
    (v_customer_nested_table2);
  IF v_result THEN
    DBMS_OUTPUT.PUT_LINE(
      'v_customer_nested_table3 not in v_customer_nested_table2'
    );
  END IF;
END in_example;
/

```

O exemplo a seguir chama in\_example():

```

CALL in_example();
v_customer_nested_table3 in v_customer_nested_table1
v_customer_nested_table3 not in v_customer_nested_table2

```

### **Operador SUBMULTISET**

O operador SUBMULTISET verifica se os elementos de uma tabela aninhada são um subconjunto de outra tabela aninhada. A procedure submultiset\_example() a seguir ilustra o uso de SUBMULTISET:

```

CREATE PROCEDURE submultiset_example AS
  TYPE t_nested_table IS TABLE OF VARCHAR2(10);
  v_customer_nested_table1 t_nested_table :=
    t_nested_table('Fred', 'George', 'Susan');
  v_customer_nested_table2 t_nested_table :=
    t_nested_table('George', 'Fred', 'Susan', 'John', 'Steve');
  v_result BOOLEAN;
BEGIN

```

```

-- usa o operador SUBMULTISET para verificar se os elementos de
-- v_customer_nested_table1 são um subconjunto de v_customer_nested_
table2
-- (eles são; portanto, v_result é configurado como verdadeiro)
v_result :=
  v_customer_nested_table1 SUBMULTISET OF v_customer_nested_table2;
IF v_result THEN
  DBMS_OUTPUT.PUT_LINE(
    'v_customer_nested_table1 subset of v_customer_nested_table2'
  );
END IF;
END submultiset_example;
/

```

O exemplo a seguir chama submultiset\_example():

```

CALL submultiset_example();
customer_nested_table1 subset of customer_nested_table2

```

### Operador MULTISET

O operador MULTISET retorna uma tabela aninhada cujos elementos são configurados com certas combinações de elementos de duas tabelas aninhadas fornecidas. Existem três operadores MULTISET:

- **MULTISET UNION** retorna uma tabela aninhada cujos elementos são configurados como a soma dos elementos de duas tabelas aninhadas fornecidas.
- **MULTISET INTERSECT** retorna uma tabela aninhada cujos elementos são configurados com os elementos comuns às duas tabelas aninhadas fornecidas.
- **MULTISET EXCEPT** retorna uma tabela aninhada cujos elementos estão na primeira tabela aninhada fornecida, mas não na segunda.

Você também pode usar uma das seguintes opções com MULTISET:

- **ALL** indica que todos os elementos aplicáveis estão na tabela aninhada retornada. ALL é o padrão. Por exemplo, MULTISET UNION ALL retorna uma tabela aninhada cujos elementos são configurados como a soma dos elementos de duas tabelas aninhadas fornecidas e todos os elementos, incluindo os duplicados, estão na tabela aninhada retornada.
- **DISTINCT** indica que somente os elementos não duplicados (isto é, distintos) estão na tabela aninhada retornada. Por exemplo, MULTISET UNION DISTINCT retorna uma tabela aninhada cujos elementos são configurados como a soma dos elementos de duas tabelas aninhadas fornecidas, mas os duplicados são removidos da tabela aninhada retornada.

A procedure multiset\_example() a seguir ilustra o uso de MULTISET:

```

CREATE PROCEDURE multiset_example AS
  TYPE t_nested_table IS TABLE OF VARCHAR2(10);
  v_customer_nested_table1 t_nested_table :=
    t_nested_table('Fred', 'George', 'Susan');
  v_customer_nested_table2 t_nested_table :=
    t_nested_table('George', 'Steve', 'Rob');
  v_customer_nested_table3 t_nested_table;

```

```

v_count INTEGER;
BEGIN
  -- usa MULTISSET UNION (retorna uma tabela aninhada cujos elementos
  -- são configurados como a soma das duas tabelas aninhadas fornecidas)
  v_customer_nested_table3 :=
    v_customer_nested_table1 MULTISSET UNION
    v_customer_nested_table2;
  DBMS_OUTPUT.PUT('UNION: ');
  FOR v_count IN 1..v_customer_nested_table3.COUNT LOOP
    DBMS_OUTPUT.PUT(v_customer_nested_table3(v_count) || ' ');
  END LOOP;
  DBMS_OUTPUT.PUT_LINE(' ');

  -- usa MULTISSET UNION DISTINCT (DISTINCT indica que somente
  -- os elementos não duplicados das duas tabelas aninhadas fornecidas
  -- são configurados na tabela aninhada retornada)
  v_customer_nested_table3 :=
    v_customer_nested_table1 MULTISSET UNION DISTINCT
    v_customer_nested_table2;
  DBMS_OUTPUT.PUT('UNION DISTINCT: ');
  FOR v_count IN 1..v_customer_nested_table3.COUNT LOOP
    DBMS_OUTPUT.PUT(v_customer_nested_table3(v_count) || ' ');
  END LOOP;
  DBMS_OUTPUT.PUT_LINE(' ');

  -- usa MULTISSET INTERSECT (retorna uma tabela aninhada cujos elementos
  -- são configurados com os elementos comuns às duas
  -- tabelas aninhadas fornecidas)
  v_customer_nested_table3 :=
    v_customer_nested_table1 MULTISSET INTERSECT
    v_customer_nested_table2;
  DBMS_OUTPUT.PUT('INTERSECT: ');
  FOR v_count IN 1..v_customer_nested_table3.COUNT LOOP
    DBMS_OUTPUT.PUT(v_customer_nested_table3(v_count) || ' ');
  END LOOP;
  DBMS_OUTPUT.PUT_LINE(' ');

  -- usa MULTISSET EXCEPT (retorna uma tabela aninhada cujos
  -- elementos estão na primeira tabela aninhada, mas não na
  -- segunda)
  v_customer_nested_table3 :=
    v_customer_nested_table1 MULTISSET EXCEPT
    v_customer_nested_table2;
  DBMS_OUTPUT.PUT_LINE('EXCEPT: ');
  FOR v_count IN 1..v_customer_nested_table3.COUNT LOOP
    DBMS_OUTPUT.PUT(v_customer_nested_table3(v_count) || ' ');
  END LOOP;
END multiset_example;
/

```

O exemplo a seguir chama `multiset_example()`:

```
CALL multiset_example();
UNION: Fred George Susan George Steve Rob
UNION DISTINCT: Fred George Susan Steve Rob
INTERSECT: George
EXCEPT:
```

### Função **CARDINALITY()**

A função `CARDINALITY()` retorna o número de elementos em uma coleção. A procedure `cardinality_example()` a seguir ilustra o uso de `CARDINALITY()`:

```
CREATE PROCEDURE cardinality_example AS
  TYPE t_nested_table IS TABLE OF VARCHAR2(10);
  v_customer_nested_table1 t_nested_table :=
    t_nested_table('Fred', 'George', 'Susan');
  v_cardinality INTEGER;
BEGIN
  -- chama CARDINALITY() para obter o número de elementos em
  -- v_customer_nested_table1
  v_cardinality := CARDINALITY(v_customer_nested_table1);
  DBMS_OUTPUT.PUT_LINE('v_cardinality = ' || v_cardinality);
END cardinality_example;
/
```

O exemplo a seguir chama `cardinality_example()`:

```
CALL cardinality_example();
v_cardinality = 3
```

### Operador **MEMBER OF**

O operador `MEMBER OF` verifica se um elemento está em uma tabela aninhada. A procedure `member_of_example()` a seguir ilustra o uso de `MEMBER OF`:

```
CREATE PROCEDURE member_of_example AS
  TYPE t_nested_table IS TABLE OF VARCHAR2(10);
  v_customer_nested_table1 t_nested_table :=
    t_nested_table('Fred', 'George', 'Susan');
  v_result BOOLEAN;
BEGIN
  -- usa MEMBER OF para verificar se 'George' está em
  -- v_customer_nested_table1 (ele está; portanto, v_result é configurado
  -- como verdadeiro)
  v_result := 'George' MEMBER OF v_customer_nested_table1;
  IF v_result THEN
    DBMS_OUTPUT.PUT_LINE(''George'' is a member');
  END IF;
END member_of_example;
/
```

O exemplo a seguir chama `member_of_example()`:

```
CALL member_of_example();
'George' is a member
```

### Função SET()

A função `SET()` primeiro converte uma tabela aninhada em um conjunto, em seguida remove os elementos duplicados do conjunto e, finalmente, retorna o conjunto como uma tabela aninhada. A procedure `set_example()` a seguir ilustra o uso de `SET()`:

```
CREATE PROCEDURE set_example AS
  TYPE t_nested_table IS TABLE OF VARCHAR2(10);
  v_customer_nested_table1 t_nested_table :=
    t_nested_table('Fred', 'George', 'Susan', 'George');
  v_customer_nested_table2 t_nested_table;
  v_count INTEGER;
BEGIN
  -- chama SET() para converter uma tabela aninhada em um conjunto,
  -- remove elementos duplicados do conjunto e retorna o conjunto
  -- como uma tabela aninhada
  v_customer_nested_table2 := SET(v_customer_nested_table1);
  DBMS_OUTPUT.PUT('v_customer_nested_table2: ');
  FOR v_count IN 1..v_customer_nested_table2.COUNT LOOP
    DBMS_OUTPUT.PUT(v_customer_nested_table2(v_count) || ' ');
  END LOOP;
  DBMS_OUTPUT.PUT_LINE(' ');
END set_example;
/
```

O exemplo a seguir chama `set_example()`:

```
CALL set_example();
v_customer_nested_table2: Fred George Susan
```

### Operador IS A SET

O operador `IS A SET` verifica se os elementos de uma tabela aninhada são distintos. A procedure `is_a_set_example()` a seguir ilustra o uso de `IS A SET`:

```
CREATE PROCEDURE is_a_set_example AS
  TYPE t_nested_table IS TABLE OF VARCHAR2(10);
  v_customer_nested_table1 t_nested_table :=
    t_nested_table('Fred', 'George', 'Susan', 'George');
  v_result BOOLEAN;
BEGIN
  -- usa o operador IS A SET para verificar se os elementos de
  -- v_customer_nested_table1 são distintos (eles não são; portanto,
  -- v_result é configurado como falso)
  v_result := v_customer_nested_table1 IS A SET;
  IF v_result THEN
    DBMS_OUTPUT.PUT_LINE('Elements are all unique');
  ELSE
    DBMS_OUTPUT.PUT_LINE('Elements contain duplicates');
  END IF;
END IF;
```



```
END is_a_set_example;
/
```

O exemplo a seguir chama `is_a_set_example()`:

```
CALL is_a_set_example();
Elements contain duplicates
```

### Operador IS EMPTY

O operador `IS EMPTY` verifica se uma tabela aninhada não contém elementos. A procedure `is_empty_example()` a seguir ilustra o uso de `IS EMPTY`:

```
CREATE PROCEDURE is_empty_example AS
  TYPE t_nested_table IS TABLE OF VARCHAR2(10);
  v_customer_nested_table1 t_nested_table :=
    t_nested_table('Fred', 'George', 'Susan');
  v_result BOOLEAN;
BEGIN
  -- usa o operador IS EMPTY para verificar se
  -- v_customer_nested_table1 está vazia (ela não está; portanto,
  -- v_result é configurado como falso)
  v_result := v_customer_nested_table1 IS EMPTY;
  IF v_result THEN
    DBMS_OUTPUT.PUT_LINE('Nested table is empty');
  ELSE
    DBMS_OUTPUT.PUT_LINE('Nested table contains elements');
  END IF;
END is_empty_example;
/
```

O exemplo a seguir chama `is_empty_example()`:

```
CALL is_empty_example();
Nested table contains elements
```

### Função COLLECT()

A função `COLLECT()` retorna uma tabela aninhada a partir de um conjunto de elementos. A consulta a seguir ilustra o uso de `COLLECT()`:

```
SELECT COLLECT(first_name)
FROM customers_with_varray;

COLLECT (FIRST_NAME)
-----
SYSTPfrFhAg+WRJGwW7ma9zylKA=('Steve', 'John')
```

Você pode usar `CAST()` para converter os elementos retornado por `COLLECT()` para um tipo específico, como mostrado na consulta a seguir:

```
SELECT CAST(COLLECT(first_name) AS t_table)
FROM customers_with_varray;

CAST(COLLECT (FIRST_NAME)AST_Tabela)
-----
T_TABLE('Steve', 'John')
```

Para sua referência, o tipo `t_table` usado no exemplo anterior é criado pela instrução a seguir no script `collection_schema3.sql`:

```
CREATE TYPE t_table AS TABLE OF VARCHAR2(10);
/
```

### **Função POWERMULTISET()**

A função `POWMULTISET()` retorna todas as combinações de elementos de uma tabela aninhada dada, como mostrado na consulta a seguir:

```
SELECT *
FROM TABLE(
    POWERMULTISET(t_table('This', 'is', 'a', 'test'))
);

COLUMN_VALUE
-----
T_TABLE('This')
T_TABLE('is')
T_TABLE('This', 'is')
T_TABLE('a')
T_TABLE('This', 'a')
T_TABLE('is', 'a')
T_TABLE('This', 'is', 'a')
T_TABLE('test')
T_TABLE('This', 'test')
T_TABLE('is', 'test')
T_TABLE('This', 'is', 'test')
T_TABLE('a', 'test')
T_TABLE('This', 'a', 'test')
T_TABLE('is', 'a', 'test')
T_TABLE('This', 'is', 'a', 'test')
```

### **Função POWERMULTISET\_BY\_CARDINALITY()**

A função `POWMULTISET_BY_CARDINALITY()` retorna as combinações dos elementos de uma tabela aninhada dada que têm um número de elementos (ou “cardinalidade”) especificado. A consulta a seguir ilustra o uso de `POWMULTISET_BY_CARDINALITY()`, especificando uma cardinalidade igual a 3:

```
SELECT *
FROM TABLE(
    POWERMULTISET_BY_CARDINALITY(
        t_table('This', 'is', 'a', 'test'), 3
    )
);

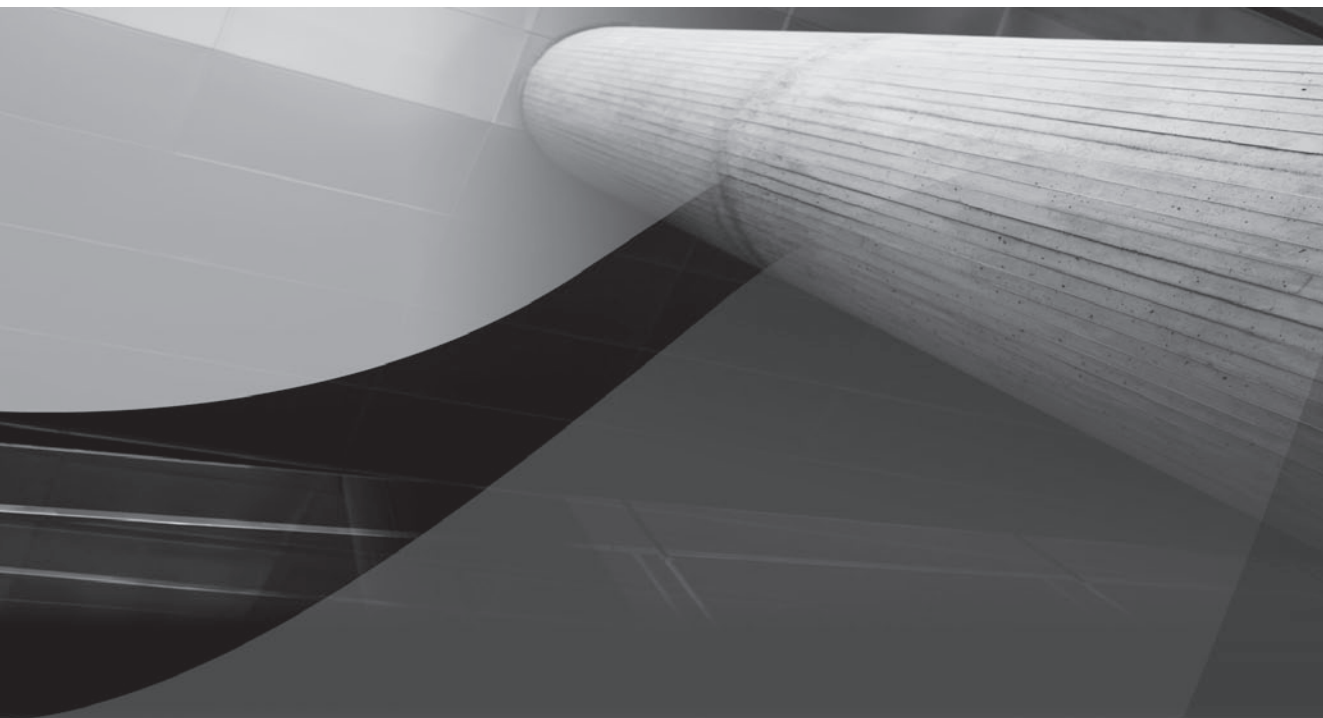
COLUMN_VALUE
-----
T_TABLE('This', 'is', 'a')
T_TABLE('This', 'is', 'test')
T_TABLE('This', 'a', 'test')
T_TABLE('is', 'a', 'test')
```

## RESUMO

Neste capítulo, você aprendeu que:

- As coleções permitem armazenar conjuntos de elementos.
- Existem três tipos de coleções: varrays, tabelas aninhadas e arrays associativos.
- Um varray é semelhante a um array em Java; você pode usar um varray para armazenar um conjunto ordenado de elementos, com cada elemento tendo um índice associado. Os elementos de um varray são do mesmo tipo e um varray tem uma só dimensão. Um varray tem um tamanho máximo que você define ao criá-lo, mas é possível alterar o tamanho posteriormente.
- Uma tabela aninhada é uma tabela incorporada dentro de outra e você pode inserir, atualizar e excluir elementos individuais em uma tabela aninhada. Como você pode modificar elementos individuais de uma tabela aninhada, ela é mais flexível do que um varray — um varray só pode ser modificado como um todo. Uma tabela aninhada não tem um tamanho máximo e você pode armazenar nela um número arbitrário de elementos.
- Uma array associativo é um conjunto de pares de chave e valor. Você pode obter o valor do array usando a chave (que pode ser uma string) ou um número inteiro que especifique a posição do valor no array. Um array associativo é semelhante a uma hash table em linguagens de programação como Java.
- Uma coleção pode ela própria conter coleções incorporadas. Tal coleção é conhecida como coleção de múltiplos níveis.

No próximo capítulo, você vai aprender sobre large objects.



# CAPÍTULO 14

Large objects  
(objetos grandes)

Neste capítulo, você vai:

- Aprender sobre large objects (LOBs)
- Ver arquivos cujo conteúdo será usado para preencher exemplos de LOBs
- Examinar as diferenças entre os diferentes tipos de LOBs
- Criar tabelas contendo LOBs
- Usar LOBs em SQL e em PL/SQL
- Examinar os tipos `LONG` e `LONG RAW`
- Ver as melhorias feitas nos LOBs pelo Oracle Database 10g e 11g

## INTRODUÇÃO AOS LARGE OBJECTS (LOBs)

Os sites atuais exigem mais do que apenas o armazenamento e a recuperação de texto e números: eles também exigem multimídia. Conseqüentemente, os bancos de dados agora estão sendo obrigados a armazenar itens como músicas e vídeos. Antes do lançamento do Oracle Database 8, você tinha que armazenar blocos grandes de dados de caractere usando o tipo `LONG` do banco de dados e os blocos grandes de dados binários tinham que ser armazenados usando-se o tipo `LONG RAW` ou o tipo mais curto `RAW`.

Com o lançamento do Oracle Database 8, foi introduzida uma nova classe de tipos de banco de dados, conhecidos como *large objects* ou LOBs. Os LOBs podem ser usados para armazenar dados binários, dados de caractere e referências a arquivos. Os dados binários podem conter imagens, músicas, vídeos, documentos, executáveis etc. Os LOBs podem armazenar até 128 terabytes de dados, dependendo da configuração do banco de dados.

## OS ARQUIVOS DE EXEMPLO

Neste capítulo, serão usados os seguintes arquivos:

- `textContent.txt` Um arquivo de texto
- `binaryContent.doc` Um arquivo do Microsoft Word



### NOTA

*Esses arquivos estão contidos no diretório `sample_files`, que é criado quando você extrai o arquivo Zip deste livro. Se quiser acompanhar os exemplos, você deve copiar o diretório `sample_files` para a partição C em seu servidor de banco de dados. Se estiver usando Linux ou Unix, você pode copiar o diretório para uma de suas partições.*

O arquivo `textContent.txt` contém um trecho da peça *Macbeth* de Shakespeare. O texto a seguir é a fala de Macbeth pouco antes de ser assassinado:

To-morrow, and to-morrow, and to-morrow,  
 Creeps in this petty pace from day to day,  
 To the last syllable of recorded time;  
 And all our yesterdays have lighted fools  
 The way to a dusty death. Out, out, brief candle!  
 Life's but a walking shadow; a poor player,  
 That struts and frets his hour upon the stage,  
 And then is heard no more: it is a tale  
 Told by an idiot, full of sound and fury,  
 Signifying nothing.\*

O arquivo `binaryContent.doc` é um documento do Word que contém o mesmo texto de `textContent.txt`. (Um documento do Word é um arquivo binário.) Embora um documento do Word seja utilizado nos exemplos, você pode usar qualquer arquivo binário, por exemplo, MP3, DivX, JPEG, MPEG, PDF ou EXE. Os exemplos foram testados com todos esses tipos de arquivos.

## TIPOS DE LARGE OBJECT

Existem quatro tipos de LOB:

- **CLOB** O tipo Character LOB, usado para armazenar dados de caractere.
- **NCLOB** O tipo National Character Set LOB, usado para armazenar dados de caractere multibyte (utilizado normalmente para caracteres que não estão no idioma inglês). Você pode aprender mais sobre os conjuntos de caracteres que não estão no idioma inglês no *Oracle Database Globalization Support Guide*, publicado pela Oracle Corporation.
- **BLOB** O tipo Binary LOB, usado para armazenar dados binários.
- **BFILE** O tipo Binary FILE, usado para armazenar um ponteiro para um arquivo. O arquivo pode estar em um disco rígido, CD, DVD, disco Blu-Ray, HD-DVD ou em qualquer outro dispositivo que seja acessível por meio do sistema de arquivos do servidor de banco de dados. O arquivo em si nunca é armazenado no banco de dados; somente um ponteiro para o arquivo.

Antes do Oracle Database 8, sua única escolha para armazenar grandes volumes de dados era usar os tipos `LONG` e `LONG RAW` (você também podia usar o tipo `RAW` para armazenar dados binários do que 4 kilobytes). Os tipos LOB têm três vantagens em relação a esses tipos mais antigos:

- Um LOB pode armazenar até 128 terabytes de dados. Isso é bem mais do que você pode armazenar em uma coluna `LONG` ou `LONG RAW`, que só pode armazenar 2 gigabytes de dados.
- Uma tabela pode ter várias colunas LOB, mas só pode ter uma coluna `LONG` ou `LONG RAW`.
- Os dados LOB podem ser acessados em ordem aleatória; os dados `LONG` e `LONG RAW` só podem ser acessados em ordem seqüencial.

---

\* N. de R.: Amanhã e amanhã, e amanhã/chegando no passo impensado de um dia após um dia/até a última sílaba do tempo registrado./ E cada dia de ontem iluminou, aos tolos que nós somos,/ o caminho para o pó da morte. Apagai-vos, vela tão pequena!/ A vida é apenas uma sombra que caminha, um pobre ator,/Que gagueja e vacila a sua hora sobre o palco/ e depois nunca mais se ouve. É uma história/ contada por um idiota, cheia de som e fúria,/ significando nada.

Um LOB consiste em duas partes:

- **O localizador do LOB** Um ponteiro que especifica a localização dos dados do LOB
- **Os dados do LOB** Os dados de caractere ou byte armazenados no LOB

Dependendo do volume de dados armazenados em uma coluna CLOB, NCLOB ou BLOB, eles serão armazenados dentro ou fora da tabela. Se os dados forem menores do que 4 kilobytes, eles serão armazenados na mesma tabela; caso contrário, eles serão armazenados fora da tabela. Com uma coluna BFILE, somente o localizador é armazenado na tabela — e o localizador aponta para um arquivo externo armazenado no sistema de arquivos.

## CRIANDO TABELAS CONTENDO LARGE OBJECTS

Nesta seção, estas três tabelas são usadas:

- A tabela `clob_content`, que contém uma coluna CLOB chamada `clob_column`
- A tabela `blob_content`, que contém uma coluna BLOB chamada `blob_column`
- A tabela `bfile_content`, que contém uma coluna BFILE chamada `bfile_column`

Fornecemos um script SQL\*Plus chamado `lob_schema.sql` no diretório SQL. Esse script pode ser executado no Oracle Database 8 e superiores. O script cria um usuário chamado `lob_user` com a senha `lob_password` e cria as tabelas e o código PL/SQL utilizados na primeira parte deste capítulo. Depois que o script terminar, você estará conectado como `lob_user`.

As três tabelas são criadas com as instruções a seguir do script:

```
CREATE TABLE clob_content (
    id          INTEGER PRIMARY KEY,
    clob_column CLOB NOT NULL
);

CREATE TABLE blob_content (
    id          INTEGER PRIMARY KEY,
    blob_column BLOB NOT NULL
);

CREATE TABLE bfile_content (
    id          INTEGER PRIMARY KEY,
    bfile_column BFILE NOT NULL
);
```

## USANDO LARGE OBJECTS EM SQL

Nesta seção, você vai aprender a usar SQL para manipular large objects. Começaremos examinando os objetos CLOB e BLOB e depois passaremos para os objetos BFILE.

### Usando CLOBs e BLOBs

As seções a seguir mostram como preencher objetos CLOB e BLOB com dados, como recuperar os dados e depois modificá-los.

**Preenchendo CLOBs e BLOBs com dados**

As instruções INSERT a seguir adicionam duas linhas na tabela `clob_content`; observe o uso da função `TO_CLOB()` para converter o texto em um CLOB:

```
INSERT INTO clob_content (
    id, clob_column
) VALUES (
    1, TO_CLOB('Creeps in this petty pace')
);

INSERT INTO clob_content (
    id, clob_column
) VALUES (
    2, TO_CLOB(' from day to day')
);
```

As instruções INSERT a seguir adicionam duas linhas na tabela `blob_content`; observe o uso da função `TO_BLOB()` para converter os números em um BLOB (a primeira instrução contém um número binário e a segunda contém um número hexadecimal):

```
INSERT INTO blob_content (
    id, blob_column
) VALUES (
    1, TO_BLOB('10011101010101111')
);

INSERT INTO blob_content (
    id, blob_column
) VALUES (
    2, TO_BLOB('A0FFB71CF90DE')
);
```

**Recuperando dados de CLOBs**

A consulta a seguir recupera as linhas da tabela `clob_content`:

```
SELECT *
FROM clob_content;

          ID
-----
CLOB_COLUMN
-----
          1
Creeps in this petty pace

          2
 from day to day
```

A próxima consulta tenta recuperar a linha da tabela `blob_content` e falha:

```
SELECT *
FROM blob_content;
SP2-0678: Column or attribute type can not be displayed by SQL*Plus
```



Esse exemplo falha porque o SQL\*Plus não pode exibir os dados binários de um BLOB. Você vai aprender a recuperar os dados de um BLOB na seção “Usando large objects em PL/SQL”. Entretanto, você pode obter as colunas da tabela que não são BLOB:

```
SELECT id
FROM blob_content;

      ID
-----
      1
      2
```

### **Modificando os dados de CLOBs e BLOBs**

Execute as instruções UPDATE e INSERT mostradas nesta seção. As instruções UPDATE a seguir mostram como você modifica o conteúdo de um CLOB e de um BLOB:

```
UPDATE clob_content
SET clob_column = TO_CLOB('What light through yonder window breaks')
WHERE id = 1;

UPDATE blob_content
SET blob_column = TO_BLOB('111001101010101111')
WHERE id = 1;
```

Você também pode inicializar o localizador do LOB sem armazenar os dados reais no LOB. Para tanto, use a função EMPTY\_CLOB() para armazenar um CLOB vazio e a função EMPTY\_BLOB() para armazenar um BLOB vazio:

```
INSERT INTO clob_content(
    id, clob_column
) VALUES (
    3, EMPTY_CLOB()
);

INSERT INTO blob_content(
    id, blob_column
) VALUES (
    3, EMPTY_BLOB()
);
```

Essas instruções inicializam o localizador do LOB, mas configuram os dados do LOB como vazios. Você também pode usar EMPTY\_CLOB() e EMPTY\_BLOB() em instruções UPDATE quando quiser esvaziar os dados do LOB. Por exemplo:

```
UPDATE clob_content
SET clob_column = EMPTY_CLOB()
WHERE id = 1;

UPDATE blob_content
SET blob_column = EMPTY_BLOB()
WHERE id = 1;
```

Se você executou alguma das instruções `INSERT` e `UPDATE` mostradas nesta seção, reverta as alterações para que sua saída corresponda às do restante deste capítulo:

```
ROLLBACK;
```

## Usando BFILEs

Um `BFILE` armazena um ponteiro para um arquivo que é acessível por meio do sistema de arquivos do servidor de banco de dados. O ponto importante a ser lembrado é que esses arquivos são armazenados fora do banco de dados. Um `BFILE` pode apontar para arquivos localizados em qualquer mídia: disco rígido, CD, DVD, Blu-Ray, HD-DVD etc.

### NOTA

*Um `BFILE` contém um ponteiro para um arquivo externo. O arquivo em si nunca é armazenado no banco de dados; somente um ponteiro para esse arquivo é armazenado. O arquivo deve ser acessível por meio do sistema de arquivos do servidor de banco de dados.*

## Criando um objeto de diretório

Antes de poder armazenar um ponteiro para um arquivo em um `BFILE`, você deve primeiro criar um objeto de diretório no banco de dados. O objeto de diretório armazena o diretório no sistema de arquivos em que os arquivos estão localizados. Um objeto de diretório é criado com a instrução `CREATE DIRECTORY`; para executar essa instrução, você precisa do privilégio de banco de dados `CREATE ANY DIRECTORY`.

O exemplo a seguir (contido em `lob_schema.sql`) cria um objeto de diretório chamado `SAMPLE_FILES_DIR` para o diretório de sistema de arquivos `C:\sample_files`:

```
CREATE DIRECTORY SAMPLE_FILES_DIR AS 'C:\sample_files';
```

### NOTA

*O Windows usa o caractere de barra invertida (\) em diretórios, enquanto o Linux e o Unix usam o caractere de barra normal (/). Além disso, se o seu diretório `sample_files` não está armazenado na partição `C`, você precisa especificar o caminho apropriado no exemplo anterior.*

Quando você cria um objeto de diretório, deve certificar-se de que:

- O diretório exista no sistema de arquivos
- A conta de usuário no sistema operacional usado para instalar o software Oracle tenha permissão de leitura no diretório e em todos os arquivos apontados por um `BFILE` no banco de dados

Se você estiver usando Windows, não deve se preocupar com o segundo ponto. O software de banco de dados Oracle provavelmente foi instalado usando uma conta de usuário com privilégios de administrador e essa conta tem permissão de leitura para todo o sistema de arquivos. Se estiver usando Linux ou Unix, você precisará conceder acesso de leitura para a conta de usuário Oracle apropriada que possua o banco de dados (para tanto, use o comando `chmod`).

### **Preenchendo uma coluna BFILE com um ponteiro para um arquivo**

Como um BFILE é apenas um ponteiro para um arquivo externo, preencher uma coluna BFILE é muito simples. Basta usar a função `BFILENAME()` do banco de dados Oracle para preencher um BFILE com um ponteiro para seu arquivo externo. A função `BFILENAME()` aceita dois parâmetros: o nome do objeto de diretório e o nome do arquivo.

Por exemplo, a instrução `INSERT` a seguir adiciona uma linha na tabela `bfile_content`; observe que a função `BFILENAME()` é usada para preencher `bfile_column` com um ponteiro para o arquivo `textContent.txt`:

```
INSERT INTO bfile_content (
    id, bfile_column
) VALUES (
    1, BFILENAME('SAMPLE_FILES_DIR', 'textContent.txt')
);
```

A próxima instrução `INSERT` adiciona uma linha na tabela `bfile_content`; observe que a função `BFILENAME()` é usada para preencher `bfile_column` com um ponteiro para o arquivo `binaryContent.doc`:

```
INSERT INTO bfile_content (
    id, bfile_column
) VALUES (
    2, BFILENAME('SAMPLE_FILES_DIR', 'binaryContent.doc')
);
```

A consulta a seguir tenta recuperar as linhas de `bfile_content` e falha, pois o SQL\*Plus não pode exibir o conteúdo de um BFILE:

```
SELECT *
FROM bfile_content;
SP2-0678: Column or attribute type can not be displayed by SQL*Plus
```

Você vai aprender a usar PL/SQL para acessar o conteúdo de um BFILE ou de um BLOB em seguida.

## **USANDO LARGE OBJECTS EM PL/SQL**

Nesta seção, você vai aprender a usar LOBs em PL/SQL. Começaremos examinando os métodos do pacote `DBMS_LOB`, o qual vem com o banco de dados. Posteriormente, você verá muitos programas PL/SQL que mostram como usar os métodos `DBMS_LOB` para ler os dados de um LOB, copiar dados de um LOB para outro, procurar dados em um LOB, copiar dados de um arquivo em um LOB, copiar dados de um LOB em um arquivo e muito mais.

A Tabela 14-1 resume os métodos mais usados do pacote `DBMS_LOB`. Nas seções a seguir, você verá os detalhes de alguns dos métodos mostrados na tabela anterior. Você pode ver todos os métodos de `DBMS_LOB` no manual *Oracle Database PL/SQL Packages and Types Reference*, publicado pela Oracle Corporation.

**Tabela 14-1** *Métodos de DBMS\_LOB*

<b>Método</b>	<b>Descrição</b>
<code>APPEND(lob_dest, lob_ori)</code>	Adiciona os dados lidos de <i>lob_ori</i> ao final de <i>lob_dest</i> .
<code>CLOSE(lob)</code>	Fecha um LOB aberto anteriormente.
<code>COMPARE(lob1, lob2, quantidade, deslocamento1, deslocamento2)</code>	<p>Compara os dados armazenados em <i>lob1</i> e <i>lob2</i>, começando no <i>deslocamento1</i> de <i>lob1</i> e no <i>deslocamento2</i> de <i>lob2</i>. Os deslocamentos sempre começam em 1, que é a posição do primeiro caractere ou byte nos dados.</p> <p>Os dados dos LOBs são comparados em cima de um número máximo de caracteres ou bytes (o máximo é especificado em <i>quantidade</i>).</p>
<code>CONVERTTOBLOB(blob_dest, clob_ori, quantidade, desl_dest, desl_ori, idcc_blob, contexto_ling, alerta)</code>	<p>Converte os dados de caractere lidos de <i>clob_ori</i> em dados binários gravados em <i>blob_dest</i>.</p> <p>A leitura começa em <i>desl_ori</i> de <i>clob_ori</i> e a gravação começa em <i>desl_dest</i> de <i>blob_dest</i>.</p> <p><i>idcc_blob</i> é o conjunto de caracteres desejado para os dados convertidos gravados em <i>blob_dest</i>. Normalmente, você deve usar <code>DBMS_LOB.DEFAULT_CSID</code>, que é o conjunto de caracteres padrão do banco de dados.</p> <p><i>contexto_ling</i> é o contexto da linguagem a ser usado na conversão dos caracteres lidos de <i>clob_ori</i>. Normalmente, você deve usar <code>DBMS_LOB.DEFAULT_LANG_CTX</code>, que é o contexto de linguagem padrão do banco de dados.</p> <p><i>alerta</i> é configurado como <code>DBMS_LOB.WARN_INCONVERTIBLE_CHAR</code> se havia um caractere que não poderia ser convertido.</p>
<code>CONVERTTOCLOB(clob_dest, blob_ori, quantidade, desl_dest, desl_ori, idcc_blob, contexto_ling, alerta)</code>	<p>Converte os dados binários lidos de <i>blob_ori</i> em dados de caractere gravados em <i>clob_dest</i>.</p> <p><i>idcc_blob</i> é o conjunto de caracteres dos dados lidos de <i>blob_dest</i>. Normalmente, você deve usar <code>DBMS_LOB.DEFAULT_CSID</code>.</p> <p><i>contexto_ling</i> é o contexto da linguagem a ser usado na gravação dos caracteres convertidos em <i>clob_dest</i>. Normalmente, você deve usar <code>DBMS_LOB.DEFAULT_LANG_CTX</code>.</p> <p><i>alerta</i> é configurado como <code>DBMS_LOB.WARN_INCONVERTIBLE_CHAR</code> se havia um caractere que não poderia ser convertido.</p>
<code>COPY(lob_dest, lob_ori, quantidade, desl_dest, desl_ori)</code>	Copia os dados de <i>lob_ori</i> em <i>lob_dest</i> , começando nos deslocamentos, para uma quantidade total de caracteres ou bytes.
<code>CREATETEMPORARY(lob, cache, duração)</code>	Cria um LOB temporário no tablespace temporário padrão do usuário.
<code>ERASE(lob, quantidade, deslocamento)</code>	Apaga os dados de um LOB, começando no deslocamento, para uma quantidade total de caracteres ou bytes.
<code>FILECLOSE(bfile)</code>	Fecha <i>bfile</i> . Você deve usar o método <code>CLOSE()</code> , mais recente, em vez de <code>FILECLOSE()</code> .
<code>FILECLOSEALL()</code>	Fecha todos os <code>BFILEs</code> abertos anteriormente.
<code>FILEEXISTS(bfile)</code>	Verifica se o arquivo externo apontado por <i>bfile</i> realmente existe.
<code>FILEGETNAME(bfile, diretório, nome_arq)</code>	Retorna o diretório e o nome de arquivo do arquivo externo apontado por <i>bfile</i> .
<code>FILEISOPEN(bfile)</code>	Verifica se <i>bfile</i> está aberto. Você deve usar o método <code>ISOPEN()</code> , mais recente, em vez de <code>FILEISOPEN()</code> .

(continua)

**Tabela 14-1** Métodos de DBMS\_LOB (continuação)

Método	Descrição
FILEOPEN( <i>bfile</i> , <i>modo_abertura</i> )	Abre <i>bfile</i> no modo indicado, o qual só pode ser configurado como DBMS_LOB.FILE_READONLY, informando que o arquivo só pode ser lido (e nunca gravado). Você deve usar o método OPEN(), mais recente, em vez de FILEOPEN().
FREETEMPORARY( <i>lob</i> )	Libera um LOB temporário.
GETCHUNKSIZE( <i>lob</i> )	Retorna o tamanho do trecho usado ao ler e gravar os dados armazenados no LOB. Um trecho é uma unidade de dados.
GET_STORAGE_LIMIT()	Retorna o tamanho máximo permitido para um LOB.
GETLENGTH( <i>lob</i> )	Obtém o comprimento dos dados armazenados no LOB.
INSTR( <i>lob</i> , <i>padrão</i> , <i>deslocamento</i> , <i>n</i> )	Retorna a posição inicial dos caracteres ou bytes correspondentes à <i>n</i> -ésima ocorrência de um padrão nos dados do LOB. Os dados são lidos no LOB a partir do deslocamento.
ISOPEN( <i>lob</i> )	Verifica se o LOB já foi aberto.
ISTEMPORARY( <i>lob</i> )	Verifica se o LOB é temporário.
LOADFROMFILE( <i>lob_dest</i> , <i>bfile_ori</i> , <i>quantidade</i> , <i>desl_dest</i> , <i>desl_ori</i> )	Carrega os dados recuperados por meio de <i>bfile_ori</i> em <i>lob_dest</i> , começando nos deslocamentos, para um quantidade total de caracteres ou bytes; <i>bfile_ori</i> é um BFILE que aponta para um arquivo externo. LOADFROMFILE() é antigo e você deve usar os métodos de alto desempenho LOADBLOBFROMFILE() ou LOADCLOBFROMFILE().
LOADBLOBFROMFILE( <i>blob_dest</i> , <i>bfile_ori</i> , <i>quantidade</i> , <i>desl_dest</i> , <i>desl_ori</i> )	Carrega os dados recuperados por meio de <i>bfile_ori</i> em <i>blob_dest</i> , começando nos deslocamentos, para uma quantidade total de bytes; <i>bfile_ori</i> é um BFILE que aponta para um arquivo externo. LOADBLOBFROMFILE() oferece desempenho melhor do que LOADFROMFILE() ao se usar um BLOB.
LOADCLOBFROMFILE( <i>clob_dest</i> , <i>bfile_ori</i> , <i>quantidade</i> , <i>desl_dest</i> , <i>desl_ori</i> , <i>csid_ori</i> , <i>con-texto_ling</i> , <i>aviso</i> )	Carrega os dados recuperados por meio de <i>bfile_ori</i> em <i>clob_dest</i> , começando nos deslocamentos, para uma quantidade total de caracteres; <i>bfile_ori</i> é um BFILE que aponta para um arquivo externo. LOADCLOBFROMFILE() oferece desempenho melhor do que LOADFROMFILE() ao se usar um CLOB/NCLOB.
LOBMAXSIZE	Retorna o tamanho máximo de um LOB em bytes (atualmente 2 <sup>64</sup> ).
OPEN( <i>lob</i> , <i>modo_abertura</i> )	Abre o LOB no modo indicado, o qual pode ser configurado como: <ul style="list-style-type: none"> <li>■ DBMS_LOB.FILE_READONLY, que indica que o LOB só pode ser lido</li> <li>■ DBMS_LOB.FILE_READWRITE, que indica que o LOB pode ser lido e gravado</li> </ul>
READ( <i>lob</i> , <i>quantidade</i> , <i>deslocamento</i> , <i>buffer</i> )	Lê os dados do LOB e os armazena na variável <i>buffer</i> , começando no deslocamento no LOB, para uma quantidade total de caracteres ou bytes.
SUBSTR( <i>lob</i> , <i>quantidade</i> , <i>deslocamento</i> )	Retorna parte dos dados do LOB, começando no deslocamento no LOB, para uma quantidade total de caracteres ou bytes.
TRIM( <i>lob</i> , <i>novocomp</i> )	Corta os dados do LOB no comprimento mais curto especificado.
WRITE( <i>lob</i> , <i>quantidade</i> , <i>deslocamento</i> , <i>buffer</i> )	Grava os dados da variável <i>buffer</i> no LOB, começando no deslocamento no LOB, para uma quantidade total de caracteres ou bytes.
WRITEAPPEND( <i>lob</i> , <i>quantidade</i> , <i>buffer</i> )	Grava os dados da variável <i>buffer</i> no final do LOB, começando no deslocamento no LOB, para uma quantidade total de caracteres ou bytes.

## APPEND()

APPEND() adiciona os dados de um LOB de origem no final de um LOB de destino. Existem duas versões de APPEND():

```
DBMS_LOB.APPEND(
    lob_dest IN OUT NOCOPY BLOB,
    lob_ori IN                BLOB
);

DBMS_LOB.APPEND(
    lob_dest IN OUT NOCOPY CLOB/NCLOB CHARACTER SET ANY_CS,
    lob_ori IN                CLOB/NCLOB CHARACTER SET lob_dest%CHARSET
);
```

onde

- *lob\_dest* é o LOB de destino no qual os dados são anexados.
- *lob\_ori* é o LOB de origem a partir do qual os dados são lidos.
- CHARACTER SET ANY\_CS significa que os dados em *lob\_dest* podem ser qualquer conjunto de caracteres.
- CHARACTER SET *lob\_dest*%CHARSET é o conjunto de caracteres de *lob\_dest*.

A tabela a seguir mostra a exceção lançada por APPEND().

Exceção	Lançada quando
VALUE_ERROR	Ou <i>lob_dest</i> ou <i>lob_ori</i> é nulo.

## CLOSE()

CLOSE() fecha um LOB aberto anteriormente. Existem três versões de CLOSE():

```
DBMS_LOB.CLOSE(
    lob IN OUT NOCOPY BLOB
);

DBMS_LOB.CLOSE(
    lob IN OUT NOCOPY CLOB/NCLOB CHARACTER SET ANY_CS
);

DBMS_LOB.CLOSE(
    lob IN OUT NOCOPY BFILE
);
```

onde *lob* é o LOB a ser fechado.

## COMPARE()

COMPARE() compara os dados armazenados em dois LOBs, começando nos deslocamentos, por uma quantidade total de caracteres ou bytes. Existem três versões de COMPARE():

```
DBMS_LOB.COMPARE (
    lob1          IN BLOB,
    lob2          IN BLOB,
    quantidade    IN INTEGER:= 4294967295,
    deslocamento1 IN INTEGER:= 1,
    deslocamento2 IN INTEGER:= 1
) RETURN INTEGER;

DBMS_LOB.COMPARE (
    lob1          IN CLOB/NCLOB CHARACTER SET ANY_CS,
    lob2          IN CLOB/NCLOB CHARACTER SET lob_1%CHARSET,
    quantidade    IN INTEGER:= 4294967295,
    deslocamento1 IN INTEGER:= 1,
    deslocamento2 IN INTEGER:= 1
) RETURN INTEGER;

DBMS_LOB.COMPARE (
    lob1          IN BFILE,
    lob2          IN BFILE,
    quantidade    IN INTEGER,
    deslocamento1 IN INTEGER:= 1,
    deslocamento2 IN INTEGER:= 1
) RETURN INTEGER;
```

onde

- *lob1* e *lob2* são os LOBs a serem comparados
- *quantidade* é o número máximo de caracteres a ler de um CLOB/NCLOB ou o número máximo de bytes a ler de um BLOB/BFILE
- *deslocamento1* e *deslocamento2* são os deslocamentos em caracteres ou bytes em *lob1* e *lob2* para iniciar a comparação (os deslocamentos começam em 1)

COMPARE() retorna

- 0 se os LOBs são idênticos
- 1 se os LOBs não são idênticos
- Nulo se
  - *quantidade* < 1
  - *quantidade* > LOBMAXSIZE (Nota: LOBMAXSIZE é o tamanho máximo do LOB)
  - *deslocamento1* ou *deslocamento2* < 1
  - *deslocamento1* ou *deslocamento2* > LOBMAXSIZE

A tabela a seguir mostra as exceções lançadas por `COMPARE()`.

Exceção	Lançada quando
<code>UNOPENED_FILE</code>	O arquivo não foi aberto ainda
<code>NOEXIST_DIRECTORY</code>	O diretório não existe
<code>NOPRIV_DIRECTORY</code>	Você não tem privilégios para acessar o diretório
<code>INVALID_DIRECTORY</code>	O diretório é inválido
<code>INVALID_OPERATION</code>	O arquivo existe, mas você não tem privilégios para acessá-lo

## COPY()

`COPY()` copia dados de um LOB de origem em um LOB de destino, começando nos deslocamentos, para uma quantidade total de caracteres ou bytes. Existem duas versões de `COPY()`:

```
DBMS_LOB.COPY(
    lob_dest    IN OUT NOCOPY BLOB,
    lob_ori     IN          BLOB,
    quantidade  IN          INTEGER,
    desl_dest   IN          INTEGER:= 1,
    desl_ori    IN          INTEGER:= 1
);

DBMS_LOB.COPY(
    lob_dest    IN OUT NOCOPY CLOB/NCLOB CHARACTER SET ANY_CS,
    lob_ori     IN          CLOB/NCLOB CHARACTER SET lob_dest%CHARSET,
    quantidade  IN          INTEGER,
    desl_dest   IN          INTEGER:= 1,
    desl_ori    IN          INTEGER:= 1
);
```

onde

- *lob\_dest* e *lob\_ori* são os LOBs a gravar e ler, respectivamente.
- *quantidade* é o número máximo de caracteres a serem lidos de um CLOB/NCLOB ou o número máximo de bytes a serem lidos de um BLOB/BFILE.
- *desl\_dest* e *desl\_ori* são os deslocamentos em caracteres ou bytes em *lob\_dest* e *lob\_ori* para iniciar a cópia (os deslocamentos começam em 1).

As exceções lançadas por `COPY()` são mostradas a seguir.

Exceção	Lançada quando
<code>VALUE_ERROR</code>	Algum dos parâmetros é nulo.
<code>INVALID_ARGVAL</code>	Qualquer um de: <ul style="list-style-type: none"> <li>■ <i>desl_ori</i> &lt; 1</li> <li>■ <i>desl_dest</i> &lt; 1</li> <li>■ <i>desl_ori</i> &gt; LOBMAXSIZE</li> <li>■ <i>desl_dest</i> &gt; LOBMAXSIZE</li> <li>■ <i>quantidade</i> &lt; 1</li> <li>■ <i>quantidade</i> &gt; LOBMAXSIZE</li> </ul>



## CREATETEMPORARY()

CREATETEMPORARY () cria um LOB temporário no tablespace temporário padrão do usuário. Existem duas versões de CREATETEMPORARY ():

```
DBMS_LOB.CREATETEMPORARY (
    lob          IN OUT NOCOPY BLOB,
    cache        IN          BOOLEAN,
    duração      IN          PLS_INTEGER:= 10
);

DBMS_LOB.CREATETEMPORARY (
    lob          IN OUT NOCOPY CLOB/NCLOB CHARACTER SET ANY_CS,
    cache        IN          BOOLEAN,
    duração      IN          PLS_INTEGER:= 10
);
```

onde

- *lob* é o LOB temporário a ser criado.
- *cache* indica se o LOB deve ser lido no cache de buffer (true para sim, false para não).
- *duração* é uma dica (pode ser configurada como SESSION, TRANSACTION ou CALL) a respeito de se o LOB temporário é removido ao final da sessão, transação ou chamada (o padrão é SESSION).

A exceção lançada por CREATETEMPORARY () é mostrada a seguir.

Exceção	Lançada quando
VALUE_ERROR	O parâmetro <i>lob</i> é nulo

## ERASE()

ERASE () remove dados de um LOB, começando no deslocamento, para uma quantidade total de caracteres ou bytes. Existem duas versões de ERASE ():

```
DBMS_LOB.ERASE (
    lob          IN OUT NOCOPY BLOB,
    quantidade   IN OUT NOCOPY INTEGER,
    deslocamento IN          INTEGER:= 1
);

DBMS_LOB.ERASE (
    lob          IN OUT NOCOPY CLOB/NCLOB CHARACTER SET ANY_CS,
    quantidade   IN OUT NOCOPY INTEGER,
    deslocamento IN          INTEGER:= 1
);
```

onde

- *lob* é o LOB a ser apagado
- *quantidade* é o número máximo de caracteres a serem lidos de um CLOB/NCLOB ou o número de bytes a serem lidos de um BLOB

- *deslocamento* é o deslocamento em caracteres ou bytes em *lob* para iniciar a deleção (o deslocamento começa em 1)

Estas são as exceções lançadas por `ERASE()`:

#### Exceção

`VALUE_ERROR`  
`INVALID_ARGVAL`

#### Lançada quando

Algum dos parâmetros é nulo.

Qualquer um de:

- *quantidade* < 1
- *quantidade* > `LOBMAXSIZE`
- *deslocamento* < 1
- *deslocamento* > `LOBMAXSIZE`

## FILECLOSE()

`FILECLOSE()` fecha um `BFILE`. Você deve usar a procedure `CLOSE()`, mais recente, pois a Oracle Corporation não pretende estender a procedure mais antiga `FILECLOSE()`. A abordagem de `FILECLOSE()` é apresentada apenas para que você possa entender programas mais antigos.

```
DBMS_LOB.FILECLOSE (
    bfile IN OUT NOCOPY BFILE
);
```

onde *bfile* é o `BFILE` a ser fechado.

A tabela a seguir mostra as exceções lançadas por `FILECLOSE()`.

#### Exceção

`VALUE_ERROR`  
`UNOPENED_FILE`  
`NOEXIST_DIRECTORY`  
`NOPRIV_DIRECTORY`  
`INVALID_DIRECTORY`  
`INVALID_OPERATION`

#### Lançada quando

O parâmetro *bfile* é nulo  
O arquivo não foi aberto ainda  
O diretório não existe  
Você não tem privilégios para acessar o diretório  
O diretório é inválido  
O arquivo existe, mas você não tem privilégios para acessá-lo

## FILECLOSEALL()

`FILECLOSEALL()` fecha todos os objetos `BFILE`.

```
DBMS_LOB.FILECLOSEALL;
```

Esta é a exceção lançada por `FILECLOSEALL()`:

#### Exceção

`UNOPENED_FILE`

#### Lançada quando

Nenhum arquivo foi aberto na sessão

## FILEEXISTS()

`FILEEXISTS()` verifica se um arquivo existe.

```
DBMS_LOB.FILEEXISTS (
    bfile IN BFILE
) RETURN INTEGER;
```

onde *bfile* é um BFILE que aponta para um arquivo externo.

FILEEXISTS() retorna

- 0 se o arquivo não existe
- 1 se o arquivo existe

As exceções lançadas por FILEEXISTS() são mostradas a seguir.

#### Exceção

VALUE\_ERROR

NOEXIST\_DIRECTORY

NOPRIV\_DIRECTORY

INVALID\_DIRECTORY

#### Lançada quando

O parâmetro *bfile* é nulo

O diretório não existe

Você não tem privilégios para acessar o diretório

O diretório é inválido

## FILEGETNAME()

FILEGETNAME() retorna o diretório e o nome de arquivo de um BFILE.

```
DBMS_LOB.FILEGETNAME (
    bfile          IN BFILE,
    diretório      OUT VARCHAR2,
    nome_arquivo   OUT VARCHAR2
);
```

onde

- *bfile* é o ponteiro para o arquivo
- *diretório* é o diretório onde o arquivo está armazenado
- *nome\_arquivo* é o nome do arquivo

Estas são as exceções lançadas por FILEGETNAME():

#### Exceção

VALUE\_ERROR

INVALID\_ARGVAL

#### Lançada quando

Algum dos parâmetros de entrada é nulo ou inválido

Os parâmetros *diretório* ou *nome\_arquivo* são nulos

## FILEISOPEN()

FILEISOPEN() verifica se um arquivo está aberto. Você deve usar a procedure ISOPEN(), mas recente, para verificar se um arquivo está aberto em seus próprios programas, pois a Oracle Corporation não pretende estender o método FILEISOPEN(), mais antigo. A abordagem de FILEISOPEN() é apresentada apenas para que você entenda programas mais antigos.

```
DBMS_LOB.FILEISOPEN(
    bfile IN BFILE
) RETURN INTEGER;
```

onde *bfile* é o ponteiro para o arquivo.

FILEISOPEN() retorna

- 0 se o arquivo não está aberto
- 1 se o arquivo está aberto

As exceções lançadas por FILEISOPEN() são mostradas a seguir.

#### Exceção

NOEXIST\_DIRECTORY  
NOPRIV\_DIRECTORY  
INVALID\_DIRECTORY  
INVALID\_OPERATION

#### Lançada quando

O diretório não existe  
Você não tem privilégios para acessar o diretório  
O diretório é inválido  
O arquivo não existe ou você não tem privilégios para acessá-lo

## FILEOPEN()

FILEOPEN() abre um arquivo. Você deve usar a procedure OPEN(), mais recente, para abrir um arquivo em seus próprios programas, pois a Oracle Corporation não pretende estender a procedure FILEOPEN(), mais antiga. A abordagem de FILEOPEN() é apresentada apenas para que você entenda os programas mais antigos.

```
DBMS_LOB.FILEOPEN(
    bfile          IN OUT NOCOPY BFILE,
    modo_abertura IN          BINARY_INTEGER:= DBMS_LOB.FILE_READONLY
);
```

onde

- *bfile* é o ponteiro para o arquivo
- *modo\_abertura* indica o modo de abertura; o único modo de abertura é DBMS\_LOB.FILE\_READONLY, que indica que o arquivo pode ser lido

Estas são as exceções lançadas por FILEOPEN():

#### Exceção

VALUE\_ERROR  
INVALID\_ARGVAL  
OPEN\_TOOMANY  
  
NOEXIST\_DIRECTORY  
INVALID\_DIRECTORY  
INVALID\_OPERATION

#### Lançada quando

Alguns dos parâmetros de entrada é nulo ou inválido  
O *modo\_abertura* não está configurado como DBMS\_LOB.FILE\_READONLY  
Foi feita uma tentativa de abrir um número de arquivos maior que o especificado em SESSION\_MAX\_OPEN\_FILES, onde SESSION\_MAX\_OPEN\_FILES é um parâmetro de inicialização do banco de dados configurado pelo DBA  
  
O diretório não existe  
O diretório é inválido  
O arquivo existe, mas você não tem privilégios para acessá-lo

## FREETEMPORARY()

`FREETEMPORARY()` libera um LOB temporário do tablespace temporário padrão do usuário. Existem duas versões de `FREETEMPORARY()`:

```
DBMS_LOB.FREETEMPORARY (
    lob IN OUT NOCOPY BLOB
);

DBMS_LOB.FREETEMPORARY (
    lob IN OUT NOCOPY CLOB/NCLOB CHARACTER SET ANY_CS
);
```

onde `lob` é o lob a ser liberado.

Esta é a exceção lançada por `FREETEMPORARY()`:

### Exceção

`VALUE_ERROR`

### Lançada quando

O parâmetro de entrada é nulo ou inválido.

## GETCHUNKSIZE()

`GETCHUNKSIZE()` retorna o tamanho do trecho ao ler e gravar dados do LOB (um trecho é uma unidade de dados). Existem duas versões de `GETCHUNKSIZE()`:

```
DBMS_LOB.GETCHUNKSIZE (
    lob IN BLOB
) RETURN INTEGER;

DBMS_LOB.GETCHUNKSIZE (
    lob IN CLOB/NCLOB CHARACTER SET ANY_CS
) RETURN INTEGER;
```

onde `lob` é o LOB do qual se vai obter o tamanho do trecho.

`GETCHUNKSIZE()` retorna

- O tamanho do trecho, em bytes, de um BLOB
- O tamanho do trecho, em caracteres, de um CLOB/NCLOB

Esta é a exceção lançada por `GETCHUNKSIZE()`:

### Exceção

`VALUE_ERROR`

### Lançada quando

O parâmetro `lob` é nulo

## GET\_STORAGE\_LIMIT()

`GET_STORAGE_LIMIT()` retorna o tamanho máximo permitido para um LOB.

```
DBMS_LOB.GET_STORAGE_LIMIT ()
RETURN INTEGER;
```

## GETLENGTH()

GETLENGTH() retorna o comprimento dos dados do LOB. Existem três versões de GETLENGTH():

```
DBMS_LOB.GETLENGTH (
    lob IN BLOB
) RETURN INTEGER;

DBMS_LOB.GETLENGTH (
    lob IN CLOB/NCLOB CHARACTER SET ANY_CS
) RETURN INTEGER;

DBMS_LOB.GETLENGTH (
    bfile IN BFILE
) RETURN INTEGER;
```

onde

- *lob* são os dados do BLOB, CLOB ou NCLOB dos quais se vai obter o comprimento
- *bfile* são os dados BFILE dos quais se vai obter o comprimento

GETLENGTH() retorna

- O comprimento, em bytes, de um BLOB ou BFILE
- O comprimento, em caracteres, de um CLOB ou NCLOB

Esta é a exceção lançada por GETLENGTH():

### Exceção

VALUE\_ERROR

### Lançada quando

O parâmetro *lob* ou *bfile* é nulo

## INSTR()

INSTR() retorna a posição inicial dos caracteres que correspondem à *n-ésima* ocorrência de um padrão nos dados do LOB, começando em um deslocamento. Existem três versões de INSTR():

```
DBMS_LOB.INSTR (
    lob          IN BLOB,
    padrão       IN RAW,
    deslocamento IN INTEGER:= 1,
    n            IN INTEGER:= 1
) RETURN INTEGER;

DBMS_LOB.INSTR (
    lob          IN CLOB/NCLOB CHARACTER SET ANY_CS,
    padrão       IN VARCHAR2 CHARACTER SET lob%CHARSET,
    deslocamento IN INTEGER := 1,
    n            IN INTEGER := 1
) RETURN INTEGER;
```

```
DBMS_LOB.INSTR(  
    bfile          IN BFILE,  
    padrão        IN RAW,  
    deslocamento IN INTEGER := 1,  
    n              IN INTEGER := 1  
) RETURN INTEGER;
```

onde

- *lob* é o BLOB, CLOB ou NCLOB a ser lido.
- *bfile* é o BFILE a ser lido.
- *padrão* é o padrão a ser pesquisado nos dados do LOB. O padrão é um grupo de bytes RAW de um BLOB ou BFILE e uma string de caracteres VARCHAR2 de um CLOB; o tamanho máximo do padrão é de 16.383 bytes.
- *deslocamento* é o deslocamento para iniciar a leitura dos dados do LOB (o deslocamento começa em 1).
- *n* é a ocorrência do padrão para pesquisar os dados.

INSTR() retorna

- O deslocamento do início do padrão (se for encontrado)
- Zero se o padrão não for encontrado
- Nulo se
  - Algum dos parâmetros de IN é nulo ou inválido
  - *deslocamento* < 1 ou *deslocamento* > LOBMAXSIZE
  - *n* < 1 ou *n* > LOBMAXSIZE

Estas são as exceções lançadas por INSTR():

#### Exceção

VALUE\_ERROR  
UNOPENED\_FILE  
NOEXIST\_DIRECTORY  
NOPRIV\_DIRECTORY  
INVALID\_DIRECTORY  
INVALID\_OPERATION

#### Lançada quando

Algum dos parâmetros de entrada é nulo ou inválido  
O BFILE não está aberto  
O diretório não existe  
O diretório existe, mas você não tem privilégios para acessá-lo  
O diretório é inválido  
O arquivo existe, mas você não tem privilégios para acessá-lo

## ISOPEN()

ISOPEN() verifica se o LOB já foi aberto. Existem três versões de ISOPEN():

```
DBMS_LOB.ISOPEN(  
    lob IN BLOB  
) RETURN INTEGER;
```

```
DBMS_LOB.ISOPEN (
  lob IN CLOB/NCLOB CHARACTER SET ANY_CS
) RETURN INTEGER;
```

```
DBMS_LOB.ISOPEN (
  bfile IN BFILE
) RETURN INTEGER;
```

onde

- *lob* é o BLOB, CLOB ou NCLOB a ser verificado
- *bfile* é o *BFILE* a ser verificado

ISOPEN() retorna

- 0 se o LOB não está aberto
- 1 se o LOB está aberto

Esta é a exceção lançada por ISOPEN():

#### Exceção

VALUE\_ERROR

#### Lançada quando

O parâmetro *lob* ou *bfile* é nulo ou inválido.

## ISTEMPORARY()

ISTEMPORARY() verifica se o LOB é temporário. Existem duas versões de ITEMPORARY():

```
DBMS_LOB.ISTEMPORARY (
  lob IN BLOB
) RETURN INTEGER;
```

```
DBMS_LOB.ISTEMPORARY (
  lob IN CLOB/NCLOB CHARACTER SET ANY_CS
) RETURN INTEGER;
```

onde

- *lob* é o LOB a ser verificado

ISTEMPORARY() retorna

- 0 se o LOB não é temporário
- 1 se o LOB é temporário

Esta é a exceção lançada por ITEMPORARY():

#### Exceção

VALUE\_ERROR

#### Lançada quando

O parâmetro *lob* é nulo ou inválido.



## LOADFROMFILE()

LOADFROMFILE() carrega os dados recuperados por meio de um BFILE em um CLOB, NCLOB ou BLOB, começando nos deslocamentos, para uma quantidade total de caracteres ou bytes. Você deve usar as procedures LOADCLOBFROMFILE() ou LOADBLOBFROMFILE(), de desempenho mais alto, em seus próprios programas. Estamos incluindo a abordagem de LOADFROMFILE() aqui somente para que você possa entender programas mais antigos.

Existem duas versões de LOADFROMFILE():

```
DBMS_LOB.LOADFROMFILE (
    lob_dest      IN OUT NOCOPY BLOB,
    bfile_ori     IN             BFILE,
    quantidade    IN             INTEGER,
    desl_dest     IN             INTEGER := 1,
    desl_ori      IN             INTEGER := 1
);

DBMS_LOB.LOADFROMFILE (
    lob_dest      IN OUT NOCOPY CLOB/NCLOB CHARACTER SET ANY_CS,
    bfile_ori     IN             BFILE,
    quantidade    IN             INTEGER,
    desl_dest     IN             INTEGER := 1,
    desl_ori      IN             INTEGER := 1
);
```

onde

- *lob\_dest* é o LOB no qual os dados devem ser gravados
- *bfile\_ori* é o ponteiro para o arquivo do qual os dados devem ser lidos
- *quantidade* é o número máximo de bytes ou caracteres a serem lidos de *bfile\_ori*
- *desl\_dest* é o deslocamento em bytes ou caracteres em *lob\_dest* para começar a gravar os dados (o deslocamento começa em 1)
- *desl\_ori* é o deslocamento em bytes em *bfile\_ori* para começar a ler os dados (o deslocamento começa em 1)

Estas são as exceções lançadas por LOADFROMFILE():

### Exceção

VALUE\_ERROR

INVALID\_ARGVAL

### Lançada quando

Algum dos parâmetros de entrada é nulo ou inválido

Qualquer um de:

- *desl\_ori* < 1
- *desl\_dest* < 1
- *desl\_ori* > LOBMAXSIZE
- *desl\_dest* > LOBMAXSIZE
- *quantidade* < 1
- *quantidade* > LOBMAXSIZE

## LOADBLOBFROMFILE()

LOADBLOBFROMFILE() carrega os dados recuperados por meio de um BFILE em um BLOB. LOADBLOBFROMFILE() oferece desempenho melhor do que o método LOADFROMFILE() ao se usar um BLOB.

```
DBMS_LOB.LOADBLOBFROMFILE (
    blob_dest    IN OUT NOCOPY BLOB,
    bfile_ori    IN             BFILE,
    quantidade   IN             INTEGER,
    desl_dest    IN OUT        INTEGER := 1,
    desl_ori     IN OUT        INTEGER := 1
);
```

onde

- *blob\_dest* é o BLOB no qual os dados devem ser gravados
- *bfile\_ori* é o ponteiro para o arquivo do qual os dados devem ser lidos
- *quantidade* é o número máximo de bytes a serem lidos de *bfile\_ori*
- *desl\_dest* é o deslocamento em bytes em *lob\_dest* para começar a gravar os dados (o deslocamento começa em 1)
- *desl\_ori* é o deslocamento em bytes em *bfile\_ori* para começar a ler os dados (o deslocamento começa em 1)

Estas são as exceções lançadas por LOADBLOBFROMFILE():

### Exceção

VALUE\_ERROR

INVALID\_ARGVAL

### Lançada quando

Algum dos parâmetros de entrada é nulo ou inválido

Qualquer um de:

- *desl\_ori* < 1
- *desl\_dest* < 1
- *desl\_ori* > LOBMAXSIZE
- *desl\_dest* > LOBMAXSIZE
- *quantidade* < 1
- *quantidade* > LOBMAXSIZE

## LOADCLOBFROMFILE()

LOADCLOBFROMFILE() carrega os dados recuperados por meio de um BFILE em um CLOB/NCLOB. LOADCLOBFROMFILE() oferece desempenho melhor do que o método LOADFROMFILE() ao se usar um CLOB/NCLOB. LOADCLOBFROMFILE() também converte dados binários em dados de caractere, automaticamente.

```
DBMS_LOB.LOADCLOBFROMFILE (
    clob_dest    IN OUT NOCOPY CLOB/NCLOB,
    bfile_ori    IN             BFILE,
    quantidade   IN             INTEGER,
    desl_dest    IN OUT        INTEGER,
    desl_ori     IN OUT        INTEGER,
```

```

        csid_ori          IN          NUMBER,
        contexto_ling     IN OUT     INTEGER,
        aviso             OUT        INTEGER
    );

```

onde

- *blob\_dest* é o CLOB/NCLOB no qual os dados devem ser gravados.
- *bfile\_ori* é o ponteiro para o arquivo do qual os dados devem ser lidos.
- *quantidade* é o número máximo de caracteres a serem lidos de *bfile\_ori*.
- *desl\_dest* é o deslocamento em caracteres em *lob\_dest* para começar a gravar os dados (o deslocamento começa em 1).
- *desl\_ori* é o deslocamento em caracteres em *bfile\_ori* para começar a ler os dados (o deslocamento começa em 1).
- *csid\_ori* é o conjunto de caracteres de *bfile\_ori* (normalmente, você deve usar DBMS\_LOB.DEFAULT\_CSID, que é o conjunto de caracteres padrão do banco de dados).
- *contexto\_ling* é o contexto da linguagem a ser usado para o carregamento (normalmente, você deve usar DBMS\_LOB.DEFAULT\_LANG\_CTX, que é o contexto de linguagem padrão do banco de dados).
- *aviso* é uma mensagem de aviso contendo informações se houve um problema no carregamento; um problema comum é um caractere em *bfile\_ori* não ser convertido em um caractere em *lob\_dest* (caso em que *aviso* é configurado como DBMS\_LOB.WARN\_INCONVERTIBLE\_CHAR).



#### NOTA

Você pode aprender tudo sobre conjuntos de caracteres, contextos e como converter caracteres de uma linguagem para outra no Oracle Database Globalization Support Guide, publicado pela Oracle Corporation.

Estas são as exceções lançadas por `LOADCLOBFROMFILE()`:

#### Exceção

VALUE\_ERROR

INVALID\_ARGVAL

#### Lançada quando

Algum dos parâmetros de entrada é nulo ou inválido

Qualquer um de:

- *desl\_ori* < 1
- *desl\_dest* < 1
- *desl\_ori* > LOBMAXSIZE
- *desl\_dest* > LOBMAXSIZE
- *quantidade* < 1
- *quantidade* > LOBMAXSIZE

## OPEN()

`OPEN()` abre um LOB. Existem três versões de `OPEN()`:

```

DBMS_LOB.OPEN(
    lob          IN OUT NOCOPY BLOB,
    modo_abertura IN          BINARY_INTEGER
);

DBMS_LOB.OPEN(
    lob          IN OUT NOCOPY CLOB/NCLOB CHARACTER SET ANY_CS,
    modo_abertura IN          BINARY_INTEGER
);

DBMS_LOB.OPEN(
    bfile        IN OUT NOCOPY BFILE,
    modo_abertura IN          BINARY_INTEGER:= DBMS_LOB.FILE_READONLY
);

```

onde

- *lob* é o LOB a ser aberto.
- *bfile* é o ponteiro para o arquivo a ser aberto.
- *modo\_abertura* indica o modo de abertura; o padrão é `DBMS_LOB.FILE_READONLY`, que indica que o LOB só pode ser lido. `DBMS_LOB.FILE_READWRITE` indica que o LOB pode ser lido e gravado.

Esta é a exceção lançada por `OPEN()`:

#### Exceção

`VALUE_ERROR`

#### Lançada quando

Algum dos parâmetros de entrada é nulo ou inválido

## READ()

`READ()` lê os dados de um LOB em um buffer. Existem três versões de `READ()`:

```

DBMS_LOB.READ(
    lob          IN          BLOB,
    quantidade   IN OUT NOCOPY BINARY_INTEGER,
    deslocamento IN          INTEGER,
    buffer       OUT          RAW
);

DBMS_LOB.READ(
    lob          IN          CLOB/NCLOB CHARACTER SET ANY_CS,
    quantidade   IN OUT NOCOPY BINARY_INTEGER,
    deslocamento IN          INTEGER,
    buffer       OUT          VARCHAR2 CHARACTER SET lob%CHARSET
);

DBMS_LOB.READ(
    bfile        IN          BFILE,
    quantidade   IN OUT NOCOPY BINARY_INTEGER,

```

```

        deslocamento IN          INTEGER,
        buffer        OUT         RAW
    );

```

onde

- *lob* é o CLOB, NCLOB ou BLOB a ser lido
- *bfile* é o BFILE a ser lido
- *quantidade* é o número máximo de caracteres a serem lidos de um CLOB/NCLOB ou o número máximo de bytes a serem lidos de um BLOB/BFILE
- *deslocamento* é o deslocamento para iniciar a leitura (o deslocamento começa em 1)
- *buffer* é a variável onde os dados lidos do LOB devem ser armazenados

Estas são as exceções lançadas por `READ()`:

#### Exceção

VALUE\_ERROR

INVALID\_ARGVAL

NO\_DATA\_FOUND

#### Lançada quando

Algum dos parâmetros de entrada é nulo

Qualquer um de:

- *quantidade* < 1
- *quantidade* > MAXBUFSIZE
- *quantidade* > capacidade do buffer em bytes ou caracteres
- *deslocamento* < 1
- *deslocamento* > LOBMAXSIZE

O final do LOB foi atingido e não existem mais bytes ou caracteres a ler do LOB

## SUBSTR()

`SUBSTR()` retorna parte dos dados do LOB, começando no deslocamento, para uma quantidade total de caracteres ou bytes. Existem três versões de `SUBSTR()`:

```

DBMS_LOB.SUBSTR (
    lob          IN BLOB,
    quantidade   IN INTEGER := 32767,
    deslocamento IN INTEGER := 1
) RETURN RAW;

DBMS_LOB.SUBSTR (
    lob          IN CLOB/NCLOB CHARACTER SET ANY_CS,
    quantidade   IN INTEGER := 32767,
    deslocamento IN INTEGER := 1
) RETURN      VARCHAR2 CHARACTER SET lob%CHARSET;

DBMS_LOB.SUBSTR (
    bfile        IN BFILE,
    quantidade   IN INTEGER := 32767,
    deslocamento IN INTEGER := 1
) RETURN RAW;

```

onde

- *lob* é o BLOB, CLOB ou NCLOB a ser lido
- *bfile* é o ponteiro para o arquivo a ser lido
- *quantidade* é o número máximo de caracteres a serem lidos de um CLOB/NCLOB ou o número máximo de bytes a serem lidos de um BLOB/BFILE
- *deslocamento* é o deslocamento para iniciar a leitura dos dados do LOB (o deslocamento começa em 1)

SUBSTR() retorna

- Dados RAW ao se ler de um BLOB/BFILE
- Dados VARCHAR2 a se ler de um CLOB/NCLOB
- Nulo se
  - *quantidade* < 1
  - *quantidade* > 32767
  - *deslocamento* < 1
  - *deslocamento* > LOBMAXSIZE

Estas são as exceções lançadas por SUBSTR() :

#### Exceção

VALUE\_ERROR  
UNOPENED\_FILE  
NOEXIST\_DIRECTORY  
NOPRIV\_DIRECTORY  
INVALID\_DIRECTORY  
INVALID\_OPERATION

#### Lançada quando

Algum dos parâmetros de entrada é nulo ou inválido.  
O BFILE não está aberto.  
O diretório não existe.  
Você não tem privilégios no diretório.  
O diretório é inválido.  
O arquivo existe, mas você não tem privilégios para acessá-lo.

## TRIM()

TRIM() corta os dados do LOB para o comprimento mais curto especificado. Existem duas versões de TRIM() :

```
DBMS_LOB.TRIM(
  lob          IN OUT NOCOPY BLOB,
  novocomp    IN          INTEGER
);

DBMS_LOB.TRIM(
  lob          IN OUT NOCOPY CLOB/NCLOB CHARACTER SET ANY_CS,
  novocomp    IN          INTEGER
);
```

onde

- *lob* é o BLOB, CLOB ou NCLOB a ser cortado
- *novocomp* é o novo comprimento (em bytes para um BLOB ou caracteres para um CLOB/NCLOB)

Estas são as exceções lançadas por `TRIM()`:

**Exceção**

VALUE\_ERROR

INVALID\_ARGVAL

**Lançada quando**

O parâmetro *lob* é nulo

Qualquer um de:

- *novocomp* < 0
- *novocomp* > LOBMAXSIZE

## WRITE()

`WRITE()` grava dados de um buffer em um LOB. Existem duas versões de `WRITE()`:

```
DBMS_LOB.WRITE (
    lob           IN OUT NOCOPY BLOB,
    quantidade    IN             BINARY_INTEGER,
    deslocamento IN             INTEGER,
    buffer        IN             RAW
);

DBMS_LOB.WRITE (
    lob           IN OUT NOCOPY CLOB/NCLOB CHARACTER SET ANY_CS,
    quantidade    IN             BINARY_INTEGER,
    deslocamento IN             INTEGER,
    buffer        IN             VARCHAR2 CHARACTER SET lob%CHARSET
);
```

onde

- *lob* é o LOB a ser gravado
- *quantidade* é o número máximo de caracteres a serem gravados em um CLOB/NCLOB ou o número máximo de bytes a serem gravados em um BLOB
- *deslocamento* é o deslocamento para começar a gravação dos dados no LOB (o deslocamento começa em 1)
- *buffer* é a variável que contém os dados a serem gravados no LOB

As exceções lançadas por `WRITE()` são mostradas a seguir.

**Exceção**

VALUE\_ERROR

INVALID\_ARGVAL

**Lançada quando**

Algun dos parâmetros de entrada é nulo ou inválido

Qualquer um de:

- *quantidade* < 1
- *quantidade* > MAXBUFSIZE
- *deslocamento* < 1
- *deslocamento* > LOBMAXSIZE

## WRITEAPPEND()

WRITEAPPEND() grava dados do buffer no final de um LOB, começando no deslocamento, para uma quantidade total de caracteres ou bytes. Existem duas versões de WRITEAPPEND():

```
DBMS_LOB.WRITEAPPEND (
    lob          IN OUT NOCOPY BLOB,
    quantidade IN          BINARY_INTEGER,
    buffer       IN          RAW
);

DBMS_LOB.WRITEAPPEND (
    lob          IN OUT NOCOPY CLOB/NCLOB CHARACTER SET ANY_CS,
    quantidade IN          BINARY_INTEGER,
    buffer       IN          VARCHAR2 CHARACTER SET lob%CHARSET
);
```

onde

- *lob* é o BLOB, CLOB ou NCLOB a ser gravado
- *quantidade* é o número máximo de caracteres a serem gravados em um CLOB/NCLOB ou o número máximo de bytes a serem gravados em um BLOB
- *buffer* é a variável que contém os dados a serem gravados no LOB

Estas são as exceções lançadas por WRITEAPPEND():

### Exceção

VALUE\_ERROR

INVALID\_ARGVAL

### Lançada quando

Algum dos parâmetros de entrada é nulo ou inválido

Qualquer um de:

- *quantidade* < 1
- *quantidade* > MAXBUFSIZE

## Exemplos de procedures em PL/SQL

Nesta seção, você verá exemplos de procedures em PL/SQL que utilizam os vários métodos descritos nas seções anteriores. Os exemplos de procedures são criados quando você executa o script `lob_schema.sql`.

### Recuperando um localizador de LOB

A procedure `get_clob_locator()` a seguir obtém um localizador de LOB da tabela `clob_content`; `get_clob_locator()` executa as seguintes tarefas:

- Aceita um parâmetro IN OUT chamado *p\_clob* de tipo CLOB; *p\_clob* é configurado como um localizador de LOB dentro da procedure. Como *p\_clob* é IN OUT, o valor é passado da procedure.
- Aceita um parâmetro IN chamado *p\_id* de tipo INTEGER, o qual especifica o valor de id de uma linha a ser recuperada da tabela `clob_content`.
- Seleciona a `clob_column` da tabela `clob_content` no *p\_clob*; isso armazena o localizador de LOB de `clob_column` em *p\_clob*.



```

CREATE PROCEDURE get_clob_locator(
    p_clob IN OUT CLOB,
    p_id   IN INTEGER
) AS
BEGIN
    -- obtém o localizador de LOB e o armazena em p_clob
    SELECT clob_column
    INTO p_clob
    FROM clob_content
    WHERE id = p_id;
END get_clob_locator;
/

```

A procedure `get_blob_locator()` a seguir faz a mesma coisa que a procedure anterior, exceto que obtém o localizador de um BLOB da tabela `blob_content`:

```

CREATE PROCEDURE get_blob_locator(
    p_blob IN OUT BLOB,
    p_id   IN INTEGER
) AS
BEGIN
    -- obtém o localizador de LOB e o armazena em p_blob
    SELECT blob_column
    INTO p_blob
    FROM blob_content
    WHERE id = p_id;
END get_blob_locator;
/

```

Essas duas procedures são usadas no código mostrado nas seções a seguir.

### ***Lendo dados de CLOBs e BLOBs***

A procedure `read_clob_example()` a seguir lê os dados de um CLOB e os exibe na tela; `read_clob_example()` executa as seguintes tarefas:

- chama `get_clob_locator()` para obter um localizador e o armazena em `v_clob`
- usa `READ()` para ler o conteúdo de `v_clob` em uma variável `VARCHAR2` chamada `v_char_buffer`
- exibe o conteúdo de `v_char_buffer` na tela

```

CREATE PROCEDURE read_clob_example(
    p_id IN INTEGER
) AS
    v_clob CLOB;
    v_offset INTEGER:= 1;
    v_amount INTEGER:= 50;
    v_char_buffer VARCHAR2(50);
BEGIN
    -- obtém o localizador de LOB e o armazena em v_clob
    get_clob_locator(v_clob, p_id);

```

```

-- lê o conteúdo de v_clob em v_char_buffer, começando na
-- posição v_offset e lê um total de v_amount caracteres
DBMS_LOB.READ(v_clob, v_amount, v_offset, v_char_buffer);

-- exibe o conteúdo de v_char_buffer
DBMS_OUTPUT.PUT_LINE('v_char_buffer = ' || v_char_buffer);
DBMS_OUTPUT.PUT_LINE('v_amount = ' || v_amount);
END read_clob_example;
/

```

O exemplo a seguir ativa a saída do servidor e chama `read_clob_example()`:

```

SET SERVEROUTPUT ON
CALL read_clob_example(1);
v_char_buffer = Creeps in this petty pace
v_amount = 25

```

A procedure `read_blob_example()` a seguir lê os dados de um BLOB; `read_blob_example()` executa as seguintes tarefas:

- chama `get_blob_locator()` para obter o localizador e o armazena em `v_blob`
- chama `READ()` para ler o conteúdo de `v_blob` em uma variável RAW chamada `v_binary_buffer`
- exibe o conteúdo de `v_binary_buffer` na tela

```

CREATE PROCEDURE read_blob_example(
    p_id IN INTEGER
) AS
    v_blob BLOB;
    v_offset INTEGER:= 1;
    v_amount INTEGER:= 25;
    v_binary_buffer RAW(25);
BEGIN
    -- obtém o localizador de LOB e o armazena em v_blob
    get_blob_locator(v_blob, p_id);

    -- lê o conteúdo de v_blob em v_binary_buffer, começando na
    -- posição v_offset e lê um total de v_amount bytes
    DBMS_LOB.READ(v_blob, v_amount, v_offset, v_binary_buffer);

    -- exibe o conteúdo de v_binary_buffer
    DBMS_OUTPUT.PUT_LINE('v_binary_buffer = ' || v_binary_buffer);
    DBMS_OUTPUT.PUT_LINE('v_amount = ' || v_amount);
END read_blob_example;
/

```

O exemplo a seguir chama `read_blob_example()`:

```

CALL read_blob_example(1);
v_binary_buffer = 10011101010101111
v_amount = 9

```

### Gravando em um CLOB

A procedure `write_example()` a seguir grava uma string `v_char_buffer` em `v_clob` usando `WRITE()`; observe que a instrução `SELECT` na procedure usa a cláusula `FOR UPDATE`, utilizada porque o CLOB é gravado através de `WRITE()`:

```
CREATE PROCEDURE write_example(
    p_id IN INTEGER
) AS
    v_clob CLOB;
    v_offset INTEGER:= 7;
    v_amount INTEGER:= 6;
    v_char_buffer VARCHAR2(10) := 'pretty';
BEGIN
    -- obtém o localizador de LOB em v_clob com a cláusula FOR UPDATE
    -- (porque o LOB é gravado por WRITE() posteriormente)
    SELECT clob_column
    INTO v_clob
    FROM clob_content
    WHERE id = p_id
    FOR UPDATE;

    -- lê e exibe o conteúdo do CLOB
    read_clob_example(p_id);

    -- grava os caracteres de v_char_buffer em v_clob, começando
    -- na posição v_offset e grava um total de v_amount caracteres
    DBMS_LOB.WRITE(v_clob, v_amount, v_offset, v_char_buffer);

    -- lê e exibe o conteúdo do CLOB
    -- e depois reverte a gravação
    read_clob_example(p_id);
    ROLLBACK;
END write_example;
/
```

O exemplo a seguir chama `write_example()`:

```
CALL write_example(1);
v_char_buffer = Creeps in this petty pace
v_amount = 25
v_char_buffer = Creepsprettyis petty pace
v_amount = 25
```

### Anexando dados em um CLOB

A procedure `append_example()` a seguir usa `APPEND()` para copiar os dados de `v_src_clob` no final de `v_dest_clob`:

```
CREATE PROCEDURE append_example AS
    v_src_clob CLOB;
    v_dest_clob CLOB;
BEGIN
    -- obtém o localizador de LOB do CLOB na linha nº 2 da
```

```

-- tabela clob_content em v_src_clob
get_clob_locator(v_src_clob, 2);

-- obtém o localizador de LOB do CLOB na linha nº 1 da
-- tabela clob_content em v_dest_clob com a cláusula FOR UPDATE
-- (porque o CLOB será adicionado usando
-- APPEND() posteriormente)
SELECT clob_column
INTO v_dest_clob
FROM clob_content
WHERE id = 1
FOR UPDATE;

-- lê e exibe o conteúdo do CLOB nº 1
read_clob_example(1);

-- usa APPEND() para copiar o conteúdo de v_src_clob em v_dest_clob
DBMS_LOB.APPEND(v_dest_clob, v_src_clob);

-- lê e exibe o conteúdo do CLOB nº 1
-- e depois reverte a alteração
read_clob_example(1);
ROLLBACK;
END append_example;
/

```

O exemplo a seguir chama `append_example()`:

```

CALL append_example();
v_char_buffer = Creeps in this petty pace
v_amount = 25
v_char_buffer = Creeps in this petty pace from day to day
v_amount = 41

```

### Comparando os dados de dois CLOBs

A procedure `compare_example()` a seguir compara os dados de `v_clob1` e `v_clob2` usando `COMPARE()`:

```

CREATE PROCEDURE compare_example AS
  v_clob1 CLOB;
  v_clob2 CLOB;
  v_return INTEGER;
BEGIN
  -- obtém os localizadores de LOB
  get_clob_locator(v_clob1, 1);
  get_clob_locator(v_clob2, 2);

  -- compara v_clob1 com v_clob2 (COMPARE() retorna 1,
  -- porque o conteúdo de v_clob1 e de v_clob2 são diferentes)
  DBMS_OUTPUT.PUT_LINE('Comparing v_clob1 with v_clob2');
  v_return:= DBMS_LOB.COMPARE(v_clob1, v_clob2);
  DBMS_OUTPUT.PUT_LINE('v_return = ' || v_return);

```

```

-- compara v_clob1 com v_clob1 (COMPARE() retorna 0,
-- porque os conteúdos são iguais)
DBMS_OUTPUT.PUT_LINE('Comparing v_clob1 with v_clob1');
v_return:= DBMS_LOB.COMPARE(v_clob1, v_clob1);
DBMS_OUTPUT.PUT_LINE('v_return = ' || v_return);
END compare_example;
/

```

O exemplo a seguir chama `compare_example()`:

```

CALL compare_example();
Comparing v_clob1 with v_clob2
v_return = 1
Comparing v_clob1 with v_clob1
v_return = 0

```

Note que `v_return` é 1 ao comparar `v_clob1` com `v_clob2`, o que indica que os dados do LOB são diferentes; `v_return` é 0 ao comparar `v_clob1` com `v_clob1`, o que indica que os dados do LOB são iguais.

### ***Copiando dados de um CLOB em outro***

A procedure `copy_example()` a seguir copia alguns caracteres de `v_src_clob` em `v_dest_clob` usando `COPY()`:

```

CREATE PROCEDURE copy_example AS
    v_src_clob CLOB;
    v_dest_clob CLOB;
    v_src_offset INTEGER:= 1;
    v_dest_offset INTEGER:= 7;
    v_amount INTEGER:= 5;
BEGIN
    -- obtém o localizador de LOB do CLOB na linha nº 2 da
    -- tabela clob_content em v_dest_clob
    get_clob_locator(v_src_clob, 2);

    -- obtém o localizador de LOB do CLOB na linha nº 1 da
    -- tabela clob_content em v_dest_clob com a cláusula FOR UPDATE
    -- (porque o CLOB será adicionado usando
    -- COPY() posteriormente)
    SELECT clob_column
    INTO v_dest_clob
    FROM clob_content
    WHERE id = 1
    FOR UPDATE;

    -- lê e exibe o conteúdo do CLOB nº 1
    read_clob_example(1);

    -- copia os caracteres de v_src_clob em v_dest_clob usando COPY(),
    -- começando nos deslocamentos especificados por v_dest_offset e
    -- v_src_offset, para um total de v_amount caracteres

```

```

DBMS_LOB.COPY(
    v_dest_clob, v_src_clob,
    v_amount, v_dest_offset, v_src_offset
);

-- lê e exibe o conteúdo do CLOB n° 1
-- e depois reverte a alteração
read_clob_example(1);
ROLLBACK;
END copy_example;
/

```

O exemplo a seguir chama `copy_example()`:

```

CALL copy_example();
v_char_buffer = Creeps in this petty pace
v_amount = 25
v_char_buffer = Creeps fromhis petty pace
v_amount = 25

```

### Usando CLOBs temporários

A procedure `temporary_lob_example()` a seguir ilustra o uso de um CLOB temporário:

```

CREATE PROCEDURE temporary_lob_example AS
    v_clob CLOB;
    v_amount INTEGER;
    v_offset INTEGER:= 1;
    v_char_buffer VARCHAR2(17):= 'Juliet is the sun';
BEGIN
    -- usa CREATETEMPORARY() para criar um CLOB temporário chamado v_clob
    DBMS_LOB.CREATETEMPORARY(v_clob, TRUE);

    -- usa WRITE() para gravar o conteúdo de v_char_buffer em v_clob
    v_amount:= LENGTH(v_char_buffer);
    DBMS_LOB.WRITE(v_clob, v_amount, v_offset, v_char_buffer);

    -- usa ISTEMPORARY() para verificar se v_clob é temporário
    IF (DBMS_LOB.ISTEMPORARY(v_clob) = 1) THEN
        DBMS_OUTPUT.PUT_LINE('v_clob is temporary');
    END IF;

    -- usa READ() para ler o conteúdo de v_clob em v_char_buffer
    DBMS_LOB.READ(
        v_clob, v_amount, v_offset, v_char_buffer
    );
    DBMS_OUTPUT.PUT_LINE('v_char_buffer = ' || v_char_buffer);

    -- usa FREETEMPORARY() para liberar v_clob
    DBMS_LOB.FREETEMPORARY(v_clob);
END temporary_lob_example;
/

```

O exemplo a seguir chama `temporary_lob_example()`:

```
CALL temporary_lob_example();
v_clob is temporary
v_char_buffer = Juliet is the sun
```

### **Apagando dados de um CLOB**

A procedure `erase_example()` a seguir apaga parte de um CLOB usando `ERASE()`:

```
CREATE PROCEDURE erase_example IS
    v_clob CLOB;
    v_offset INTEGER:= 2;
    v_amount INTEGER:= 5;
BEGIN
    -- obtém o localizador de LOB do CLOB na linha nº 1 da
    -- tabela clob_content em v_dest_clob com a cláusula FOR UPDATE
    -- (porque o CLOB será apagado usando
    -- ERASE() posteriormente)
    SELECT clob_column
    INTO v_clob
    FROM clob_content
    WHERE id = 1
    FOR UPDATE;

    -- lê e exibe o conteúdo do CLOB nº 1
    read_clob_example(1);

    -- usa ERASE() para apagar um total de v_amount caracteres
    -- de v_clob, começando em v_offset
    DBMS_LOB.ERASE(v_clob, v_amount, v_offset);

    -- lê e exibe o conteúdo do CLOB nº 1
    -- e depois reverte a alteração
    read_clob_example(1);
    ROLLBACK;
END erase_example;
/
```

O exemplo a seguir chama `erase_example()`:

```
CALL erase_example();
v_char_buffer = Creeps in this petty pace
v_amount = 25
v_char_buffer = C          in this petty pace
v_amount = 25
```

### **Pesquisando os dados em um CLOB**

A procedure `instr_example()` a seguir usa `INSTR()` para pesquisar os dados de caractere armazenados em um CLOB:

```

CREATE PROCEDURE instr_example AS
    v_clob CLOB;
    v_char_buffer VARCHAR2(50) := 'It is the east and Juliet is the sun';
    v_pattern VARCHAR2(5);
    v_offset INTEGER := 1;
    v_amount INTEGER;
    v_occurrence INTEGER;
    v_return INTEGER;
BEGIN
    -- usa CREATETEMPORARY() para criar um CLOB temporário chamado v_clob
    DBMS_LOB.CREATETEMPORARY(v_clob, TRUE);

    -- usa WRITE() para gravar o conteúdo de v_char_buffer em v_clob
    v_amount := LENGTH(v_char_buffer);
    DBMS_LOB.WRITE(v_clob, v_amount, v_offset, v_char_buffer);

    -- usa READ() para ler o conteúdo de v_clob em v_char_buffer
    DBMS_LOB.READ(v_clob, v_amount, v_offset, v_char_buffer);
    DBMS_OUTPUT.PUT_LINE('v_char_buffer = ' || v_char_buffer);

    -- usa INSTR() para procurar a segunda ocorrência de is em v_clob,
    -- e INSTR() retorna 27
    DBMS_OUTPUT.PUT_LINE('Searching for second ''is''');
    v_pattern := 'is';
    v_occurrence := 2;
    v_return := DBMS_LOB.INSTR(v_clob, v_pattern, v_offset, v_occurrence);
    DBMS_OUTPUT.PUT_LINE('v_return = ' || v_return);

    -- usa INSTR() para procurar a primeira ocorrência de Moon em v_clob,
    -- e INSTR() retorna 0, pois Moon não aparece em v_clob
    DBMS_OUTPUT.PUT_LINE('Searching for ''Moon''');
    v_pattern := 'Moon';
    v_occurrence := 1;
    v_return := DBMS_LOB.INSTR(v_clob, v_pattern, v_offset, v_occurrence);
    DBMS_OUTPUT.PUT_LINE('v_return = ' || v_return);

    -- usa FREETEMPORARY() para liberar v_clob
    DBMS_LOB.FREETEMPORARY(v_clob);
END instr_example;
/

```

O exemplo a seguir chama `instr_example()`:

```

CALL instr_example();
v_char_buffer = It is the east and Juliet is the sun
Searching for second 'is'
v_return = 27
Searching for 'Moon'
v_return = 0

```



***Copiando dados de um arquivo em um CLOB e em um BLOB***

A procedure `copy_file_data_to_clob()` a seguir mostra como se lê texto de um arquivo e o armazena em um CLOB:

```
CREATE PROCEDURE copy_file_data_to_clob(
    p_clob_id INTEGER,
    p_directory VARCHAR2,
    p_file_name VARCHAR2
) AS
    v_file UTL_FILE.FILE_TYPE;
    v_chars_read INTEGER;
    v_dest_clob CLOB;
    v_amount INTEGER:= 32767;
    v_char_buffer VARCHAR2(32767);
BEGIN
    -- insere um CLOB vazio
    INSERT INTO clob_content(
        id, clob_column
    ) VALUES (
        p_clob_id, EMPTY_CLOB()
    );

    -- obtém o localizador de LOB do CLOB
    SELECT clob_column
    INTO v_dest_clob
    FROM clob_content
    WHERE id = p_clob_id
    FOR UPDATE;

    -- abre o arquivo para ler o texto (até v_amount caracteres por linha)
    v_file:= UTL_FILE.FOPEN(p_directory, p_file_name, 'r', v_amount);

    -- copia os dados do arquivo em v_dest_clob, uma linha por vez
    LOOP
        BEGIN
            -- lê uma linha do arquivo em v_char_buffer;
            -- GET_LINE() não copia o caractere de nova linha em
            -- v_char_buffer
            UTL_FILE.GET_LINE(v_file, v_char_buffer);
            v_chars_read:= LENGTH(v_char_buffer);

            -- anexa a linha em v_dest_clob
            DBMS_LOB.WRITEAPPEND(v_dest_clob, v_chars_read, v_char_buffer);

            -- anexa um caractere de nova linha em v_dest_clob porque v_char_
            buffer;
            -- o valor ASCII do caractere de nova linha é 10; portanto, CHR(10)
            retorna o caractere --de nova linha
            DBMS_LOB.WRITEAPPEND(v_dest_clob, 1, CHR(10));
        EXCEPTION
            -- quando não há mais dados no arquivo, sai
```

```

        WHEN NO_DATA_FOUND THEN
            EXIT;
        END;
    END LOOP;

    -- fecha o arquivo
    UTL_FILE.FCLOSE(v_file);

    DBMS_OUTPUT.PUT_LINE('Copy successfully completed.');
```

END copy\_file\_data\_to\_clob;  
/

Há alguns aspectos a observar sobre essa procedure:

- UTL\_FILE é um pacote incluído com o banco de dados e contém métodos e tipos que permitem ler e gravar arquivos. Por exemplo, UTL\_FILE.FILE\_TYPE é um tipo de objeto utilizado para representar um arquivo.
- A variável `v_amount` é configurada como 32767, que é o número máximo de caracteres que podem ser lidos de um arquivo durante cada operação de leitura.
- A variável `v_char_buffer` é usada para armazenar os resultados lidos do arquivo, antes que eles sejam anexados em `v_dest_clob`. O comprimento máximo de `v_char_buffer` é configurado como 32767; esse comprimento é grande o suficiente para armazenar o número máximo de caracteres lidos do arquivo durante cada operação de leitura.
- UTL\_FILE.FOPEN(*diretório*, *nome\_arquivo*, *modo\_abertura*, *quantidade*) abre um arquivo; *modo\_abertura* pode ser configurado como um dos modos a seguir:
  - r para ler texto
  - w para gravar texto
  - a para anexar texto
  - rb para ler bytes
  - wb para gravar bytes
  - ab para anexar bytes
- UTL\_FILE.GET\_LINE(`v_file`, `v_char_buffer`) insere uma linha de texto de `v_file` em `v_char_buffer`. GET\_LINE() não adiciona o caractere de nova linha em `v_char_buffer`; como queremos o caractere de nova linha, o adicionamos usando DBMS\_LOB.WRITEAPPEND(`v_dest_clob`, 1, CHR(10)).

O exemplo a seguir chama `copy_file_data_to_clob()` para copiar o conteúdo do arquivo `textContent.txt` em um novo CLOB com um valor de `id` igual a 3:

```
CALL copy_file_data_to_clob(3, 'SAMPLE_FILES_DIR', 'textContent.txt');
```

Copy successfully completed.

A procedure `copy_file_data_to_blob()` a seguir mostra como se lê dados binários de um arquivo e os armazena em um BLOB; observe que um array RAW é usado para armazenar os dados binários lidos do arquivo:

```
CREATE PROCEDURE copy_file_data_to_blob(
    p_blob_id INTEGER,
    p_directory VARCHAR2,
    p_file_name VARCHAR2
) AS
    v_file UTL_FILE.FILE_TYPE;
    v_bytes_read INTEGER;
    v_dest_blob BLOB;
    v_amount INTEGER:= 32767;
    v_binary_buffer RAW(32767);
BEGIN
    -- insere um BLOB vazio
    INSERT INTO blob_content(
        id, blob_column
    ) VALUES (
        p_blob_id, EMPTY_BLOB()
    );

    -- obtém o localizador de LOB do BLOB
    SELECT blob_column
    INTO v_dest_blob
    FROM blob_content
    WHERE id = p_blob_id
    FOR UPDATE;

    -- abre o arquivo para ler os bytes (até v_amount bytes por vez)
    v_file:= UTL_FILE.FOPEN(p_directory, p_file_name, 'rb', v_amount);

    -- copia os dados do arquivo em v_dest_blob
    LOOP
        BEGIN
            -- lê os dados binários do arquivo em v_binary_buffer
            UTL_FILE.GET_RAW(v_file, v_binary_buffer, v_amount);
            v_bytes_read:= LENGTH(v_binary_buffer);

            -- anexa v_binary_buffer em v_dest_blob
            DBMS_LOB.WRITEAPPEND(v_dest_blob, v_bytes_read/2,
                v_binary_buffer);
        EXCEPTION
            -- quando não há mais dados no arquivo, sai
            WHEN NO_DATA_FOUND THEN
                EXIT;
        END;
    END LOOP;

    -- fecha o arquivo
    UTL_FILE.FCLOSE(v_file);
```

```
DBMS_OUTPUT.PUT_LINE('Copy successfully completed.');
```

```
END copy_file_data_to_blob;
```

```
/
```

O exemplo a seguir chama `copy_file_data_to_blob()` para copiar o conteúdo do arquivo `binaryContent.doc` em um novo BLOB com um valor de `id` igual a 3:

```
CALL copy_file_data_to_blob(3, 'SAMPLE_FILES_DIR', 'binaryContent.doc');
```

```
Copy successfully completed.
```

Evidentemente, `copy_file_data_to_blob()` pode ser usada para gravar todos os dados binários contidos em um arquivo em um BLOB. Os dados binários podem conter música, vídeo, imagens, executáveis etc. Experimente usando seus próprios arquivos.

### DICA

*Você também pode carregar dados em massa em um LOB, usando os utilitários Oracle SQL\*Loader e Data Pump; consulte o Oracle Database Large Objects Developer's Guide, publicado pela Oracle Corporation, para ver os detalhes.*

### Copiando dados de um CLOB e de um BLOB em um arquivo

A procedure `copy_clob_data_to_file()` a seguir mostra como ler texto de um CLOB e salvá-lo em um arquivo:

```
CREATE PROCEDURE copy_clob_data_to_file(
    p_clob_id INTEGER,
    p_directory VARCHAR2,
    p_file_name VARCHAR2
) AS
    v_src_clob CLOB;
    v_file UTL_FILE.FILE_TYPE;
    v_offset INTEGER:= 1;
    v_amount INTEGER:= 32767;
    v_char_buffer VARCHAR2(32767);
BEGIN
    -- obtém o localizador de LOB do CLOB
    SELECT clob_column
    INTO v_src_clob
    FROM clob_content
    WHERE id = p_clob_id;

    -- abre o arquivo para gravar o texto (até v_amount caracteres por vez)
    v_file:= UTL_FILE.FOPEN(p_directory, p_file_name, 'w', v_amount);

    -- copia os dados de v_src_clob no arquivo
    LOOP
        BEGIN
            -- lê caracteres de v_src_clob em v_char_buffer
            DBMS_LOB.READ(v_src_clob, v_amount, v_offset, v_char_buffer);

            -- copia os caracteres de v_char_buffer no arquivo
            UTL_FILE.PUT(v_file, v_char_buffer);
```

```

        -- adiciona v_amount em v_offset
        v_offset:= v_offset + v_amount;
    EXCEPTION
        -- quando não há mais dados no arquivo, sai
        WHEN NO_DATA_FOUND THEN
            EXIT;
    END;
END LOOP;

-- descarrega todos os dados restantes no arquivo
UTL_FILE.FFLUSH(v_file);

-- fecha o arquivo
UTL_FILE.FCLOSE(v_file);

DBMS_OUTPUT.PUT_LINE('Copy successfully completed.');
```

END copy\_clob\_data\_to\_file;

/

O exemplo a seguir chama `copy_clob_data_to_file()` para copiar o conteúdo do CLOB nº 3 em um novo arquivo chamado `textContent2.txt`:

```
CALL copy_clob_data_to_file(3, 'SAMPLE_FILES_DIR', 'textContent2.txt');
Copy successfully completed.
```

No diretório `C:\sample_files`, você encontrará o novo arquivo `textContent2.txt`. Esse arquivo contém um texto idêntico a `textContent.txt`.

A procedure `copy_blob_data_to_file()` a seguir mostra como ler dados binários de um BLOB e salvá-los em um arquivo:

```
CREATE PROCEDURE copy_blob_data_to_file(
    p_blob_id INTEGER,
    p_directory VARCHAR2,
    p_file_name VARCHAR2
) AS
    v_blob_ori BLOB;
    v_file UTL_FILE.FILE_TYPE;
    v_offset INTEGER:= 1;
    v_amount INTEGER:= 32767;
    v_binary_buffer RAW(32767);
BEGIN
    -- obtém o localizador de LOB do BLOB
    SELECT blob_column
    INTO v_blob_ori
    FROM blob_content
    WHERE id = p_blob_id;

    -- abre o arquivo para gravar os bytes (até v_amount bytes por vez)
    v_file:= UTL_FILE.FOPEN(p_directory, p_file_name, 'wb', v_amount);

    -- copia os dados de v_src_blob no arquivo
```

```

LOOP
  BEGIN
    -- lê caracteres de v_src_blob em v_binary_buffer
    DBMS_LOB.READ(v_src_blob, v_amount, v_offset, v_binary_buffer);

    -- copia os dados binários de v_binary_buffer no arquivo
    UTL_FILE.PUT_RAW(v_file, v_binary_buffer);

    -- adiciona v_amount em v_offset
    v_offset:= v_offset + v_amount;
  EXCEPTION
    -- quando não há mais dados no arquivo, sai
    WHEN NO_DATA_FOUND THEN
      EXIT;
  END;
END LOOP;

-- descarrega os dados restantes no arquivo
UTL_FILE.FFLUSH(v_file);

-- fecha o arquivo
UTL_FILE.FCLOSE(v_file);

  DBMS_OUTPUT.PUT_LINE('Copy successfully completed.');
```

END copy\_blob\_data\_to\_file;

/

O exemplo a seguir chama `copy_blob_data_to_file()` para copiar o conteúdo do BLOB nº 3 em um novo arquivo chamado `binaryContent2.doc`:

```

CALL copy_blob_data_to_file(3, 'SAMPLE_FILES_DIR', 'binaryContent2.doc');
```

Copy successfully completed.

No diretório `C:\sample_files`, você encontrará o novo arquivo `binaryContent2.doc`. Esse arquivo contém texto idêntico a `binaryContent.doc`.

Evidentemente, `copy_blob_data_to_file()` pode ser usada para gravar qualquer dado binário contido em um BLOB em um arquivo. Os dados binários podem conter música, vídeo, imagens, executáveis etc.

### ***Copiando dados de um BFILE em um CLOB e em um BLOB***

A procedure `copy_bfile_data_to_clob()` a seguir mostra como ler texto de um BFILE e salvá-lo em um CLOB:

```

CREATE PROCEDURE copy_bfile_data_to_clob(
  p_bfile_id INTEGER,
  p_clob_id INTEGER
) AS
  v_src_bfile BFILE;
  v_directory VARCHAR2(200);
  v_filename VARCHAR2(200);
  v_length INTEGER;
  v_dest_clob CLOB;
```

```

v_amount INTEGER:= DBMS_LOB.LOBMAXSIZE;
v_dest_offset INTEGER:= 1;
v_src_offset INTEGER:= 1;
v_src_csid INTEGER:= DBMS_LOB.DEFAULT_CSID;
v_lang_context INTEGER:= DBMS_LOB.DEFAULT_LANG_CTX;
v_warning INTEGER;
BEGIN
    -- obtém o localizador do BFILE
    SELECT bfile_column
    INTO v_src_bfile
    FROM bfile_content
    WHERE id = p_bfile_id;

    -- usa FILEEXISTS() para verificar se o arquivo existe
    -- (FILEEXISTS() retorna 1 se o arquivo existe)
    IF (DBMS_LOB.FILEEXISTS(v_src_bfile) = 1) THEN
        -- usa OPEN() para abrir o arquivo
        DBMS_LOB.OPEN(v_src_bfile);

        -- usa FILEGETNAME() para obter o nome do arquivo e o diretório
        DBMS_LOB.FILEGETNAME(v_src_bfile, v_directory, v_filename);
        DBMS_OUTPUT.PUT_LINE('Directory = ' || v_directory);
        DBMS_OUTPUT.PUT_LINE('Filename = ' || v_filename);

        -- insere um CLOB vazio
        INSERT INTO clob_content (
            id, clob_column
        ) VALUES (
            p_clob_id, EMPTY_CLOB()
        );

        -- obtém o localizador de LOB do CLOB (com a cláusula FOR UPDATE)
        SELECT clob_column
        INTO v_dest_clob
        FROM clob_content
        WHERE id = p_clob_id
        FOR UPDATE;

        -- usa LOADCLOBFROMFILE() para obter até v_amount caracteres
        -- de v_src_bfile e os armazenar em v_dest_clob, começando
        -- no deslocamento 1 em v_src_bfile e v_dest_clob
        DBMS_LOB.LOADCLOBFROMFILE(
            v_dest_clob, v_src_bfile,
            v_amount, v_dest_offset, v_src_offset,
            v_src_csid, v_lang_context, v_warning
        );

        -- verifica se existe um caractere que não pode ser convertido em v_
        warning
        IF (v_warning = DBMS_LOB.WARN_INCONVERTIBLE_CHAR) THEN
            DBMS_OUTPUT.PUT_LINE('Warning! Inconvertible character. ');
        END IF;
    
```

```

-- usa CLOSE() para fechar v_src_bfile
DBMS_LOB.CLOSE(v_src_bfile);
DBMS_OUTPUT.PUT_LINE('Copy successfully completed.');
```

ELSE

```

  DBMS_OUTPUT.PUT_LINE('File does not exist');
END IF;
END copy_bfile_data_to_clob;
/
```

O exemplo a seguir chama `copy_bfile_data_to_clob()` para copiar o conteúdo do BFILE nº 1 em um novo CLOB com um valor de id igual a 4:

```

CALL copy_bfile_data_to_clob(1, 4);
Copy successfully completed.
```

O exemplo a seguir chama `copy_clob_data_to_file()` para copiar o conteúdo do CLOB nº 4 em um novo arquivo chamado `textContent3.txt`:

```

CALL copy_clob_data_to_file(4, 'SAMPLE_FILES_DIR', 'textContent3.txt');
Copy successfully completed.
```

Você encontrará o novo arquivo `textContent3.txt` no diretório `C:\sample_files`. Esse arquivo contém texto idêntico a `textContent.txt`.

A procedure `copy_bfile_data_to_blob()` a seguir mostra como ler dados binários de um BFILE e salvá-los em um BLOB:

```

CREATE PROCEDURE copy_bfile_data_to_blob(
  p_bfile_id INTEGER,
  p_blob_id INTEGER
) AS
  v_src_bfile BFILE;
  v_directory VARCHAR2(200);
  v_filename VARCHAR2(200);
  v_length INTEGER;
  v_dest_blob BLOB;
  v_amount INTEGER:= DBMS_LOB.LOBMAXSIZE;
  v_dest_offset INTEGER:= 1;
  v_src_offset INTEGER:= 1;
BEGIN
  -- obtém o localizador do BFILE
  SELECT bfile_column
  INTO v_src_bfile
  FROM bfile_content
  WHERE id = p_bfile_id;

  -- usa FILEEXISTS() para verificar se o arquivo existe
  -- (FILEEXISTS() retorna 1 se o arquivo existe)
  IF (DBMS_LOB.FILEEXISTS(v_src_bfile) = 1) THEN
    -- usa OPEN() para abrir o arquivo
    DBMS_LOB.OPEN(v_src_bfile);

    -- usa FILEGETNAME() para obter o nome do arquivo e
```



```

-- o diretório
DBMS_LOB.FILEGETNAME(v_src_bfile, v_directory, v_filename);
DBMS_OUTPUT.PUT_LINE('Directory = ' || v_directory);
DBMS_OUTPUT.PUT_LINE('Filename = ' || v_filename);

-- insere um BLOB vazio
INSERT INTO blob_content(
    id, blob_column
) VALUES (
    p_blob_id, EMPTY_BLOB()
);

-- obtém o localizador de LOB do BLOB (pcom a cláusula FOR UPDATE)
SELECT blob_column
INTO v_dest_blob
FROM blob_content
WHERE id = p_blob_id
FOR UPDATE;

-- usa LOADBLOBFROMFILE() para obter até v_amount bytes
-- de v_src_bfile e os armazena em v_dest_blob, começando
-- no deslocamento 1 em v_src_bfile e v_dest_blob
DBMS_LOB.LOADBLOBFROMFILE(
    v_dest_blob, v_src_bfile,
    v_amount, v_dest_offset, v_src_offset
);

-- usa CLOSE() para fechar v_src_bfile
DBMS_LOB.CLOSE(v_src_bfile);
DBMS_OUTPUT.PUT_LINE('Copy successfully completed.');
```

ELSE

```

    DBMS_OUTPUT.PUT_LINE('File does not exist');
END IF;
END copy_bfile_data_to_blob;
/
```

O exemplo a seguir chama `copy_bfile_data_to_blob()` para copiar o conteúdo do BFILE nº 2 em um novo BLOB com um valor de id igual a 4:

```
CALL copy_bfile_data_to_blob(2, 4);
Copy successfully completed.
```

O exemplo a seguir chama `copy_blob_data_to_file()` para copiar o conteúdo do BLOB nº 4 em um novo arquivo chamado `binaryContent3.doc`:

```
CALL copy_blob_data_to_file(4, 'SAMPLE_FILES_DIR', 'binaryContent3.doc');
Copy successfully completed.
```

Você encontrará o novo arquivo `binaryContent3.doc` no diretório `C:\sample_files`. Esse arquivo contém texto idêntico a `binaryContent.doc`.

Com isso terminamos o conteúdo sobre large objects. Na próxima seção, você vai aprender sobre os tipos `LONG` e `LONG RAW`.

## TIPOS LONG E LONG RAW

Mencionamos no início deste capítulo que os LOBs são o tipo de armazenamento preferencial para grandes blocos de dados, mas você poderá encontrar bancos de dados que ainda utilizam os seguintes tipos:

- **LONG** Usado para armazenar até 2 gigabytes de dados de caractere
- **LONG RAW** Usado para armazenar até 2 gigabytes de dados binários
- **RAW** Usado para armazenar até 4 kilobytes de dados binários

Nesta seção, você vai aprender a usar os tipos **LONG** e **LONG RAW**. O tipo **RAW** é usado da mesma maneira que **LONG RAW**; portanto, omitiremos a abordagem de **RAW**.

### As tabelas de exemplo

Nesta seção, serão abordadas estas tabelas:

- **long\_content** Contém uma coluna **LONG** chamada `long_column`
- **long\_raw\_content** Contém uma coluna **LONG RAW** chamada `long_raw_column`

Essas duas tabelas são criadas pelo script `lob_schema.sql` usando as seguintes instruções:

```
CREATE TABLE long_content (
    id          INTEGER PRIMARY KEY,
    long_column LONG NOT NULL
);

CREATE TABLE long_raw_content (
    id          INTEGER PRIMARY KEY,
    long_raw_column LONG RAW NOT NULL
);
```

### Adicionando dados em colunas LONG e LONG RAW

As instruções **INSERT** a seguir adicionam linhas na tabela `long_content`:

```
INSERT INTO long_content (
    id, long_column
) VALUES (
    1, 'Creeps in this petty pace'
);

INSERT INTO long_content (
    id, long_column
) VALUES (
    2, ' from day to day'
);
```

As instruções **INSERT** a seguir adicionam linhas na tabela `long_raw_content` (a primeira instrução **INSERT** contém um número binário e a segunda, um número hexadecimal):

```
INSERT INTO long_raw_content (
    id, long_raw_column
```

```

) VALUES (
  1, '10011101010101111'
);

INSERT INTO long_raw_content (
  id, long_raw_column
) VALUES (
  2, 'A0FFB71CF90DE'
);

```

Na próxima seção, você vai aprender a converter colunas LONG e LONG RAW em LOBs.

## Convertendo colunas LONG e LONG RAW em LOBs

Você pode converter um tipo LONG em um CLOB usando a função TO\_LOB(). Por exemplo, a instrução a seguir converte long\_column em um CLOB usando TO\_LOB() e armazena o resultado na tabela clob\_content:

```

INSERT INTO clob_content
SELECT 10 + id, TO_LOB(long_column)
FROM long_content;

```

2 rows created.

Você pode converter um tipo LONG RAW em um BLOB usando a função TO\_LOB(). Por exemplo, a instrução a seguir converte long\_raw\_column em um BLOB usando TO\_LOB() e armazena o resultado na tabela blob\_content:

```

INSERT INTO blob_content
SELECT 10 + id, TO_LOB(long_raw_column)
FROM long_raw_content;

```

2 rows created.

Você também pode usar a instrução ALTER TABLE para converter colunas LONG e LONG RAW diretamente. Por exemplo, a instrução a seguir converte long\_column em um CLOB:

```

ALTER TABLE long_content MODIFY (long_column CLOB);

```

O exemplo a seguir converte long\_raw\_column em um BLOB:

```

ALTER TABLE long_raw_content MODIFY (long_raw_column BLOB);

```

### CUIDADO

*Você não deve modificar tabelas que estão sendo usadas em uma aplicação de produção.*

Depois que uma coluna LONG ou LONG RAW for convertida em um LOB, você pode aproveitar a produtividade dos métodos PL/SQL descritos anteriormente para acessar o LOB.

## APRIMORAMENTOS FEITOS PELO ORACLE DATABASE 10G NOS LARGE OBJECTS

Nesta seção, você vai aprender sobre as seguintes melhorias feitas nos large objects no Oracle Database 10g:

- Conversão implícita entre objetos CLOB e NCLOB
- Uso do atributo: new ao utilizar LOBs em um trigger

Fornecemos um script SQL\*Plus chamado `lob_schema2.sql` no diretório `SQL`. Esse script pode ser executado pelo Oracle Database 10g e por versões superiores. O script cria um usuário chamado `lob_user2` com a senha `lob_password` e cria as tabelas e o código PL/SQL utilizados nesta seção. Depois que o script terminar, você estará conectado como `lob_user2`.

### Conversão implícita entre objetos CLOB e NCLOB

No ambiente corporativo global atual, talvez você tenha que lidar com conversões entre Unicode e um conjunto de caracteres de idioma nacional. O Unicode é um conjunto de caracteres universal que permite armazenar texto que pode ser convertido em qualquer idioma; ele faz isso fornecendo um código único para cada caractere, independentemente do idioma. Um conjunto de caracteres nacional armazena texto em um idioma específico.

Nas versões do banco de dados anteriores ao Oracle Database 10g, você precisa converter entre texto Unicode e texto do conjunto de caracteres nacional explicitamente, utilizando as funções `TO_CLOB()` e `TO_NCLOB()`. `TO_CLOB()` permite converter texto armazenado em um tipo `VARCHAR2`, `NVARCHAR2` ou `NCLOB` para um `CLOB`. Analogamente, `TO_NCLOB()` permite converter texto armazenado em um tipo `VARCHAR2`, `NVARCHAR2` ou `CLOB` para um `NCLOB`.

O Oracle Database 10g e superiores convertem texto Unicode e texto do conjunto de caracteres nacional em objetos `CLOB` e `NCLOB` implicitamente, o que evita o uso de `TO_CLOB()` e `TO_NCLOB()`. Você pode usar essa conversão implícita para variáveis `IN` e `OUT` em consultas e instruções DML, assim como para parâmetros de métodos PL/SQL e atribuições de variável.

Vejam os exemplos. A instrução a seguir cria uma tabela chamada `nclob_content` que contém uma coluna `NCLOB` chamada `nclob_column`:

```
CREATE TABLE nclob_content (
    id INTEGER PRIMARY KEY,
    nclob_column NCLOB
);
```

A procedure `nclob_example()` a seguir mostra a conversão implícita de um `CLOB` em um `NCLOB` e vice-versa:

```
CREATE PROCEDURE nclob_example
AS
    v_clob CLOB:= 'It is the east and Juliet is the sun';
    v_nclob NCLOB;
BEGIN
```

```

-- insere v_clob em nclob_column; isso converte
-- o CLOB v_clob em um NCLOB implicitamente, armazenando
-- o conteúdo de v_clob na tabela nclob_content
INSERT INTO nclob_content (
    id, nclob_column
) VALUES (
    1, v_clob
);

-- seleciona nclob_column em v_clob; isso converte
-- o NCLOB armazenado em nclob_column em um
-- CLOB implicitamente, recuperando o conteúdo de nclob_column
-- em v_clob
SELECT nclob_column
INTO v_clob
FROM nclob_content
WHERE id = 1;

-- exibe o conteúdo de v_clob
DBMS_OUTPUT.PUT_LINE('v_clob = ' || v_clob);
END nclob_example;
/

```

O exemplo a seguir ativa a saída do servidor e chama `nclob_example()`:

```

SET SERVEROUTPUT ON
CALL nclob_example();
v_clob = It is the east and Juliet is the sun

```

## Uso do atributo :new ao utilizar LOBs em um trigger

No Oracle Database 10g e superiores, você pode usar o atributo `:new` ao referenciar LOBs em um trigger em nível de linha `BEFORE UPDATE` ou `BEFORE INSERT`. O exemplo a seguir cria um trigger chamado `before_clob_content_update`; o trigger dispara quando a tabela `clob_content` é atualizada e exibe o comprimento dos novos dados em `clob_column`; observe que `:new` é usado para acessar os novos dados em `clob_column`:

```

CREATE TRIGGER before_clob_content_update
BEFORE UPDATE
ON clob_content
FOR EACH ROW
BEGIN
    DBMS_OUTPUT.PUT_LINE('clob_content changed');
    DBMS_OUTPUT.PUT_LINE(
        'Length = ' || DBMS_LOB.GETLENGTH(:new.clob_column)
    );
END before_clob_content_update;
/

```

O exemplo a seguir atualiza a tabela `clob_content`, causando o disparo do trigger:

```

UPDATE clob_content
SET clob_column = 'Creeps in this petty pace'
WHERE id = 1;
clob_content changed
Length = 25

```

## APRIMORAMENTOS FEITOS PELO ORACLE DATABASE 11g NOS LARGE OBJECTS

Nesta seção, você vai aprender sobre as seguintes melhorias feitas nos large objects pelo Oracle Database 11g:

- Criptografia de dados BLOB, CLOB e NCLOB, o que impede a visualização e a modificação não autorizadas dos dados
- Compactação para reduzir o tamanho de dados BLOB, CLOB e NCLOB
- Eliminação da duplicação (deduplication) de dados BLOB, CLOB e NCLOB para detectar e remover dados repetidos, automaticamente

### Criptografia de dados de LOB

Você pode mascarar seus dados usando criptografia para que usuários não autorizados não possam vê-los nem modificá-los. Você deve criptografar dados sigilosos, como números de cartão de crédito, números de CPF etc.

Antes de poder criptografar dados, você ou o administrador do banco de dados precisa configurar uma “wallet” para armazenar detalhes de segurança. Os dados que estão em uma wallet incluem uma chave privada para criptografar e descriptografar dados. Nesta seção, você verá como criar uma wallet, como criptografar dados de LOB e como criptografar dados de coluna normais.

#### *Criando uma wallet*

Para criar uma wallet, você precisa primeiro criar um diretório chamado wallet no diretório \$ORACLE\_BASE\admin\\${ORACLE\_SID}, onde ORACLE\_BASE é o diretório base onde o software de banco de dados Oracle está instalado e ORACLE\_SID é o identificador de sistema do banco de dados no qual a wallet vai ser criada. Por exemplo, em um computador executando Windows XP e Oracle Database 11g, é possível criar o diretório wallet em C:\oracle\_11g\admin\orcl.

Uma vez criado o diretório wallet, você precisa executar o SQL\*Plus, conectar-se ao banco de dados usando uma conta de usuário privilegiado (por exemplo, system) e executar um comando ALTER SYSTEM para definir a senha da chave de criptografia da wallet, como mostrado aqui:

```

SQL> CONNECT system/manager
SQL> ALTER SYSTEM SET ENCRYPTION KEY IDENTIFIED BY "testpassword123";
System altered.

```

Feito isso, um arquivo chamado ewallet.p12 aparece no diretório wallet e o banco de dados abre a wallet automaticamente. A senha da chave de criptografia é armazenada na wallet e é utilizada para criptografar e descriptografar dados de forma transparente aos usuários.

Fornecemos um script SQL\*Plus chamado `lob_schema3.sql` no diretório SQL. Esse script pode ser executado com o Oracle Database 11g. O script cria um usuário chamado `lob_user3` com a senha `lob_password` e também cria as tabelas utilizadas posteriormente nesta seção. Depois que o script terminar, você estará conectado como `lob_user3`.

### **Criptografia de dados de LOB**

Você pode criptografar os dados armazenados em um BLOB, CLOB ou NCLOB para impedir o acesso não autorizado aos dados; você não pode criptografar um BFILE, pois o arquivo em si é armazenado fora do banco de dados.

Você pode usar os seguintes algoritmos para criptografar dados:

- **3DES168** O algoritmo Triple-DES (Data Encryption Standard) com um comprimento de chave de 168 bits.
- **AES128** O algoritmo Advanced Encryption Standard com um comprimento de chave de 128 bits. Os algoritmos AES foram desenvolvidos para substituir os algoritmos baseados em DES, mais antigos.
- **AES192** O algoritmo Advanced Encryption Standard com um comprimento de chave de 192 bits.
- **AES256** O algoritmo Advanced Encryption Standard com um comprimento de chave de 256 bits. Este é o algoritmo de criptografia mais seguro suportado pelo banco de dados Oracle.

A instrução a seguir cria uma tabela com um CLOB cujo conteúdo deve ser criptografado com o algoritmo AES128; observe o uso das palavras-chave `ENCRYPT` e `SECUREFILE`, que são obrigatórias ao se criptografar dados:

```
CREATE TABLE clob_content (
    id INTEGER PRIMARY KEY,
    clob_column CLOB ENCRYPT USING 'AES128'
) LOB(clob_column) STORE AS SECUREFILE (
    CACHE
);
```

Como você pode ver, o conteúdo de `clob_column` será criptografado com o algoritmo AES128. Se você omitir a palavra-chave `USING` e o algoritmo, o algoritmo AES192 padrão será usado.

A palavra-chave `CACHE` na instrução `CREATE TABLE` indica que o banco de dados coloca os dados do LOB no cache de buffer para obter acesso mais rápido. As opções que você pode usar para cache de buffer são:

- **CACHE READS** Use quando os dados do LOB forem lidos freqüentemente, mas gravados apenas uma vez ou ocasionalmente.
- **CACHE** Use quando os dados do LOB forem lidos e gravados freqüentemente.
- **NOCACHE** Use quando os dados do LOB forem lidos uma vez ou ocasionalmente e gravados uma vez ou ocasionalmente. Esta é a opção padrão.

As instruções `INSERT` a seguir adicionam duas linhas na tabela `clob_content`:

```
INSERT INTO clob_content (
    id, clob_column
) VALUES (
    1, TO_CLOB('Creeps in this petty pace')
```

```
);

INSERT INTO clob_content (
    id, clob_column
) VALUES (
    2, TO_CLOB(' from day to day')
);
```

Os dados fornecidos para `clob_column` nessas instruções são criptografados automaticamente e de forma transparente pelo banco de dados.

A consulta a seguir recupera as linhas da tabela `clob_content`:

```
SELECT *
FROM clob_content;

          ID
-----
CLOB_COLUMN
-----
          1
Creeps in this petty pace

          2
from day to day
```

Quando os dados são recuperados, eles são descriptografados automaticamente pelo banco de dados e, então, retornados para o SQL\*Plus.

Desde que a wallet esteja aberta, você pode armazenar e recuperar dados criptografados; quando a wallet está fechada, isso não é possível. Vejamos o que acontece quando a wallet está fechada; as instruções a seguir se conectam como o usuário `system` e fecham a wallet:

```
CONNECT system/manager
ALTER SYSTEM SET WALLET CLOSE;
```

Se agora você tentar se conectar como `lob_user3` e recuperar `clob_column` da tabela `clob_content`, obterá o erro ORA-28365: wallet is not open:

```
CONNECT lob_user3/lob_password
SELECT clob_column
FROM clob_content;
ORA-28365: wallet is not open
```

Você ainda pode recuperar e modificar o conteúdo de colunas não criptografadas; por exemplo, a consulta a seguir recupera a coluna `id` da tabela `clob_content`:

```
SELECT id
FROM clob_content;

          ID
-----
          1
          2
```



As instruções a seguir se conectam como o usuário `system` e reabrem a wallet:

```
CONNECT system/manager  
ALTER SYSTEM SET WALLET OPEN IDENTIFIED BY "testpassword123";
```

Depois de fazer isso, você pode recuperar e modificar o conteúdo de `clob_column` da tabela `clob_content`.

### **Criptografando dados de coluna**

Você também pode criptografar dados de coluna normais. Esse recurso foi introduzido no Oracle Database 10g release 2. Por exemplo, a instrução a seguir cria uma tabela chamada `credit_cards` com uma coluna criptografada chamada `card_number`:

```
CREATE TABLE credit_cards (  
    card_number NUMBER(16, 0) ENCRYPT,  
    first_name VARCHAR2(10),  
    last_name VARCHAR2(10),  
    expiration DATE  
);
```

Para criptografar uma coluna, você pode usar os mesmos algoritmos utilizados para um LOB: 3DES168, AES128, AES192 (o padrão) e AES256. Como não especificamos um algoritmo após a palavra-chave `ENCRYPT` para a coluna `card_number`, o algoritmo AES192 padrão é usado.

As instruções `INSERT` a seguir adicionam duas linhas na tabela `credit_cards`:

```
INSERT INTO credit_cards (  
    card_number, first_name, last_name, expiration  
) VALUES (  
    1234, 'Jason', 'Bond', '03-FEB-2008'  
);  
  
INSERT INTO credit_cards (  
    card_number, first_name, last_name, expiration  
) VALUES (  
    5768, 'Steve', 'Edwards', '07-MAR-2009'  
);
```

Contanto que a wallet esteja aberta, você pode recuperar e modificar o conteúdo da coluna `card_number`. Se a wallet estiver fechada, você obterá o erro `ORA-28365: wallet is not open`. Você viu exemplos que ilustram esses conceitos na seção anterior; portanto, não repetiremos exemplos semelhantes aqui.

O acesso aos dados de uma coluna criptografada causa um overhead adicional. O overhead para criptografar e descriptografar uma coluna é estimado pela Oracle Corporation em cerca de 5%; isso significa que uma instrução `SELECT` ou `INSERT` leva cerca de 5% de tempo a mais para terminar. O overhead total depende do número de colunas criptografadas e de sua frequência de acesso; portanto, você só deve criptografar colunas que contenham dados sigilosos.

### **NOTA**

*Se você quiser aprender mais sobre wallets e segurança de banco de dados em geral, consulte o Advanced Security Administrator's Guide, publicado pela Oracle Corporation.*

## Compactando dados de LOB

Você pode compactar os dados armazenados em um BLOB, CLOB ou NCLOB para reduzir o espaço de armazenamento. Por exemplo, a instrução a seguir cria uma tabela com um CLOB cujo conteúdo deve ser compactado; observe o uso da palavra-chave `COMPRESS`:

```
CREATE TABLE clob_content3 (
  id          INTEGER PRIMARY KEY,
  clob_column CLOB
) LOB(clob_column) STORE AS SECUREFILE (
  COMPRESS
  CACHE
);
```

### NOTA

*Mesmo que a tabela não contenha dados criptografados, a cláusula `SECUREFILE` deve ser usada.*

Quando você adicionar dados no LOB, eles serão compactados automaticamente pelo banco de dados. Da mesma forma, quando você ler dados de um LOB, eles serão descompactados automaticamente. Você pode usar `COMPRESS HIGH` para obter a máxima compactação dos dados; o padrão é `COMPRESS MEDIUM` e a palavra-chave `MEDIUM` é opcional. Quanto mais alta a compactação, maior o overhead ao ler e gravar dados de LOB.

## Removendo dados de LOB duplicados

Você pode configurar um BLOB, CLOB ou NCLOB de modo que os dados duplicados fornecidos a ele sejam removidos automaticamente; esse processo é conhecido como eliminação de duplicação (deduplication) de dados e pode economizar espaço de armazenamento. Por exemplo, a instrução a seguir cria uma tabela com um CLOB cujo conteúdo precisa ter a duplicação eliminada; observe o uso das palavras-chave `DEDUPLICATE LOB`:

```
CREATE TABLE clob_content2 (
  id          INTEGER PRIMARY KEY,
  clob_column CLOB
) LOB(clob_column) STORE AS SECUREFILE (
  DEDUPLICATE LOB
  CACHE
);
```

Todos os dados duplicados adicionados no LOB serão removidos automaticamente pelo banco de dados. O banco de dados utiliza o algoritmo de hashing seguro SHA1 para detectar dados duplicados.

Se você quiser aprender mais sobre large objects, consulte o *Oracle Database Large Objects Developer's Guide*, publicado pela Oracle Corporation.

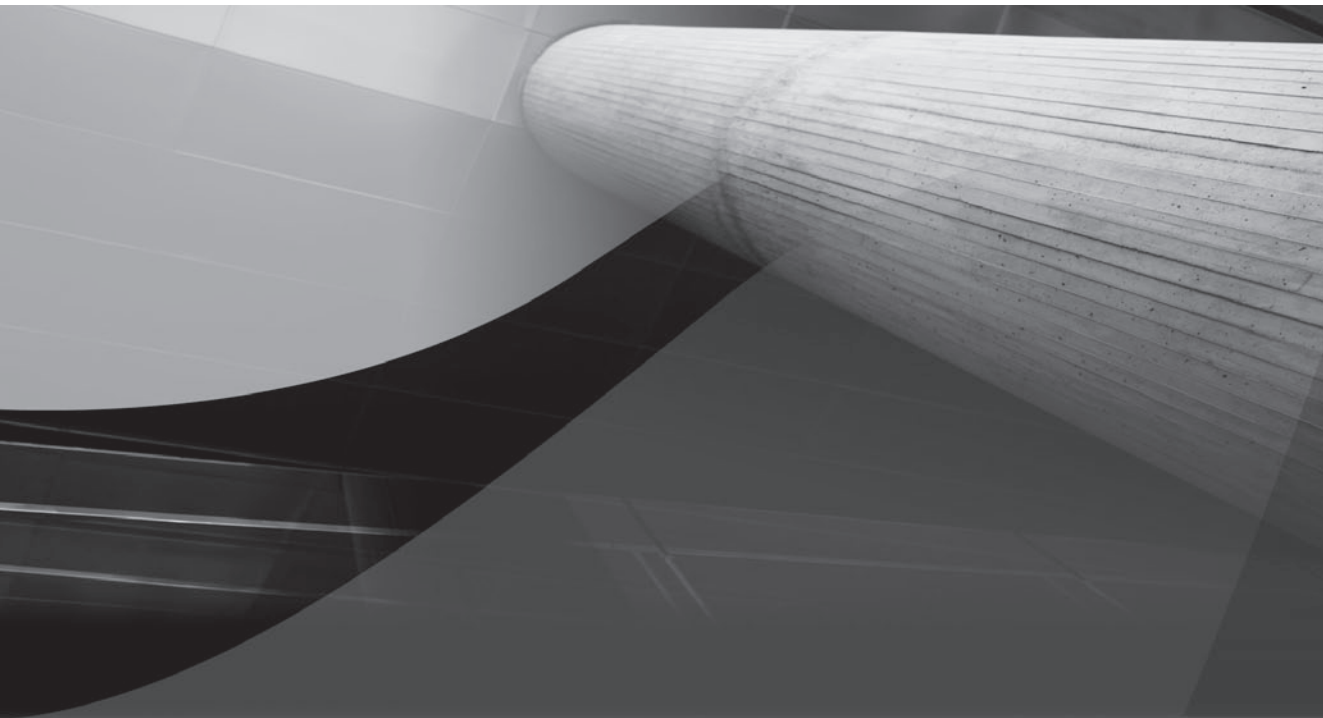
## RESUMO

Neste capítulo, você aprendeu que:

- Os LOBs podem ser usados para armazenar dados binários, dados de caractere e referências a arquivos externos. Os LOBs podem armazenar até 128 terabytes de dados.

- Existem quatro tipos de LOB: CLOB, NCLOB, BLOB e BFILE.
- Um CLOB armazena dados de caractere.
- Um NCLOB armazena dados de caractere multi-byte.
- Um BLOB armazena dados binários.
- Um BFILE armazena um ponteiro para um arquivo localizado no sistema de arquivos.
- Um LOB consiste em duas partes: um localizador (o qual especifica a posição dos dados do LOB) e os dados em si.
- O pacote PL/SQL DBMS\_LOB contém métodos para acessar LOBs.

No próximo capítulo, você vai aprender a executar instruções SQL a partir de um programa Java.



# CAPÍTULO 15

Executando SQL  
usando Java

Neste capítulo, você vai:

- Aprender a executar SQL a partir de programas Java usando a API (Interface de Programação de Aplicativo) JDBC (Java Database Connectivity)
- Examinar os vários drivers JDBC da Oracle que podem ser utilizados para se conectar com um banco de dados Oracle
- Executar consultas e instruções DML em SQL para acessar tabelas de banco de dados
- Usar os vários tipos Java para obter e configurar valores de coluna no banco de dados
- Aprender a executar instruções de controle de transação e instruções DDL em SQL
- Tratar as exceções de banco de dados que podem ocorrer quando um programa Java é executado
- Examinar as extensões do software de banco de dados Oracle para JDBC
- Ver programas Java completos que ilustram o uso de JDBC



#### NOTA

*Este capítulo fornece uma introdução ao JDBC. Para saber mais sobre o uso de JDBC com um banco de dados Oracle, consulte o livro Oracle9i JDBC Programming (McGraw-Hill/Osborne, 2002).*

## COMEÇANDO

Antes de executar os exemplos deste capítulo, você precisa instalar uma versão do SDK (Software Development Kit) Java da Sun. Você pode fazer o download do SDK e ver instruções de instalação completas no site para Java da Sun, no endereço [java.sun.com](http://java.sun.com).



#### NOTA

*Este capítulo foi escrito usando Java 1.6.0, que é instalado com Java EE 5 SDK Update 2.*

O diretório onde você instalou o software Oracle em sua máquina é chamado ORACLE\_HOME. Em meu computador Windows, esse diretório é E:\oracle\_11g\product\11.1.0\db1. Dentro de ORACLE\_HOME existem muitos subdiretórios, entre eles o jdbc. O diretório jdbc contém o seguinte:

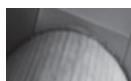
- Um arquivo de texto chamado Readme.txt. Você deve abrir e ler esse arquivo, pois ele contém itens importantes, como informações de versão e as instruções de instalação mais recentes.
- Um diretório chamado lib, que contém diversos arquivos JAR (Java Archive).

## CONFIGURANDO SEU COMPUTADOR

Depois de fazer o download e instalar o software necessário, o próximo passo é configurar seu computador para desenvolver e executar programas Java contendo instruções JDBC. Você precisa configurar quatro variáveis de ambiente em sua máquina:

- ORACLE\_HOME
- JAVA\_HOME
- PATH
- CLASSPATH

Se estiver usando Unix ou Linux, você também precisará configurar a variável de ambiente adicional `LD_LIBRARY_PATH`. Você vai aprender a configurar essas variáveis de ambiente nas seções a seguir.



### CUIDADO

*As informações desta seção estavam corretas quando este livro foi redigido. Você precisa ler o arquivo `Readme.txt` do diretório `ORACLE_HOME\jdbc` para conferir as observações de versão e instruções de instalação mais recentes.*

## Configurando a variável de ambiente ORACLE\_HOME

O subdiretório `ORACLE_HOME` está localizado no diretório onde você instalou o software Oracle. Você precisará configurar em sua máquina uma variável de ambiente chamada `ORACLE_HOME` que especifique esse diretório.

### Configurando uma variável de ambiente no Windows XP

Para configurar uma variável de ambiente no Windows XP, execute os passos a seguir:

1. Abra o Painel de controle.
2. Clique duas vezes em Sistema. Isso exibe a caixa de diálogo Propriedades do sistema.
3. Selecciona a guia Avançado.
4. Clique no botão Variáveis de ambiente. Isso exibe a caixa de diálogo Variáveis de ambiente.
5. Clique no botão Novo na área Variáveis de sistema (o painel inferior da caixa de diálogo).
6. Configure o nome da variável como `ORACLE_HOME` e configure o valor com o seu diretório `ORACLE_HOME`. (Em minha máquina com Windows XP, `ORACLE_HOME` está configurado como `E:\oracle_11g\product\11.1.0\db1.`)

### Configurando uma variável de ambiente com Unix ou Linux

Para configurar uma variável de ambiente no Unix ou Linux, você precisa adicionar linhas em um arquivo especial; o arquivo que você precisa modificar depende do shell que está usando. Se estiver usando o shell Bourne, Korn ou Bash, adicione linhas semelhantes às seguintes em `.profile` (ao usar shell Bourne ou Korn) ou em `.bash_profile` (shell Bash):

```
ORACLE_HOME=/u01/app/oracle/product/11.1.0/db_1
export ORACLE_HOME
```

**NOTA**

*Você precisará substituir o diretório mostrado no exemplo anterior pelo ORACLE\_HOME correto de sua configuração.*

Se você estiver usando o shell C, adicione a linha a seguir em seu arquivo .login:

```
setenv ORACLE_HOME /u01/app/oracle/product/11.1.0/db_1
```

## Configurando a variável de ambiente JAVA\_HOME

A variável de ambiente `JAVA_HOME` especifica o diretório onde você instalou o SDK Java. Por exemplo, se você instalou o SDK Java no diretório `E:\java\jdk`, crie uma variável de sistema `JAVA_HOME` e configure-a como `E:\java\jdk`. Para tanto, use passos semelhantes àqueles mostrados na seção anterior.

## Configurando a variável de ambiente PATH

A variável de ambiente `PATH` contém uma lista de diretórios. Quando você digita um comando usando a linha de comando do sistema operacional, o computador procura o executável nos diretórios do `PATH`. Você precisa adicionar os dois diretórios a seguir em sua variável `PATH` existente:

- O subdiretório `bin` onde instalou o SDK Java
- O subdiretório `BIN` de `ORACLE_HOME`

Por exemplo, se você instalou o SDK Java no diretório `E:\java\jdk` e `ORACLE_HOME` é `E:\oracle_11g\product\11.1.0\db1`, então adicione `E:\java\jdk\bin`; `E:\oracle_11g\product\11.1.0\db1` em sua variável `PATH` (observe que um ponto-e-vírgula separa os dois diretórios). Para adicionar diretórios na variável `PATH` no Windows XP, você pode usar passos semelhantes àqueles mostrados anteriormente.

Para adicionar uma variável `PATH` existente no Unix ou Linux, modifique o arquivo apropriado de seu shell. Por exemplo, se estiver usando o shell Bash com Linux, adicione linhas semelhantes às seguintes no arquivo `.bash_profile`:

```
PATH=$PATH:$JAVA_HOME/bin:$ORACLE_HOME/BIN
export PATH
```

Note que dois-pontos (:) separam os diretórios.

## Configurando a variável de ambiente CLASSPATH

A variável de ambiente `CLASSPATH` contém uma lista de locais onde os pacotes de classe Java são encontrados. Um local pode ser um nome de diretório ou o nome de um arquivo Zip ou JAR contendo classes. O diretório `ORACLE_HOME\jdbc\lib` contém vários arquivos JAR; os que você adiciona em sua variável `CLASSPATH` dependem do SDK Java que está usando.

Quando este livro estava sendo redigido, o seguinte era correto para configurar uma variável `CLASSPATH`:

- Se estiver usando JDK 1.6 (ou superior), adicione `ORACLE_HOME\jdbc\lib\ojdbc6.jar` em sua variável `CLASSPATH`.
- Se estiver usando JDK 1.5, adicione `ORACLE_HOME\jdbc\lib\ojdbc5.jar` em sua variável `CLASSPATH`.
- Se você precisa de suporte para National Language, adicione `ORACLE_HOME\jlib\orai18n.jar` em sua variável `CLASSPATH`.

- Se você precisa dos recursos JTA e JNDI, adicione `ORACLE_HOME\jlib\jta.jar` e `ORACLE_HOME\jlib\jndi.jar` em sua variável `CLASSPATH`. JNDI é a *Java Naming and Directory Interface*. JTA é a *Java Transaction API*.
- Você também precisa adicionar o diretório atual em sua variável `CLASSPATH`. Para tanto, adicione um ponto-final (.) em sua variável `CLASSPATH`. Desse modo, as classes de seu diretório atual serão encontradas pela linguagem Java quando você executar seus programas.

Quando Java 1.6 é usado e o diretório `ORACLE_HOME` é `E:\oracle_11g\product\11.1.0\db1`, um exemplo de `CLASSPATH` para Windows XP é:

```
.;E:\oracle_11g\product\11.1.0\db1\jdbc\lib\ojdbc6.jar;  
E:\oracle_11g\product\11.1.0\db1\jlib\orai18n.jar
```

Se você está usando Windows XP, utilize os passos descritos anteriormente para criar uma variável de ambiente de sistema chamada `CLASSPATH`. Se estiver usando Linux e Java 1.6, você deve adicionar as linhas a seguir em `.bash_profile`:

```
CLASSPATH=$CLASSPATH:.$ORACLE_HOME/jdbc/lib/ojdbc6.jar:  
$ORACLE_HOME/jlib/orai18n.jar  
export CLASSPATH
```

## Configurando a variável de ambiente `LD_LIBRARY_PATH`

Se estiver usando Unix ou Linux, precisará configurar a variável de ambiente `LD_LIBRARY_PATH` como `$ORACLE_HOME/jdbc/lib`. Esse diretório contém bibliotecas compartilhadas utilizadas pelo driver JDBC OCI. Você adiciona `LD_LIBRARY_PATH` no arquivo apropriado, por exemplo:

```
LD_LIBRARY_PATH=$ORACLE_HOME/jdbc/lib  
export CLASSPATH
```

Isso conclui a configuração de seu computador. Você vai aprender sobre os drivers JDBC da Oracle a seguir.

## OS DRIVERS JDBC DA ORACLE

Nesta seção, você vai aprender sobre os vários drivers JDBC da Oracle. Esses drivers permitem que as instruções JDBC de um programa Java acessem um banco de dados Oracle. Existem quatro drivers JDBC da Oracle:

- Driver Thin
- Driver OCI
- Driver interno server-side
- Driver Thin server-side

As seções a seguir descrevem cada um desses drivers.

### O driver Thin

O driver Thin tem o menor footprint de todos os drivers, significando que ele exige a menor quantidade de recursos do sistema para executar. O driver Thin é escrito inteiramente em Java. Se você estiver escrevendo um applet Java, deve usar o driver Thin. O driver Thin também pode ser usado



em aplicativos Java independentes e para acessar todas as versões do banco de dados Oracle. O driver Thin só funciona com TCP/IP e exige que Oracle Net esteja configurado e em execução. Para ver os detalhes sobre Oracle Net, consulte o *Oracle Database Net Services Administrator's Guide*, publicado pela Oracle Corporation.

**NOTA**

*Você não precisa instalar nada no computador cliente para utilizar o driver Thin e, portanto, pode usá-lo para applets.*

**O driver OCI**

O driver OCI exige mais recursos do que o driver Thin, mas geralmente tem melhor desempenho. O driver OCI é conveniente para programas implantados na camada intermediária (middle-tier) — um servidor de aplicações Web, por exemplo.

**NOTA**

*O driver OCI exige que você o instale no computador cliente e, portanto, não é conveniente para applets.*

O driver OCI tem vários recursos para melhorar o desempenho, incluindo a capacidade de trabalhar com pool de conexões de banco de dados e realizar pré-fetch em linhas do banco de dados. O driver OCI funciona com todas as versões do banco de dados e com todos os protocolos Oracle Net suportados.

**O driver interno server-side**

O driver interno server-side fornece acesso direto ao banco de dados e é utilizado pela JVM Oracle para se comunicar com esse banco de dados. A JVM Oracle é uma máquina virtual Java integrada com o banco de dados. Você pode carregar uma classe Java no banco de dados e, então, publicar e executar os métodos contidos nessa classe usando a JVM Oracle; o código Java é executado no servidor de banco de dados e pode acessar dados de uma única sessão do Oracle.

**O driver Thin server-side**

O driver Thin server-side também é usado pela JVM Oracle e fornece acesso a bancos de dados remotos. Assim como o driver Thin, esse driver também é escrito inteiramente em Java. O código Java que utiliza o driver Thin server-side pode acessar outra sessão no mesmo servidor de banco de dados ou em um servidor remoto.

**IMPORTANDO PACOTES JDBC**

Para que seus programas usem JDBC, você precisa importar os pacotes JDBC necessários para seus programas Java. Existem dois conjuntos de pacotes JDBC:

- Pacotes JDBC padrão da Sun Microsystems
- Pacotes de extensão da Oracle Corporation

Os pacotes JDBC padrão permitem que seus programas Java acessem os recursos básicos da maioria dos bancos de dados, incluindo o Oracle, SQL Server, DB2 e MySQL. As extensões da Oracle para JDBC permitem que seus programas acessem todos os recursos específicos do Oracle, assim como as extensões de desempenho específicas do Oracle. Você vai aprender sobre alguns recursos específicos do Oracle posteriormente neste capítulo.

Para usar JDBC em seus programas, você precisa importar os pacotes `java.sql.*` padrão, como mostrado na instrução `import` a seguir:

```
import java.sql.*;
```

Importar `java.sql.*` importa *todos* os pacotes JDBC padrão. Quando você se tornar mais experiente em JDBC, verá que nem sempre precisa importar todas as classes: é possível importar apenas os pacotes realmente utilizados por seu programa.

## REGISTRANDO OS DRIVERS JDBC DA ORACLE

Antes de poder abrir uma conexão de banco de dados, você primeiro precisa registrar os drivers JDBC da Oracle em seu programa Java. Conforme mencionado anteriormente, os drivers JDBC permitem que suas instruções JDBC acessem o banco de dados.

Existem duas maneiras de registrar os drivers JDBC da Oracle:

- Com o método `forName()` da classe `java.lang.Class`
- Com o método `registerDriver()` da classe `JDBC DriverManager`

O exemplo a seguir ilustra o uso do método `forName()`:

```
Class.forName("oracle.jdbc.OracleDriver");
```

A segunda maneira de registrar os drivers JDBC da Oracle é com o método `registerDriver()` da classe `java.sql.DriverManager`, como mostrado no exemplo a seguir:

```
DriverManager.registerDriver(new oracle.jdbc.OracleDriver());
```

Uma vez que você tenha registrado os drivers JDBC da Oracle, pode abrir uma conexão com um banco de dados.

## ABRINDO UMA CONEXÃO DE BANCO DE DADOS

Antes de poder executar instruções SQL em seus programas Java, você precisa abrir uma conexão de banco de dados. Existem duas maneiras principais de abrir uma conexão de banco de dados:

- Com o método `getConnection()` da classe `DriverManager`
- Com um objeto de origem de dados Oracle, o qual deve primeiro ser criado e depois conectado. Esse método utiliza uma maneira padronizada de configurar detalhes da conexão de banco de dados e um objeto de origem de dados Oracle pode ser usado com JNDI (Java Naming and Directory Interface).

Essas duas maneiras de abrir uma conexão de banco de dados serão descritas nas seções a seguir, começando com o método `getConnection()` da classe `DriverManager`.

### Conectando-se no banco de dados com `getConnection()`

O método `getConnection()` retorna um objeto `JDBC Connection`, o qual deve ser armazenado em seu programa para que possa ser referenciado posteriormente. A sintaxe de uma chamada do método `getConnection()` é:

```
DriverManager.getConnection(URL, nomeusuário, senha);
```

onde

- *URL* é o banco de dados em que seu programa se conecta, junto com o driver JDBC que você deseja usar. (Consulte a seção “A URL do banco de dados” a seguir para ver os detalhes sobre a URL.)
- *nomeusuário* é o nome do usuário do banco de dados com o qual seu programa se conecta.
- *senha* é a senha do nome de usuário.

O exemplo a seguir mostra o método `getConnection()` sendo usado para conectar a um banco de dados:

```
Connection myConnection = DriverManager.getConnection(
    "jdbc:oracle:thin:@localhost:1521:ORCL",
    "store",
    "store_password"
);
```

Nesse exemplo, a conexão é estabelecida com um banco de dados em execução na máquina identificada como `localhost`, com o SID (System Identifier — identificador de sistema) `ORCL`; o driver Thin JDBC Oracle é usado. A conexão é estabelecida com o nome de usuário `store` e a senha `store_password`. O objeto `Connection` retornado pela chamada de `getConnection()` é armazenado em `myConnection`. A conexão com um banco de dados é estabelecida por meio do Oracle Net, que deve estar configurado e funcionando quando essas linhas do programa forem executadas.

## A URL do banco de dados

A URL do banco de dados especifica o local do banco de dados. A estrutura da URL do banco de dados depende do fornecedor dos drivers JDBC. No caso dos drivers JDBC da Oracle, a estrutura da URL do banco de dados é:

```
nome_driver:@informações_driver
```

onde

- *nome\_driver* Nome do driver JDBC da Oracle utilizado pelo seu programa. Ele pode ser configurado como um dos seguintes:
  - **jdbc:oracle:thin** O driver Thin JDBC da Oracle
  - **jdbc:oracle:oci** O driver OCI JDBC da Oracle
- *informações\_driver* As informações específicas do driver necessárias para conectar o banco de dados. Isso depende do driver que está sendo usado. No caso do driver Thin JDBC da Oracle, as informações específicas do driver podem ser especificadas no seguinte formato:
  - **nome\_host:porta:SID\_bancodedados** Onde *nome\_host* é o nome do computador, *porta* é a porta para acessar o banco de dados e *SID\_bancodedados* é o SID do banco de dados

Para todos os drivers JDBC da Oracle, inclusive o driver Thin e os vários drivers OCI, as informações específicas do driver também podem ser especificadas usando pares de palavra-chave e valor do Oracle Net, os quais podem ser especificados no seguinte formato:

```
(description=(address=(host=nome_host) (protocol=tcp) (port=porta))
(connect_data=(sid=SID_bancodedados)))
```

onde

- *nome\_host* é o nome do computador em que o banco de dados está sendo executado.
- *porta* é o número da porta em que o listener do banco de dados espera requisições; 1521 é o número de porta padrão. O administrador do banco de dados pode fornecer o número da porta.
- *SID\_bancodedados* é o SID Oracle da instância do banco de dados em que você deseja se conectar. O administrador do banco de dados pode fornecer o SID.

O exemplo a seguir mostra o método `getConnection()` sendo usado para conectar a um banco de dados utilizando o driver OCI da Oracle, com as informações do driver especificadas com pares de palavra-chave e valor do Oracle Net:

```
Connection myConnection = DriverManager.getConnection(
    "jdbc:oracle:oci:@(description=(address=(host=localhost) " +
    "(protocol=tcp) (port=1521)) (connect_data=(sid=ORCL)))",
    "store",
    "store_password"
);
```

Uma conexão é estabelecida com um banco de dados em execução na máquina identificada como `localhost`, com o SID Oracle `ORCL`, usando o driver OCI da Oracle. A conexão com o banco de dados é estabelecida com o nome de usuário `store` e a senha `store_password`. O objeto `Connection` retornado pela chamada de `getConnection()` é armazenado em `myConnection`.

#### NOTA

Para o driver OCI da Oracle, você também pode usar uma string `TNSNAMES` do Oracle Net. Para mais informações, fale com o administrador do seu banco de dados ou consulte o Oracle Database Net Services Administrator's Guide, publicado pela Oracle Corporation.

## Conectando-se com o banco de dados usando uma origem de dados Oracle

Você também pode usar uma *origem de dados* Oracle para conectar a um banco de dados. Uma origem de dados Oracle usa uma maneira mais padronizada para fornecer os diversos parâmetros para conectar a um banco de dados do que o método anterior, que utilizava `DriverManager.getConnection()`. Além disso, uma origem de dados Oracle também pode ser registrada no JNDI. Usar JNDI com JDBC é muito útil, pois permite que você registre (ou faça um *bind* nas) origens de dados e depois acesse essas origens de dados em seu programa, sem ter de fornecer os detalhes exatos da conexão de banco de dados. Assim, se esses detalhes mudarem, somente o objeto JNDI precisará ser alterado.

#### NOTA

Você pode aprender sobre JNDI no livro *Oracle9i JDBC Programming* (McGraw-Hill/Osborne, 2002).

Três passos devem ser executados para usar uma origem de dados Oracle:

1. Criar um objeto de origem de dados Oracle da classe `oracle.jdbc.pool.OracleDataSource`.
2. Configurar os atributos do objeto de origem de dados Oracle usando os métodos `set`, os quais são definidos na classe.
3. Conectar-se no banco de dados por meio do objeto de origem de dados Oracle, usando o método `getConnection()`.

As seções a seguir descrevem esses três passos.

### **Passo 1: Criar um objeto de origem de dados Oracle**

O primeiro passo é criar um objeto de origem de dados Oracle da classe `oracle.jdbc.pool.OracleDataSource`. O exemplo a seguir cria um objeto `OracleDataSource` chamado `myDataSource` (suponha que a classe `oracle.jdbc.pool.OracleDataSource` foi importada):

```
OracleDataSource myDataSource = new OracleDataSource();
```

Uma vez que você tenha o objeto `OracleDataSource`, o segundo passo é configurar os atributos desse objeto usando os métodos `set`.

### **Passo 2: Configurar os atributos do objeto de origem de dados Oracle**

Antes de poder usar o objeto `OracleDataSource` para conectar a um banco de dados, você deve configurar vários atributos nesse objeto para indicar os detalhes da conexão, utilizando vários métodos `set` definidos na classe. Esses detalhes incluem itens como o nome do banco de dados, o driver JDBC a ser usado etc.; cada um desses detalhes tem um atributo correspondente em um objeto `OracleDataSource`.

A classe `oracle.jdbc.pool.OracleDataSource` implementa a interface `javax.sql.DataSource` fornecida com o JDBC. A interface `javax.sql.DataSource` define vários atributos, os quais estão listados na Tabela 15-1. Essa tabela mostra o nome, a descrição e o tipo de cada atributo.

A classe `oracle.jdbc.pool.OracleDataSource` fornece um conjunto de atributos adicionais, os quais estão listados na Tabela 15-2. Você pode usar vários métodos para ler e gravar cada um dos atributos listados nas tabelas 15-1 e 15-2. Os métodos que lêem os atributos são conhecidos como `get` e os métodos que gravam nos atributos são conhecidos como `set`.

Os nomes de método `set` e `get` são fáceis de lembrar: você pega o nome do atributo, converte a primeira letra em maiúscula e adiciona a palavra “set” ou “get” no início. Por exemplo, para configurar o nome do banco de dados (armazenado no atributo `databaseName`), você usa o método `setDatabaseName()`; para obter o nome do banco de dados atualmente configurado, você usa o método `getDatabaseName()`. Há uma exceção: não existe um método `getPassword()` (isso acontece por motivos de segurança — você não quer que alguém consiga sua senha de forma programática).

A maioria dos atributos é composta de objetos Java `String`, de modo que a maioria dos métodos `set` aceita um único parâmetro `String` e a maioria dos métodos `get` retorna uma `String`. A exceção é o atributo `portNumber`, que é um `int`. Portanto, seu método `set`, `setPortNumber()`, aceita um `int` e seu método `get`, `getPortNumber()`, retorna um `int`.

**Tabela 15-1** *Atributos de DataSource*

Nome do atributo	Descrição do atributo	Tipo do atributo
databaseName	O nome do banco de dados (SID).	String
dataSourceName	O nome da classe de origem de dados subjacente.	String
description	Descrição da origem de dados.	String
networkProtocol	O protocolo de rede a ser usado para se comunicar com o banco de dados. Isso se aplica somente aos drivers OCI do JDBC da Oracle e o padrão é <code>tcp</code> . Para mais detalhes, consulte o <i>Oracle Database Net Services Administrator's Guide</i> , publicado pela Oracle Corporation.	String
password	A senha do nome de usuário fornecido.	String
portNumber	A porta na qual o listener Oracle Net espera requisições de conexão ao banco de dados. O padrão é 1521.	int
serverName	O nome da máquina do servidor de banco de dados (endereço TCP/IP ou apelido DNS).	String
user	O nome de usuário do banco de dados.	String

**Tabela 15-2** *Atributos de OracleDataSource*

Nome do atributo	Descrição do atributo	Tipo do atributo
driverType	O driver JDBC a ser usado. Se você estiver usando o driver interno server-side, ele é configurado como <code>kprb</code> e as outras configurações dos atributos são ignoradas.	String
url	Pode ser usado para especificar uma URL de banco de dados Oracle, a qual pode ser utilizada como uma alternativa para a configuração do local do banco de dados. Para ver os detalhes, consulte a seção anterior sobre URLs de banco de dados.	String
tnsEntryName	Pode ser usado para especificar uma string TNSNAMES do Oracle Net, a qual também pode ser utilizada para especificar o local do banco de dados ao se usar os drivers OCI.	String

Os exemplos a seguir ilustram o uso dos métodos `set` para gravar nos atributos do objeto `myDataSource` de `OracleDataSource` que foi criado anteriormente no passo 1:

```
myDataSource.setServerName("localhost");
myDataSource.setDatabaseName("ORCL");
myDataSource.setDriverType("oci");
myDataSource.setNetworkProtocol("tcp");
myDataSource.setPortNumber(1521);
myDataSource.setUser("scott");
myDataSource.setPassword("tiger");
```

Os exemplos a seguir ilustram o uso de alguns dos métodos `get` para ler os atributos configurados anteriormente em `myDataSource`:

```
String serverName = myDataSource.getServerName();
String databaseName = myDataSource.getDatabaseName();
String driverType = myDataSource.getDriverType();
String networkProtocol = myDataSource.getNetworkProtocol();
int portNumber = myDataSource.getPortNumber();
```

Uma vez configurados os atributos do objeto `OracleDataSource`, você pode utilizá-lo para conectar-se no banco de dados.

### ***Passo 3: Conectar-se no banco de dados por meio do objeto de origem de dados Oracle***

O terceiro passo é conectar-se no banco de dados por meio do objeto `OracleDataSource`. Para tanto, chame o método `getConnection()` de seu objeto `OracleDataSource`. O método `getConnection()` retorna um objeto `JDBC Connection`, o qual deve ser armazenado.

O exemplo a seguir mostra como se chama o método `getConnection()` usando o objeto `myDataSource` preenchido no passo anterior:

```
Connection myConnection = myDataSource.getConnection();
```

O objeto `Connection` retornado por `getConnection()` é armazenado em `myConnection`. Você também pode passar um nome de usuário e uma senha como parâmetros para o método `getConnection()`, como mostrado no exemplo a seguir:

```
Connection myConnection = myDataSource.getConnection(
    "store", "store_password"
);
```

Nesse exemplo, o nome de usuário e a senha irão sobrescrever o nome de usuário e a senha configurados anteriormente em `myDataSource`. Portanto, a conexão com o banco de dados será estabelecida com o nome de usuário `store` e a senha `store_password`, em vez de `scott` e `tiger`, que foram configurados em `myDataSource` na seção anterior.

Uma vez que tenha o objeto `Connection`, você pode utilizá-lo para criar um objeto `JDBC Statement`.

## **CRIANDO UM OBJETO JDBC STATEMENT**

Em seguida, você precisa criar um objeto `JDBC Statement` da classe `java.sql.Statement`. Um objeto `Statement` é usado para representar uma instrução SQL, como uma consulta, uma instrução

DML (INSERT, UPDATE ou DELETE) ou uma instrução DDL (como CREATE TABLE). Você vai aprender a executar consultas, instruções DML e DDL posteriormente neste capítulo.

Para criar um objeto Statement, use o método `createStatement()` de um objeto `Connection`. No exemplo a seguir, um objeto Statement chamado `myStatement` é criado com o método `createStatement()` do objeto `myConnection` criado na seção anterior:

```
Statement myStatement = myConnection.createStatement();
```

O método utilizado na classe Statement para executar a instrução SQL dependerá da instrução SQL que você deseja executar. Se quiser executar uma consulta, use o método `executeQuery()`. Se quiser executar uma instrução INSERT, UPDATE ou DELETE, use o método `executeUpdate()`. Se não sabe antecipadamente qual tipo de instrução SQL vai ser executada, use o método `execute()`, o qual pode ser utilizado para executar qualquer instrução SQL.

Existe outra classe JDBC que pode ser usada para representar uma instrução SQL: a classe `PreparedStatement`. Ela oferece funcionalidades mais avançadas do que a classe `Statement`; vamos discutir a classe `PreparedStatement` depois de explicarmos o uso da classe `Statement`.

Uma vez que você tenha um objeto Statement, está pronto para executar instruções SQL usando JDBC.

## RECUPERANDO LINHAS DO BANCO DE DADOS

Para executar uma consulta usando JDBC, use o método `executeQuery()` do objeto Statement, o qual aceita uma String Java contendo o texto da consulta.

Como uma consulta pode retornar mais de uma linha, o método `executeQuery()` retorna um objeto que armazena a linha (ou linhas) retornada pela consulta. Esse objeto é conhecido como *conjunto de resultados* JDBC e é da classe `java.sql.ResultSet`. Ao usar um objeto `ResultSet` para ler linhas do banco de dados, siga três passos:

1. Crie um objeto `ResultSet` e o preencha com os resultados retornados por uma consulta.
2. Leia os valores de coluna do objeto `ResultSet` usando métodos `get`.
3. Feche o objeto `ResultSet`.

Agora, veja um exemplo que utiliza um objeto `ResultSet` para recuperar as linhas da tabela `customers`.

### Passo 1: Criar e preencher um objeto ResultSet

Primeiro, você cria um objeto `ResultSet` e o preenche com os resultados retornados por uma consulta. O exemplo a seguir cria um objeto `ResultSet` chamado `customerResultSet` e o preenche com as colunas `customer_id`, `first_name`, `last_name`, `dob` e `phone` da tabela `customers`:

```
ResultSet customerResultSet = myStatement.executeQuery(
    "SELECT customer_id, first_name, last_name, dob, phone " +
    "FROM customers"
);
```

Depois que essa instrução for executada, o objeto `ResultSet` conterá os valores de coluna das linhas recuperadas pela consulta. O objeto `ResultSet` poderá, então, ser usado para acessar os valores de coluna das linhas recuperadas. No exemplo, `customerResultSet` conterá as cinco linhas recuperadas da tabela `customers`.



Como o método `execute()` aceita uma `String` Java, você pode construir suas instruções SQL quando seu programa for executado. Isso significa que você pode fazer algumas coisas poderosas em JDBC. Por exemplo, você poderia fazer o usuário de seu programa digitar uma string contendo uma cláusula `WHERE` para uma consulta quando executar o programa ou mesmo digitar a consulta inteira. O exemplo a seguir mostra uma string de cláusula `WHERE`:

```
String whereClause = "WHERE customer_id = 1";
ResultSet customerResultSet2 = myStatement.executeQuery(
    "SELECT customer_id, first_name, last_name, dob, phone " +
    "FROM customers " +
    whereClause
);
```

Você não está limitado às consultas ao usar esse método de construção dinâmica: é possível construir outras instruções SQL de maneira semelhante.

Passo 2: Ler os valores de coluna do objeto `ResultSet`

Para ler os valores de coluna das linhas armazenadas em um objeto `ResultSet`, a classe `ResultSet` fornece uma série de métodos `get`. Antes de entrarmos nos detalhes desses métodos `get`, você precisa entender como os tipos de dados usados para representar valores no Oracle podem ser mapeados em tipos de dados Java compatíveis.

Tipos Oracle e Java

Um programa Java usa um conjunto de tipos diferente dos tipos de banco de dados Oracle para representar valores. Felizmente, os tipos utilizados pelo Oracle são compatíveis com certos tipos Java. Isso permite que a linguagem Java e o Oracle troquem os dados armazenados em seus respectivos tipos. A Tabela 15-3 mostra um conjunto de mapeamentos de tipo compatíveis.

A partir dessa tabela, você pode ver que um `INTEGER` Oracle é compatível com um `int` Java. (Discutiremos outros tipos numéricos mais adiante neste capítulo, na seção “Manipulando números”.) Isso significa que a coluna `customer_id` da tabela `customers` (que é definida como

Tabela 15-3 Mapeamentos de tipo compatíveis

Tipo Oracle	Tipo Java
CHAR	String
VARCHAR2	String
DATE	java.sql.Date java.sql.Time java.sql.Timestamp
INTEGER	short int long
NUMBER	float double java.math.BigDecimal

um `INTEGER` Oracle) pode ser armazenada em uma variável `int` Java. Da mesma forma, os valores de coluna `first_name`, `last_name` e `phone` (`VARCHAR2`) podem ser armazenados em variáveis `String` Java.

O tipo Oracle `DATE` armazena um ano, mês, dia, hora, minuto e segundo. Você pode usar um objeto `java.sql.Date` para armazenar a parte referente à data do valor de coluna `dob` e uma variável `java.sql.Time` para armazenar a parte referente à hora. Você também pode usar um objeto `java.sql.Timestamp` para armazenar as partes referentes à data e à hora da coluna `dob`. Mais adiante, neste capítulo, discutiremos o tipo `oracle.sql.DATE`, uma extensão da Oracle para o padrão JDBC e que fornece uma maneira superior para armazenar datas e horas.

Voltando ao exemplo, as colunas `customer_id`, `first_name`, `last_name`, `dob` e `phone` são recuperadas pela consulta da seção anterior e os exemplos a seguir declaram variáveis e objetos Java compatíveis com essas colunas:

```
int customerId = 0;
String firstName = null;
String lastName = null;
java.sql.Date dob = null;
String phone = null;
```

Os tipos `int` e `String` fazem parte da linguagem Java básica, enquanto `java.sql.Date` faz parte do JDBC e é uma extensão da linguagem Java básica. O JDBC fornece diversos tipos que permitem a troca de dados entre Java e um banco de dados relacional. Entretanto, o JDBC não tem tipos para manipular todos os tipos utilizados pelo Oracle; um exemplo é o tipo `ROWID` — você precisa usar o tipo `oracle.sql.ROWID` para armazenar um `ROWID` Oracle.

Para manipular todos os tipos Oracle, a Oracle fornece vários tipos adicionais, os quais são definidos no pacote `oracle.sql`. Além disso, o Oracle tem vários tipos que podem ser usados como uma alternativa aos tipos Java e JDBC básicos e, em alguns casos, essas alternativas oferecem uma melhor funcionalidade e desempenho do que os tipos Java e JDBC básicos. Discutiremos mais sobre os tipos Oracle definidos no pacote `oracle.sql` posteriormente neste capítulo.

Agora que você entende um pouco sobre tipos Java e Oracle compatíveis, veja como usar os métodos `get` para ler valores de coluna.

### Usando os métodos `get` para ler valores de coluna

Os métodos `get` são usados para ler os valores armazenados em um objeto `ResultSet`. O nome de cada método `get` é simples de entender: pegue o nome do tipo Java no qual você deseja que o valor de coluna seja retornado e adicione a palavra “`get`” no início. Por exemplo, use `getInt()` para ler um valor de coluna como um `int` Java e use `getString()` para ler um valor de coluna como uma `String` Java. Para ler o valor como `java.sql.Date`, você usaria `getDate()`. Cada método `get` aceita um parâmetro: ou um `int` representando a posição da coluna na consulta original ou uma `String` contendo o nome da coluna. Vamos examinar alguns exemplos baseados no exemplo anterior que recuperava as colunas da tabela `customers` no objeto `customerResultSet`.

Para obter o valor da coluna `customer_id`, que foi a primeira coluna especificada na consulta, use `getInt(1)`. Também é possível usar o nome da coluna no método `get`; portanto, você também poderia usar `getInt("customer_id")` para obter o mesmo valor.

#### DICA

Usar o nome da coluna em vez do número da posição da coluna em um método `get` torna seu código mais fácil de ler.

Para obter o valor da coluna `first_name`, que foi a segunda coluna especificada na consulta, use `getString(2)` ou `getString("first_name")`. Você usa chamadas de método semelhantes para obter os valores de coluna `last_name` e `phone`, pois essas colunas também são strings de texto. Para obter o valor da coluna `dob`, você poderia usar `getDate(4)` ou `getDate("dob")`. Para ler os valores armazenados em um objeto `ResultSet`, é necessário chamar os métodos `get` utilizando esse objeto `ResultSet`.

Como um objeto `ResultSet` pode conter mais de uma linha, o JDBC fornece um método chamado `next()` que permite percorrer cada linha armazenada em um objeto `ResultSet`. Você deve chamar o método `next()` para acessar a primeira linha no objeto `ResultSet` e cada chamada sucessiva de `next()` passa para a linha seguinte. Quando não há mais linhas para ler no objeto `ResultSet`, o método `next()` retorna o valor booleano `false`.

Agora vamos voltar ao nosso exemplo: temos um objeto `ResultSet` chamado `customerResultSet` que tem cinco linhas contendo os valores de coluna recuperados das colunas `customer_id`, `first_name`, `last_name`, `dob` e `phone` da tabela `customers`. O exemplo a seguir mostra um loop `while` que lê os valores de coluna de `customerResultSet` nos objetos `customerId`, `firstName`, `lastName`, `dob` e `phone` criados anteriormente:

```
while (customerResultSet.next()) {
    customerId = customerResultSet.getInt("customer_id");
    firstName = customerResultSet.getString("first_name");
    lastName = customerResultSet.getString("last_name");
    dob = customerResultSet.getDate("dob");
    phone = customerResultSet.getString("phone");

    System.out.println("customerId = " + customerId);
    System.out.println("firstName = " + firstName);
    System.out.println("lastName = " + lastName);
    System.out.println("dob = " + dob);
    System.out.println("phone = " + phone);
} // fim do loop while
```

Quando não existem mais linhas para ler de `customerResultSet`, o método `next()` retorna `false` e o loop termina. Você irá observar que o exemplo passa o nome da coluna a ser lida, em vez de posições numéricas, para os métodos `get`. Além disso, copiamos os valores de coluna em variáveis e objetos Java; por exemplo, o valor retornado de `customerResultSet.getInt("customer_id")` é copiado em `customerId`. Não é preciso fazer essa cópia: você poderia simplesmente usar a chamada do método `get` quando precisasse do valor. Entretanto, geralmente é melhor se ele for copiado em uma variável ou em um objeto Java, pois economizará tempo caso esse valor seja utilizado mais de uma vez (o tempo é economizado porque você não precisa chamar o método `get` novamente).

### Passo 3: Fechar o objeto `ResultSet`

Uma vez que você tenha terminado de usar o objeto `ResultSet`, deve fechá-lo utilizando o método `close()`. O exemplo a seguir fecha `customerResultSet`:

```
customerResultSet.close();
```

**NOTA**

*É importante lembrar-se de fechar o objeto `ResultSet` quando você tiver terminado de usá-lo. Isso garante que o objeto seja coletado pelo `Garbage Collector`.*

Agora que você já viu como recuperar linhas, mostraremos como adicionar linhas em uma tabela de banco de dados usando JDBC.

## ADICIONANDO LINHAS NO BANCO DE DADOS

Você usa a instrução SQL `INSERT` para adicionar linhas em uma tabela. Existem duas maneiras principais de executar uma instrução `INSERT` usando JDBC:

- Com o método `executeUpdate()` definido na classe `Statement`.
- Com o método `execute()` definido na classe `PreparedStatement`. (Abordaremos essa classe posteriormente neste capítulo.)

Os exemplos desta seção mostram como adicionar uma linha na tabela `customers`. As colunas `customer_id`, `first_name`, `last_name`, `dob` e `phone` dessa nova linha serão configuradas como 6; Jason; Price; February 22, 1969; e 800-555-1216, respectivamente.

Para adicionar essa nova linha, usaremos o mesmo objeto `Statement` declarado anteriormente (`myStatement`), junto com as mesmas variáveis e objetos utilizados para recuperar as linhas da tabela `customers` na seção anterior. Primeiro, vamos configurar essas variáveis e objetos com os valores com os quais queremos definir nas colunas da tabela `customers` do banco de dados:

```
customerid = 6;
firstName = "Jason";
lastName = "Red";
dob = java.sql.Date.valueOf("1969-02-22");
phone = "800-555-1216";
```

**NOTA**

*A classe `java.sql.Date` armazena datas usando o formato `YYYY-MMDD`, onde `YYYY` é o ano, `MM` é o número do mês e `DD` é o número do dia. Você também pode usar as classes `java.sql.Time` e `java.sql.Timestamp` para representar horas e datas contendo horas, respectivamente.*

Quando você tenta especificar uma data em uma instrução SQL, primeiro a converte para um formato que o banco de dados possa entender, usando a função `TO_DATE()`. `TO_DATE()` aceita uma string contendo uma data, junto com o formato dessa data. Você verá o uso da função `TO_DATE()` em breve, no exemplo de instrução `INSERT`. Mais adiante neste capítulo, discutiremos as extensões para JDBC da Oracle e você verá uma maneira superior de representar datas específicas do Oracle utilizando o tipo `oracle.sql.DATE`.

Você já pode executar uma instrução `INSERT` para adicionar a nova linha na tabela `customers`. O objeto `myStatement` é usado para executar a instrução `INSERT`, configurando os valores

de coluna `customer_id`, `first_name`, `last_name`, `dob` e `phone` iguais aos valores configurados anteriormente nas variáveis `customerId`, `firstName`, `lastName`, `dob` e `phone`.

```
myStatement.executeUpdate(
    "INSERT INTO customers " +
    "(customer_id, first_name, last_name, dob, phone) VALUES (" +
    customerId + ", '" + firstName + "', '" + lastName + "', " +
    "TO_DATE('" + dob + "', 'YYYY, MM, DD'), '" + phone + "')"
);
```

Observe o uso da função `TO_DATE()` para converter o conteúdo do objeto `dob` em uma data aceitável para o banco de dados Oracle. Quando essa instrução tiver terminado, a tabela `customers` conterá a nova linha.

## MODIFICANDO LINHAS NO BANCO DE DADOS

Você usa a instrução SQL `UPDATE` para modificar linhas existentes em uma tabela. Assim como na execução de uma instrução `INSERT` com JDBC, você pode usar o método `executeUpdate()` definido na classe `Statement` ou o método `execute()` definido na classe `PreparedStatement`. O uso da classe `PreparedStatement` será abordado posteriormente neste capítulo. O exemplo a seguir mostra como modificar a linha onde a coluna `customer_id` é igual a 1:

```
first_name = "Jean";
myStatement.executeUpdate(
    "UPDATE customers " +
    "SET first_name = '" + firstName + "' " +
    "WHERE customer_id = 1"
);
```

Após a execução dessa instrução, o nome do cliente nº 1 será definido como "Jean".

## EXCLUINDO LINHAS DO BANCO DE DADOS

Você usa a instrução SQL `DELETE` para excluir linhas existentes de uma tabela. Você pode usar o método `executeUpdate()` definido na classe `Statement` ou o método `execute()` definido na classe `PreparedStatement`. O exemplo a seguir mostra como excluir o cliente nº 5 da tabela `customers`:

```
myStatement.executeUpdate(
    "DELETE FROM customers " +
    "WHERE customer_id = 5"
);
```

Após a execução dessa instrução, a linha do cliente nº 5 terá sido removida da tabela `customers`.

## MANIPULANDO NÚMEROS

Esta seção descreve as questões associadas ao armazenamento de números em seus programas Java. Um banco de dados Oracle é capaz de armazenar números com precisão de até 38 dígitos. No contexto da representação numérica, a precisão se refere à exatidão com a qual um número de ponto flutuante pode ser representado na memória de um computador digital. O nível de precisão de 38 dígitos oferecido pelo banco de dados permite armazenar números muito grandes.

Essa capacidade de precisão é ótima ao se trabalhar com números no banco de dados, mas a linguagem Java utiliza seu próprio conjunto de tipos para representar números. Isso significa que você precisa tomar cuidado quando selecionar o tipo Java que será usado para representar números em seus programas, especialmente se esses números serão armazenados em um banco de dados.

Para armazenar valores inteiros em seu programa Java, você pode usar os tipos `short`, `int`, `long` ou `java.math.BigInteger`, dependendo do tamanho do valor que deseja armazenar. A Tabela 15-4 mostra o número de bits usados para armazenar tipos `short`, `int` e `long`, junto com os valores inferiores e superiores suportados por cada tipo.

Para armazenar números de ponto flutuante em seus programas Java, você pode usar os tipos `float`, `double` ou `java.math.BigDecimal`. A Tabela 15-5 mostra as mesmas colunas da Tabela 15-4 para os tipos `float` e `double`, junto com a precisão suportada por cada um desses tipos.

Como você pode ver, um tipo `float` pode ser usado para armazenar números de ponto flutuante com precisão de até 6 dígitos e um tipo `double` pode ser usado para números de ponto flutuante com precisão de até 15 dígitos. Se você tiver um número de ponto flutuante que exija mais de 15 dígitos de precisão para armazenamento em seu programa Java, pode usar o tipo `java.math.BigDecimal`, que pode armazenar um número de ponto flutuante arbitrariamente longo.

Além desses tipos, existe um tipo de extensão da Oracle para JDBC que você pode usar para armazenar valores inteiros ou números de ponto flutuante. Esse tipo é o `oracle.sql.NUMBER` e ele permite armazenar números com até 38 dígitos de precisão. Você vai aprender mais sobre o tipo `oracle.sql.NUMBER` posteriormente neste capítulo. No Oracle Database 10g e em versões superiores, você pode usar os tipos `oracle.sql.BINARY_FLOAT` e `oracle.sql.BINARY_DOUBLE`. Esses tipos permitem armazenar os números `BINARY_FLOAT` e `BINARY_DOUBLE`.

Vejamos alguns exemplos de uso desses tipos inteiros e de ponto flutuante para armazenar os valores de coluna `product_id` e `price` de uma linha recuperada da tabela `products`. Suponha que um objeto `ResultSet` chamado `productResultSet` foi preenchido com as colunas `product_id` e `price` de uma linha da tabela `products`. A coluna `product_id` é definida como

**Tabela 15-4** Os tipos `short`, `int` e `long`

Tipo	Bits	Valor inferior	Valor superior
<code>short</code>	16	-32768	32767
<code>int</code>	32	-2147483648	2147483647
<code>long</code>	64	-9223372036854775808	9223372036854775807

Tabela 15-5 Os tipos float e double

Tipo	Bits	Valor inferior	Valor superior	Precisão
float	32	-3,4E+38	3,4E+38	6 dígitos
double	64	-1,7E+308	1,7E+308	15 dígitos

um tipo `INTEGER` do banco de dados e a coluna `price` é definida como um tipo `NUMBER` do banco de dados. O exemplo a seguir cria variáveis dos vários tipos inteiros e de ponto flutuante e recupera os valores de coluna `product_id` e `price` nessas variáveis:

```
short productIdShort = productResultSet.getShort("product_id");
int productIdInt = productResultSet.getInt("product_id");
long productIdLong = productResultSet.getLong("product_id");
float priceFloat = productResultSet.getFloat("price");
double priceDouble = productResultSet.getDouble("price");
java.math.BigDecimal priceBigDec = productResultSet.getBigDecimal("price").
```

Observe o uso dos diferentes métodos `get` para recuperar os valores de coluna como os diferentes tipos, cuja saída é então armazenada em uma variável Java de tipo apropriado.

## MANIPULANDO VALORES NULOS NO BANCO DE DADOS

Uma coluna em uma tabela do banco de dados pode ser definida como `NULL` ou `NOT NULL`. `NULL` indica que a coluna pode armazenar um valor `NULL`; `NOT NULL` indica que a coluna não pode conter um valor `NULL`. Um valor `NULL` significa que o valor é desconhecido. Quando uma tabela é criada no banco de dados e você não especifica se uma coluna é `NULL` ou `NOT NULL`, o banco de dados presume que ela deve ser `NULL`.

Os tipos de objeto Java, como `String`, podem ser usados para armazenar valores de banco de dados `NULL`. Quando uma consulta for usada para recuperar uma coluna que contém um valor `NULL` em uma `String` Java, essa `String` conterá um valor `null` Java. Por exemplo, a coluna `phone` (`VARCHAR2`) do cliente nº 5 é `NULL` e a instrução a seguir usa o método `getString()` para ler esse valor em uma `String` chamada `phone`:

```
phone = customerResultSet.getString("phone");
```

Quando a instrução for executada, a `String` Java `phone` conterá o valor `null`.

Esse método é ótimo para valores `NULL` armazenados em objetos Java, mas e quanto aos tipos numéricos, lógicos e de bit em Java? Se você recuperar um valor `NULL` em uma variável numérica, lógica ou de bit em Java — `int`, `float`, `boolean` ou `byte`, por exemplo — essa variável conterá o valor zero. Para o banco de dados, zero e `NULL` são valores diferentes: zero é um valor definido; `NULL` significa que o valor é desconhecido. Isso causa um problema se você deseja diferenciar entre zero e `NULL` em seu programa Java.

Existem duas maneiras de solucionar esse problema:

- Você pode usar o método `wasNull()` no `ResultSet`. O método `wasNull()` retorna `true` se o valor recuperado do banco de dados era `NULL`; caso contrário, o método retorna `false`.
- Você pode usar uma classe wrapper (envoltória) Java. Uma classe wrapper é uma classe Java que permite definir um objeto wrapper, o qual pode então ser usado para armazenar

o valor de coluna retornado do banco de dados. Um objeto wrapper armazena valores de banco de dados `NULL` como valores `null` Java e os valores não `NULL` são armazenados como valores normais.

Vejamos um exemplo que ilustra a primeira técnica, utilizando o produto nº 12 da tabela `products`. Essa linha tem um valor `NULL` na coluna `product_type_id` e essa coluna é definida como o tipo `INTEGER` do banco de dados. Além disso, suponha que um objeto `ResultSet` chamado `productResultSet` foi preenchido com as colunas `product_id` e `product_type_id` do produto nº 12 da tabela `products`. O exemplo a seguir usa o método `wasNull()` para verificar se o valor lido da coluna `product_type_id` era `NULL`:

```
System.out.println("product_type_id = " +
    productResultSet.getInt("product_type_id"));
if (productResultSet.wasNull()) {
    System.out.println("Last value read was NULL");
}
```

Como a coluna `product_type_id` contém um valor `NULL`, `wasNull()` retornaria `true` e, assim, a string "Last value read was NULL" seria exibida.

Antes de vermos um exemplo do segundo método, que utiliza as classes wrapper Java, é preciso entender o que são as classes wrapper. As classes wrapper são definidas no pacote `java.lang`, que possui sete dessas classes definidas:

- `java.lang.Short`
- `java.lang.Integer`
- `java.lang.Long`
- `java.lang.Float`
- `java.lang.Double`
- `java.lang.Boolean`
- `java.lang.Byte`

Os objetos declarados com essas classes wrapper podem ser usados para representar valores de banco de dados `NULL` para os vários tipos de números, assim como para o tipo `Boolean`. Quando um valor de banco de dados `NULL` for recuperado em tal objeto, ele conterá o valor `null` Java. O exemplo a seguir declara um objeto `java.lang.Integer` chamado `productTypeId`:

```
java.lang.Integer productTypeId;
```

Um valor de banco de dados `NULL` pode então ser armazenado em `productTypeId`, usando-se uma chamada para o método `getObject()`, como mostrado no exemplo a seguir:

```
productTypeId =
    (java.lang.Integer) productResultSet.getObject("product_type_id");
```

O método `getObject()` retorna uma instância da classe `java.lang.Object` e deve ser feito um `type cast` para o tipo apropriado; neste caso, para `java.lang.Integer`. Supondo que esse exemplo lê a mesma linha de `productResultSet` do exemplo anterior, `getObject()` retornará um valor `null` em Java e esse valor será copiado em `productTypeId`. Evidentemente, se o valor recuperado do



banco de dados tivesse um valor diferente de `NULL`, `productTypeId` conteria esse valor. Por exemplo, se o valor recuperado do banco de dados fosse 1, `productTypeId` conteria o valor 1.

Você também pode usar um objeto wrapper em uma instrução JDBC que execute uma instrução `INSERT` ou `UPDATE` para configurar uma coluna com um valor normal ou com um valor `NULL`. Se quisesse configurar um valor de coluna como `NULL` usando um objeto wrapper, você configuraria esse objeto wrapper como `null` e o utilizaria em uma instrução `INSERT` ou `UPDATE` para configurar a coluna do banco de dados como `NULL`. O exemplo a seguir configura a coluna `price` do produto nº 12 como `NULL` usando um objeto `java.lang.Double` definido como `null`:

```
java.lang.Double price = null;
myStatement.executeUpdate(
    "UPDATE products " +
    "SET price = " + price + " " +
    "WHERE product_id = 12"
);
```

## CONTROLANDO TRANSAÇÕES DE BANCO DE DADOS

No Capítulo 8, você aprendeu sobre transações de banco de dados e a usar a instrução SQL `COMMIT` para registrar permanentemente as alterações feitas no conteúdo de tabelas. Você também viu como utilizar a instrução `ROLLBACK` para desfazer alterações em uma transação de banco de dados. Os mesmos conceitos se aplicam às instruções SQL executadas utilizando instruções JDBC dentro de seus programas Java.

Por padrão, os resultados das instruções `INSERT`, `UPDATE` e `DELETE` executadas usando JDBC são confirmadas imediatamente. Isso é conhecido como modo de *autocommit*. Geralmente, usar o modo de *autocommit* não é a maneira preferencial para confirmar alterações, pois isso vai contra a idéia de considerar as transações como unidades lógicas de trabalho. Com o modo de *autocommit*, todas as instruções são consideradas transações individuais e essa suposição normalmente é incorreta. Além disso, o modo de *autocommit* pode fazer com que suas instruções SQL demorem mais para terminar, devido ao fato de que cada instrução sempre sofre `commit`.

Felizmente, você pode ativar ou desativar o modo de *autocommit* usando o método `setAutoCommit()` da classe `Connection`, passando a ele um valor booleano `true` ou `false`. O exemplo a seguir desativa o modo de *autocommit* para o objeto `Connection` chamado `myConnection`:

```
myConnection.setAutoCommit(false);
```

### DICA

*Você deve desativar o modo de autocommit. Em geral, isso faz com que seus programas executem mais rapidamente.*

Uma vez desativado o *autocommit*, você pode efetuar `commit` nas suas transações usando o método `commit()` da classe `Connection` ou reverter as alterações usando o método `rollback()`. No exemplo a seguir, o método `commit()` é utilizado para confirmar as alterações feitas no banco de dados usando o objeto `myConnection`:

```
myConnection.commit();
```

No próximo exemplo, o método `rollback()` é usado para reverter as alterações feitas no banco de dados:

```
myConnection.rollback();
```

Se o autocommit foi desativado e você fechar seu objeto `Connection`, um commit implícito é realizado. Portanto, todas as instruções DML que você tiver executado até esse ponto e que ainda não foram confirmadas sofrerão commit automaticamente.

## EXECUTANDO INSTRUÇÕES DATA DEFINITION LANGUAGE

As instruções DDL (Data Definition Language) da linguagem SQL são usadas para criar usuários de banco de dados, tabelas e muitos outros tipos de estruturas que compõem um banco de dados. Instruções DDL consistem em `CREATE`, `ALTER`, `DROP`, `TRUNCATE` e `RENAME`. As instruções DDL podem ser executadas no JDBC com o método `execute()` da classe `Statement`. No exemplo a seguir, a instrução `CREATE TABLE` é usada para criar uma tabela chamada `addresses`, a qual pode ser utilizada para armazenar endereços de clientes:

```
myStatement.execute(
    "CREATE TABLE addresses (" +
    " address_id INTEGER CONSTRAINT addresses_pk PRIMARY KEY," +
    " customer_id INTEGER CONSTRAINT addresses_fk_customers " +
    " REFERENCES customers(customer_id)," +
    " street VARCHAR2(20) NOT NULL," +
    " city VARCHAR2(20) NOT NULL," +
    " state CHAR(2) NOT NULL" +
    ") "
);
```

### NOTA

*A execução de uma instrução DDL resulta na execução implícita de um commit. Portanto, se você tiver executado instruções DML não confirmadas antes de executar uma instrução DDL, essas instruções DML também irão sofrer commit.*

## TRATAMENTO DE EXCEÇÕES

Quando ocorrer um erro no banco de dados ou no driver JDBC, uma exceção `java.sql.SQLException` será lançada. A classe `java.sql.SQLException` é uma subclasse de `java.lang.Exception`. Por isso, você deve colocar todas as suas instruções JDBC dentro de um bloco `try/catch` para que seu código não lance uma exceção `java.sql.SQLException`. Quando essa exceção ocorre, a linguagem Java tenta localizar o handler (tratador) apropriado para processá-la.

Se você incluir um handler para uma exceção `java.sql.SQLException` em uma cláusula `catch`, quando ocorrer um erro no banco de dados ou no driver JDBC, a execução do código Java passará para esse handler e executará o código apropriado incluído nessa cláusula `catch`. No código do handler, você pode, por exemplo, exibir o código de erro e uma mensagem de erro, o que ajudará a determinar o que aconteceu.

A instrução `try/catch` a seguir contém um handler para exceções do tipo `java.sql.SQLException` que podem ocorrer na instrução `try`:

```
try {
    ...
} catch (SQLException e) {
    ...
}
```

**NOTA**

*Se `java.sql.*` foi importada, é possível simplesmente usar `SQLException` na cláusula `catch`, em vez de referenciar `java.sql.SQLException`.*

A instrução `try` conterá as instruções JDBC que podem fazer uma exceção `SQLException` ser lançada e a cláusula `catch` conterá o código de tratamento de erro. A classe `SQLException` define quatro métodos úteis para descobrir o que fez a exceção ocorrer:

- **`getErrorCode()`** No caso de erros que ocorrem no banco de dados ou no driver JDBC, este método retorna o código de erro Oracle, que é um número de cinco dígitos.
- **`getMessage()`** No caso de erros que ocorrem no banco de dados, este método retorna a mensagem de erro, junto com o código de erro Oracle de cinco dígitos. No caso de erros que ocorrem no driver JDBC, este método retorna apenas a mensagem de erro.
- **`getSQLState()`** No caso de erros que ocorrem no banco de dados, este método retorna um código de cinco dígitos contendo o estado da SQL. No caso de erros que ocorrem no driver JDBC, este método não retorna nada de interesse.
- **`printStackTrace()`** Este método exibe o conteúdo da pilha quando a exceção ocorreu. Essa informação pode ajudá-lo a descobrir o que deu errado.

A instrução `try/catch` a seguir ilustra o uso desses quatro métodos:

```
try {
    ...
} catch (SQLException e) {
    System.out.println("Error code = " + e.getErrorCode());
    System.out.println("Error message = " + e.getMessage());
    System.out.println("SQL state = " + e.getSQLState());
    e.printStackTrace();
}
```

Se o seu código lançar uma exceção `SQLException`, em vez de tratar dela de forma local, como acabamos de mostrar, a linguagem Java procurará um handler apropriado na procedure ou função que fez a chamada, até encontrar um. Se nenhum for encontrado, a exceção será tratada pelo handler de exceção padrão, que exibe o código de erro Oracle, a mensagem de erro e o rastreamento da pilha.

## FECHANDO SEUS OBJETOS JDBC

Nos exemplos mostrados neste capítulo, criamos vários objetos JDBC: um objeto `Connection` chamado `myConnection`, um objeto `Statement` chamado `myStatement` e dois objetos `ResultSet` chamados `customerResultSet` e `productResultSet`. Os objetos `ResultSet` devem ser fechados quando não são mais necessários com o método `close()`. Da mesma forma, você também deve fechar os objetos `Statement` e `Connection` quando eles não forem mais necessários.

No exemplo a seguir, os objetos `myStatement` e `myConnection` são fechados com o método `close()`:

```
myStatement.close();
myConnection.close();
```

Normalmente, você deve fechar seus objetos `Statement` e `Connection` em uma cláusula `finally`. Todo código contido em uma cláusula `finally` é executado, independentemente de como o controle saia da instrução `try`. Se você quiser adicionar uma cláusula `finally` para fechar seus objetos `Statement` e `Connection`, esses objetos devem ser declarados antes da primeira instrução `try/catch` utilizada para capturar exceções. O exemplo a seguir mostra como estruturar o método `main()` para que os objetos `Statement` e `Connection` possam ser fechados em uma cláusula `finally`:

```
public static void main (String args []) {
    // declara os objetos Connection e Statement
    Connection myConnection = null;
    Statement myStatement = null;

    try {
        // registra os drivers JDBC da Oracle
        DriverManager.registerDriver(
            new oracle.jdbc.driver.OracleDriver()
        );

        // conecta-se no banco de dados como store
        // usando o driver Thin JDBC da Oracle
        myConnection = DriverManager.getConnection(
            "jdbc:oracle:thin:@localhost:1521:ORCL",
            "store",
            "store_password"
        );

        // cria um objeto Statement
        myStatement = myConnection.createStatement();

        // mais código entra aqui
        ...
    } catch (SQLException e) {
        e.printStackTrace();
    } finally {
        try {
            // fecha o objeto Statement usando o método close()
            if (myStatement != null) {
                myStatement.close();
            }

            // fecha o objeto Connection usando o método close()
            if (myConnection != null) {
                myConnection.close();
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
} // fim de main()
```

Note que o código na cláusula `finally` verifica se os objetos `Statement` e `Connection` não são iguais a `null` antes de fechá-los usando o método `close()`. Se eles são iguais a `null`, não há necessidade de fechá-los. Como o código na cláusula `finally` é o último a ser executado e com certeza é executado, os objetos `Statement` e `Connection` são sempre fechados, independentemente do que aconteça em seu programa. Por brevidade, somente o primeiro programa apresentado neste capítulo utiliza uma cláusula `finally` para fechar os objetos `Statement` e `Connection`.

Agora você já viu como escrever instruções JDBC que conectam a um banco de dados, como executar instruções DML e DDL, como controlar transações, manipular exceções e fechar objetos JDBC. A seção a seguir contém um programa completo que ilustra o uso de JDBC.

## EXEMPLO DE PROGRAMA: BASICEXAMPLE1.JAVA

O programa `BasicExample1.java` ilustra os conceitos abordados até aqui neste capítulo. Esse programa e os outros apresentados neste capítulo podem ser encontrados na pasta `Java` onde você extraiu o arquivo `Zip` deste livro. Todos os programas contêm comentários detalhados que você deve estudar.

```

/*
BasicExample1.java mostra como:
- importar os pacotes JDBC
- carregar os drivers JDBC da Oracle
- conectar-se em um banco de dados
- executar instruções DML
- controlar transações
- usar objetos ResultSet para recuperar linhas
- usar os métodos get
- executar instruções DDL
*/

// importa os pacotes JDBC
import java.sql.*;

public class BasicExample1 {
    public static void main (String args []) {
        // declara os objetos Connection e Statement
        Connection myConnection = null;
        Statement myStatement = null;

        try {
            // registra os drivers JDBC da Oracle
            DriverManager.registerDriver(
                new oracle.jdbc.OracleDriver()
            );

            // EDITE CONFORME FOR NECESSÁRIO PARA CONECTAR A SEU BANCO DE DADOS
            // cria um objeto Connection e conecta-se no banco de dados
            // como o usuário store usando o driver Thin JDBC da Oracle
            myConnection = DriverManager.getConnection(
                "jdbc:oracle:thin:@localhost:1521:ORCL",
                "store",

```

```

    "store_password"
);

// desativa o modo de autocommit
myConnection.setAutoCommit(false);

// cria um objeto Statement
myStatement = myConnection.createStatement();
// cria as variáveis e objetos usados para representar
// valores de coluna
int customerId = 6;
String firstName = "Jason";
String lastName = "Red";
java.sql.Date dob = java.sql.Date.valueOf("1969-02-22");
java.sql.Time dobTime;
java.sql.Timestamp dobTimestamp;
String phone = "800-555-1216";

// executa a instrução SQL INSERT para adicionar uma nova linha na
// tabela customers usando os valores configurados no
// passo anterior - o método executeUpdate() do objeto
// Statement é usado para executar a instrução INSERT
myStatement.executeUpdate(
    "INSERT INTO customers " +
    "(customer_id, first_name, last_name, dob, phone) VALUES (" +
    customerId + ", '" + firstName + "', '" + lastName + "', " +
    "TO_DATE('" + dob + "', 'YYYY, MM, DD'), '" + phone + "')"
);
System.out.println("Added row to customers table");

// executa a instrução SQL UPDATE para modificar a coluna
// first_name do cliente nº 1
firstName = "Jean";
myStatement.executeUpdate(
    "UPDATE customers " +
    "SET first_name = '" + firstName + "' " +
    "WHERE customer_id = 1"
);
System.out.println("Updated row in customers table");

// executa a instrução SQL DELETE para remover o cliente nº 5
myStatement.executeUpdate(
    "DELETE FROM customers " +
    "WHERE customer_id = 5"
);
System.out.println("Deleted row from customers table");

// cria um objeto ResultSet e o preenche com o
// resultado de uma instrução SELECT que recupera as
// colunas customer_id, first_name, last_name, dob e phone

```

```

// de todas as linhas da tabela customers - o
// método executeQuery() do objeto Statement é usado
// para executar a instrução SELECT
ResultSet customerResultSet = myStatement.executeQuery(
    "SELECT customer_id, first_name, last_name, dob, phone " +
    "FROM customers"
);
System.out.println("Retrieved rows from customers table");

// faz loop pelas linhas do objeto ResultSet usando o
// método next() e usa os métodos get para ler os valores
// recuperados das colunas do banco de dados
while (customerResultSet.next()) {
    customerId = customerResultSet.getInt("customer_id");
    firstName = customerResultSet.getString("first_name");
    lastName = customerResultSet.getString("last_name");
    dob = customerResultSet.getDate("dob");
    dobTime = customerResultSet.getTime("dob");
    dobTimestamp = customerResultSet.getTimestamp("dob");
    phone = customerResultSet.getString("phone");

    System.out.println("customerId = " + customerId);
    System.out.println("firstName = " + firstName);
    System.out.println("lastName = " + lastName);
    System.out.println("dob = " + dob);
    System.out.println("dobTime = " + dobTime);
    System.out.println("dobTimestamp = " + dobTimestamp);
    System.out.println("phone = " + phone);
} // fim do loop while

// fecha o objeto ResultSet usando o método close()
customerResultSet.close();

// reverte as alterações feitas no banco de dados
myConnection.rollback();

// cria variáveis numéricas para armazenar as colunas product_id e
// price
short productIdShort;
int productIdInt;
long productIdLong;
float priceFloat;
double priceDouble;
java.math.BigDecimal priceBigDec;

// cria outro objeto ResultSet e recupera as
// colunas product_id, product_type_id e price do produto nº 12
// (essa linha tem um valor NULL na coluna product_type_id)
ResultSet productResultSet = myStatement.executeQuery(
    "SELECT product_id, product_type_id, price " +
    "FROM products " +
    "WHERE product_id = 12"
);
System.out.println("Retrieved row from products table");

```

```

while (productResultSet.next()) {
    System.out.println("product_id = " +
        productResultSet.getInt("product_id"));
    System.out.println("product_type_id = " +
        productResultSet.getInt("product_type_id"));

    // verifica se o valor que acabou de ser lido pelo método get era
    NULL
    if (productResultSet.isNull()) {
        System.out.println("Last value read was NULL");
    }

    // usa o método getObject() para ler o valor e efetua um type cast
    para
    // um objeto wrapper - isso converte um valor NULL do banco de
    dados em um
    // valor null Java
    java.lang.Integer productTypeId =
        (java.lang.Integer) productResultSet.getObject("product_type_id");
    System.out.println("productTypeId = " + productTypeId);

    // recupera os valores de coluna product_id e price nas
    // diversas variáveis numéricas criadas anteriormente
    productIdShort = productResultSet.getShort("product_id");
    productIdInt = productResultSet.getInt("product_id");
    productIdLong = productResultSet.getLong("product_id");
    priceFloat = productResultSet.getFloat("price");
    priceDouble = productResultSet.getDouble("price");
    priceBigDec = productResultSet.getBigDecimal("price");
    System.out.println("productIdShort = " + productIdShort);
    System.out.println("productIdInt = " + productIdInt);
    System.out.println("productIdLong = " + productIdLong);
    System.out.println("priceFloat = " + priceFloat);
    System.out.println("priceDouble = " + priceDouble);
    System.out.println("priceBigDec = " + priceBigDec);
} // fim do loop while

// fecha o objeto ResultSet
productResultSet.close();

// executa a instrução SQL DDL CREATE TABLE para criar uma nova tabela
// que pode ser usada para armazenar endereços de clientes
myStatement.execute(
    "CREATE TABLE addresses (" +
    "  address_id INTEGER CONSTRAINT addresses_pk PRIMARY KEY," +
    "  customer_id INTEGER CONSTRAINT addresses_fk_customers " +
    "    REFERENCES customers(customer_id)," +
    "  street VARCHAR2(20) NOT NULL," +
    "  city VARCHAR2(20) NOT NULL," +
    "  state CHAR(2) NOT NULL" +
    ") "
);
System.out.println("Created addresses table");
// elimina essa tabela usando a instrução SQL DDL DROP TABLE

```



```

        myStatement.execute("DROP TABLE addresses");
        System.out.println("Dropped addresses table");
    } catch (SQLException e) {
        System.out.println("Error code = " + e.getErrorCode());
        System.out.println("Error message = " + e.getMessage());
        System.out.println("SQL state = " + e.getSQLState());
        e.printStackTrace();
    } finally {
        try {
            // fecha o objeto Statement usando o método close()
            if (myStatement != null) {
                myStatement.close();
            }

            // fecha o objeto Connection usando o método close()
            if (myConnection != null) {
                myConnection.close();
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
} // fim de main()
}

```

**NOTA**

*Talvez você precise editar a parte do código que contém a linha rotulada com o texto EDITE CONFORME FOR NECESSÁRIO... com as configurações corretas para acessar seu banco de dados.*

## Compilando BasicExample1

Para compilar BasicExample1.java, digite o comando a seguir usando o prompt de comando de seu sistema operacional:

```
javac BasicExample1.java
```

Se você não tiver configurado a variável de ambiente CLASSPATH corretamente, obterá a seguinte mensagem de erro quando tentar compilar o programa FirstExample.java:

```

FirstExample.java:22: cannot resolve symbol
symbol   : class OracleDriver
location: package jdbc
    new oracle.jdbc.OracleDriver()
                    ^
1 error

```

Você deve verificar a configuração de sua variável de ambiente CLASSPATH — é provável que no CLASSPATH esteja ausente o arquivo de classes Oracle JDBC (ojdbc6.jar, por exemplo). Consulte a seção anterior “Configurando a variável de ambiente CLASSPATH”.

**DICA**

*Você pode digitar javac -help para obter ajuda sobre o compilador Java.*

## Executando BasicExample1

Uma vez que `BasicExample1.java` esteja compilado, você pode executar o arquivo de classe executável resultante (chamado `BasicExample1.class`) digitando o seguinte comando:

```
java BasicExample1
```

### CUIDADO

*A linguagem Java faz diferenciação de letras maiúsculas e minúsculas, portanto, certifique-se de digitar `BasicExample1` com os caracteres B e E maiúsculos.*

Se o programa falhar com o código de erro e a mensagem a seguir, isso significa que o usuário `store` com a senha `store_password` não existe em seu banco de dados:

```
Error code = 1017
Error message = ORA-01017: invalid username/password; logon denied
```

Se esse erro aparecer, verifique se o usuário `store` está no banco de dados.

Talvez o programa também não encontre seu banco de dados, caso em que você obterá o seguinte erro:

```
Error code = 17002
Error message = Io exception: The Network Adapter could not establish the
connection
```

Normalmente, existem dois motivos pelos quais você poderia obter esse erro:

- Não há um banco de dados em execução em sua máquina `localhost` com o SID `ORCL`.
- O Oracle Net não está em execução ou não está recebendo conexões na porta 1521.

Você deve certificar-se de ter a string de conexão correta no programa e também de que o banco de dados e o Oracle Net estejam em execução.

Supondo que o programa funcione, você deverá obter a seguinte saída:

```
Added row to customers table
Updated row in customers table
Deleted row from customers table
Retrieved rows from customers table
customerId = 1
firstName = Jean
lastName = Brown
dob = 1965-01-01
dobTime = 00:00:00
dobTimestamp = 1965-01-01 00:00:00.0
phone = 800-555-1211
customerId = 2
firstName = Cynthia
lastName = Green
dob = 1968-02-05
dobTime = 00:00:00
dobTimestamp = 1968-02-05 00:00:00.0
phone = 800-555-1212
customerId = 3
```

```

firstName = Steve
lastName = White
dob = 1971-03-16
dobTime = 00:00:00
dobTimestamp = 1971-03-16 00:00:00.0
phone = 800-555-1213
customerId = 4
firstName = Gail
lastName = Black
dob = null
dobTime = null
dobTimestamp = null
phone = 800-555-1214
customerId = 6
firstName = Jason
lastName = Red
dob = 1969-02-22
dobTime = 00:00:00
dobTimestamp = 1969-02-22 00:00:00.0
phone = 800-555-1216
Retrieved row from products table
product_id = 12
product_type_id = 0
Last value read was NULL
productTypeId = null
productIdShort = 12
productIdInt = 12
productIdLong = 12
priceFloat = 13.49
priceDouble = 13.49
priceBigDec = 13.49
Created addresses table
Dropped addresses table

```

## SQL PREPARED STATEMENTS

Quando você envia uma instrução SQL para o banco de dados, o software do banco de dados a lê e verifica se está correta. Isso é conhecido como *parsing* da instrução SQL. Então, o software do banco de dados constrói um plano, conhecido como *plano de execução*, para executar de fato a instrução. Até aqui, todas as instruções SQL enviadas para o banco de dados através de JDBC exigiram a construção de um novo plano de execução. Isso porque cada instrução SQL enviada ao banco de dados era diferente.

Suponha que você tivesse um aplicativo Java que estivesse executando a mesma instrução INSERT repetidas vezes; um exemplo é o carregamento de muitos produtos novos em nosso exemplo de loja, um processo que exigiria adicionar muitas linhas na tabela `products` usando instruções INSERT. Vejamos as instruções Java que fariam isso. Suponha que uma classe chamada `Product` fosse definida como segue:

```

class Product {
    int productId;
    int productTypeId;
    String name;

```

```

    String description;
    double price;
}

```

O código a seguir cria um array de cinco objetos de `Product`. Como a tabela `products` já contém linhas com valores de `product_id` de 1 a 12, os atributos `productId` dos novos objetos de `Product` começam em 13:

```

Product [] productArray = new Product[5];
for (int counter = 0; counter < productArray.length; counter++) {
    productArray[counter] = new Product();
    productArray[counter].productId = counter + 13; //começa em 13
    productArray[counter].productTypeId = 1;
    productArray[counter].name = "Test product";
    productArray[counter].description = "Test product";
    productArray[counter].price = 19.95;
} // fim do loop

```

Para adicionar as linhas na tabela `products`, usaremos um loop `for` contendo uma instrução JDBC para executar uma instrução `INSERT` e os valores de coluna virão de `productArray`:

```

Statement myStatement = myConnection.createStatement();
for (int counter = 0; counter < productArray.length; counter++) {
    myStatement.executeUpdate(
        "INSERT INTO products " +
        "(product_id, product_type_id, name, description, price) VALUES (" +
        productArray[counter].productId + ", " +
        productArray[counter].productTypeId + ", '" +
        productArray[counter].name + "', '" +
        productArray[counter].description + "', " +
        productArray[counter].price + ")"
    );
} // fim do loop

```

Cada iteração pelo loop resulta no envio de uma instrução `INSERT` para o banco de dados. Como a string que representa cada instrução `INSERT` contém valores diferentes, a instrução `INSERT` enviada ao banco de dados é ligeiramente diferente a cada vez. Isso significa que o banco de dados cria um plano de execução diferente para cada instrução `INSERT` — isso é muito ineficiente.

O JDBC oferece uma maneira melhor para executar essas instruções SQL. Em vez de usar um objeto JDBC `Statement`, você pode usar um objeto JDBC `PreparedStatement`. Um objeto `PreparedStatement` permite executar a mesma instrução SQL, mas fornece valores diferentes para cada execução dessa instrução. Isso é mais eficiente, pois o mesmo plano de execução é utilizado pelo banco de dados quando a instrução SQL é executada. O exemplo a seguir cria um objeto `PreparedStatement` contendo uma instrução `INSERT` semelhante àquela usada no loop anterior:

```

PreparedStatement myPreparedStatement = myConnection.prepareStatement(
    "INSERT INTO products " +
    "(product_id, product_type_id, name, description, price) VALUES (" +
    "?, ?, ?, ?, ?"
    ") "
);

```

Observe dois aspectos desse exemplo:

- O método `prepareStatement()` é usado para especificar a instrução SQL.
- Caracteres de ponto de interrogação (?) são usados para indicar as posições onde você fornecerá as variáveis a serem usadas quando a instrução SQL for executada.

As posições dos pontos de interrogação são importantes: eles são referenciados de acordo com sua posição, com o primeiro ponto de interrogação sendo referenciado pelo número 1, o segundo pelo número 2 e assim por diante.

O processo de fornecimento de variáveis Java para um prepared statement é conhecido como *bind* (vínculo) das variáveis com a instrução e as variáveis em si são conhecidas como *variáveis de bind*. Para fornecer variáveis para o prepared statement SQL, você deve usar métodos `set`. Esses métodos são semelhantes aos métodos `get` discutidos anteriormente, exceto que os métodos `set` são utilizados para fornecer valores de variável, em vez de lê-los.

Por exemplo, para fazer um bind da variável `int` chamada `intVar` à coluna `product_id` no objeto `PreparedStatement`, você usa `setInt(1, intVar)`. O primeiro parâmetro indica a posição numérica do ponto de interrogação (?) na string especificada anteriormente na chamada do método `prepareStatement()`. Para esse exemplo, o valor 1 corresponde ao primeiro ponto de interrogação, o qual fornece um valor para a coluna `product_id` na instrução `INSERT`. Da mesma forma, para fazer o bind da variável `String` chamada `stringVar` à coluna `name`, você usa `setString(3, stringVar)`, pois o terceiro ponto de interrogação corresponde à coluna `name`. Outros métodos que você pode chamar em um objeto `PreparedStatement` incluem `setFloat()` e `setDouble()`, que são usados para configurar números de ponto flutuante de precisão simples e de precisão dupla.

O exemplo a seguir apresenta um loop que mostra o uso de métodos `set` para fazer o bind dos atributos dos objetos de `Product` em `productArray` ao objeto `PreparedStatement`; observe que o método `execute()` é usado para executar a instrução SQL:

```
for (int counter = 0; counter < productArray.length; counter++) {
    myPreparedStatement.setInt(1, productArray[counter].productId);
    myPreparedStatement.setInt(2, productArray[counter].productTypeId);
    myPreparedStatement.setString(3, productArray[counter].name);
    myPreparedStatement.setString(4, productArray[counter].description);
    myPreparedStatement.setDouble(5, productArray[counter].price);
    myPreparedStatement.execute();
} // fim do loop
```

Depois que esse código for executado, a tabela `products` conterá cinco novas linhas.

Para configurar uma coluna do banco de dados como `NULL` usando um objeto `PreparedStatement`, você pode usar o método `setNull()`. Por exemplo, a instrução a seguir configura a coluna `description` como `NULL`:

```
myPreparedStatement.setNull(4, java.sql.Types.VARCHAR);
```

O primeiro parâmetro na chamada de `setNull()` é a posição numérica da coluna que você deseja configurar como `NULL`. O segundo parâmetro é um valor `int` que corresponde ao tipo da coluna do banco de dados que deve ser configurada como `NULL`. Esse segundo parâmetro deve ser especificado por uma das constantes definidas na classe `java.sql.Types`. Para uma coluna `VARCHAR2` (a coluna `description` é `VARCHAR2`), você deve usar `java.sql.Types.VARCHAR`.

## EXEMPLO DE PROGRAMA: BASICEXAMPLE2.JAVA

O programa `BasicExample2.java` a seguir contém as instruções mostradas nas seções anteriores.

```

/*
   BasicExample2.java mostra como usar instruções preparadas SQL
 */

// importa os pacotes JDBC
import java.sql.*;

class Product {
    int productId;
    int productId;
    String name;
    String description;
    double price;
}

public class BasicExample2 {
    public static void main (String args []) {
        try {
            // registra os drivers JDBC da Oracle
            DriverManager.registerDriver(
                new oracle.jdbc.OracleDriver()
            );

            // EDITE CONFORME FOR NECESSÁRIO PARA CONECTAR EM SEU BANCO DE DADOS
            // cria um objeto Connection e conecta-se no banco de dados
            // como o usuário store usando o driver Thin JDBC da Oracle
            Connection myConnection = DriverManager.getConnection(
                "jdbc:oracle:thin:@localhost:1521:ORCL",
                "store",
                "store_password"
            );

            // desativa modo de auto-commit
            myConnection.setAutoCommit(false);

            Product [] productArray = new Product[5];
            for (int counter = 0; counter < productArray.length; counter++) {
                productArray[counter] = new Product();
                productArray[counter].productId = counter + 13;
                productArray[counter].productId = 1;
                productArray[counter].name = "Test product";
                productArray[counter].description = "Test product";
                productArray[counter].price = 19.95;
            } // fim do loop

            // cria um objeto PreparedStatement

```

```

PreparedStatement myPreparedStatement = myConnection.prepareStatement(
    "INSERT INTO products " +
    "(product_id, product_type_id, name, description, price) VALUES (" +
    "?, ?, ?, ?, ?" +
    ")"
);

// inicializa os valores das novas linhas usando os
// métodos set apropriados
for (int counter = 0; counter < productArray.length; counter++) {
    myPreparedStatement.setInt(1, productArray[counter].productId);
    myPreparedStatement.setInt(2, productArray[counter].productTypeId);
    myPreparedStatement.setString(3, productArray[counter].name);
    myPreparedStatement.setString(4, productArray[counter].description);
    myPreparedStatement.setDouble(5, productArray[counter].price);
    myPreparedStatement.execute();
} // fim do loop

// fecha o objeto PreparedStatement
myPreparedStatement.close();

// recupera as colunas product_id, product_type_id, name, description
// e price dessas novas linhas usando um objeto ResultSet
Statement myStatement = myConnection.createStatement();
ResultSet productResultSet = myStatement.executeQuery(
    "SELECT product_id, product_type_id, " +
    " name, description, price " +
    "FROM products " +
    "WHERE product_id > 12"
);

// exibe os valores de coluna
while (productResultSet.next()) {
    System.out.println("product_id = " +
        productResultSet.getInt("product_id"));
    System.out.println("product_type_id = " +
        productResultSet.getInt("product_type_id"));
    System.out.println("name = " +
        productResultSet.getString("name"));
    System.out.println("description = " +
        productResultSet.getString("description"));
    System.out.println("price = " +
        productResultSet.getDouble("price"));
} // fim do loop while

// fecha o objeto ResultSet usando o método close()
productResultSet.close();

// reverte as alterações feitas no banco de dados
myConnection.rollback();

// fecha os outros objetos JDBC

```

```

        myStatement.close();
        myConnection.close();

    } catch (SQLException e) {
        System.out.println("Error code = " + e.getErrorCode());
        System.out.println("Error message = " + e.getMessage());
        System.out.println("SQL state = " + e.getSQLState());
        e.printStackTrace();
    }
} // fim de main()
}

```

A saída desse programa é a seguinte:

```

product_id = 13
product_type_id = 1
name = Test product
description = Test product
price = 19.95
product_id = 14
product_type_id = 1
name = Test product
description = Test product
price = 19.95
product_id = 15
product_type_id = 1
name = Test product
description = Test product
price = 19.95
product_id = 16
product_type_id = 1
name = Test product
description = Test product
price = 19.95
product_id = 17
product_type_id = 1
name = Test product
description = Test product
price = 19.95

```

## AS EXTENSÕES DA ORACLE PARA JDBC

As extensões da Oracle para JDBC permitem acessar todos os tipos de dados fornecidos pela Oracle, junto com extensões de desempenho específicas. Nesta seção, você vai aprender sobre a manipulação de strings, números, datas e identificadores de linha. Consulte o livro *Oracle9i JDBC Programming* para conhecer todos os tipos e aprimoramentos de desempenho da Oracle.

Dois pacotes de extensão JDBC são fornecidos pela Oracle Corporation:

- **oracle.sql** contém as classes que suportam todos os tipos do banco de dados Oracle
- **oracle.jdbc** contém as interfaces que suportam o acesso a um banco de dados Oracle



Para importar os pacotes JDBC da Oracle em seus programas Java, adicione neles as instruções import a seguir:

```
import oracle.sql.*;
import oracle.jdbc.*;
```

Não é necessário importar todos os pacotes: você poderia importar apenas as classes e interfaces que realmente utiliza em seu programa. Nas seções a seguir, você vai aprender sobre os principais recursos dos pacotes `oracle.sql` e `oracle.jdbc`.

O pacote `oracle.sql`

O pacote `oracle.sql` contém as classes que suportam todos os tipos de banco de dados Oracle. Usar objetos das classes definidas nesse pacote para armazenar valores de banco de dados é mais eficiente do que usar objetos Java normais. Isso porque os valores de banco de dados não precisam ser convertidos primeiro para um tipo Java de base apropriado. Além disso, usar um valor `float` ou `double` Java para representar um número de ponto flutuante pode resultar em perda de precisão para esse número. Se você usar um objeto `oracle.sql.NUMBER`, seus números nunca perderão a precisão.



DICA

*Se você estiver escrevendo um programa que movimenta muitos dados no banco de dados, use as classes `oracle.sql.*`.*

As classes `oracle.sql` estendem a classe `oracle.sql.Datum`, a qual contém a funcionalidade comum a todas as classes. A Tabela 15-6 mostra um subconjunto das classes `oracle.sql`, junto com o mapeamento para os tipos compatíveis no banco de dados Oracle.

A partir da Tabela 15-6, você pode ver que um objeto `oracle.sql.NUMBER` é compatível com um tipo `INTEGER` ou `NUMBER` do banco de dados. Um objeto `oracle.sql.CHAR` é compatível com um tipo `CHAR`, `VARCHAR2`, `NCHAR` e `NVARCHAR2` do banco de dados (`NCHAR` e `NVARCHAR2` são normalmente usados para armazenar caracteres que não são do idioma inglês).

Tabela 15-6 Classes e tipos compatíveis no banco de dados Oracle

Classe	Tipo compatível no banco de dados
oracle.sql.NUMBER	INTEGER
	NUMBER
oracle.sql.CHAR	CHAR
	VARCHAR2
	NCHAR
	NVARCHAR2
oracle.sql.DATE	DATE
oracle.sql.BINARY_FLOAT	BINARY_FLOAT
oracle.sql.BINARY_DOUBLE	BINARY_DOUBLE
oracle.sql.ROWID	ROWID

Os objetos declarados com as classes `oracle.sql` armazenam dados como arrays de bytes — também conhecidos como *formato SQL* — e não formatam novamente os dados recuperados do banco de dados. Isso significa que nenhuma informação é perdida na conversão entre um objeto de formato SQL e um valor armazenado no banco de dados.

Cada objeto `oracle.sql` tem um método `getBytes()` que retorna os dados binários armazenados no objeto. Cada objeto também tem um método `toJdbc()` que retorna os dados binários como um tipo Java compatível (uma exceção é `oracle.sql.ROWID`, onde `toJdbc()` sempre retorna um `oracle.sql.ROWID`).

Cada classe `oracle.sql` também fornece métodos para converter seus dados binários de formato SQL em um tipo Java básico. Por exemplo, `stringValue()` retorna o valor como `String`, `intValue()` retorna um `int`, `floatValue()` retorna um `float`, `doubleValue()` retorna um `double`, `bigDecimalValue()` retorna um `java.math.BigDecimal`, `dateValue()` retorna um `java.sql.Date` etc. Você usa esses métodos quando quer armazenar os dados de formato SQL em um tipo Java básico ou para exibir os dados SQL na tela. Cada classe `oracle.sql` também contém um construtor que aceita como entrada uma variável, objeto ou array de bytes Java.

Conforme você verá posteriormente, a classe `oracle.jdbc.OraclePreparedStatement` contém vários métodos `set` que são utilizados para especificar valores para objetos `oracle.sql`. A classe `OracleResultSet` define vários métodos `get` que você usa para ler valores de objetos `oracle.sql`.

As seções a seguir descrevem as principais classes `oracle.sql`.

### **A classe `oracle.sql.NUMBER`**

A classe `oracle.sql.NUMBER` é compatível com os tipos de banco de dados `INTEGER` e `NUMBER`. A classe `oracle.sql.NUMBER` pode ser usada para representar um número com até 38 dígitos de precisão. O exemplo a seguir cria um objeto `oracle.sql.NUMBER` chamado `customerId`, o qual é configurado com o valor 6 usando o construtor:

```
oracle.sql.NUMBER customerId = new oracle.sql.NUMBER(6);
```

Você pode ler o valor armazenado em `customerId` usando o método `intValue()`, o qual retorna o valor como um `int`. Por exemplo:

```
int customerIdInt = customerId.intValue();
```

Você também pode configurar um objeto `oracle.sql.NUMBER` como um número em ponto flutuante. O exemplo a seguir passa o valor 19,95 para o construtor de um objeto chamado `price`:

```
oracle.sql.NUMBER price = new oracle.sql.NUMBER(19.95);
```

Você pode ler o número de ponto flutuante armazenado em `price` usando os métodos `floatValue()`, `doubleValue()` e `bigDecimalValue()`, os quais retornam um `float`, `double` e `BigDecimal`, respectivamente. Você também pode obter um número em ponto flutuante truncado para um `int`, usando `intValue()` (por exemplo, 19,95 seria retornado como 19). Os exemplos a seguir mostram o uso desses métodos:

```
float priceFloat = price.floatValue();
double priceDouble = price.doubleValue();
java.math.BigDecimal priceBigDec = price.bigDecimalValue();
int priceInt = price.intValue();
```

O método `stringValue()` retorna o valor como uma `String`:

```
String priceString = price.stringValue();
```

### A classe `oracle.sql.CHAR`

A classe `oracle.sql.CHAR` é compatível com os tipos `CHAR`, `VARCHAR2`, `NCHAR` e `NVARCHAR2` do banco de dados. Tanto o banco de dados Oracle como a classe `oracle.sql.CHAR` contêm suporte de globalização para muitos idiomas diferentes. Para saber os detalhes completos dos vários idiomas suportados pelo Oracle, consulte o *Oracle Database Globalization Support Guide*, publicado pela Oracle Corporation.

Quando você recupera dados de caractere do banco de dados em um objeto `oracle.sql.CHAR`, o driver JDBC da Oracle retorna esse objeto usando o conjunto de caracteres do banco de dados (o padrão), o conjunto de caracteres `WE8ISO8859P1` (ISO 8859-1 — Europa Ocidental) ou o conjunto de caracteres `UTF8` (Unicode 3.0 UTF-8 Universal).

Ao se passar um objeto `oracle.sql.CHAR` para o banco de dados, existem restrições sobre o conjunto de caracteres do objeto. O conjunto de caracteres depende do tipo de coluna do banco de dados em que o objeto será armazenado. Se você estiver armazenando o objeto `oracle.sql.CHAR` em uma coluna `CHAR` ou `VARCHAR2`, deverá usar `US7ASCII` (ASCII 7-bit Americano), `WE8ISO8859P1` (ISO 8859-1 Europa Ocidental) ou `UTF8` (Unicode 3.0 UTF-8 Universal). Se estiver armazenando o objeto `oracle.sql.CHAR` em uma coluna `NCHAR` ou `NVARCHAR2`, deverá usar o conjunto de caracteres utilizado pelo banco de dados.

Ao criar seu objeto `oracle.sql.CHAR`, você deve seguir dois passos:

1. Crie um objeto `oracle.sql.CharacterSet` com o conjunto de caracteres que deseja usar.
2. Crie um objeto `oracle.sql.CHAR` através do objeto `oracle.sql.CharacterSet`.

As seções a seguir abordam esses passos.

**Passo 1: Crie um objeto `oracle.sql.CharacterSet`** O exemplo a seguir cria um objeto `oracle.sql.CharacterSet` chamado `myCharSet`:

```
oracle.sql.CharacterSet myCharSet =
    CharacterSet.make(CharacterSet.US7ASCII_CHARSET);
```

O método `make()` aceita um valor `int` que especifica o conjunto de caracteres. No exemplo, a constante `US7ASCII_CHARSET` (definida na classe `oracle.sql.CharacterSet`) especifica o conjunto de caracteres `US7ASCII`. Outros valores `int` incluem `UTF8_CHARSET` (para `UTF8`) e `DEFAULT_CHARSET` (para o conjunto de caracteres usado pelo banco de dados).

**Passo 2: Crie um objeto `oracle.sql.CHAR`** O exemplo a seguir cria um objeto `oracle.sql.CHAR` chamado `firstName`, usando o objeto `myCharSet` criado no passo anterior:

```
oracle.sql.CHAR firstName = new oracle.sql.CHAR("Jason", myCharSet);
```

O objeto `firstName` é preenchido com a string `Jason`. Você pode ler essa string usando o método `stringValue()`, por exemplo:

```
String firstNameString = firstName.stringValue();
System.out.println("firstNameString = " + firstNameString);
```

Isso exibirá `firstNameString = Jason`.

De modo semelhante, o exemplo a seguir cria outro objeto `oracle.sql.CHAR` chamado `lastName`:

```
oracle.sql.CHAR lastName = new oracle.sql.CHAR("Price", myCharSet);
```

Você também pode exibir o valor de um objeto `oracle.sql.CHAR` diretamente, como mostrado no exemplo a seguir:

```
System.out.println("lastName = " + lastName);
```

Essa instrução exibe:

```
lastName = Price
```

### A classe `oracle.sql.DATE`

A classe `oracle.sql.DATE` é compatível com o tipo `DATE` do banco de dados. O exemplo a seguir cria um objeto `oracle.sql.DATE` chamado `dob`:

```
oracle.sql.DATE dob = new oracle.sql.DATE("1969-02-22 13:54:12");
```

Note que o construtor pode aceitar uma string no formato `YYY-MM-DD HH:MI:SS`, onde `YYYY` é o ano, `MM` é o mês, `DD` é o dia, `HH` é a hora, `MI` é o minuto e `SS` é o segundo. Você pode ler o valor armazenado em `dob` como uma String Java usando o método `stringValue()`, como mostrado no exemplo a seguir:

```
String dobString = dob.stringValue();
```

Nesse exemplo, `dobString` conterá `2/22/1969 13:54:12` (o formato muda para `MM/DD/YYYY HH:MI:SS` ao se usar uma String Java).

Você também pode passar um objeto `java.sql.Date` para o construtor `oracle.sql.DATE`, como mostrado no exemplo a seguir:

```
oracle.sql.DATE anotherDob =
    new oracle.sql.DATE(java.sql.Date.valueOf("1969-02-22"));
```

Nesse exemplo, `anotherDob` conterá o valor `oracle.sql.DATE 1969-02-22 00:00:00`.

### A classe `oracle.sql.ROWID`

A classe `oracle.sql.ROWID` é compatível com o tipo `ROWID` do banco de dados. O `ROWID` contém o endereço físico de uma linha no banco de dados. O exemplo a seguir cria um objeto `oracle.sql.ROWID` chamado `rowid`:

```
oracle.sql.ROWID rowid;
```

## O pacote `oracle.jdbc`

As classes e interfaces do pacote `oracle.jdbc` permitem ler e gravar valores de coluna no banco de dados por meio de objetos `oracle.sql`. O pacote `oracle.jdbc` também contém várias melhorias de desempenho. Nesta seção, você vai aprender sobre o conteúdo do pacote `oracle.jdbc` e a criar uma linha na tabela `customers`. Em seguida, vai aprender a ler essa linha usando objetos `oracle.sql`.

### **As classes e interfaces do pacote *oracle.jdbc***

A Tabela 15-7 mostra as classes e interfaces do pacote `oracle.jdbc`.

### **Usando um objeto *OraclePreparedStatement***

A interface `OraclePreparedStatement` implementa `java.sql.PreparedStatement`. Essa interface suporta os vários métodos `set` para efetuar `bind` em objetos `oracle.sql`.

Na seção anterior, você viu os seguintes objetos `oracle.sql`:

- Um objeto `oracle.sql.NUMBER` chamado `customerId`, que foi configurado como 6.
- Um objeto `oracle.sql.CHAR` chamado `firstName`, que foi configurado como Jason.
- Outro objeto `oracle.sql.CHAR` chamado `lastName`, que foi configurado como Price.
- Um objeto `oracle.sql.DATE` chamado `dob`, que foi configurado como 1969-02-22 13:54:12.

Para usar esses objetos em uma instrução DML da linguagem SQL, você precisa utilizar um objeto `OraclePreparedStatement`, o qual contém métodos `set` para manipular objetos `oracle.sql`. O exemplo a seguir cria um objeto `OraclePreparedStatement` chamado `myPreparedStatement`, o qual será usado posteriormente para adicionar uma linha na tabela `customers`:

```
OraclePreparedStatement myPreparedStatement =
    (OraclePreparedStatement) myConnection.prepareStatement (
        "INSERT INTO customers " +
        "(customer_id, first_name, last_name, dob, phone) VALUES (" +
        "?, ?, ?, ?, ?" +
        ") "
    );
```

Note que acontece um `type cast` do objeto `PreparedStatement` retornado pelo método `prepareStatement()` para um objeto `OraclePreparedStatement` e em seguida é armazenado em `myPreparedStatement`.

O próximo passo é efetuar o `bind` dos objetos `oracle.sql` com `myPreparedStatement` usando os métodos `set`. Isso envolve atribuir valores para os lugares reservados, marcados por caracteres de ponto de interrogação (?) em `myPreparedStatement`. Você usa métodos `set`, como `setInt()`, `setFloat()`, `setString()` e `setDate()`, tanto para efetuar o `bind` de variáveis Java a um objeto `PreparedStatement`, como para efetuar o `bind` de objetos `oracle.sql` a um objeto `OraclePreparedStatement` (esses métodos `set` incluem `setNUMBER()`, `setCHAR()`, `setDATE()` etc.).

O exemplo a seguir ilustra o modo de realizar o `bind` dos objetos `customerId`, `firstName`, `lastName` e `dob` a `myPreparedStatement`, usando os métodos `set` apropriados:

```
myPreparedStatement.setNUMBER(1, customerId);
myPreparedStatement.setCHAR(2, firstName);
myPreparedStatement.setCHAR(3, lastName);
myPreparedStatement.setDATE(4, dob);
```

O exemplo a seguir configura o quinto ponto de interrogação (?) em `myPreparedStatement` como `NULL` (o quinto ponto de interrogação (?) corresponde à coluna `phone` na tabela `customers`):

```
myPreparedStatement.setNull(5, OracleTypes.CHAR);
```

**Tabela 15-7** *Classes e interfaces do pacote `oracle.jdbc`*

Nome	Classe ou interface	Descrição
<code>OracleDriver</code>	Classe	Implementa <code>java.sql.Driver</code> . Você insere um objeto dessa classe ao registrar os drivers JDBC Oracle usando o método <code>registerDriver()</code> da classe <code>java.sql.DriverManager</code> .
<code>OracleConnection</code>	Interface	Implementa <code>java.sql.Connection</code> . Essa interface estende a funcionalidade de conexão JDBC padrão para usar objetos <code>OracleStatement</code> . Ela também melhora o desempenho em relação às funções JDBC padrão.
<code>OracleStatement</code>	Interface	Implementa <code>java.sql.Statement</code> e é a superclasse das classes <code>OraclePreparedStatement</code> e <code>OracleCallableStatement</code> .
<code>OraclePreparedStatement</code>	Interface	Implementa <code>java.sql.PreparedStatement</code> e é a superclasse de <code>OracleCallableStatement</code> . Essa interface suporta os vários métodos <code>set</code> para efetuar bind de objetos <code>oracle.sql</code> .
<code>OracleCallableStatement</code>	Interface	Implementa <code>java.sql.CallableStatement</code> . Essa interface contém vários métodos <code>get</code> e <code>set</code> para efetuar bind de objetos <code>oracle.sql</code> .
<code>OracleResultSet</code>	Interface	Implementa <code>java.sql.ResultSet</code> . Essa interface contém vários métodos <code>get</code> para efetuar bind de objetos <code>oracle.sql</code> .
<code>OracleResultSetMetaData</code>	Interface	Implementa <code>java.sql.ResultSetMetaData</code> . Essa interface contém métodos para recuperar metadados sobre conjuntos de resultados Oracle (como nomes de coluna e seus tipos).
<code>OracleDatabaseMetaData</code>	Classe	Implementa <code>java.sql.DatabaseMetaData</code> . Essa classe contém métodos para recuperar metadados sobre o banco de dados Oracle (como a versão do software de banco de dados).
<code>OracleTypes</code>	Classe	Define constantes inteiras para os tipos do banco de dados. Essa classe duplica a classe padrão <code>java.sql.Types</code> e contém constantes inteiras adicionais para todos os tipos Oracle.

A constante `int OracleTypes.CHAR` especifica que o tipo do banco de dados é compatível com o tipo `oracle.sql.CHAR`; `OracleTypes.CHAR` é usado porque a coluna `phone` é `VARCHAR2`. Só resta executar a instrução `INSERT` usando o método `execute()`:

```
myPreparedStatement.execute();
```

Isso adiciona a linha na tabela `customers`.

### Usando um objeto `OracleResultSet`

A interface `OracleResultSet` implementa `java.sql.ResultSet` e contém métodos `get` para manipular objetos `oracle.sql`. Nesta seção, você verá como utilizar um objeto `OracleResultSet` para recuperar a linha adicionada anteriormente na tabela `customers`.

O primeiro item necessário é um objeto `JDBC Statement` através do qual uma instrução SQL possa ser executada:

```
Statement myStatement = myConnection.createStatement();
```

O exemplo a seguir cria um objeto `OracleResultSet` chamado `customerResultSet`, o qual é preenchido com as colunas `ROWID`, `customer_id`, `first_name`, `last_dob` e `phone` recuperadas do cliente nº 6:

```
OracleResultSet customerResultSet =
    (OracleResultSet) myStatement.executeQuery(
        "SELECT ROWID, customer_id, first_name, last_name, dob, phone " +
        "FROM customers " +
        "WHERE customer_id = 6"
    );
```

Definimos anteriormente os objetos `oracle.sql` a seguir: `rowid`, `customerId`, `firstName`, `lastName` e `dob`. Eles podem ser usados para conter os cinco primeiros valores de coluna. Para armazenar o valor da coluna `phone`, é necessário um objeto `oracle.sql.CHAR`:

```
oracle.sql.CHAR phone = new oracle.sql.CHAR("", myCharSet);
```

Um objeto `OracleResultSet` contém métodos `get` que retornam objetos `oracle.sql`. Você usa `getCHAR()` para obter um `oracle.sql.CHAR`, `getNUMBER()` para obter um `oracle.sql.NUMBER`, `getDate()` para obter um `oracle.sql.DATE` etc.

O loop `while` a seguir contém chamadas para os métodos `get` apropriados a fim de copiar os valores de `customerResultSet` em `rowid`, `customerId`, `firstName`, `lastName`, `dob` e `phone`:

```
while (customerResultSet.next()) {
    rowid = customerResultSet.getRowID("ROWID");
    customerId = customerResultSet.getNUMBER("customer_id");
    firstName = customerResultSet.getCHAR("first_name");
    lastName = customerResultSet.getCHAR("last_name");
    dob = customerResultSet.getDate("dob");
    phone = customerResultSet.getCHAR("phone");

    System.out.println("rowid = " + rowid.stringValue());
    System.out.println("customerId = " + customerId.stringValue());
    System.out.println("firstName = " + firstName);
    System.out.println("lastName = " + lastName);
    System.out.println("dob = " + dob.stringValue());
```

```

    System.out.println("phone = " + phone);
} // fim do loop while

```

Para exibir os valores, os exemplos utilizam chamadas para o método `stringValue()`, para converter os objetos `rowid`, `customerId` e `dob` em valores `String` Java. Para os objetos `firstName`, `lastName` e `phone`, o exemplo simplesmente usa esses objetos diretamente na chamadas de `System.out.println()`.

A seção a seguir mostra um programa completo contendo as instruções abordadas nas seções anteriores.

## Exemplo de programa: BasicExample3.java

O programa `BasicExample3.java` a seguir contém as instruções mostradas nas seções anteriores:

```

/*
   BasicExample3.java mostra como se usa as extensões da Oracle para JDBC
   para adicionar uma linha na tabela customers e, depois, recuperar essa linha
*/

// importa os pacotes JDBC
import java.sql.*;

// importa os pacotes de extensão JDBC da Oracle
import oracle.sql.*;
import oracle.jdbc.*;

public class BasicExample3 {
    public static void main (String args []) {
        try {
            // registra os drivers JDBC da Oracle
            DriverManager.registerDriver(
                new oracle.jdbc.OracleDriver()
            );

            // EDITE CONFORME FOR NECESSÁRIO PARA CONECTAR EM SEU BANCO DE DADOS
            // cria um objeto Connection e conecta-se no banco de dados
            // como o usuário store usando o driver Thin JDBC da Oracle
            Connection myConnection = DriverManager.getConnection(
                "jdbc:oracle:thin:@localhost:1521:ORCL",
                "store",
                "store_password"
            );

            // desativa o modo de autocommit
            myConnection.setAutoCommit(false);

            // cria um objeto oracle.sql.NUMBER
            oracle.sql.NUMBER customerId = new oracle.sql.NUMBER(6);
            int customerIdInt = customerId.intValue();
            System.out.println("customerIdInt = " + customerIdInt);

            // cria dois objetos oracle.sql.CHAR

```



```

oracle.sql.CharacterSet myCharSet =
    CharacterSet.make(CharacterSet.US7ASCII_CHARSET);
oracle.sql.CHAR firstName = new oracle.sql.CHAR("Jason", myCharSet);
String firstNameString = firstName.stringValue();
System.out.println("firstNameString = " + firstNameString);
oracle.sql.CHAR lastName = new oracle.sql.CHAR("Price", myCharSet);
System.out.println("lastName = " + lastName);

// cria um objeto oracle.sql.DATE
oracle.sql.DATE dob = new oracle.sql.DATE("1969-02-22 13:54:12");
String dobString = dob.stringValue();
System.out.println("dobString = " + dobString);

// cria um objeto OraclePreparedStatement
OraclePreparedStatement myPrepStatement =
    (OraclePreparedStatement) myConnection.prepareStatement(
        "INSERT INTO customers " +
        "(customer_id, first_name, last_name, dob, phone) VALUES (" +
        "?, ?, ?, ?, ?" +
        ")");

// efetua o bind dos objetos a OraclePreparedStatement usando os
// métodos set apropriados
myPrepStatement.setNUMBER(1, customerId);
myPrepStatement.setCHAR(2, firstName);
myPrepStatement.setCHAR(3, lastName);
myPrepStatement.setDATE(4, dob);

// configura a coluna phone como NULL
myPrepStatement.setNull(5, OracleTypes.CHAR);

// executa PreparedStatement
myPrepStatement.execute();
System.out.println("Added row to customers table");

// recupera as colunas ROWID, customer_id, first_name, last_name, dob e
// phone para essa nova linha, usando um
// objeto OracleResultSet
Statement myStatement = myConnection.createStatement();
OracleResultSet customerResultSet =
    (OracleResultSet) myStatement.executeQuery(
        "SELECT ROWID, customer_id, first_name, last_name, dob, phone " +
        "FROM customers " +
        "WHERE customer_id = 6"
    );
System.out.println("Retrieved row from customers table");

// declara um objeto oracle.sql.ROWID para armazenar o ROWID e
// um objeto oracle.sql.CHAR para armazenar a coluna phone
oracle.sql.ROWID rowid;
oracle.sql.CHAR phone = new oracle.sql.CHAR("", myCharSet);

```

```

// exibe os valores de coluna da linha usando os
// métodos get para ler os valores
while (customerResultSet.next()) {
    rowid = customerResultSet.getRowID("ROWID");
    customerId = customerResultSet.getNUMBER("customer_id");
    firstName = customerResultSet.getCHAR("first_name");
    lastName = customerResultSet.getCHAR("last_name");
    dob = customerResultSet.getDATE("dob");
    phone = customerResultSet.getCHAR("phone");

    System.out.println("rowid = " + rowid.stringValue());
    System.out.println("customerId = " + customerId.stringValue());
    System.out.println("firstName = " + firstName);
    System.out.println("lastName = " + lastName);
    System.out.println("dob = " + dob.stringValue());
    System.out.println("phone = " + phone);
} // fim do loop while

// fecha o objeto OracleResultSet usando o método close()
customerResultSet.close();

// reverte as alterações feitas no banco de dados
myConnection.rollback();

// fecha os outros objetos JDBC
myPreparedStatement.close();
myConnection.close();

} catch (SQLException e) {
    System.out.println("Error code = " + e.getErrorCode());
    System.out.println("Error message = " + e.getMessage());
    System.out.println("SQL state = " + e.getSQLState());
    e.printStackTrace();
}
} // fim de main()
}

```

A saída desse programa é:

```

customeridInt = 6
firstNameString = Jason
lastName = Price
dobString = 2/22/1969 13:54:12
Added row to customers table
Retrieved row from customers table
rowid = AAARk5AAEAAAAGPAAF
customerId = 6
firstName = Jason
lastName = Price
dob = 2/22/1969 13:54:12
phone = null
dobString2 = 2/22/1969 0:0:0

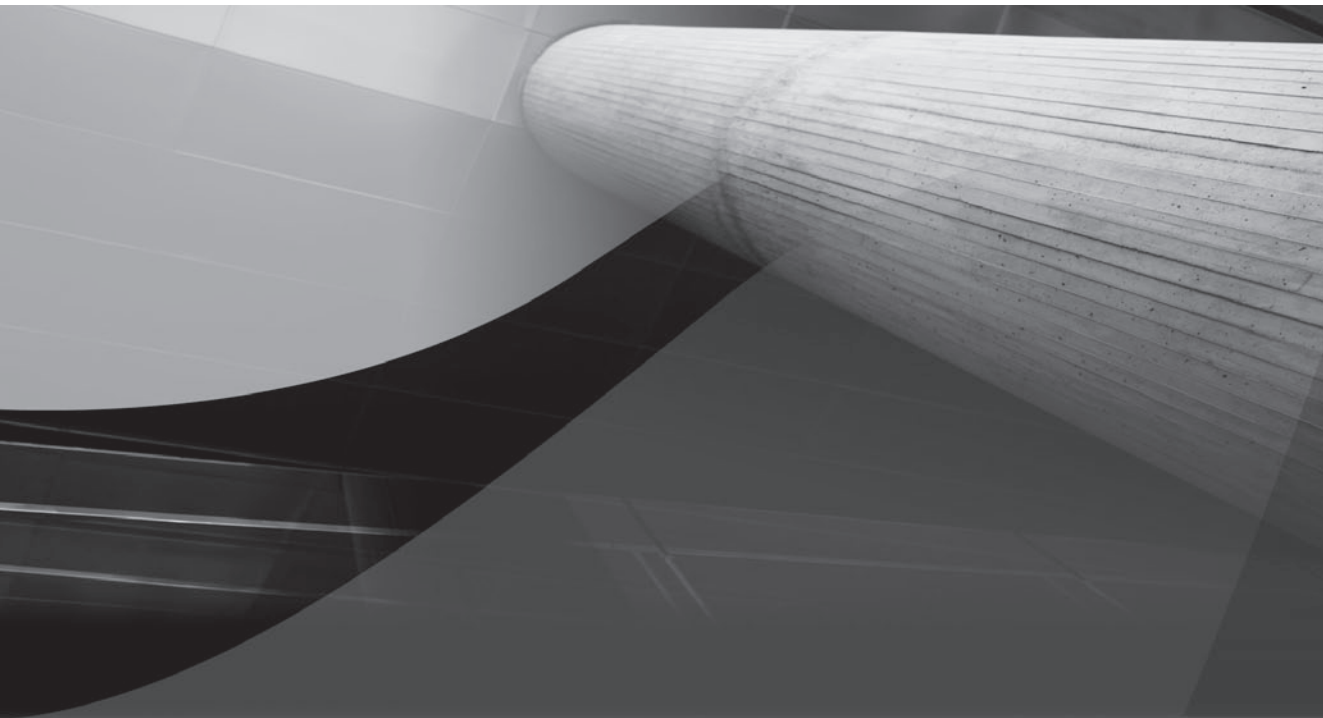
```

## RESUMO

Neste capítulo, você aprendeu que:

- A API JDBC permite que um programa Java acesse um banco de dados.
- Drivers JDBC da Oracle são usados para conectar a um banco de dados Oracle.
- Instruções SQL podem ser executadas usando JDBC.
- A Oracle desenvolveu várias extensões para o JDBC padrão que permitem obter acesso a todos os tipos de banco de dados Oracle.

No próximo capítulo, você vai aprender a ajustar suas instruções SQL para obter o máximo desempenho.



# CAPÍTULO 16

## Ajuste de SQL

Neste capítulo, você vai:

- Aprender sobre o ajuste de SQL
- Conhecer dicas de ajuste de SQL que pode usar para diminuir o tempo de execução de suas consultas
- Aprender sobre o otimizador Oracle
- Aprender a comparar o custo da execução de consultas
- Examinar dicas de otimizador
- Aprender sobre algumas ferramentas de ajuste adicionais

## INTRODUÇÃO AO AJUSTE DE SQL

Uma das principais vantagens da linguagem SQL é que você não precisa dizer ao banco de dados exatamente como ele deve obter os dados solicitados. Basta executar uma consulta especificando as informações desejadas e o software de banco de dados descobre a melhor maneira de obtê-las. Às vezes, você pode melhorar o desempenho de suas instruções SQL “ajustando-as”. Nas seções a seguir, você verá dicas de ajuste que podem fazer suas consultas executarem mais rapidamente e técnicas de ajuste mais avançadas.

## USE UMA CLÁUSULA WHERE PARA FILTRAR LINHAS

Muitos iniciantes recuperam todas as linhas de uma tabela quando só querem uma delas (ou algumas poucas). Isso é muito desperdício. Uma estratégia melhor é adicionar uma cláusula **WHERE** em uma consulta. Desse modo, você restringe as linhas recuperadas apenas àquelas realmente necessárias.

Por exemplo, digamos que você queira os detalhes dos clientes nº 1 e 2. A consulta a seguir recupera todas as linhas da tabela **customers** no esquema **store** (desperdício):

```
-- RUIM (recupera todas as linhas da tabela customers)
SELECT *
FROM customers;
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB	PHONE
1	John	Brown	01-JAN-65	800-555-1211
2	Cynthia	Green	05-FEB-68	800-555-1212
3	Steve	White	16-MAR-71	800-555-1213
4	Gail	Black		800-555-1214
5	Doreen	Blue	20-MAY-70	

A próxima consulta adiciona uma cláusula **WHERE** ao exemplo anterior para obter apenas os clientes nº 1 e 2:

```
-- BOM (usa uma cláusula WHERE para limitar as linhas recuperadas)
SELECT *
FROM customers
WHERE customer_id IN (1, 2);
```

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB	PHONE
1	John	Brown	01-JAN-65	800-555-1211
2	Cynthia	Green	05-FEB-68	800-555-1212

Você deve evitar o uso de funções na cláusula `WHERE`, pois isso aumenta o tempo de execução.

## USE JOINS DE TABELA EM VEZ DE VÁRIAS CONSULTAS

Se você precisa de informações de várias tabelas relacionadas, deve usar condições de join, em vez de várias consultas. No exemplo inadequado a seguir, são usadas duas consultas para obter o nome e o tipo do produto nº 1 (usar duas consultas é desperdício). A primeira consulta obtém os valores de coluna `name` e `product_type_id` da tabela `products` para o produto nº 1. A segunda consulta utiliza esse valor de `product_type_id` para obter a coluna `name` da tabela `product_types`.

```
-- RUIM (duas consultas separadas, quando uma seria suficiente)
```

```
SELECT name, product_type_id
FROM products
WHERE product_id = 1;
```

NAME	PRODUCT_TYPE_ID
Modern Science	1

```
SELECT name
FROM product_types
WHERE product_type_id = 1;
```

NAME
Book

Em vez de usar duas consultas, você deve escrever uma única consulta que utilize um join entre as tabelas `products` e `product_types`. A consulta correta a seguir mostra isso:

```
-- BOM (uma única consulta com um join)
```

```
SELECT p.name, pt.name
FROM products p, product_types pt
WHERE p.product_type_id = pt.product_type_id
AND p.product_id = 1;
```

NAME	NAME
Modern Science	Book

Essa consulta resulta na recuperação do mesmo nome e tipo de produto do primeiro exemplo, mas os resultados são obtidos com uma única consulta. Uma só consulta geralmente é mais eficiente do que duas.

Você deve escolher a ordem de junção em sua consulta de modo a juntar menos linhas nas tabelas posteriormente. Por exemplo, digamos que você estivesse juntando três tabelas relacionadas, chamadas `tab1`, `tab2` e `tab3`. Suponha que `tab1` contenha 1.000 linhas; `tab2`, 100 linhas; e `tab3`, 10 linhas. Você deve juntar primeiro `tab1` com `tab2`, seguido de `tab2` e `tab3`.

Além disso, evite um join de visões complexas em suas consultas, pois isso faz as consultas das visões serem executadas primeiro, seguidas de sua consulta real. Escreva sua consulta usando as tabelas em vez de visões.

## USE REFERÊNCIAS DE COLUNA TOTALMENTE QUALIFICADAS AO FAZER JOINS

Sempre inclua apelidos de tabela em suas consultas e utilize o apelido de cada coluna (isso é conhecido como “qualificar totalmente” suas referências de coluna). Desse modo, o banco de dados não precisará procurar nas tabelas cada coluna utilizada em sua consulta.

O exemplo inadequado a seguir usa os apelidos p e pt para as tabelas products e product\_types respectivamente, mas a consulta não qualifica totalmente as colunas description e price:

```
-- RUIM (as colunas description e price não estão totalmente qualificadas)
SELECT p.name, pt.name, description, price
FROM products p, product_types pt
WHERE p.product_type_id = pt.product_type_id
AND p.product_id = 1;
```

NAME	NAME	
DESCRIPTION		PRICE
Modern Science	Book	
A description of modern science		19.95

Esse exemplo funciona, mas o banco de dados precisa procurar as colunas description e price nas tabelas products e product\_types; isso porque não há um apelido que diga ao banco de dados em qual tabela essas colunas estão. O tempo extra gasto pelo banco de dados para fazer a pesquisa é tempo desperdiçado.

O exemplo correto a seguir inclui o apelido de tabela p para qualificar totalmente as colunas description e price:

```
-- BOM (todas as colunas estão totalmente qualificadas)
SELECT p.name, pt.name, p.description, p.price
FROM products p, product_types pt
WHERE p.product_type_id = pt.product_type_id
AND p.product_id = 1;
```

NAME	NAME	
DESCRIPTION		PRICE
Modern Science	Book	
A description of modern science		19.95

Como todas as referências às colunas incluem um apelido de tabela, o banco de dados não precisa perder tempo procurando-as nas tabelas e o tempo de execução é reduzido.

## USE EXPRESSÕES CASE EM VEZ DE VÁRIAS CONSULTAS

Use expressões CASE, em vez de várias consultas, quando precisar efetuar muitos cálculos nas mesmas linhas em uma tabela. O exemplo inadequado a seguir usa várias consultas para contar o número de produtos dentro de diversos intervalos de preço:

```
-- RUIM (três consultas separadas, quando uma única instrução CASE
funcionaria)
SELECT COUNT(*)
FROM products
WHERE price < 13;

COUNT(*)
-----
2

SELECT COUNT(*)
FROM products
WHERE price BETWEEN 13 AND 15;

COUNT(*)
-----
5

SELECT COUNT(*)
FROM products
WHERE price > 15;

COUNT(*)
-----
5
```

Em vez de usar três consultas, você deve escrever uma única que utilize expressões CASE. Isso está mostrado no exemplo correto a seguir:

```
-- BOM (uma única consulta com uma expressão CASE)
SELECT
  COUNT(CASE WHEN price < 13 THEN 1 ELSE null END) low,
  COUNT(CASE WHEN price BETWEEN 13 AND 15 THEN 1 ELSE null END) med,
  COUNT(CASE WHEN price > 15 THEN 1 ELSE null END) high
FROM products;

      LOW      MED      HIGH
-----
      2        5        5
```

Note que as contagens dos produtos com preços menores do que US\$13 são rotuladas como low, os produtos entre US\$13 e US\$15 são rotulados como med e os produtos maiores do que US\$15 são rotulados como high.

### NOTA

Evidentemente, você pode utilizar intervalos sobrepostos e funções diferentes em suas expressões CASE.



## ADICIONE ÍNDICES NAS TABELAS

Ao procurar um tópico específico em um livro, você pode percorrer o livro inteiro ou utilizar o índice para encontrar o local. Conceitualmente, um índice de uma tabela de banco de dados é semelhante ao índice de um livro, exceto que os índices de banco de dados são usados para encontrar linhas específicas em uma tabela. O inconveniente dos índices é que, quando uma linha é adicionada na tabela, é necessário tempo adicional para atualizar o índice da nova linha.

Geralmente, você deve criar um índice em uma coluna quando está recuperando um pequeno número de linhas de uma tabela que contenha muitas linhas. Uma boa regra geral é:

*Crie um índice quando uma consulta recuperar  $\leq 10\%$  do total de linhas de uma tabela.*

Isso significa que a coluna do índice deve conter uma ampla variedade de valores. Uma boa candidata à indexação seria uma coluna contendo um valor exclusivo para cada linha (por exemplo, um número de CPF). Uma candidata ruim para indexação seria uma coluna que contivesse somente uma pequena variedade de valores (por exemplo, N, S, E, O ou 1, 2, 3, 4, 5, 6). Um banco de dados Oracle cria um índice automaticamente para a chave primária de uma tabela e para as colunas incluídas em uma restrição única.

Além disso, se o seu banco de dados é acessado por muitas consultas hierárquicas (isto é, uma consulta contendo uma cláusula `CONNECT BY`), você deve adicionar índices nas colunas referenciadas nas cláusulas `START WITH` e `CONNECT BY` (consulte o Capítulo 7 para saber mais sobre consultas hierárquicas).

Por fim, para uma coluna que contenha uma pequena variedade de valores e seja usada freqüentemente na cláusula `WHERE` de consultas, você deve considerar a adição de um índice de bitmap nessa coluna. Os índices de bitmap são normalmente usados em ambientes de data warehouse, que são bancos de dados contendo volumes de dados muito grandes. Os dados de um data warehouse normalmente são lidos por muitas consultas, mas não são modificados por muitas transações concorrentes.

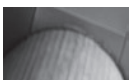
Normalmente, o administrador do banco de dados é responsável pela criação de índices. Entretanto, como desenvolvedor de aplicativos, você poderá fornecer informações para ele sobre quais colunas são boas candidatas à indexação, pois talvez saiba mais sobre o aplicativo do que o DBA. Para relembrar os índices, volte ao Capítulo 10.

## USE WHERE EM VEZ DE HAVING

A cláusula `WHERE` é usada para filtrar linhas; a cláusula `HAVING`, para filtrar grupos de linhas. Como a cláusula `HAVING` filtra grupos de linhas *depois* que elas foram agrupadas (o que leva algum tempo para ser feito), quando possível, você deve primeiro filtrar as linhas usando uma cláusula `WHERE`. Desse modo, você evita o tempo gasto para agrupar as linhas filtradas.

A consulta inadequada a seguir recupera o valor de `product_type_id` e o preço médio dos produtos cujo valor de `product_type_id` é 1 ou 2. Para tanto, a consulta:

- Utiliza a cláusula `GROUP BY` para agrupar as linhas em blocos com o mesmo valor de `product_type_id`
- Utiliza a cláusula `HAVING` para filtrar os resultados retornados nos grupos que têm o valor de `product_type_id` 1 ou 2 (isso é inadequado, pois uma cláusula `WHERE` funcionaria)



```
-- RUIM (usa HAVING em vez de WHERE)
SELECT product_type_id, AVG(price)
FROM products
GROUP BY product_type_id
HAVING product_type_id IN (1, 2);
```

```

PRODUCT_TYPE_ID AVG(PRICE)
-----
                1      24.975
                2      26.22

```

A consulta correta a seguir reescreve o exemplo anterior usando `WHERE`, em vez de `HAVING`, para primeiro filtrar as linhas naquelas cujo valor de `product_type_id` é 1 ou 2:

```

-- BOM (usa WHERE em vez de HAVING)
SELECT product_type_id, AVG(price)
FROM products
WHERE product_type_id IN (1, 2)
GROUP BY product_type_id;

PRODUCT_TYPE_ID AVG(PRICE)
-----
                1      24.975
                2      26.22

```

## USE UNION ALL EM VEZ DE UNION

Você usa `UNION ALL` para obter todas as linhas recuperadas por duas consultas, incluindo as linhas duplicadas; `UNION` é usado para obter todas as linhas não duplicadas recuperadas pelas consultas. Como `UNION` remove as linhas duplicadas (o que leva algum tempo para ser feito), quando possível, você deve usar `UNION ALL`.

A consulta inadequada a seguir usa `UNION` (ruim, porque `UNION ALL` funcionaria) para obter as linhas das tabelas `products` e `more_products`. Observe que todas as linhas não duplicadas de `products` e `more_products` são recuperadas:

```

-- RUIM (usa UNION em vez de UNION ALL)
SELECT product_id, product_type_id, name
FROM products
UNION
SELECT prd_id, prd_type_id, name
FROM more_products;

PRODUCT_ID PRODUCT_TYPE_ID NAME
-----
                1      1 Modern Science
                2      1 Chemistry
                3      2 Supernova
                3      Supernova
                4      2 Lunar Landing
                4      2 Tank War
                5      2 Submarine
                5      2 Z Files
                6      2 2412: The Return
                7      3 Space Force 9
                8      3 From Another Planet
                9      4 Classical Music
               10      4 Pop 3
               11      4 Creative Yell
               12      My Front Line

```

A consulta correta a seguir reescreve o exemplo anterior para usar `UNION ALL`. Observe que todas as linhas de `products` e `more_products` são recuperadas, incluindo as duplicadas:

```
-- BOM (usa UNION ALL em vez de UNION)
SELECT product_id, product_type_id, name
FROM products
UNION ALL
SELECT prd_id, prd_type_id, name
FROM more_products;

PRODUCT_ID PRODUCT_TYPE_ID NAME
-----
1          1 Modern Science
2          1 Chemistry
3          2 Supernova
4          2 Tank War
5          2 Z Files
6          2 2412: The Return
7          3 Space Force 9
8          3 From Another Planet
9          4 Classical Music
10         4 Pop 3
11         4 Creative Yell
12         My Front Line
1          1 Modern Science
2          1 Chemistry
3          Supernova
4          2 Lunar Landing
5          2 Submarine
```

## USE EXISTS EM VEZ DE IN

Você usa `IN` para verificar se um valor está contido em uma lista. `EXISTS` é usado para verificar a existência de linhas retornadas por uma subconsulta. `EXISTS` é diferente de `IN`: `EXISTS` apenas verifica a existência de linhas, enquanto `IN` verifica os valores reais. Normalmente, `EXISTS` oferece melhor desempenho do que `IN` com subconsultas. Portanto, quando possível, use `EXISTS` em vez de `IN`.

Consulte a seção intitulada “Usando `EXISTS` e `NOT EXISTS` em uma subconsulta correlacionada”, no Capítulo 6, para ver os detalhes completos sobre quando usar `EXISTS` com uma subconsulta correlacionada (um ponto importante a lembrar é que as subconsultas correlacionadas podem trabalhar com valores nulos).

A consulta inadequada a seguir usa `IN` (ruim, porque `EXISTS` funcionaria) para recuperar os produtos que foram comprados:

```
-- RUIM (usa IN em vez de EXISTS)
SELECT product_id, name
FROM products
WHERE product_id IN
  (SELECT product_id
   FROM purchases);

PRODUCT_ID NAME
-----
1 Modern Science
```

```

2 Chemistry
3 Supernova

```

A consulta correta a seguir reescreve o exemplo anterior usando EXISTS:

```

-- BOM (usa EXISTS em vez de IN)
SELECT product_id, name
FROM products outer
WHERE EXISTS
  (SELECT 1
   FROM purchases inner
   WHERE inner.product_id = outer.product_id);

PRODUCT_ID NAME
-----
1 Modern Science
2 Chemistry
3 Supernova

```

## USE EXISTS EM VEZ DE DISTINCT

Você pode suprimir a exibição de linhas duplicadas usando DISTINCT. EXISTS é usado para verificar a existência de linhas retornadas por uma subconsulta. Quando possível, use EXISTS em vez de DISTINCT, pois DISTINCT classifica as linhas recuperadas antes de suprimir as linhas duplicadas.

A consulta inadequada a seguir usa DISTINCT (ruim, porque EXISTS funcionaria) para recuperar os produtos que foram comprados:

```

-- RUIM (usa DISTINCT quando EXISTS funcionaria)
SELECT DISTINCT pr.product_id, pr.name
FROM products pr, purchases pu
WHERE pr.product_id = pu.product_id;

PRODUCT_ID NAME
-----
1 Modern Science
2 Chemistry
3 Supernova

```

A consulta correta a seguir reescreve o exemplo anterior usando EXISTS em vez de DISTINCT:

```

-- BOM (usa EXISTS em vez de DISTINCT)
SELECT product_id, name
FROM products outer
WHERE EXISTS
  (SELECT 1
   FROM purchases inner
   WHERE inner.product_id = outer.product_id);

PRODUCT_ID NAME
-----
1 Modern Science
2 Chemistry
3 Supernova

```

## USE GROUPING SETS EM VEZ DE CUBE

Normalmente, a cláusula `GROUPING SETS` oferece melhor desempenho do que `CUBE`. Portanto, quando possível, você deve usar `GROUPING SETS` em vez de `CUBE`. Isso foi abordado detalhadamente na seção intitulada “Usando a cláusula `GROUPING SETS`”, no Capítulo 7.

## USE VARIÁVEIS DE BIND

O software de banco de dados Oracle coloca as instruções SQL em cache; uma instrução SQL colocada no cache é reutilizada se uma instrução idêntica é enviada para o banco de dados. Quando uma instrução SQL é reutilizada, o tempo de execução é reduzido. Entretanto, a instrução SQL deve ser *absolutamente idêntica* para ser reutilizada. Isso significa que:

- Todos os caracteres na instrução SQL devem ser iguais
- Todas as letras na instrução SQL devem ter a mesma caixa
- Todos os espaços na instrução SQL devem ser iguais

Se você precisa fornecer valores de coluna diferentes em uma instrução, pode usar variáveis de bind em vez de valores de coluna literais. Exemplos que esclarecem essas idéias são mostrados a seguir.

### Instruções SQL não idênticas

Nesta seção, você verá algumas instruções SQL não idênticas. As consultas não idênticas a seguir recuperam os produtos nº 1 e 2:

```
SELECT * FROM products WHERE product_id = 1;
SELECT * FROM products WHERE product_id = 2;
```

Essas consultas não são idênticas, pois o valor 1 é usado na primeira instrução, mas o valor 2 é usado na segunda. As consultas não idênticas têm espaços em posições diferentes:

```
SELECT * FROM products WHERE product_id = 1;
SELECT * FROM products WHERE product_id = 1;
```

As consultas não idênticas a seguir usam uma caixa diferente para alguns dos caracteres:

```
select * from products where product_id = 1;
SELECT * FROM products WHERE product_id = 1;
```

Agora que você já viu algumas instruções não idênticas, vejamos instruções SQL idênticas que utilizam variáveis de bind.

### Instruções SQL idênticas que usam variáveis de bind

Você pode garantir que uma instrução seja idêntica utilizando variáveis de bind para representar valores de coluna. Uma variável de bind é criada com o comando `VARIABLE` do SQL\*Plus. Por exemplo, o comando a seguir cria uma variável chamada `v_product_id` de tipo `NUMBER`:

```
VARIABLE v_product_id NUMBER
```

**NOTA**

*Você pode usar os tipos mostrados na Tabela A-1 do apêndice para definir o tipo de uma variável de bind.*

Você referencia uma variável de bind em uma instrução SQL ou PL/SQL usando dois-pontos, seguidos do nome da variável (como em :v\_product\_id). Por exemplo, o bloco PL/SQL a seguir configura v\_product\_id como 1:

```
BEGIN
    :v_product_id:= 1;
END;
/
```

A consulta a seguir usa v\_product\_id para configurar o valor de coluna product\_id na cláusula WHERE; como v\_product\_id foi configurada como 1 no bloco PL/SQL anterior, a consulta recupera os detalhes do produto nº 1:

```
SELECT * FROM products WHERE product_id =:v_product_id;
PRODUCT_ID PRODUCT_TYPE_ID NAME
-----
DESCRIPTION PRICE
-----
          1          1 Modern Science
A description of modern science          19.95
```

O exemplo a seguir configura v\_product\_id como 2 e repete a consulta:

```
BEGIN
    :v_product_id:= 2;
END;
/
SELECT * FROM products WHERE product_id =:v_product_id;

PRODUCT_ID PRODUCT_TYPE_ID NAME
-----
DESCRIPTION PRICE
-----
          2          1 Chemistry
Introduction to Chemistry          30
```

Como a consulta usada neste exemplo é idêntica à consulta anterior, a consulta colocada no cache é reutilizada e há o desempenho melhora.

**DICA**

*Normalmente, você deve usar variáveis de bind se estiver executando a mesma consulta muitas vezes. Além disso, no exemplo, as variáveis de bind são específicas da sessão e precisarão ser reconfiguradas, caso a sessão seja perdida.*

## Listando e imprimindo variáveis de bind

Você lista variáveis de bind no SQL\*Plus usando o comando `VARIABLE`. Por exemplo:

```
VARIABLE
variable  v_product_id
datatype  NUMBER
```

O comando `PRINT` exibe o valor de uma variável de bind no SQL\*Plus. Por exemplo:

```
PRINT v_product_id
V_PRODUCT_ID
-----
                2
```

## Usando uma variável de bind para armazenar um valor retornado por uma função PL/SQL

Você também pode usar uma variável de bind para armazenar valores retornados de uma função PL/SQL. O exemplo a seguir cria uma variável de bind chamada `v_average_product_price` e armazena o resultado retornado pela função `average_product_price()` (essa função foi descrita no Capítulo 11 e calcula o preço médio do produto para o valor de `product_type_id` fornecido):

```
VARIABLE v_average_product_price NUMBER
BEGIN
    :v_average_product_price:= average_product_price(1);
END;
/
PRINT v_average_product_price

V_AVERAGE_PRODUCT_PRICE
-----
                24.975
```

## Usando uma variável de bind para armazenar linhas de um REFCURSOR

Você também pode usar uma variável de bind para armazenar os valores retornados de um `REFCURSOR` (um `REFCURSOR` é um ponteiro para uma lista de linhas). O exemplo a seguir cria uma variável de bind chamada `v_products_refcursor` e armazena o resultado retornado pela função `product_package.get_products_ref_cursor()` (essa função foi apresentada no Capítulo 11; ela retorna um ponteiro para as linhas da tabela `products`):

```
VARIABLE v_products_refcursor REFCURSOR
BEGIN
    :v_products_refcursor:= product_package.get_products_ref_cursor();
END;
/
PRINT v_products_refcursor

PRODUCT_ID NAME                                PRICE
-----
          1 Modern Science                      19.95
          2 Chemistry                           30
```

3	Supernova	25.99
4	Tank War	13.95
5	Z Files	49.99
6	2412: The Return	14.95
7	Space Force 9	13.49
8	From Another Planet	12.99
9	Classical Music	10.99
10	Pop 3	15.99
11	Creative Yell	14.99
PRODUCT_ID NAME		PRICE
-----		-----
12	My Front Line	13.49

## COMPARANDO O CUSTO DA EXECUÇÃO DE CONSULTAS

O software de banco de dados Oracle usa um subsistema conhecido como *otimizador* para gerar o caminho mais eficiente para acessar os dados armazenados nas tabelas. O caminho gerado pelo otimizador é conhecido como *plano de execução*. O Oracle Database 10g e as versões superiores reúnem estatísticas sobre os dados de suas tabelas e índices automaticamente, para gerar o melhor plano de execução (isso é conhecido como *otimização baseada em custo*).

A comparação dos planos de execução gerados pelo otimizador permite a você julgar o custo relativo de uma instrução SQL em relação à outra. É possível usar os resultados para aprimorar suas instruções SQL. Nesta seção, você vai aprender a ver e interpretar dois exemplos de planos de execução.



### NOTA

As versões de banco de dados anteriores ao Oracle Database 10g não reúnem estatísticas automaticamente e, por padrão, o otimizador utiliza a otimização baseada em regra. A otimização baseada em regra usa regras sintáticas para gerar o plano de execução. A otimização baseada em custo normalmente é melhor do que a otimização baseada em regra, pois utiliza as informações reais reunidas dos dados das tabelas e índices. Se estiver usando Oracle Database 9i ou versões inferiores, você mesmo pode reunir estatísticas (você vai aprender a fazer isso na seção “Reunindo estatísticas de tabela”).

## Examinando planos de execução

O otimizador gera um plano de execução para uma instrução SQL. Você pode examinar o plano de execução usando o comando `EXPLAIN PLAN` do SQL\*Plus. O comando `EXPLAIN PLAN` preenche uma tabela chamada `plan_table` com o plano de execução da instrução SQL (`plan_table` é frequentemente referida como “tabela de plano”). Você pode então examinar esse plano de execução consultando a tabela de plano. A primeira coisa que você deve fazer é verificar se a tabela de plano já existe no banco de dados.

### Verificando se a tabela de plano já existe no banco de dados

Para verificar se a tabela de plano já existe no banco de dados, conecte-se no banco de dados como o usuário `store` e execute o seguinte comando `DESCRIBE`:

```
SQL> DESCRIBE plan_table
```

Name	Null?	Type
-----	-----	-----
STATEMENT_ID		VARCHAR2 (30)



PLAN_ID	NUMBER
TIMESTAMP	DATE
REMARKS	VARCHAR2 (4000)
OPERATION	VARCHAR2 (30)
OPTIONS	VARCHAR2 (255)
OBJECT_NODE	VARCHAR2 (128)
OBJECT_OWNER	VARCHAR2 (30)
OBJECT_NAME	VARCHAR2 (30)
OBJECT_ALIAS	VARCHAR2 (65)
OBJECT_INSTANCE	NUMBER (38)
OBJECT_TYPE	VARCHAR2 (30)
OPTIMIZER	VARCHAR2 (255)
SEARCH_COLUMNS	NUMBER
ID	NUMBER (38)
PARENT_ID	NUMBER (38)
DEPTH	NUMBER (38)
POSITION	NUMBER (38)
COST	NUMBER (38)
CARDINALITY	NUMBER (38)
BYTES	NUMBER (38)
OTHER_TAG	VARCHAR2 (255)
PARTITION_START	VARCHAR2 (255)
PARTITION_STOP	VARCHAR2 (255)
PARTITION_ID	NUMBER (38)
OTHER	LONG
OTHER_XML	CLOB
DISTRIBUTION	VARCHAR2 (30)
CPU_COST	NUMBER (38)
IO_COST	NUMBER (38)
TEMP_SPACE	NUMBER (38)
ACCESS_PREDICATES	VARCHAR2 (4000)
FILTER_PREDICATES	VARCHAR2 (4000)
PROJECTION	VARCHAR2 (4000)
TIME	NUMBER (38)
QBLOCK_NAME	VARCHAR2 (30)

Se obtiver uma descrição de tabela semelhante a esses resultados, você já tem a tabela de plano. Se obtiver um erro, então precisa criar a tabela de plano.

### ***Criando a tabela de plano***

Se você não tem a tabela de plano, deve criá-la. Para fazer isso, execute o script SQL\*Plus `utlxplan.sql` (em meu computador Windows, o script está localizado no diretório `E:\oracle_11g\product\11.1.0\db_1\RDBMS\ADMIN`). O exemplo a seguir mostra o comando para executar o script `utlxplan.sql`:

```
SQL> @ E:\oracle_11g\product\11.1.0\db_1\RDBMS\ADMIN\utlxplan.sql
```

### **NOTA**

*Você precisará substituir o caminho de diretório pelo caminho de seu ambiente.*

As colunas mais importantes na tabela de plano estão mostradas na Tabela 16-1.

**Criando uma tabela de plano central**

Se necessário, o administrador de banco de dados pode criar uma única tabela de plano central. Desse modo, os usuários individuais não precisam criar suas próprias tabelas de planos. Para fazer isso, o DBA executa os passos a seguir:

- 1. Cria a tabela de plano em um esquema de sua escolha, executando o script utlxplan.sql
- 2. Cria um sinônimo público para a tabela de plano
- 3. Concede acesso à tabela de plano para a atribuição PUBLIC

Eis um exemplo desses passos:

```
@ E:\oracle_11g\product\11.1.0\db_1\RDBMS\ADMIN\utlxplan.sql
CREATE PUBLIC SYNONYM plan_table FOR plan_table;
GRANT SELECT, INSERT, UPDATE, DELETE ON plan_table TO PUBLIC;
```

**Tabela 16-1** Colunas da tabela de plano

Coluna	Descrição
statement_id	Nome atribuído ao plano de execução.
operation	Operação de banco de dados executada, que pode ser: <ul style="list-style-type: none"><li>■ Percorrer uma tabela</li><li>■ Percorrer um índice</li><li>■ Acessar linhas de uma tabela usando um índice</li><li>■ Juntar duas tabelas</li><li>■ Classificar um conjunto de linhas</li></ul> Por exemplo, a operação para acessar uma tabela é TABLE ACCESS.
options	Nome da opção usada na operação. Por exemplo, a opção para uma varredura integral é FULL.
object_name	Nome do objeto de banco de dados referenciado na operação.
object_type	Atributo do objeto. Por exemplo, um índice exclusivo tem o atributo UNIQUE.
id	Número atribuído a essa operação no plano de execução.
parent_id	Número pai do passo atual no plano de execução. O valor de parent_id se relaciona a um valor de id de um passo pai.
position	Ordem de processamento dos passos que têm o mesmo valor de parent_id.
cost	Estimativa das unidades de trabalho da operação. A otimização baseada em custo usa E/S de disco, utilização de CPU e utilização da memória como unidades de trabalho. Portanto, o custo é uma estimativa do número de E/Ss de disco e da quantidade de CPU e memória utilizada para executar uma operação.

### Gerando um plano de execução

Uma vez que você tenha uma tabela de plano, pode usar o comando `EXPLAIN PLAN` para gerar um plano de execução para uma instrução SQL. A sintaxe do comando `EXPLAIN PLAN` é:

```
EXPLAIN PLAN SET STATEMENT_ID = id_instrução FOR instrução_sql;
```

onde

- *id\_instrução* é o nome que você deseja dar ao plano de execução. Pode ser qualquer texto alfanumérico.
- *instrução\_sql* é a instrução SQL para a qual você deseja gerar um plano de execução.

O exemplo a seguir gera o plano de execução para uma consulta que recupera todas as linhas da tabela `customers` (observe que o valor de *id\_instrução* é configurado como `'CUSTOMERS'`):

```
EXPLAIN PLAN SET STATEMENT_ID = 'CUSTOMERS' FOR
SELECT customer_id, first_name, last_name FROM customers;
Explained
```

Depois que o comando terminar, você pode examinar o plano de execução armazenado na tabela de plano. Você vai aprender a fazer isso a seguir.

#### NOTA

A consulta na instrução `EXPLAIN PLAN` não retorna linhas da tabela `customers`. A instrução `EXPLAIN PLAN` simplesmente gera o plano de execução que seria usado se a consulta fosse executada.

### Consultando a tabela de plano

Para consultar a tabela de plano, fornecemos um script SQL\*Plus chamado `explain_plan.sql` no diretório SQL. O script solicita o valor de `statement_id` (*id\_instrução*) e depois exibe o plano de execução para essa instrução.

O script `explain_plan.sql` contém as seguintes instruções:

```
-- Exibe o plano de execução da statement_id especificada

UNDEFINE v_statement_id;

SELECT
  id ||
  DECODE(id, 0, '', LPAD(' ', 2*(level - 1))) || ' ' ||
  operation || ' ' ||
  options || ' ' ||
  object_name || ' ' ||
  object_type || ' ' ||
  DECODE(cost, NULL, '', 'Cost = ' || position)
AS execution_plan
FROM plan_table
CONNECT BY PRIOR id = parent_id
AND statement_id = '&&v_statement_id'
START WITH id = 0
AND statement_id = '&v_statement_id';
```

Um plano de execução é organizado em uma hierarquia de operações de banco de dados semelhante a uma árvore; os detalhes dessas operações são armazenados na tabela de plano. A operação com o valor de `id` igual a 0 é a raiz da hierarquia e todas as outras operações do plano procedem dessa raiz. A consulta do script recupera os detalhes das operações, começando com a operação raiz e, então, percorre a árvore a partir da raiz.

O exemplo a seguir mostra como executar o script `explain_plan.sql` para recuperar o plano 'CUSTOMERS' criado anteriormente:

```
SQL> @ c:\sql_book\sql\explain_plan.sql
Enter value for v_statement_id: CUSTOMERS
old 12: statement_id = '&&v_statement_id'
new 12: statement_id = 'CUSTOMERS'
old 14: statement_id = '&v_statement_id'
new 14: statement_id = 'CUSTOMERS'

EXECUTION_PLAN
-----
0 SELECT STATEMENT      Cost = 3
1  TABLE ACCESS FULL CUSTOMERS TABLE Cost = 1
```

As operações mostradas na coluna `EXECUTION_PLAN` são executadas na seguinte ordem:

- A operação recuada mais à direita é executada primeiro, seguida de todas as operações pai que estão acima dela.
- Para operações com o mesmo recuo, a operação mais acima é executada primeiro, seguida de todas as operações pai que estão acima dela.

Cada operação envia seus resultados de volta no encadeamento até sua operação pai imediata e, então, a operação pai é executada. Na coluna `EXECUTION_PLAN`, a `ID` da operação é mostrada na extremidade esquerda. No exemplo de plano de execução, a operação 1 é executada primeiro, com seus resultados sendo passados para a operação 0. O exemplo a seguir ilustra a ordem para um exemplo mais complexo:

```
0 SELECT STATEMENT Cost = 6
1  MERGE JOIN Cost = 1
2    TABLE ACCESS BY INDEX ROWID PRODUCT_TYPES TABLE Cost = 1
3      INDEX FULL SCAN PRODUCT_TYPES_PK INDEX (UNIQUE) Cost = 1
4    SORT JOIN Cost = 2
5      TABLE ACCESS FULL PRODUCTS TABLE Cost = 1
```

A ordem em que as operações são executadas nesse exemplo é 3, 2, 5, 4, 1 e 0.

Agora que você já conhece a ordem na qual as operações são executadas, é hora de aprender para o que elas fazem realmente. O plano de execução da consulta 'CUSTOMERS' era:

```
0 SELECT STATEMENT Cost = 3
1  TABLE ACCESS FULL CUSTOMERS TABLE Cost = 1
```

A operação 1 é executada primeiro, com seus resultados sendo passados para a operação 0. A operação 1 envolve uma varredura integral — indicada pela string `TABLE ACCESS FULL` — da tabela `customers`. Este é o comando original usado para gerar a consulta 'CUSTOMERS':

```
EXPLAIN PLAN SET STATEMENT_ID = 'CUSTOMERS' FOR
SELECT customer_id, first_name, last_name FROM customers;
```

Uma varredura integral da tabela é realizada porque a instrução `SELECT` especifica que todas as linhas da tabela `customers` devem ser recuperadas.

O custo total da consulta é de três unidades de trabalho, conforme indicado na parte referente ao custo mostrada à direita da operação 0 no plano de execução (0 `SELECT STATEMENT Cost = 3`). Uma unidade de trabalho é a quantidade de processamento que o software precisa para realizar determinada operação. Quanto mais alto o custo, mais trabalho o software do banco de dados precisa realizar para concluir a instrução SQL.

### NOTA

*Se você estiver usando uma versão do banco de dados anterior ao Oracle Database 10g, a saída do custo da instrução global poderá estar em branco. Isso ocorre porque as versões de banco de dados anteriores não reúnem estatísticas de tabela automaticamente. Para reunir estatísticas, você precisa usar o comando `ANALYZE`. Você vai aprender a fazer isso na seção “Reunindo estatísticas de tabela”.*

### Planos de execução envolvendo joins de tabela

Os planos de execução para consultas com joins de tabelas são mais complexos. O exemplo a seguir gera o plano de execução de uma consulta que junta as tabelas `products` e `product_types`:

```
EXPLAIN PLAN SET STATEMENT_ID = 'PRODUCTS' FOR
SELECT p.name, pt.name
FROM products p, product_types pt
WHERE p.product_type_id = pt.product_type_id;
```

O plano de execução dessa consulta está mostrado no exemplo a seguir:

```
@ c:\sql_book\sql\explain_plan.sql
Enter value for v_statement_id: PRODUCTS

EXECUTION_PLAN
-----
0 SELECT STATEMENT Cost = 6
1 MERGE JOIN Cost = 1
2 TABLE ACCESS BY INDEX ROWID PRODUCT_TYPES TABLE Cost = 1
3 INDEX FULL SCAN PRODUCT_TYPES_PK INDEX (UNIQUE) Cost = 1
4 SORT JOIN Cost = 2
5 TABLE ACCESS FULL PRODUCTS TABLE Cost = 1
```

### NOTA

*Se você executar o exemplo, talvez obtenha um plano de execução ligeiramente diferente, dependendo da versão do banco de dados que esteja usando e das configurações dos parâmetros no arquivo de configuração `init.ora` do banco de dados.*

O plano de execução anterior é mais complexo e você pode ver as relações hierárquicas entre as diversas operações. A ordem de execução das operações é 3, 2, 5, 4, 1 e 0. A Tabela 16-2 descreve cada operação na ordem em que elas são executadas.

Tabela 16-2 Operação do plano de execução

ID da operação	Descrição
3	Varredura integral do índice <code>product_types_pk</code> (que é um índice exclusivo) para obter os endereços das linhas na tabela <code>product_types</code> . Os endereços estão na forma de valores de ROWID, os quais são passados para a operação 2.
2	Acesso às linhas da tabela <code>product_types</code> usando a lista de valores de ROWID passada da operação 3. As linhas são passadas para a operação 1.
5	Acesso às linhas da tabela <code>products</code> . As linhas são passadas para a operação 4.
4	Classificação das linhas passadas da operação 5. As linhas classificadas são passadas para a operação 1.
1	Mesclagem das linhas passadas das operações 2 e 5. As linhas mescladas são passadas para a operação 0.
0	Retorno das linhas da operação 1 para o usuário. O custo total da consulta é de 6 unidades de trabalho.

Reunindo estatísticas de tabela

Se estiver usando uma versão do banco de dados anterior ao Oracle Database 10g (como a 9i), você mesmo terá de reunir estatísticas de tabela usando o comando `ANALYZE`. Por padrão, se nenhuma estatística estiver disponível, a otimização baseada em regra será utilizada. Normalmente, a otimização baseada em regra não é tão boa quanto a otimização baseada em custo. Os exemplos a seguir usam o comando `ANALYZE` para reunir estatísticas para as tabelas `products` e `product_types`:

```
ANALYZE TABLE products COMPUTE STATISTICS;
ANALYZE TABLE product_types COMPUTE STATISTICS;
```

Uma vez reunidas as estatísticas, a otimização baseada em custo será usada em vez da otimização baseada em regra.

Comparando planos de execução

Comparando o custo total mostrado no plano de execução para diferentes instruções SQL, você pode determinar o valor do ajuste de seu código SQL. Nesta seção, você verá como comparar dois planos de execução e a vantagem de usar `EXISTS` em vez de `DISTINCT` (uma dica dada anteriormente). O exemplo a seguir gera um plano de execução para uma consulta que usa `EXISTS`:

```
EXPLAIN PLAN SET STATEMENT_ID = 'EXISTS_QUERY' FOR
SELECT product_id, name
FROM products outer
WHERE EXISTS
  (SELECT 1
   FROM purchases inner
   WHERE inner.product_id = outer.product_id);
```

O plano de execução dessa consulta está mostrado no exemplo a seguir:

```
@ c:\sql_book\sql\explain_plan.sql
Enter value for v_statement_id: EXISTS_QUERY

EXECUTION_PLAN
-----
0 SELECT STATEMENT Cost = 4
1   MERGE JOIN SEMI Cost = 1
2     TABLE ACCESS BY INDEX ROWID PRODUCTS TABLE Cost = 1
3       INDEX FULL SCAN PRODUCTS_PK INDEX (UNIQUE) Cost = 1
4     SORT UNIQUE Cost = 2
5       INDEX FULL SCAN PURCHASES_PK INDEX (UNIQUE) Cost = 1
```

O custo total da consulta é de 4 unidades de trabalho. O exemplo a seguir gera um plano de execução para uma consulta que usa DISTINCT:

```
EXPLAIN PLAN SET STATEMENT_ID = 'DISTINCT_QUERY' FOR
SELECT DISTINCT pr.product_id, pr.name
FROM products pr, purchases pu
WHERE pr.product_id = pu.product_id;
```

O plano de execução dessa consulta está mostrado no exemplo a seguir:

```
@ c:\sql_book\sql\explain_plan.sql
Enter value for v_statement_id: DISTINCT_QUERY

EXECUTION_PLAN
-----
0 SELECT STATEMENT Cost = 5
1   HASH UNIQUE Cost = 1
2     MERGE JOIN Cost = 1
3       TABLE ACCESS BY INDEX ROWID PRODUCTS TABLE Cost = 1
4         INDEX FULL SCAN PRODUCTS_PK INDEX (UNIQUE) Cost = 1
5       SORT JOIN Cost = 2
6         INDEX FULL SCAN PURCHASES_PK INDEX (UNIQUE) Cost = 1
```

O custo da consulta é de 5 unidades de trabalho. Essa consulta é mais dispendiosa do que a anterior, que usou EXISTS (essa consulta tinha um custo de apenas 4 unidades de trabalho). Esses resultados provam que é melhor usar EXISTS do que DISTINCT.

## PASSANDO DICAS PARA O OTIMIZADOR

Você pode passar dicas para o otimizador. Uma dica é uma diretiva do otimizador que influencia sua escolha de plano de execução. A dica correta pode melhorar o desempenho de uma instrução SQL. Você pode verificar a eficácia de uma dica comparando o custo no plano de execução de uma instrução SQL com e sem a dica.

Nesta seção, você verá um exemplo de consulta que utiliza uma das dicas mais úteis: a dica `FIRST_ROWS (n)`. A dica `FIRST_ROWS (n)` faz com que o otimizador gere um plano de execução que minimiza o tempo necessário para retornar as primeiras *n* linhas em uma consulta. Essa dica pode ser útil quando você não quer esperar muito tempo para obter *algumas* linhas de sua consulta, mas ainda assim quer ver todas as linhas.

O exemplo a seguir gera um plano de execução para uma consulta que usa `FIRST_ROWS(2)`; observe que a dica é colocada dentro das strings `/*+` e `*/`:

```
EXPLAIN PLAN SET STATEMENT_ID = 'HINT' FOR
SELECT /*+ FIRST_ROWS(2) */ p.name, pt.name
FROM products p, product_types pt
WHERE p.product_type_id = pt. product_type_id;
```

### CAUIDADO

*Sua dica deve usar a sintaxe exata mostrada, caso contrário, ela será ignorada. A sintaxe é: `/*+` seguido de um espaço, a dica seguida de um espaço e `*/`.*

O plano de execução dessa consulta está mostrado no exemplo a seguir; observe que o custo é de 4 unidades de trabalho:

```
@ c:\sql_book\sql\explain_plan.sql
Enter value for v_statement_id: HINT

EXECUTION_PLAN
-----
0 SELECT STATEMENT Cost = 4
1   NESTED LOOPS
2     NESTED LOOPS Cost = 1
3       TABLE ACCESS FULL PRODUCTS TABLE Cost = 1
4         INDEX UNIQUE SCAN PRODUCT_TYPES_PK INDEX (UNIQUE) Cost = 2
5       TABLE ACCESS BY INDEX ROWID PRODUCT_TYPES TABLE Cost = 2
```

O exemplo a seguir gera um plano de execução para a mesma consulta sem a dica:

```
EXPLAIN PLAN SET STATEMENT_ID = 'NO_HINT' FOR
SELECT p.name, pt.name
FROM products p, product_types pt
WHERE p.product_type_id = pt. product_type_id;
```

O plano de execução da consulta está mostrado no exemplo a seguir; observe que o custo é de 6 unidades de trabalho (maior do que a consulta com a dica):

```
@ c:\sql_book\sql\explain_plan.sql
Enter value for v_statement_id: NO_HINT

EXECUTION_PLAN
-----
0 SELECT STATEMENT Cost = 6
1   MERGE JOIN Cost = 1
2     TABLE ACCESS BY INDEX ROWID PRODUCT_TYPES TABLE Cost = 1
3       INDEX FULL SCAN PRODUCT_TYPES_PK INDEX (UNIQUE) Cost = 1
4     SORT JOIN Cost = 2
5       TABLE ACCESS FULL PRODUCTS TABLE Cost = 1
```

Esses resultados mostram que a inclusão da dica reduz o custo da execução da consulta em 2 unidades de trabalho.



Existem muitas dicas que você pode usar e esta seção forneceu apenas uma visão superficial do assunto.

## FERRAMENTAS DE AJUSTE ADICIONAIS

Nesta última seção, mencionaremos algumas outras ferramentas de ajuste. Uma abordagem completa dessas ferramentas está fora do escopo deste livro. Consulte o livro *Oracle Database Performance Tuning Guide*, publicado pela Oracle Corporation, para obter detalhes completos sobre as ferramentas mencionadas nesta seção e para ver uma lista abrangente das dicas.

### Oracle Enterprise Manager Diagnostics Pack

O Oracle Enterprise Manager Diagnostics Pack captura dados de desempenho do sistema operacional, do middle tier e do aplicativo, assim como dados de desempenho do banco de dados. O Diagnostics Pack analisa esses dados de desempenho e exibe os resultados graficamente. Um administrador de banco de dados também pode configurar o Diagnostics Pack para alertá-lo imediatamente sobre problemas de desempenho por e-mail ou pager. O Oracle Enterprise Manager também inclui orientações de software para ajudar a resolver problemas de desempenho.

### Automatic Database Diagnostic Monitor

O ADDM (Automatic Database Diagnostic Monitor) é um módulo de auto-diagnóstico incorporado ao software de banco de dados Oracle. O ADDM permite ao administrador de banco de dados monitorar problemas de desempenho do banco de dados, analisando o desempenho do sistema por um longo período de tempo. O administrador de banco de dados pode ver as informações de desempenho geradas pelo ADDM no Oracle Enterprise Manager. Quando o ADDM encontra problemas de desempenho, ele sugere soluções de ação corretiva. Alguns exemplos de sugestões do ADDM incluem:

- Alterações no hardware — por exemplo, adicionar CPUs no servidor de banco de dados
- Configuração do banco de dados — por exemplo, alterar as configurações dos parâmetros de inicialização do banco de dados
- Alterações no aplicativo — por exemplo, usar a opção de cache para seqüências ou usar variáveis de bind
- Uso de outros advisors — por exemplo, executar o SQL Tuning Advisor e o SQL Access Advisor em instruções SQL que estão consumindo a maior parte dos recursos do banco de dados

Você vai aprender sobre o SQL Tuning Advisor e o SQL Access Advisor a seguir.

### SQL Tuning Advisor (Supervisor de Ajuste SQL)

O SQL Tuning Advisor permite que um desenvolvedor ou administrador de banco de dados ajuste uma instrução SQL usando os seguintes itens:

- O texto da instrução SQL
- O identificador SQL da instrução (obtido na visão `V$SQL_PLAN`, uma das visões disponíveis para o DBA)
- O intervalo de identificadores de snapshot
- O nome do SQL Tuning Set

Um SQL Tuning Set é um conjunto de instruções SQL com seu plano de execução associado e estatísticas de execução. Os SQL Tuning Sets são analisados para gerar SQL Profiles que ajudam o otimizador a escolher um plano de execução otimizado. Os SQL Profiles contêm coleções de informações que possibilitam a otimização do plano de execução.

### **SQL Access Advisor**

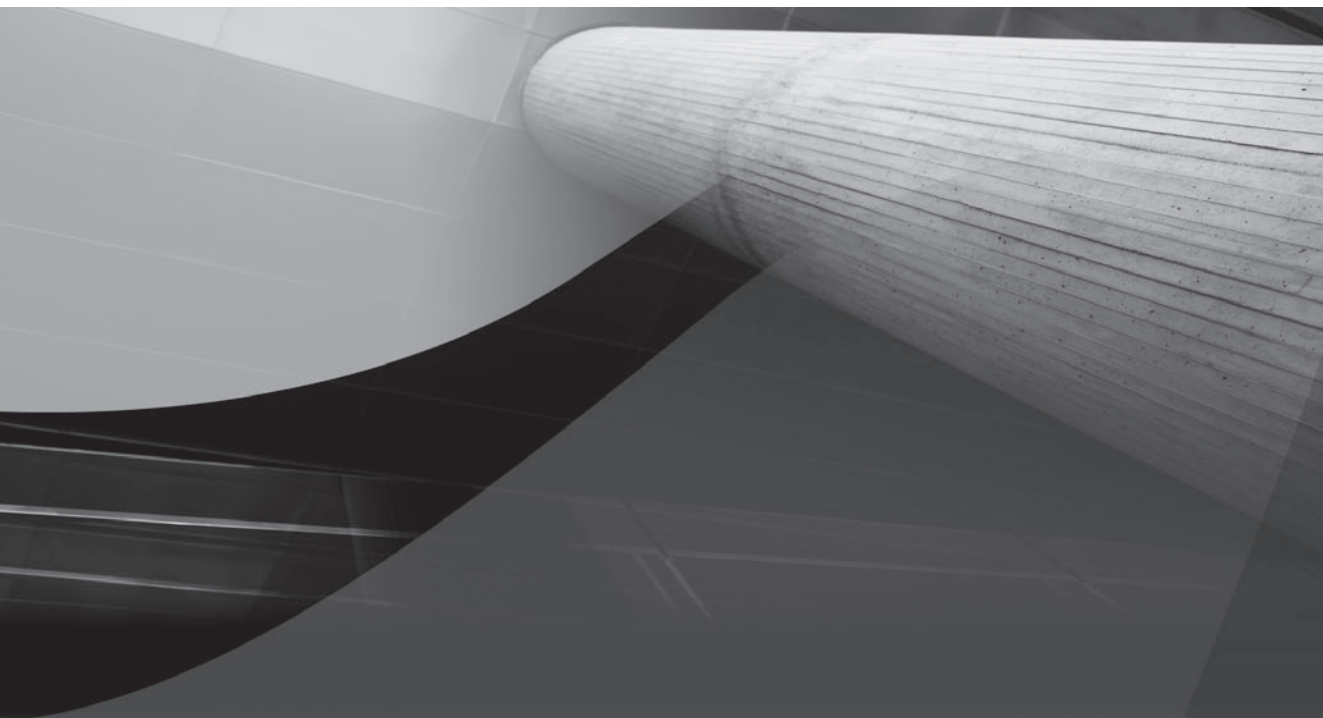
O SQL Access Advisor fornece ao desenvolvedor ou administrador de banco de dados orientações sobre desempenho em visões materializadas, índices e logs de visões materializadas. O SQL Access Advisor examina a utilização de espaço e o desempenho das consultas e recomenda a configuração com o melhor custo/benefício de visões materializadas e índices novos e já existentes.

## **RESUMO**

Neste capítulo, você aprendeu que:

- O ajuste é o processo de fazer suas instruções SQL serem executadas mais rapidamente.
- O otimizador é um subsistema do software de banco de dados Oracle que gera um plano de execução, um conjunto de operações utilizadas para executar uma instrução SQL específica.
- Dicas podem ser passadas ao otimizador para influenciar o plano de execução gerado para uma instrução SQL.
- Existem várias ferramentas de software adicionais que o DBA pode usar para ajustar o banco de dados.

No próximo capítulo, você vai aprender sobre XML.



# CAPÍTULO 17

XML e o banco de  
dados Oracle

Neste capítulo, você vai:

- Ser apresentado à XML
- Aprender a gerar código XML a partir de dados relacionais
- Investigar como salvar código XML no banco de dados

## INTRODUÇÃO À XML

A XML (Extensible Markup Language) é uma linguagem de marcação de propósito geral. Ela permite compartilhar dados estruturados na Internet e pode ser usada para codificar dados e outros documentos.

A XML apresenta as seguintes vantagens:

- Pode ser lida por seres humanos e computadores e é armazenada como texto puro
- É independente de plataforma
- Suporta Unicode, o que significa que ela pode armazenar informações escritas em muitos idiomas
- Usa um formato autodocumentado que contém a estrutura do documento, nomes de elemento e valores de elemento

Por causa dessas vantagens, a XML é muito usada para armazenamento e processamento de documentos, sendo aplicada por muitas organizações para o envio de dados entre seus sistemas de computador. Por exemplo, muitos fornecedores permitem que seus clientes enviem pedidos de compra como arquivos XML pela Internet.

O Oracle Database 9i introduziu a capacidade de armazenar código XML no banco de dados, junto com ampla funcionalidade para manipulá-lo e processá-lo. O Oracle Database 10g release 2 adicionou mais funções de geração de código XML e o Oracle Database 11g acrescenta recursos como processamento de código XML binário em Java e C (o código XML binário fornece armazenamento e manipulação mais eficientes de XML no banco de dados). Este capítulo se concentra em um subconjunto útil dos recursos de XML no banco de dados Oracle.

Se você é iniciante em XML, encontrará muitas informações nestes sites:

- <http://www.w3.org/XML>
- <http://www.wikipedia.org/wiki/XML>

## GERANDO CÓDIGO XML A PARTIR DE DADOS RELACIONAIS

O banco de dados Oracle contém várias funções SQL que podem ser usadas para gerar código XML e, nesta seção, você vai ver como gerar código XML a partir de dados relacionais utilizando algumas dessas funções.

## XMLELEMENT()

A função `XMLELEMENT()` é usada para gerar elementos XML a partir de dados relacionais. Você fornece um nome para o elemento e a coluna que deseja recuperar para a função `XMLELEMENT()` e ela retorna os elementos como objetos `XMLType`. `XMLType` é um tipo interno do banco de dados Oracle utilizado para representar dados XML. Por padrão, um objeto `XMLType` armazena os dados XML como texto em um CLOB (Character Large Object).

O exemplo a seguir se conecta como o usuário `store` e obtém os valores da coluna `customer_id` como objetos `XMLType`.

```
CONNECT store/store_password
SELECT XMLELEMENT("customer_id", customer_id)
AS xml_customers
FROM customers;
```

```
XML_CUSTOMERS
-----
<customer_id>1</customer_id>
<customer_id>2</customer_id>
<customer_id>3</customer_id>
<customer_id>4</customer_id>
<customer_id>5</customer_id>
```

Conforme você pode ver a partir desses resultados, `XMLELEMENT("customer_id", customer_id)` retorna os valores de `customer_id` dentro de uma tag `customer_id`. Você pode usar o nome de tag que desejar, como mostrado no exemplo a seguir, que utiliza a tag `"cust_id"`:

```
SELECT XMLELEMENT("cust_id", customer_id)
AS xml_customers
FROM customers;
```

```
XML_CUSTOMERS
-----
<cust_id>1</cust_id>
<cust_id>2</cust_id>
<cust_id>3</cust_id>
<cust_id>4</cust_id>
<cust_id>5</cust_id>
```

O exemplo a seguir obtém os valores de `first_name` e `dob` do cliente nº 2:

```
SELECT XMLELEMENT("first_name", first_name) || XMLELEMENT("dob", dob)
AS xml_customer
FROM customers
WHERE customer_id = 2;
```

```
XML_CUSTOMER
-----
<first_name>Cynthia</first_name><dob>1968-02-05</dob>
```

O exemplo a seguir usa a função `TO_CHAR()` para alterar o formato de data do valor de `dob`:

```
SELECT XMLELEMENT("dob", TO_CHAR(dob, 'MM/DD/YYYY'))
AS xml_dob
FROM customers
WHERE customer_id = 2;

XML_DOB
-----
<dob>02/05/1968</dob>
```

O exemplo a seguir incorpora duas chamadas de `XMLELEMENT()` dentro de uma chamada externa de `XMLELEMENT()`. Observe que os elementos `customer_id` e `name` retornados estão contidos dentro de um elemento `customer` externo:

```
SELECT XMLELEMENT(
    "customer",
    XMLELEMENT("customer_id", customer_id),
    XMLELEMENT("name", first_name || ' ' || last_name)
)
AS xml_customers
FROM customers
WHERE customer_id IN (1, 2);

XML_CUSTOMERS
-----
<customer>
  <customer_id>1</customer_id>
  <name>John Brown</name>
</customer>

<customer>
  <customer_id>2</customer_id>
  <name>Cynthia Green</name>
</customer>
```

#### NOTA

*Algumas quebras de linhas e espaços foram adicionados no código XML retornado por essa consulta para torná-lo mais fácil de ler. O mesmo foi feito em alguns dos outros exemplos deste capítulo.*

Você pode recuperar dados relacionais normais, assim como código XML, conforme mostrado no exemplo a seguir, que recupera a coluna `customer_id` como um resultado relacional normal e as colunas `first_name` e `last_name` concatenadas como elementos XML:

```
SELECT customer_id,
    XMLELEMENT("customer", first_name || ' ' || last_name) AS xml_customer
FROM customers;
```

```
CUSTOMER_ID XML_CUSTOMER
```

```
-----
1 <customer>John Brown</customer>
2 <customer>Cynthia Green</customer>
3 <customer>Steve White</customer>
4 <customer>Gail Black</customer>
5 <customer>Doreen Blue</customer>
```

Você pode gerar código XML para objetos de banco de dados, conforme mostrado no exemplo a seguir, que se conecta como `object_user` e obtém as colunas `id` e `address` para o cliente nº 1 da tabela `object_customers` (a coluna `address` armazena um objeto de tipo `t_address`):

```
CONNECT object_user/object_password
SELECT XMLELEMENT("id", id) || XMLELEMENT("address", address)
AS xml_object_customer
FROM object_customers
WHERE id = 1;
```

```
XML_OBJECT_CUSTOMER
```

```
-----
<id>1</id>
<address>
  <T_ADDRESS>
    <STREET>2 State Street</STREET>
    <CITY>Beantown</CITY>
    <STATE>MA</STATE>
    <ZIP>12345</ZIP>
  </T_ADDRESS>
</address>
```

Você pode gerar código XML para coleções, conforme mostrado no exemplo a seguir, que se conecta como `collection_user` e obtém as colunas `id` e `addresses` para o cliente nº 1 armazenado em `customers_with_nested_table` (a coluna `addresses` armazena um objeto de tipo `t_nested_table_address`, que é uma tabela aninhada de objetos `t_address`):

```
CONNECT collection_user/collection_password
SELECT XMLELEMENT("id", id) || XMLELEMENT("addresses", addresses)
AS xml_customer
FROM customers_with_nested_table
WHERE id = 1;
```

```
XML_CUSTOMER
```

```
-----
<id>1</id>
<addresses>
  <T_NESTED_TABLE_ADDRESS>
    <T_ADDRESS>
      <STREET>2 State Street</STREET><CITY>Beantown</CITY>
      <STATE>MA</STATE><ZIP>12345</ZIP>
    </T_ADDRESS>
```

```

    <T_ADDRESS>
      <STREET>4 Hill Street</STREET>
      <CITY>Lost Town</CITY>
      <STATE>CA</STATE>
      <ZIP>54321</ZIP>
    </T_ADDRESS>
  </T_NESTED_TABLE_ADDRESS>
</addresses>

```

## XMLATTRIBUTES()

XMLATTRIBUTES() é usada em conjunto com XMLELEMENT() para especificar os atributos dos elementos XML recuperados por XMLELEMENT(). O exemplo a seguir se conecta como o usuário store e usa XMLATTRIBUTES() para definir nomes de atributo para os elementos customer\_id, first\_name, last\_name e dob:

```

CONNECT store/store_password
SELECT XMLELEMENT(
  "customer",
  XMLATTRIBUTES(
    customer_id AS "id",
    first_name || ' ' || last_name AS "name",
    TO_CHAR(dob, 'MM/DD/YYYY') AS "dob"
  )
)
AS xml_customers
FROM customers
WHERE customer_id IN (1, 2);

XML_CUSTOMERS
-----
<customer id="1" name="John Brown" dob="01/01/1965"></customer>
<customer id="2" name="Cynthia Green" dob="02/05/1968"></customer>

```

Note que os atributos id, name e dob são retornados dentro de customer.

## XMLFOREST()

Você usa XMLFOREST() para gerar uma “floresta” de elementos XML. XMLFOREST() concatena elementos XML sem que você precise usar o operador de concatenação || com várias chamadas de XMLELEMENT(). O exemplo a seguir usa XMLFOREST() para obter customer\_id, phone e dob dos clientes nº 1 e 2:

```

SELECT XMLELEMENT(
  "customer",
  XMLFOREST(
    customer_id AS "id",
    phone AS "phone",
    TO_CHAR(dob, 'MM/DD/YYYY') AS "dob"
  )
)
AS xml_customers

```



```

FROM customers
WHERE customer_id IN (1, 2);

XML_CUSTOMERS
-----
<customer>
  <id>1</id>
  <phone>800-555-1211</phone>
  <dob>01/01/1965</dob>
</customer>

<customer>
  <id>2</id>
  <phone>800-555-1212</phone>
  <dob>02/05/1968</dob>
</customer>

```

O comando a seguir configura o parâmetro `LONG` do SQL\*Plus como 500, para que você possa ver todo o código XML retornado pelas consultas subseqüentes (`LONG` controla o comprimento máximo dos dados de texto exibidos pelo SQL\*Plus):

```
SET LONG 500
```

A consulta a seguir coloca o nome do cliente dentro da tag de elemento `customer` usado `XMLATTRIBUTES()`:

```

SELECT XMLELEMENT(
  "customer",
  XMLATTRIBUTES(first_name || ' ' || last_name AS "name"),
  XMLFOREST(phone AS "phone", TO_CHAR(dob, 'MM/DD/YYYY') AS "dob")
)
AS xml_customers
FROM customers
WHERE customer_id IN (1, 2);

XML_CUSTOMERS
-----
<customer name="John Brown">
  <phone>800-555-1211</phone>
  <dob>01/01/1965</dob>
</customer>

<customer name="Cynthia Green">
  <phone>800-555-1212</phone>
  <dob>02/05/1968</dob>
</customer>

```

## XMLAGG()

Você usa `XMLAGG()` para gerar uma floresta de elementos XML a partir de uma coleção de elementos XML. Normalmente, `XMLAGG()` é utilizada para agrupar código XML em uma lista de itens comuns abaixo de um único pai ou para recuperar dados de coleções. Você pode usar a cláusula

GROUP BY de uma consulta para agrupar o conjunto de linhas retornado em vários grupos e pode usar uma cláusula ORDER BY de XMLAGG() para classificar as linhas.

Por padrão, ORDER BY classifica os resultados em ordem crescente, mas você pode adicionar DESC após a lista de colunas para classificar as linhas em ordem decrescente. É possível adicionar ASC para indicar uma classificação crescente explicitamente. Você também pode adicionar NULLS LAST para colocar os valores nulos ao final dos resultados.

O exemplo a seguir recupera os valores de first\_name e last\_name e os retorna em uma lista chamada customer\_list; observe que a cláusula ORDER BY é usada com XMLAGG() para classificar os resultados pela coluna first\_name.ASC foi adicionado para indicar uma classificação crescente de forma explícita:

```
SELECT XMLELEMENT(
    "customer_list",
    XMLAGG(
        XMLELEMENT("customer", first_name || ' ' || last_name)
        ORDER BY first_name ASC
    )
)
AS xml_customers
FROM customers
WHERE customer_id IN (1, 2);

XML_CUSTOMERS
-----
<customer_list>
  <customer>Cynthia Green</customer>
  <customer>John Brown</customer>
</customer_list>
```

O exemplo a seguir recupera o valor de product\_type\_id e o valor médio de price para cada grupo de produtos; observe que os produtos são agrupados por product\_type\_id usando a cláusula GROUP BY da consulta e que NULLS LAST é usado na cláusula ORDER BY de XMLAGG() para colocar a linha com o valor de product\_type\_id nulo ao final dos resultados retornados.

```
SELECT XMLELEMENT(
    "product_list",
    XMLAGG(
        XMLELEMENT(
            "product_type_and_avg", product_type_id || ' ' || AVG(price)
        )
        ORDER BY product_type_id NULLS LAST
    )
)
AS xml_products
FROM products
GROUP BY product_type_id;

XML_PRODUCTS
-----
<product_list>
  <product_type_and_avg>1 24.975</product_type_and_avg>
  <product_type_and_avg>2 26.22</product_type_and_avg>
  <product_type_and_avg>3 13.24</product_type_and_avg>
```

```

    <product_type_and_avg>4 13.99</product_type_and_avg>
    <product_type_and_avg> 13.49</product_type_and_avg>
  </product_list>

```

**NOTA**

*Você também pode colocar a linha nula primeiro, especificando NULLS FIRST na cláusula ORDER BY de XMLAGG().*

O exemplo a seguir recupera product\_type\_id e name para os produtos com os valores 1 e 2 de product\_type\_id e os produtos são agrupados por product\_type\_id:

```

SELECT XMLELEMENT(
  "products_in_group",
  XMLATTRIBUTES(product_type_id AS "prd_type_id"),
  XMLAGG(
    XMLELEMENT("name", name)
  )
)
AS xml_products
FROM products
WHERE product_type_id IN (1, 2)
GROUP BY product_type_id;

```

```

XML_PRODUCTS
-----
<products_in_group prd_type_id="1">
  <name>Modern Science</name>
  <name>Chemistry</name>
</products_in_group>

<products_in_group prd_type_id="2">
  <name>Supernova</name>
  <name>2412: The Return</name>
</products_in_group>

```

O exemplo a seguir conecta-se como collection\_user e recupera os endereços do cliente nº 1 de customers\_with\_nested\_table:

```

CONNECT collection_user/collection_password
SELECT XMLELEMENT("customer",
  XMLAGG(
    XMLELEMENT("addresses", addresses)
  )
)
AS xml_customer
FROM customers_with_nested_table
WHERE id = 1;

```

```

XML_CUSTOMER
-----
<customer>
  <addresses>

```

```

<T_NESTED_TABLE_ADDRESS>
  <T_ADDRESS>
    <STREET>2 State Street</STREET>
    <CITY>Beantown</CITY>
    <STATE>MA</STATE>
    <ZIP>21345</ZIP>
  </T_ADDRESS>
  <T_ADDRESS>
    <STREET>4 Hill Street</STREET>
    <CITY>Lost Town</CITY>
    <STATE>CA</STATE>
    <ZIP>54321</ZIP>
  </T_ADDRESS>
</T_NESTED_TABLE_ADDRESS>
</addresses>
</customer>

```

## XMLCOLATTVAL()

Você usa `XMLCOLATTVAL()` para criar um fragmento de código XML e depois expandir o código resultante. Cada fragmento tem a coluna de nome com o nome do atributo. Você pode usar a cláusula `AS` para alterar o nome do atributo.

O exemplo a seguir se conecta como o usuário `store` e recupera os valores de `customer_id`, `dob` e `phone` dos clientes de nº 1 e 2:

```

CONNECT store/store_password
SELECT XMLELEMENT(
  "customer",
  XMLCOLATTVAL(
    customer_id AS "id",
    dob AS "dob",
    phone AS "phone"
  )
)
AS xml_customers
FROM customers
WHERE customer_id IN (1, 2);

XML_CUSTOMERS
-----
<customer>
  <column name = "id">1</column>
  <column name = "dob">1965-01-01</column>
  <column name = "phone">800-555-1211</column>
</customer>

<customer>
  <column name = "id">2</column>
  <column name = "dob">1968-02-05</column>
  <column name = "phone">800-555-1212</column>
</customer>

```

## XMLCONCAT()

XMLCONCAT() concatena uma série de elementos de cada linha. O exemplo a seguir concatena os elementos XML dos valores de first\_name, last\_name e phone dos clientes nº 1 e 2:

```
SELECT XMLCONCAT(
    XMLELEMENT("first name", first_name),
    XMLELEMENT("last name", last_name),
    XMLELEMENT("phone", phone)
)
AS xml_customers
FROM customers
WHERE customer_id IN (1, 2);

XML_CUSTOMERS
-----
<first name>John</first name>
<last name>Brown</last name>
<phone>800-555-1211</phone>

<first name>Cynthia</first name>
<last name>Green</last name>
<phone>800-555-1212</phone>
```

## XMLPARSE()

XMLPARSE() é usada para analisar e gerar código XML a partir do resultado avaliado de uma expressão. A expressão deve transformar-se em uma string; se ela se transformar em um valor nulo, XMLPARSE() retornará um valor nulo. Você deve especificar um dos seguintes itens antes da expressão:

- CONTENT, que significa que a expressão deve se transformar em um valor XML válido
- DOCUMENT, que significa que a expressão deve se transformar em um documento XML de raiz individual

Você também pode adicionar WELLFORMED após a expressão, o que significa que você está garantindo que sua expressão se transforma em um documento XML bem formatado. Isso também significa que o banco de dados não realizará verificações de validade em sua expressão.

O exemplo a seguir analisa uma expressão contendo os detalhes de um cliente:

```
SELECT XMLPARSE(
    CONTENT
    '<customer><customer_id>1</customer_id><name>John Brown</name></customer>'
    WELLFORMED
)
AS xml_customer
FROM dual;

XML_CUSTOMER
-----
<customer>
  <customer_id>1</customer_id>
  <name>John Brown</name>
</customer>
```

**NOTA**

Leia mais sobre documentos XML bem formatados no endereço <http://www.w3.org/TR/REC-xml>.

**XMLPI()**

XMLPI() gera uma instrução de processamento XML. Normalmente, uma instrução de processamento é usada para fornecer a um aplicativo informações associada a dados XML; o aplicativo pode então utilizar a instrução de processamento para determinar como vai processar os dados XML.

O exemplo a seguir gera uma instrução de processamento para um status de pedido:

```
SELECT XMLPI (
    NAME "order_status",
    'PLACED, PENDING, SHIPPED'
)
AS xml_order_status_pi
FROM dual;

XML_ORDER_STATUS_PI
-----
<?order_status PLACED, PENDING, SHIPPED?>
```

O próximo exemplo gera uma instrução de processamento para exibir um documento XML usando um arquivo de folha de estilo em cascata chamado `example.css`:

```
SELECT XMLPI (
    NAME "xml-stylesheet",
    'type="text/css" href="example.css"'
)
AS xml_stylesheet_pi
FROM dual;

XML_STYLESHEET_PI
-----
<?xml-stylesheet type="text/css" href="example.css"?>
```

**XMLCOMMENT()**

Você usa XMLCOMMENT() para gerar um comentário em XML, que é uma string de texto colocada entre `<!--` e `-->`. Por exemplo:

```
SELECT XMLCOMMENT(
    'An example XML Comment'
)
AS xml_comment
FROM dual;

XML_COMMENT
-----
<!--An example XML Comment-->
```

## XMLSEQUENCE()

`XMLSEQUENCE()` gera um objeto `XMLSequenceType`, que é um varray de objetos `XMLType`. Como `XMLSEQUENCE()` retorna um varray, você pode utilizá-lo na cláusula `FROM` de uma consulta. Por exemplo:

```
SELECT VALUE(list_of_values).GETSTRINGVAL() order_values
FROM TABLE(
  XMLSEQUENCE(
    EXTRACT(
      XMLType(' <A><B>PLACED</B><B>PENDING</B><B>SHIPPED</B></A>' ),
      '/A/B'
    )
  ) list_of_values;

ORDER_VALUES
-----
<B>PLACED</B>
<B>PENDING</B>
<B>SHIPPED</B>
```

Vamos decompor esse exemplo. A chamada de `XMLType()` é:

```
XMLType(' <A><B>PLACED</B><B>PENDING</B><B>SHIPPED</B></A>' )
```

Isso cria um objeto `XMLType` contendo o código XML a seguir:

```
<A><B>PLACED</B><B>PENDING</B><B>SHIPPED</B></A>.
```

A chamada da função `EXTRACT()` é:

```
EXTRACT(
  XMLType(' <A><B>PLACED</B><B>PENDING</B><B>SHIPPED</B></A>' ),
  '/A/B'
)
```

`EXTRACT()` extrai os dados XML do objeto `XMLType` retornados pela chamada de `XMLType()`. O segundo parâmetro de `EXTRACT()` é uma string XPath. XPath é uma linguagem que permite acessar elementos específicos em dados XML. Por exemplo, na chamada de `EXTRACT()` anterior, `'/A/B'` retorna todos os elementos de B que são filhos dos elementos de A, portanto, a função `EXTRACT()` retorna o seguinte:

```
<B>PLACED</B>
<B>PENDING</B>
<B>SHIPPED</B>
```

A chamada de `XMLSEQUENCE()` do exemplo simplesmente retorna um varray contendo os elementos retornados por `EXTRACT()`. `TABLE()` converte o varray em uma tabela de linhas e aplica o apelido `list_of_values` à tabela. A instrução `SELECT` recupera o valor de string das linhas da tabela usando `GETSTRINGVAL()`. Você verá mais exemplos de `EXTRACT()` e XPath posteriormente neste capítulo.

## XMLSERIALIZE()

`XMLSERIALIZE()` gera uma representação de string ou LOB (Large Object) de dados XML a partir do resultado avaliado de uma expressão. Antes da expressão, você deve especificar um dos itens a seguir:

- `CONTENT`, que significa que a expressão deve se transformar em um valor XML válido
- `DOCUMENT`, que significa que a expressão deve se transformar em um documento XML de raiz única

O exemplo a seguir usa `XMLSERIALIZE()` com `CONTENT` para gerar um valor XML:

```
SELECT XMLSERIALIZE(
  CONTENT XMLType('<order_status>SHIPPED</order_status>')
)
AS xml_order_status
FROM DUAL;

XML_ORDER_STATUS
-----
<order_status>SHIPPED</order_status>
```

O próximo exemplo usa `XMLSERIALIZE()` com `DOCUMENT` para gerar um documento XML, com o documento retornado em um CLOB (Character Large Object):

```
SELECT XMLSERIALIZE(
  DOCUMENT XMLType('<description>Description of a product</description>')
  AS CLOB
)
AS xml_product_description
FROM DUAL;

XML_PRODUCT_DESCRIPTION
-----
<description>Description of a product</description>
```

## Um exemplo em PL/SQL que grava os dados XML em um arquivo

Nesta seção, você vai ver um exemplo completo em PL/SQL que grava nomes de clientes em um arquivo XML. Primeiro você precisa se conectar como um usuário privilegiado (por exemplo, o usuário `system`) e conceder o privilégio `CREATE ANY DIRECTORY` ao usuário `store`:

```
CONNECT system/manager;
GRANT CREATE ANY DIRECTORY TO store;
```

Em seguida, você precisa se conectar como o usuário `store` e criar um objeto de diretório:

```
CONNECT store/store_password;
CREATE DIRECTORY TEMP_FILES_DIR AS 'C:\temp_files';
```

Você também precisará criar um diretório chamado `temp_files` na partição `C`. (Se estiver usando Linux ou Unix, pode criar o diretório em uma de suas partições e utilizar o comando `CREATE DIRECTORY` apropriado, com o caminho correto. Além disso, certifique-se de conceder permissões de gravação no diretório para a conta de usuário Oracle que você usou para instalar o software de banco de dados).



Em seguida, você precisa executar o script `xml_examples.sql` localizado no diretório SQL, como mostrado:

```
@ "E:\Oracle SQL book\sql_book\SQL\xml_examples.sql"
```

### ATENÇÃO

*Execute somente o script `xml_examples.sql` neste ponto. Você poderá observar que existe um script chamado `xml_schema.sql` no diretório SQL. Não execute esse script ainda.*

O script `xml_examples.sql` cria duas procedures; a que você vai ver nesta seção é chamada `write_xml_data_to_file()`, a qual recupera o nome dos clientes e os grava num arquivo XML. A procedure `write_xml_data_to_file()` é definida como segue:

```
CREATE PROCEDURE write_xml_data_to_file(
    p_directory VARCHAR2,
    p_file_name VARCHAR2
) AS
    v_file UTL_FILE.FILE_TYPE;
    v_amount INTEGER:= 32767;
    v_xml_data XMLType;
    v_char_buffer VARCHAR2(32767);
BEGIN
    -- abre o arquivo para gravar o texto (até v_amount
    -- caracteres por vez)
    v_file:= UTL_FILE.FOPEN(p_directory, p_file_name, 'w', v_amount);

    -- grava a linha inicial em v_file
    UTL_FILE.PUT_LINE(v_file, '<?xml version="1.0"?>');

    -- recupera os clientes e os armazena em v_xml_data
    SELECT
        EXTRACT(
            XMLELEMENT(
                "customer_list",
                XMLAGG(
                    XMLELEMENT("customer", first_name || ' ' || last_name)
                    ORDER BY last_name
                )
            ),
        )
    AS xml_customers
    INTO v_xml_data
    FROM customers;

    -- obtém o valor da string de v_xml_data e o armazena em v_char_buffer
    v_char_buffer:= v_xml_data.GETSTRINGVAL();

    -- copia os caracteres de v_char_buffer no arquivo
    UTL_FILE.PUT(v_file, v_char_buffer);
```

```

-- descarrega os dados restantes no arquivo
UTL_FILE.FFLUSH(v_file);

-- fecha o arquivo
UTL_FILE.FCLOSE(v_file);
END write_xml_data_to_file;
/

```

A instrução a seguir chama `write_xml_data_to_file()`:

```
CALL write_xml_data_to_file('TEMP_FILES_DIR', 'customers.xml');
```

Após executar essa instrução, você encontrará um arquivo chamado `customers.xml` em `C:\temp_files` ou em qualquer que seja o diretório utilizado ao executar o comando `CREATE DIRECTORY` anteriormente. O conteúdo do arquivo `customers.xml` é:

```

<?xml version="1.0"?>
<customer_list><customer>Gail Black</customer><customer>Doreen Blue
</customer><customer>John Brown</customer><customer>Cynthia Green
</customer><customer>Steve White</customer></customer_list>

```

Você pode modificar a procedure `write_xml_data_to_file()` para recuperar qualquer dado relacional do banco de dados e gerá-lo em um arquivo XML.

## XMLQUERY()

`XMLQUERY()` é usada para construir ou consultar códigos XML. Você passa uma expressão XQuery para `XMLQUERY()`. A XQuery é uma linguagem de consulta que permite construir e consultar código XML. `XMLQUERY()` retorna o resultado da avaliação da expressão XQuery.

O exemplo simples a seguir ilustra o uso de `XMLQUERY()`:

```

SELECT XMLQUERY(
  '(1, 2 + 5, "d", 155 to 161, <A>text</A>)'
  RETURNING CONTENT
)
AS xml_output
FROM DUAL;

XML_OUTPUT
-----
1 7 d 155 156 157 158 159 160 161<A>text</A>

```

Eis algumas observações sobre o exemplo:

- A string passada para `XMLQUERY()` é a expressão XQuery; ou seja, a expressão XQuery é `(1, 2 + 5, "d", 155 to 161, <A>text</A>)`. 1 é um inteiro literal, `2 + 5` é uma expressão aritmética, `d` é uma string literal, `155 to 161` é uma seqüência de inteiros e `<A>text</A>` é um elemento XML.
- Na XQuery cada um dos itens é avaliado por sua vez. Por exemplo, `2 + 5` é avaliado e 7 é retornado. Da mesma forma, `155 to 161` é avaliado e é retornado 155 156 157 158 159 160 161.

- **RETURNING CONTENT** significa que um fragmento XML é retornado. Esse fragmento é um único elemento XML com qualquer número de “filhos”, os quais podem ser qualquer tipo de elemento XML, inclusive elementos textuais. O fragmento XML também obedece ao modelo de dados Infoset estendido. Infoset é uma especificação que descreve um modelo de dados abstrato de um documento XML. Para saber mais sobre Infoset, visite o endereço <http://www.w3.org/TR/xml-infoset>.

Vamos explorar um exemplo mais complexo. A instrução a seguir (contida no script `xml_examples.sql`) cria uma procedure chamada `create_xml_resources()`; essa procedure cria strings XML para produtos e tipos de produto. Ela utiliza métodos do pacote `PL/SQL DBMS_XDB` para excluir e criar arquivos de recursos XML no XML DB Repository do Oracle (o XML DB Repository é uma área de armazenamento para dados XML dentro do banco de dados):

```
CREATE PROCEDURE create_xml_resources AS
    v_result BOOLEAN;

    -- cria string contendo código XML para produtos
    v_products VARCHAR2(300) :=
        '<?xml version="1.0"?>' ||
        '<products>' ||
        '  <product product_id="1" product_type_id="1" name="Modern Science"'
        || '    price="19.95"/>' ||
        '  <product product_id="2" product_type_id="1" name="Chemistry"'
        || '    price="30"/>' ||
        '  <product product_id="3" product_type_id="2" name="Supernova"'
        || '    price="25.99"/>' ||
        '</products>';

    -- cria string contendo código XML para tipos de produto
    v_product_types VARCHAR2(300) :=
        '<?xml version="1.0"?>' ||
        '<product_types>' ||
        '  <product_type product_type_id="1" name="Book"/>' ||
        '  <product_type product_type_id="2" name="Video"/>' ||
        '</product_types>';

BEGIN
    -- exclui o recurso existente para produtos
    DBMS_XDB.DELETETERESOURCE('/public/products.xml',
        DBMS_XDB.DELETE_RECURSIVE_FORCE);

    -- cria um recurso para produtos
    v_result:= DBMS_XDB.CREATETERESOURCE('/public/products.xml',
        v_products);

    -- exclui o recurso existente para tipos de produto
    DBMS_XDB.DELETETERESOURCE('/public/product_types.xml',
        DBMS_XDB.DELETE_RECURSIVE_FORCE);

    -- cria um recurso para tipos de produto
    v_result:= DBMS_XDB.CREATETERESOURCE('/public/product_types.xml',
        v_product_types);
END create_xml_resources;
/
```

Eis algumas observações sobre `create_xml_resources()`:

- A procedure `DBMS_XDB.DELETERESOURCE()` exclui um recurso XML do banco de dados. Essa procedure é chamada por `create_xml_resources()` para que você não precise remover os recursos manualmente, caso execute `create_xml_resources()` mais de uma vez.
- A constante `DBMS_XDB.DELETE_RECURSIVE_FORCE` força a exclusão do recurso, incluindo todos os objetos filhos.
- A função `DBMS_XDB.CREATERESOURCE()` cria um recurso XML no banco de dados e retorna um valor booleano `true/false` indicando se a operação foi bem-sucedida. As duas chamadas dessa função criam recursos para os produtos e tipos de produto em `/public`, que é o caminho absoluto para armazenar os recursos.

A instrução a seguir chama `create_xml_resources()`:

```
CALL create_xml_resources();
```

A consulta a seguir usa `XMLQUERY()` para recuperar os produtos do recurso `/public/products.xml`:

```
SELECT XMLQUERY(
    'for $product in doc("/public/products.xml")/products/product
    return <product name="{ $product/@name }"/>'
    RETURNING CONTENT
)
AS xml_products
FROM DUAL;

XML_PRODUCTS
-----
<product name="Modern Science"></product>
<product name="Chemistry"></product>
<product name="Supernova"></product>
```

A expressão XQuery dentro da função `XMLQUERY()` no exemplo anterior é:

```
for $product in doc("/public/products.xml")/products/product
return <product name="{ $product/@name }"/>
```

Vamos decompor essa expressão XQuery:

- O loop `for` itera sobre os produtos de `/public/products.xml`.
- `$product` é uma variável de bind vinculada à seqüência de produtos retornada por `doc("/public/products.xml")/products/product`; `doc("/public/products.xml")` retorna o documento `products.xml` armazenado em `/public`. A cada iteração do loop, `$product` é configurada como cada produto de `products.xml`, um após o outro.
- A parte `return` da expressão retorna o nome do produto que está em `$product`.

A próxima consulta recupera os tipos de produto a partir do recurso `/public/product_types.xml`:

```
SELECT XMLQUERY(
  'for $product_type in
    doc("/public/product_types.xml")/product_types/product_type
    return <product_type name="{ $product_type/@name}"/>'
  RETURNING CONTENT
)
AS xml_product_types
FROM DUAL;

XML_PRODUCT_TYPES
-----
<product_type name="Book"></product_type>
<product_type name="Video"></product_type>
```

A consulta a seguir recupera os produtos cujo preço é maior do que 20, junto com o tipo de produto:

```
SELECT XMLQUERY(
  'for $product in doc("/public/products.xml")/products/product
    let $product_type:=
      doc("/public/product_types.xml")//product_type[@product_type_id =
        $product/@product_type_id]/@name
    where $product/@price > 20
    order by $product/@product_id
    return <product name="{ $product/@name}"
      product_type="{ $product_type}"/>'
  RETURNING CONTENT
)
AS xml_query_results
FROM DUAL;

XML_QUERY_RESULTS
-----
<product name="Chemistry" product_type="Book"></product>
<product name="Supernova" product_type="Video"></product>
```

Vamos decompor a expressão XQuery desse exemplo:

- São usadas duas variáveis de bind: `$product` e `$product_type`. Essas variáveis são utilizadas para armazenar os produtos e os tipos de produto.
- A parte `let` da expressão configura `$product_type` com o tipo de produto recuperado de `$product`. A expressão no lado direito de `:=` realiza um join usando o valor de `product_type_id` armazenado em `$product_type` e `$product`. O sinal `//` significa “recuperar todos os elementos”.
- A parte `where` recupera somente os produtos cujo preço é maior do que 20.
- A parte `order by` ordena os resultados pela identificação do produto (em ordem crescente, por padrão).

O próximo exemplo mostra o uso das seguintes funções XQuery:

- `count()`, que conta o número de objetos passados a ela.
- `avg()`, que calcula a média dos números passados a ela.
- `integer()`, que trunca um número e retorna o inteiro. A função `integer()` está no namespace `xs`. (as funções `count()` e `avg()` estão no namespace `fn`, que é referenciado automaticamente pelo banco de dados, permitindo com isso que você omita o namespace ao chamar essas funções.)

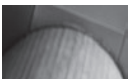
O exemplo a seguir retorna o nome do tipo de produto, o número de produtos em cada tipo de produto e o preço médio dos produtos em cada tipo de produto (truncado em um inteiro):

```
SELECT XMLQUERY (
  'for $product_type in
    doc("/public/product_types.xml")/product_types/product_type
  let $product:=
    doc("/public/products.xml")//product[@product_type_id =
      $product_type/@product_type_id]
  return
    <product_type name="{ $product_type/@name }"
      num_products="{count($product)}"
      average_price="{xs:integer(avg($product/@price))}"
    />'
  RETURNING CONTENT
)
AS xml_query_results
FROM DUAL;

XML_QUERY_RESULTS
-----
<product_type name="Book" num_products="2" average_price="24">
</product_type>

<product_type name="Video" num_products="1" average_price="25">
</product_type>
```

Como você pode ver a partir dos resultados, existem dois livros e um vídeo.



#### NOTA

Leia mais sobre funções no endereço <http://www.w3.org/TR/xquery-operators>. Para mais informações sobre `XMLQUERY()`, visite o endereço <http://www.sqlx.org>.

## SALVANDO XML NO BANCO DE DADOS

Nesta seção, você vai aprender a armazenar um documento XML no banco de dados e recuperar informações do documento XML armazenado.

## O arquivo de exemplo XML

O arquivo `purchase_order.xml` é um arquivo XML que contém uma ordem de compra. Esse arquivo está contido no diretório `xml_files`, que foi criado quando você extraiu o arquivo Zip deste livro. Se quiser acompanhar os exemplos, copie o diretório `xml_files` na partição `C` em seu servidor de banco de dados (se estiver usando Linux e Unix, você pode copiar o diretório em uma de suas partições).



### NOTA

*Se você copiar o diretório `xml_files` em um local diferente de `C`, precisará editar o script `xml_schema.sql` (você verá esse script em breve).*

O conteúdo do arquivo `purchase_order.xml` é:

```
<?xml version="1.0"?>
<purchase_order>
  <customer_order_id>176</customer_order_id>
  <order_date>2007-05-17</order_date>
  <customer_name>Best Products 456 Inc.</customer_name>
  <street>10 Any Street</street>
  <city>Any City</city>
  <state>CA</state>
  <zip>94440</zip>
  <phone_number>555-121-1234</phone_number>
  <products>
    <product>
      <product_id>1</product_id>
      <name>Supernova video</name>
      <quantity>5</quantity>
    </product>
    <product>
      <product_id>2</product_id>
      <name>Oracle SQL book</name>
      <quantity>4</quantity>
    </product>
  </products>
</purchase_order>
```

Nas seções a seguir, você vai aprender a armazenar esse arquivo XML no banco de dados.

Em um exemplo real, a ordem de compra poderia ser enviada pela Internet para uma loja online, que então despacharia os itens solicitados para o cliente.

## Criando o esquema de exemplo XML

Fornecemos um script SQL\*Plus chamado `xml_schema.sql` no diretório `SQL`. O script cria um usuário chamado `xml_user` com a senha `xml_password` e cria os itens utilizados no restante deste capítulo. Não execute esse script ainda.

O script contém as instruções a seguir, que criam um tipo de objeto chamado `t_product` (usado para representar produtos), um tipo de tabela aninhada chamado `t_nested_table_product` (usado para representar uma tabela aninhada de produtos) e uma tabela chamada `purchase_order`:

```
CREATE TYPE t_product AS OBJECT (
    product_id INTEGER,
    name VARCHAR2(15),
    quantity INTEGER
);
/

CREATE TYPE t_nested_table_product AS TABLE OF t_product;
/

CREATE TABLE purchase_order (
    purchase_order_id INTEGER CONSTRAINT purchase_order_pk PRIMARY KEY,
    customer_order_id INTEGER,
    order_date DATE,
    customer_name VARCHAR2(25),
    street VARCHAR2(15),
    city VARCHAR2(15),
    state VARCHAR2(2),
    zip VARCHAR2(5),
    phone_number VARCHAR2(12),
    products t_nested_table_product,
    xml_purchase_order XMLType
)
NESTED TABLE products
STORE AS nested_products;
```

Note que a coluna `xml_purchase_order` é de tipo `XMLType`, que é um tipo interno do banco de dados Oracle que permite armazenar dados XML. Por padrão uma coluna `XMLType` armazena os dados XML como texto em um CLOB (Character Large Object). O script `xml_schema.sql` também contém a instrução a seguir, que cria um objeto de diretório chamado `XML_FILES_DIR`:

```
CREATE OR REPLACE DIRECTORY XML_FILES_DIR AS 'C:\xml_files';
```

Você precisará modificar essa linha se tiver copiado o diretório `xml_files` em um local diferente de C. Se esse for o caso, faça isso agora; em seguida, salve o script. A instrução `INSERT` a seguir (também contida no script) adiciona uma linha na tabela `purchase_order`:

```
INSERT INTO purchase_order (
    purchase_order_id,
    xml_purchase_order
) VALUES (
    1,
    XMLType(
        BFILENAME('XML_FILES_DIR', 'purchase_order.xml'),
        NLS_CHARSET_ID('AL32UTF8')
    )
);
```



O construtor `XMLType()` aceita dois parâmetros. O primeiro é um `BFILE`, que é um ponteiro para um arquivo externo. O segundo parâmetro é o conjunto de caracteres do texto XML no arquivo externo. Na instrução `INSERT` anterior, o parâmetro `BFILE` aponta para o arquivo `purchase_order.xml` e o conjunto de caracteres é `AL32UTF8`, que é a codificação UTF-8 padrão. Quando a instrução `INSERT` é executada, o código XML do arquivo `purchase_order.xml` é lido e, então, armazenado no banco de dados como um texto `CLOB` na coluna `xml_purchase_order`.

#### NOTA

*Quando estiver trabalhando com arquivos XML escritos em inglês, use o conjunto de caracteres `AL32UTF8`. Para mais informações sobre os diferentes conjuntos de caracteres, consulte o Oracle Database Globalization Support Guide, publicado pela Oracle Corporation.*

Você deve ter notado que as colunas `customer_order_id`, `order_date`, `customer_name`, `street`, `city`, `state`, `zip`, `phone_number` e `products` da tabela `purchase_order` estão vazias. Os dados dessas colunas podem ser extraídos do código XML armazenado na coluna `xml_purchase_order`. Posteriormente neste capítulo, você verá uma procedure `PL/SQL` que lê o código XML e configura as outras colunas de forma correspondente.

Execute o script `xml_schema.sql` como um usuário privilegiado (como o usuário `system`):

```
CONNECT system/manager
@ "E:\Oracle SQL book\sql_book\SQL\xml_schema.sql"
```

Depois que o script terminar, você estará conectado como `xml_user`.

## Recuperando informações do esquema XML de exemplo

Nesta seção, você vai ver como recuperar informações do esquema `xml_user`. O exemplo a seguir recupera a linha da tabela `purchase_order`:

```
SET LONG 1000
SET PAGESIZE 500
SELECT purchase_order_id, xml_purchase_order
FROM purchase_order;

PURCHASE_ORDER_ID
-----
XML_PURCHASE_ORDER
-----
1
<?xml version="1.0"?>
<purchase_order>
  <customer_order_id>176</customer_order_id>
  <order_date>2007-05-17</order_date>
  <customer_name>Best Products 456 Inc.</customer_name>
  <street>10 Any Street</street>
  <city>Any City</city>
  <state>CA</state>
  <zip>94440</zip>
  <phone_number>555-121-1234</phone_number>
  <products>
```

```

<product>
  <product_id>1</product_id>
  <name>Supernova video</name>
  <quantity>5</quantity>
</product>
<product>
  <product_id>2</product_id>
  <name>Oracle SQL book</name>
  <quantity>4</quantity>
</product>
</products>
</purchase_order>

```

A próxima consulta extrai os valores de `customer_order_id`, `order_date`, `customer_name` e `phone_number` do código XML armazenado na coluna `xml_purchase_order` usando a função `EXTRACT()`:

```

SELECT
  EXTRACT(xml_purchase_order,
    '/purchase_order/customer_order_id') cust_order_id,
  EXTRACT(xml_purchase_order, '/purchase_order/order_date') order_date,
  EXTRACT(xml_purchase_order, '/purchase_order/customer_name') cust_name,
  EXTRACT(xml_purchase_order, '/purchase_order/phone_number') phone_number
FROM purchase_order
WHERE purchase_order_id = 1;

CUST_ORDER_ID
-----
ORDER_DATE
-----
CUST_NAME
-----
PHONE_NUMBER
-----
<customer_order_id>176</customer_order_id>
<order_date>2007-05-17</order_date>
<customer_name>Best Products 456 Inc.</customer_name>
<phone_number>555-121-1234</phone_number>

```

A função `EXTRACT()` retorna os valores como objetos `XMLType`. Você pode usar a função `EXTRACTVALUE()` para obter os valores como strings. Por exemplo, a consulta a seguir extrai os mesmos valores como strings usando a função `EXTRACTVALUE()`:

```

SELECT
  EXTRACTVALUE(xml_purchase_order,
    '/purchase_order/customer_order_id') cust_order_id,
  EXTRACTVALUE(xml_purchase_order,
    '/purchase_order/order_date') order_date,
  EXTRACTVALUE(xml_purchase_order,
    '/purchase_order/customer_name') cust_name,
  EXTRACTVALUE(xml_purchase_order,

```

```

        '/purchase_order/phone_number') phone_number
FROM purchase_order
WHERE purchase_order_id = 1;

CUST_ORDER_ID
-----
ORDER_DATE
-----
CUST_NAME
-----
PHONE_NUMBER
-----
176
2007-05-17
Best Products 456 Inc.
555-121-1234

```

A próxima consulta extrai e converte `order_date` em um valor `DATE` usando a função `TO_DATE()`. Note que o formato da data, conforme armazenado no código XML, é fornecido pelo segundo parâmetro de `TO_DATE()` e que a função retorna a data no formato padrão utilizado pelo banco de dados (DD-MON-YY):

```

SELECT
  TO_DATE(
    EXTRACTVALUE(xml_purchase_order, '/purchase_order/order_date'),
    'YYYY-MM-DD'
  ) AS ord_date
FROM purchase_order
WHERE purchase_order_id = 1;

ORD_DATE
-----
17-MAY-07

```

A consulta a seguir recupera todos os produtos de `xml_purchase_order` como XML usando `EXTRACT()`; observe o uso de `//` para obter todos os produtos:

```

SELECT
  EXTRACT(xml_purchase_order, '/purchase_order//products') xml_products
FROM purchase_order
WHERE purchase_order_id = 1;

XML_PRODUCTS
-----
<products>
  <product>
    <product_id>1</product_id>
    <name>Supernova video</name>
    <quantity>5</quantity>
  </product>
</product>

```

```

    <product_id>2</product_id>
    <name>Oracle SQL book</name>
    <quantity>4</quantity>
  </product>
</products>

```

A próxima consulta recupera o produto nº 2 de `xml_purchase_order`; observe que `product[2]` retorna o produto nº 2:

```

SELECT
  EXTRACT(
    xml_purchase_order,
    '/purchase_order/products/product[2]'
  ) xml_product
FROM purchase_order
WHERE purchase_order_id = 1;

XML_PRODUCT
-----
<product>
  <product_id>2</product_id>
  <product>Oracle SQL book</name>
  <quantity>4</quantity>
</product>

```

A consulta a seguir recupera o produto “Supernova video” de `xml_purchase_order`. Note que o nome do produto a ser recuperado é colocado entre colchetes:

```

SELECT
  EXTRACT(
    xml_purchase_order,
    '/purchase_order/products/product[name="Supernova video"]'
  ) xml_product
FROM purchase_order
WHERE purchase_order_id = 1;

XML_PRODUCT
-----
<product>
  <product_id>1</product_id>
  <name>Supernova video</name>
  <quantity>5</quantity>
</product>

```

A função `EXISTSNODE()` verifica se um elemento XML existe: ela retorna 1 se o elemento existe; caso contrário, ela retorna 0. Por exemplo, a consulta a seguir retorna a string 'Exists', pois o produto nº 1 existe:

```

SELECT 'Exists' AS "EXISTS"
FROM purchase_order
WHERE purchase_order_id = 1
AND EXISTSNODE(

```

```

    xml_purchase_order,
    '/purchase_order/products/product[product_id=1]'
) = 1;

EXISTS
-----
Exists

```

A próxima consulta não retorna uma linha porque o produto nº 3 não existe:

```

SELECT 'Exists'
FROM purchase_order
WHERE purchase_order_id = 1
AND EXISTSNODE(
    xml_purchase_order,
    '/purchase_order/products/product[product_id=3]'
) = 1;

no rows selected

```

A consulta a seguir recupera os produtos como um varray de objetos XMLType usando a função XMLSEQUENCE(). Observe o uso de product.\* para recuperar todos os produtos e seus elementos XML:

```

SELECT product.*
FROM TABLE(
    SELECT
        XMLSEQUENCE(EXTRACT(xml_purchase_order, '/purchase_order//product'))
    FROM purchase_order
    WHERE purchase_order_id = 1
) product;

COLUMN_VALUE
-----
<product>
  <product_id>1</product_id>
  <name>Supernova video</name>
  <quantity>5</quantity>
</product>

<product>
  <product_id>2</product_id>
  <name>Oracle SQL book</name>
  <quantity>4</quantity>
</product>

```

A próxima consulta recupera os valores de product\_id, name, e quantity dos produtos como strings usando a função EXTRACTVALUE():

```

SELECT
    EXTRACTVALUE(product.COLUMN_VALUE, '/product/product_id') AS product_id,
    EXTRACTVALUE(product.COLUMN_VALUE, '/product/name') AS name,

```

```

    EXTRACTVALUE(product.COLUMN_VALUE, '/product/quantity') AS quantity
FROM TABLE (
    SELECT
        XMLSEQUENCE(EXTRACT(xml_purchase_order, '/purchase_order//product'))
    FROM purchase_order
    WHERE purchase_order_id = 1
) product;

PRODUCT_ID
-----
PRODUCT
-----
QUANTITY
-----
1
Supernova video
5
2
Oracle SQL book
4

```

## Atualizando informações no esquema de exemplo XML

As colunas `customer_order_id`, `order_date`, `customer_name`, `street`, `city`, `state`, `zip`, `phone_number` e `products` da tabela `purchase_order` estão vazias. Os dados dessas colunas podem ser extraídos do código XML da coluna `xml_purchase_order`. Nesta seção, você verá uma procedure PL/SQL chamada `update_purchase_order()` que lê o código XML e configura as outras colunas de forma correspondente.

O aspecto mais complexo de `update_purchase_order()` é o processo de leitura dos produtos do código XML e seu armazenamento na coluna da tabela aninhada `products` da tabela `purchase_order`. Nessa procedure, um cursor é usado para ler os produtos do código XML, depois o código XML é convertido em strings com `EXTRACTVALUE()` e as strings são armazenadas em uma tabela aninhada.

A instrução a seguir (contida no script `xml_schema.sql`) cria a procedure `update_purchase_order()`:

```

CREATE PROCEDURE update_purchase_order(
    p_purchase_order_id IN purchase_order.purchase_order_id%TYPE
) AS
    v_count INTEGER:= 1;

    -- declara uma tabela aninhada para armazenar produtos
    v_nested_table_products t_nested_table_product:=
        t_nested_table_product();

    -- declara um tipo para representar um registro de produto
    TYPE t_product_record IS RECORD (
        product_id INTEGER,
        name VARCHAR2(15),
        quantity INTEGER
    );

```

```

-- declara um tipo REF CURSOR para apontar para registros de produto
TYPE t_product_cursor IS REF CURSOR RETURN t_product_record;

-- declara um cursor
v_product_cursor t_product_cursor;

-- declara uma variável para armazenar um registro de produto
v_product t_product_record;
BEGIN
-- abre v_product_cursor para ler o valor de product_id, name e quantity de
-- cada produto armazenado no código XML da coluna xml_purchase_order
-- da tabela purchase_order
OPEN v_product_cursor FOR
SELECT
  EXTRACTVALUE(product.COLUMN_VALUE, '/product/product_id')
    AS product_id,
  EXTRACTVALUE(product.COLUMN_VALUE, '/product/name') AS name,
  EXTRACTVALUE(product.COLUMN_VALUE, '/product/quantity') AS quantity
FROM TABLE(
  SELECT
    XMLSEQUENCE(EXTRACT(xml_purchase_order, '/purchase_order//product'))
  FROM purchase_order
  WHERE purchase_order_id = p_purchase_order_id
) product;

-- loop pelo conteúdo de v_product_cursor
LOOP
-- busca os registros de produto de v_product_cursor e sai quando
-- não encontra mais registros
FETCH v_product_cursor INTO v_product;
EXIT WHEN v_product_cursor%NOTFOUND;

-- estende v_nested_table_products para que um produto possa ser
armazenado aí
v_nested_table_products.EXTEND;

-- cria um novo produto e o armazena em v_nested_table_products
v_nested_table_products(v_count) :=
  t_product(v_product.product_id, v_product.name, v_product.quantity);

-- exibe o novo produto armazenado em v_nested_table_products
DBMS_OUTPUT.PUT_LINE('product_id = ' ||
  v_nested_table_products(v_count).product_id);
DBMS_OUTPUT.PUT_LINE('name = ' ||
  v_nested_table_products(v_count).name);
DBMS_OUTPUT.PUT_LINE('quantity = ' ||
  v_nested_table_products(v_count).quantity);
-- incrementa v_count, pronto para a próxima iteração do loop
v_count:= v_count + 1;
END LOOP;

-- fecha v_product_cursor

```

```

CLOSE v_product_cursor;

-- atualiza a tabela purchase_order usando os valores extraídos do código
-- XML armazenado na coluna xml_purchase_order (a tabela aninhada
-- products é configurada como v_nested_table_products, já preenchida pelo
-- loop anterior)
UPDATE purchase_order
SET
  customer_order_id =
    EXTRACTVALUE(xml_purchase_order,
      '/purchase_order/customer_order_id'),
  order_date =
    TO_DATE(EXTRACTVALUE(xml_purchase_order,
      '/purchase_order/order_date'), 'YYYY-MM-DD'),
  customer_name =
    EXTRACTVALUE(xml_purchase_order, '/purchase_order/customer_name'),
  street =
    EXTRACTVALUE(xml_purchase_order, '/purchase_order/street'),
  city =
    EXTRACTVALUE(xml_purchase_order, '/purchase_order/city'),
  state =
    EXTRACTVALUE(xml_purchase_order, '/purchase_order/state'),
  zip =
    EXTRACTVALUE(xml_purchase_order, '/purchase_order/zip'),
  phone_number =
    EXTRACTVALUE(xml_purchase_order, '/purchase_order/phone_number'),
  products = v_nested_table_products
WHERE purchase_order_id = p_purchase_order_id;

-- confirma a transação
COMMIT;
END update_purchase_order;
/

```

O exemplo a seguir ativa a saída do servidor e chama `update_purchase_order()` para atualizar a ordem de compra nº 1:

```

SET SERVEROUTPUT ON
CALL update_purchase_order(1);
product_id = 1
name = Supernova video
quantity = 5
product_id = 2
name = Oracle SQL book
quantity = 4

```

A consulta a seguir recupera as colunas do pedido de compra número 1:

```

SELECT purchase_order_id, customer_order_id, order_date, customer_name,
       street, city, state, zip, phone_number, products
FROM purchase_order

```



```
WHERE purchase_order_id = 1;
PURCHASE_ORDER_ID CUSTOMER_ORDER_ID ORDER_DAT CUSTOMER_NAME
-----
STREET              CITY              ST ZIP    PHONE_NUMBER
-----
PRODUCTS(PRODUCT_ID, NAME, QUANTITY)
-----
                1              176 17-MAY-07 Best Products 456 Inc.
10 Any Street    Any City              CA 94440    555-121-1234
T_NESTED_TABLE_PRODUCT(
  T_PRODUCT(1, 'Supernova video', 5),
  T_PRODUCT(2, 'Oracle SQL book', 4)
)
```

A tabela aninhada `products` contém os mesmos dados armazenados nos elementos de produto XML da coluna `xml_purchase_order`. Algumas quebras de linha foram adicionadas para separar os produtos nos resultados do exemplo para facilitar a leitura.

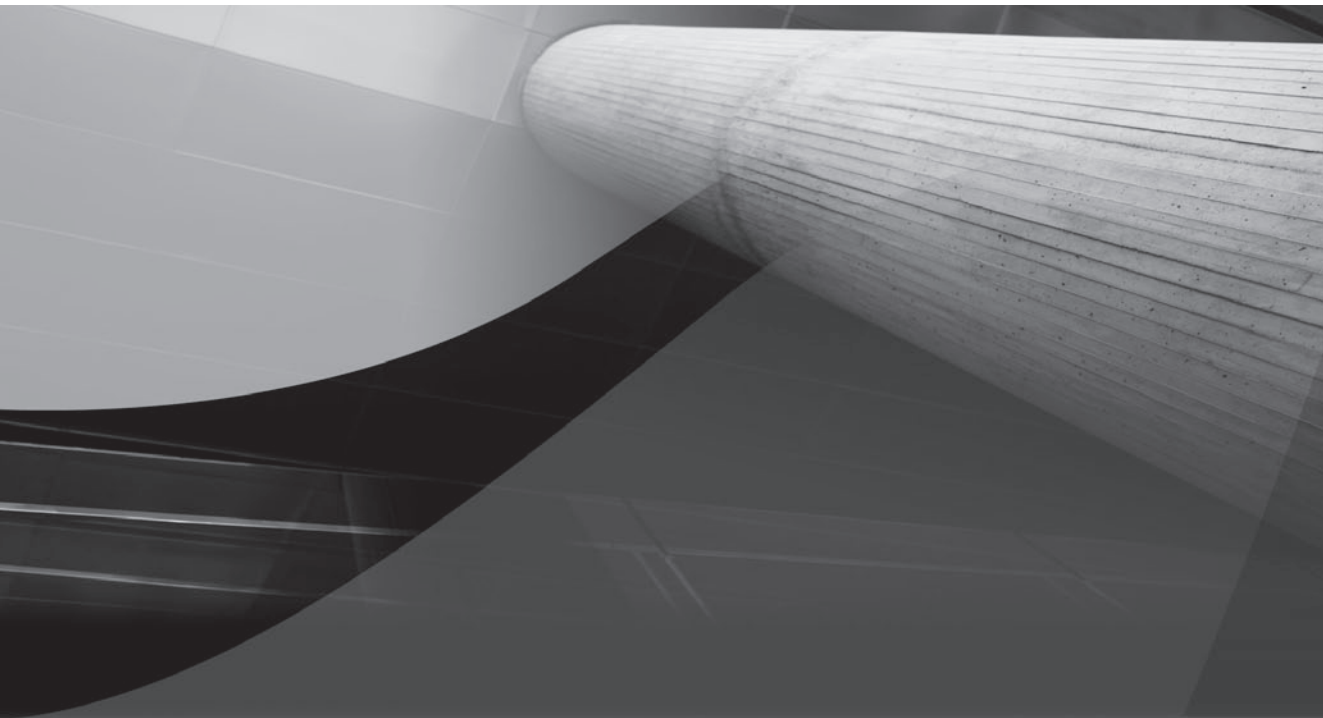
## RESUMO

Neste capítulo, você aprendeu a:

- Gerar código XML a partir de dados relacionais
- Salvar código XML no banco de dados e, depois, ler esse código para atualizar colunas relacionais

Este breve capítulo é apenas uma introdução à rica funcionalidade de XML disponível no banco de dados Oracle. Para mais informações, consulte os livros *Oracle XML Developer's Kit* e *Oracle XML DB Developer's Guide*, ambos publicados pela Oracle Corporation.

Chegamos ao fim do livro. Espero que esta obra tenha sido instrutiva e útil e que você tenha gostado de lê-la.



# APÊNDICE

## Tipos de dados Oracle

Este apêndice contém duas tabelas documentando os tipos de dados disponíveis na linguagem SQL do Oracle, que podem ser usados para definir colunas em uma tabela, junto com os tipos adicionais suportados pelo PL/SQL do Oracle.

TIPOS SQL DO ORACLE

A Tabela A-1 mostra os tipos SQL do Oracle.

Tabela A-1 Tipos SQL do Oracle

Tipo	Descrição
CHAR [( <i>comprimento</i> [BYTE   CHAR])]1	Dados de caractere de comprimento fixo bytes de <i>comprimento</i> ou caracteres e preenchimento com espaços à direita. O comprimento máximo é de 2.000 bytes.
VARCHAR2 ( <i>comprimento</i> [BYTE   CHAR])1	Dados de caractere de comprimento variável de até <i>comprimento</i> bytes ou caracteres. O comprimento máximo é de 4.000 bytes.
NCHAR [( <i>comprimento</i> )]	Dados de caractere Unicode de comprimento fixo de <i>comprimento</i> caracteres. O número de bytes armazenados é 2 multiplicado por <i>comprimento</i> para codificação AL16UTF16 e 3 multiplicado por <i>comprimento</i> para codificação UTF8. O comprimento máximo é de 2.000 bytes.
NVARCHAR2 ( <i>comprimento</i> )	Dados de caractere Unicode de comprimento variável de <i>comprimento</i> caracteres. O número de bytes armazenados é 2 multiplicado por <i>comprimento</i> para codificação AL16UTF16 e 3 multiplicado por <i>comprimento</i> para codificação UTF8. O comprimento máximo é de 4.000 bytes.
BINARY_FLOAT	Introduzido no Oracle Database 10g, armazena um número de ponto flutuante de 32 bits e precisão simples. Normalmente, as operações envolvendo BINARY_FLOAT são executadas mais rapidamente do que as operações que usam valores NUMBER. BINARY_FLOAT exige 5 bytes de espaço de armazenamento.
BINARY_DOUBLE	Introduzido no Oracle Database 10g, armazena um número de ponto flutuante de 64 bits e precisão dupla. Normalmente, as operações envolvendo BINARY_DOUBLE são executadas mais rapidamente do que as operações que usam valores NUMBER. BINARY_DOUBLE exige 9 bytes de espaço de armazenamento.
NUMBER ( <i>precisão</i> , <i>escala</i> ) e NUMERIC ( <i>precisão</i> , <i>escala</i> )	Número de comprimento variável; <i>precisão</i> é o número máximo de dígitos (à esquerda e à direita de um ponto decimal, se for usado) que podem ser usados para o número. A precisão máxima suportada é 38; <i>escala</i> é o número máximo de dígitos à direita de um ponto decimal (se for usado). Se nem <i>precisão</i> nem <i>escala</i> forem especificadas, então um número com uma precisão e uma escala de até 38 dígitos pode ser fornecido (significando que você pode fornecer um número com até 38 dígitos e que qualquer um desses 38 dígitos pode estar à direita ou à esquerda do ponto decimal.
DEC e DECIMAL	Subtipo de NUMBER. Um número decimal de ponto flutuante fixo com até 38 dígitos de precisão decimal.
DOUBLE PRECISION e FLOAT	Subtipo de NUMBER. Um número em ponto flutuante com até 38 dígitos de precisão.
REAL	Subtipo de NUMBER. Um número em ponto flutuante com até 18 dígitos de precisão.
INT, INTEGER, e SMALLINT	Subtipo de NUMBER. Um número inteiro com até 38 dígitos de precisão decimal.
DATE	Data e hora com o século; todos os quatro dígitos do ano, mês, dia, hora (no formato de 24 horas), minuto e segundo. Pode ser usado para armazenar uma data e hora entre 1° de janeiro de 4.712 a.C. e 31 de dezembro de 4.712 d.C. O formato padrão é especificado pelo parâmetro de banco de dados NLS_DATE_FORMAT (por exemplo: DD-MMM-AA).
INTERVAL YEAR [( <i>precisão_anos</i> )] TO MONTH	Intervalo de tempo medido em anos e meses; <i>precisão_anos</i> especifica a precisão dos anos, que pode ser um número inteiro de 0 a 9 (o padrão é 2). Pode ser usado para representar um intervalo de tempo positivo ou negativo.

(continua)

**Tabela A-1** *Tipos SQL do Oracle (continuação)*

Tipo	Descrição
INTERVAL DAY [(precisão_dias)] TO SECOND [(precisão_segundos)]	Intervalo de tempo medido em dias e segundos; <i>precisão_dias</i> especifica a precisão dos dias, que é um valor inteiro de 0 a 9 (o padrão é 2); <i>precisão_segundos</i> especifica a precisão da parte fracionária dos segundos, que é um valor inteiro 0 a 9 (o padrão é 6). Pode ser usado para representar um intervalo de tempo positivo ou negativo.
TIMESTAMP [(precisão_segundos)]	Data e hora com o século; todos os quatro dígitos do ano, mês, dia, hora (no formato de 24 horas), minuto e segundos; <i>precisão_segundos</i> especifica o número de dígitos da parte fracionária dos segundos, que pode ser um valor inteiro de 0 a 9 (o padrão é 6). O formato padrão é especificado pelo parâmetro de banco de dados NLS_TIMESTAMP_FORMAT.
TIMESTAMP [(precisão_segundos)] WITH TIME ZONE	Estende TIMESTAMP para armazenar um fuso horário. O fuso horário pode ser uma diferença em relação UTC, como -8:0, ou um nome de região, como US/Pacific ou PST. O formato padrão é especificado pelo parâmetro de banco de dados NLS_TIMESTAMP_TZ_FORMAT.
TIMESTAMP [(precisão_segundos)] WITH LOCAL TIME ZONE	Estende TIMESTAMP para converter um valor de data/horário fornecido no fuso horário local definido para o banco de dados. O processo de conversão é conhecido como <i>normalização</i> do valor de data/horário. O formato padrão é especificado pelo parâmetro de banco de dados NLS_TIMESTAMP_FORMAT.
CLOB	Dados de caracteres de um byte e comprimento variável de até 128 terabytes.
NCLOB	Dados do conjunto de caracteres Unicode nacional de comprimento variável de até 128 terabytes.
BLOB	Dados binários de comprimento variável de até 128 terabytes.
BFILE	Ponteiro para um arquivo externo. O arquivo externo não é armazenado no banco de dados.
LONG	Dados de caractere de comprimento variável de até 2 gigabytes. Substituído pelos tipos CLOB e NCLOB, mas suportado para compatibilidade com versões anteriores.
RAW (comprimento)	Dados binários de comprimento variável de até <i>comprimento</i> bytes. O comprimento máximo é de 2.000 bytes. Substituído pelo tipo BLOB, mas suportado para compatibilidade com versões anteriores.
LONG RAW	Dados binários de até 2 gigabytes. Substituído pelo tipo BLOB, mas suportado para compatibilidade com versões anteriores.
ROWID	String hexadecimal usada para representar um endereço de linha.
UROWID [(comprimento)]	String hexadecimal representando o endereço lógico de uma linha de uma tabela indexada; <i>comprimento</i> especifica o número de bytes. O comprimento máximo é de 4.000 bytes (também será o comprimento padrão, se nenhum for especificado).
REF tipo_objeto	Referência a um tipo de objeto. Semelhante a um ponteiro na linguagem de programação C++.
VARRAY	Array de comprimento variável. Este é um tipo composto e armazena um conjunto ordenado de elementos.
NESTED TABLE	Tabela aninhada. Este é um tipo composto e armazena um conjunto não ordenado de elementos.
XMLType	Armazena dados XML.
Tipo de objeto definido pelo usuário	Você pode definir seu próprio tipo de objeto e criar objetos desse tipo. Para ver os detalhes, consulte o Capítulo 12.

<sup>1</sup> As palavras-chave BYTE e CHAR funcionam apenas com o Oracle Database 9i e versões superiores. Se nem BYTE nem CHAR forem especificados, o padrão será BYTE.

# TIPOS PL/SQL DO ORACLE

O PL/SQL do Oracle suporta todos os tipos mostrados anteriormente na Tabela A-1, além dos seguintes tipos adicionais, específicos PL/SQL, mostrados na Tabela A-2.

**Tabela A-2** *Tipos PL/SQL do Oracle*

Tipo	Descrição
BOOLEAN	Valor booleano (TRUE, FALSE ou NULL).
BINARY_INTEGER	Valor inteiro entre $-2^{31}$ (−2.147.483.648) e $2^{31}$ (2.147.483.648).
NATURAL	Subtipo de BINARY_INTEGER. Um valor inteiro não negativo.
NATURALN	Subtipo de BINARY_INTEGER. Um valor inteiro não negativo (não pode ser NULL).
POSITIVE	Subtipo de BINARY_INTEGER. Um valor inteiro positivo.
POSITIVEN	Subtipo de BINARY_INTEGER. Um valor inteiro positivo (não pode ser NULL).
SIGNTYPE	Subtipo de BINARY_INTEGER. Um valor inteiro configurado como −1, 0 ou 1.
PLS_INTEGER	Valor inteiro entre $-2^{31}$ (−2.147.483.648) e $2^{31}$ (2.147.483.648). Idêntico a BINARY_INTEGER.
SIMPLE_INTEGER	Novidade do Oracle Database 11g, SIMPLE_INTEGER é um subtipo de BINARY_INTEGER. SIMPLE_INTEGER pode armazenar o mesmo intervalo de valores que BINARY_INTEGER, exceto quanto aos valores NULL, que não podem ser armazenados em um SIMPLE_INTEGER. Além disso, um estouro aritmético não causa um erro ao se usar valores SIMPLE_INTEGER; em vez disso, o resultado é simplesmente truncado.
STRING	Igual a VARCHAR2.
RECORD	Composição de um grupo de outros tipos. Semelhante a uma estrutura na linguagem de programação C++.
REF CURSOR	Ponteiro para um conjunto de linhas.

# Índice

## SÍMBOLOS

- (menos), 59
- # (número), 135
- % (porcentagem), 68–69
- %TYPE, 370, 374
- () (parênteses), 62, 196
- \* (asterisco)
  - metacaracteres, 138
  - na aritmética, 59
  - recuperação de coluna, 57–58
  - tabela customers, 43
- / (barra normal)
  - caminhos de diretório SQL, 38
  - editando instruções SQL, 93–94
  - instruções PL/SQL, 369
  - na aritmética, 59
- \ (barra invertida), 38
- ; (ponto-e-vírgula), 34, 369
- ? (ponto de interrogação), 592
- != (diferente), 66–67
- @ nome\_arquivo, 95
- \_ (sublinhado), 68–69
- || (operador de concatenação), 62
- + (mais)
  - joins externas, 79–83
  - na aritmética, 59

- < (menor do que)
  - comparação de valor, 66–67
  - comparando valores de objeto, 421
  - não-equiijoins, 78
- <= (menor ou igual)
  - comparação de valor, 66–67
  - comparando valores de objeto, 421
  - não-equiijoins, 78
- <> (diferente)
  - comparação de valor, 66–67
  - comparando valores de objeto, 420–422
  - suporte para tabela aninhada ANSI, 492–493
- = (igual)
  - comparação de valor, 66–67
  - comparando valores de objeto, 420–421
  - equiijoins, 78
  - suporte para tabela aninhada ANSI, 492–493
- > (maior do que)
  - comparação de valor, 66–67
  - comparando valores de objeto, 421–422
  - não-equiijoins, 78

- >= (maior ou igual)
  - comparação de valor, 66–67
  - comparando valores de objeto, 421
  - não-equiijoins, 78
- “ (aspas), 62, 282

---

## A

- A[PPEND], 93–94
- abrindo cursores, 374
- ABS(), 126, 128
- ACCEPT, 106–109
- ACCESS\_INT0\_NULL, 382
- ACOS(), 126
- ADD\_MONTHS(), 170–171
- ADDM (Automatic Database Diagnostic Monitor), 628–629
- agrupando funções
  - agrupando linhas, 148–155
  - tipos de, 145–148
- ajuda, SQL\*Plus, 113–114
- ajuste de SQL (Structured Query Language), 607–629
  - adicionando índices em tabelas, 612
  - comparação de custos de consulta, 619–626

EXISTS *versus* DISTINCT, 615  
 EXISTS *versus* IN, 614–615  
 expressões CASE *versus* várias consultas, 611  
 ferramentas adicionais, 628–629  
 filtrando linhas com WHERE, 608–609  
 GROUPING SETS *versus* CUBE, 616  
 introdução, 608  
 joins com referências de coluna qualificadas, 610  
 joins de tabela *versus* várias consultas, 609–610  
 passando dicas para o otimizador, 626–628  
 resumo, 629  
 UNION ALL *versus* UNION, 613–614  
 variáveis de bind, 616–619  
 WHERE *versus* HAVING, 612–613  
 algoritmos, criptografia de dados LOB, 554  
 ALL, 66–67, 203  
 ALTER  
   definido, 31  
   INDEX, 352  
   SEQUENCE, 348  
   TABLE, 330–334  
 alterando conteúdo de tabela. *Consulte* conteúdo de tabela, alterando  
 AND, 71, 266  
 anos  
   formato padrão de data/horário, 158  
   interpretação de ano de dois dígitos, 168–170  
   INTERVAL YEAR TO MONTH, 187–190  
   parâmetros de formatação de data/horário, 161  
 ANSI (American National Standards Institute), 75, 492–500  
 ANY  
   acessando todas as células com, 266–267  
   comparação de valor, 66–67  
   subconsultas de várias linhas, 202  
 apelidos  
   colunas, 62–63  
   tabela, 75–76

apelidos de tabela  
   autojoins, 83  
   definidos, 75–76  
 APPEND()  
   dados para CLOBs, 534–535  
   definido, 511–513  
 aritmética, 59–62  
 aritmética de data, 59–60  
 armazenamento  
   data/horário. *Consulte* data/horário  
   definido, 51  
   linhas com variável de bind, 618–619  
   números em Java, 577–578  
   tabelas, 491–492  
   valores retornados com variável de bind, 618  
 arquivos  
   BFILES. *Consulte* BFILES (tipos binários FILE)  
   copiando dados de, em CLOBs e BLOBs, 540–543  
   salvando/recuperando/executando, 94–98  
 arrays, associativos, 456  
   associativos, 490  
 arrays associativos, 456, 490  
 arrays, varray. *Consulte* varrays  
 arredondamento transparente, 51  
 ASCII(), 119–120  
 ASIN(), 126  
 aspas ("), 62, 282  
 asterisco (\*). *Consulte* \* (asterisco)  
 ATAN(), 126  
 ATAN2(), 127  
 ativando constraints, 337–338  
 ativando triggers, 402  
 atribuições  
   concedendo privilégios, 318  
   concedendo privilégios de objeto, 320–321  
   concedidas a usuários, 318–319  
   criando, 317  
   eliminando, 323  
   padrão, 322  
   resumo, 325–326  
   revogando, 322  
   uso de privilégio concedido, 321  
   verificando privilégios, 319–320  
 atribuições padrão, 322

atributos  
   :new, 552–553  
   OracleDataSource, 568–571  
   XMLATTRIBUTES(), 636  
 atributo :new, 552–553  
 atualizando  
   esquema XML, 658–661  
   RULES UPDATE, 271–272  
 auditoria  
   usando triggers, 397–402  
   visão geral, 323–325  
 autojoins, 78, 83–84, 88  
 Automatic Database Diagnostic Monitor (ADDM), 628–629  
 AVG(), 145–146

## B

bancos de dados  
   abrindo conexão, 565–571  
   criando usuários, 39–40  
   fusos horários, 175–177  
   integridade, 284–286  
   tabelas. *Consulte* tabelas  
   tipos comuns, 40–41  
   transações, 290–297, 579–581  
 bancos de dados relacionais, 30–31  
 barra invertida (\), 38  
 barra normal (/). *Consulte* / (barra normal)  
 BasicExample1.java, 584–590  
 BasicExample2.java, 593–595  
 BasicExample3.java, 603–605  
 BETWEEN  
   acesso a intervalo de células usando, 266  
   comparando valores de objeto, 421  
   definido, 67  
   não-equi-joins, 78–79  
   visão geral, 70–71  
 BFILES (tipos binários FILE)  
   copiando dados de, em CLOBs e BLOBs, 545–548  
   definidos, 504–506  
   FILECLOSE() e FILECLOSE-ALL(), 516–517  
   FILEEXISTS(), 517–518  
   FILEGETNAME(), 517–518  
   FILEISOPEN(), 517–519  
   FILEOPEN(), 519  
   LOADBLOBFROMFILE(), 525–526

LOADCLOBFROMFILE(), 525-526  
 LOADFROMFILE(), 523-524  
 usando em SQL, 509-510  
 Binary LOBs (BLOBs). *Consulte*  
 BLOBs (binary LOBs)  
 BINARY\_DOUBLE, 41, 51-53  
 BINARY\_FLOAT, 41, 51-53  
 binds, 567  
 BITAND(), 127  
 BLOBs (binary LOBs)  
 compactação, 557  
 copiando dados de, em arquivos, 543-545  
 copiando dados de BFILE em, 545-548  
 criptografia, 554-556  
 definidos, 504-506  
 lendo dados de, 532-534  
 LOADBLOBFROMFILE(), 525-526  
 usando em SQL, 506-509  
 bloqueio de transação, 294-295  
 bloqueio padrão, 294-295  
 bloqueios, transação, 294-295  
 BREAK ON, 111-113  
 BTITLE, 110-111

## C

C[HANGE], 93-94  
 cabeçalho HEA[DING], 98-99  
 cabeçalhos, 110-111  
 cache de dados de LOB, 554  
 CACHE em seqüências, 347  
 calculando subtotaís, 111-113  
 cálculos com MODEL. *Consulte*  
 MODEL  
 cálculos numéricos, 126-130  
 caminhos de diretório, 38  
 caracteres  
 arquivos XML escritos em inglês, 653  
 definição de variável, 104  
 caracteres de definição, 104  
 CARDINALITY(), 497  
 CASE\_NOT\_FOUND, 382  
 CAST(), 136-137, 471-472  
 CEIL(), 127, 128  
 células  
 acessando todas usando ANY e IS ANY, 266-267  
 acesso a intervalo usando BETWEEN e AND, 266  
 atualizando existentes, 271-272

loops FOR, 268-269  
 notação posicional e simbólica, 265-266  
 chamando funções PL/SQL, 392, 394-396  
 chamando procedures PL/SQL, 388-389, 394-396  
 CHAR(), 41  
 chaves  
 arrays associativos, 490  
 primária e estrangeira, 285-286  
 chaves estrangeiras  
 adicionando, 336-337  
 definidas, 334  
 imposição de constraints, 285-286  
 chaves primárias  
 definidas, 334  
 imposição de constraints, 285  
 preenchendo com seqüências, 347  
 CHECK, 334, 335  
 CHECK OPTION, 334, 357  
 CHR(), 119-120  
 classes  
 pacote oracle.jdbc, 601  
 tipos de banco de dados Oracle compatíveis, 596  
 tipos de objeto como, 408  
 classes wrapper, 578-580  
 classificação, funções hipotéticas, 263  
 classificando linhas, 72-74  
 CLE[AR], 98-99  
 CLEAR, 101  
 CLOBs (character LOBs)  
 anexando dados em, 534-535  
 apagando dados de, 538  
 compactação, 557  
 comparando dados, 535-536  
 conversão implícita, 551-552  
 copiando dados de, 536-537  
 copiando dados de, em arquivos, 543-545  
 copiando dados de arquivo em, 540-543  
 copiando dados de BFILE em, 545-548  
 criptografia, 554-556  
 definidos, 504-506  
 gravando em, 534  
 lendo dados de, 532-534  
 LOADCLOBFROMFILE(), 525-526

pesquisando dados, 538-539  
 temporários, 537-538  
 usando em SQL, 506-509  
 CLOSE(), 511-513  
 Codd, E. F., 30, 31  
 coleções, 455-501  
 aprimoramentos do Oracle Database 10g, 489-490  
 arrays associativos, 490  
 aumentando elementos em  
 varray, 491  
 comparando conteúdo de tabela aninhada com método de mapeamento, 468-470  
 convertendo tipos com CAST(), 471-472  
 COUNT(), 476-479  
 de vários níveis, 486-489  
 definindo coluna com tipo, 458  
 DELETE(), 479-480  
 dimensionamento de tipo de elemento, 491  
 em PL/SQL, 472-476  
 EXISTS(), 480-481  
 EXTEND(), 481  
 FIRST(), 482  
 gerando XML para, 635-636  
 introdução, 456-457  
 LAST(), 482-483  
 modificando elementos, 466-468  
 NEXT(), 483-484  
 preenchendo com elementos, 462-463  
 PRIOR(), 484-485  
 recuperação de informações, 459-461  
 recuperando elementos de, 463-464  
 resumo, 501  
 suporte para tabela aninhada ANSI, 492-500  
 TABLE(), 464-466  
 tablespaces de tabela de armazenamento de uma tabela aninhada, 491-492  
 tipos, 456-458  
 TRIM(), 485-486  
 varrays em tabelas temporárias, 491  
 XMLAGG(), 637-640  
 coleções de vários níveis, 486-489  
 COLLECT(), 499-500



- COLLECTION\_IS\_NULL, 382
- coluna numéricas, 333
- colunas
  - acessando usando cláusula MODEL, 264–265
  - adicionando/modificando/removendo, 331–334
  - alterando conteúdo de tabela. *Consulte* conteúdo de tabela, alterando
  - apelidos, 62–63
  - combinando saída, 63
  - comentários, 341–342
  - criptografando dados de LOB, 556
  - declarando variáveis para armazenar valores, 374
  - definidas, 30
  - definindo com tipo de coleção, 458
  - em aritmética, 60–62
  - GROUPING(), 237–239
  - joins com referências totalmente qualificadas, 610
  - lendo valores do objeto ResultSet, 572–574
  - na cláusula ROLLUP, 233–235
  - objetos, 411–414
  - recuperação de informações de constraint, 339–340
  - recuperação de informações em índices, 351–352
  - recuperando todas, 57–58
  - SQL\*Plus, formatando, 98–100
  - SQL\*Plus, limpando formatos, 101
  - subconsultas de várias colunas, 203
  - substituindo nomes usando variáveis, 104–105
  - tabela de plano, 621
  - tipos LONG e LONG RAW, 549–550
  - UNPIVOT, 276–277
  - usando pivô em várias, 274–275
  - usando várias em grupo, 150
- colunas virtuais, 332–333
- comandos
  - editando instruções SQL, 93–94
  - formatação de coluna, 98–100
  - HELP, 113–114
  - relatório simples, 107–113
  - salvando/recuperando/executando arquivos, 94–98
  - SET LINESIZE, 101
  - SET PAGESIZE, 100–101
  - variáveis definidas, 105–107
- combinando funções, 126
- comentários
  - em tabelas, 341–342
  - XML, 642
- COMMIT
  - definido, 32
  - modo de autocommit, 579–581
  - transações, 290–291
- Communications of the ACM*, 30
- compactação, LOB, 557
- comparando valores
  - definido, 65–67
  - objetos de tabela, 420–422
- COMPARE()
  - dados em dois CLOBs, 535–536
  - definido, 511–512, 514–515
- comprimento, 41
- COMPUTE, 111–113
- concatenação
  - combinando saída de coluna, 63
  - CONCAT(), 119–120
  - XMLCONCAT(), 641
- concedendo atribuições, 318–319
- concedendo privilégios
  - a usuários, 306–307
  - atribuições, 318
  - objeto, 310
  - usando com atribuições, 321
- condições, trigger, 397
- condições em consultas hierárquicas, 230–231
- conectividade, banco de dados, 565–571
- configuração do computador para executar SQL usando Java, 561–563
- configurações de tamanho de linha, 101
- configurando o computador para executar SQL usando Java, 561–563
- conjunto de resultados
  - definido, 56
  - recuperando linhas usando JDBC, 571–575
- CONNECT BY, 226
- constraints
  - criando visões com, 357–358
  - impondo, 285–286
  - obtendo informações sobre visão, 359–360
  - tabela, 334–340
- constraints adiadas, 338
- construtores
  - definidos pelo usuário, 446–450
  - objetos de coluna, 412
  - tabelas de objeto, 414
- construtores definidos pelo usuário, 446–450
- consultas
  - ajuste de SQL. *Consulte* ajuste de SQL (Structured Query Language)
  - comparação de custos, 619–626
  - definidas, 56
  - flashbacks, 298–301
  - subconsultas. *Consulte* subconsultas
  - usando JDBC, 571–575
  - XMLQUERY(), 646–650
- consultas avançadas, 211–278
  - acesso à célula usando loop FOR, 268–269
  - acesso a intervalo de célula usando BETWEEN e AND, 266
  - acesso a todas as células usando ANY e IS ANY, 266–267
  - atualizando células existentes, 271–272
  - CUBE, 235–236
  - DECODE(), 219–221
  - eliminando nós e ramos, 229–230
  - exemplo de PIVOT, 272–274
  - expressões CASE, 221–224
  - formatando resultados hierárquicos, 227–228
  - função de classificação hipotética e distribuição, 263
  - funções analíticas, 244–245
  - funções de classificação, 245–252
  - funções de janela, 253–258
  - funções de percentil inversas, 252–253
  - funções de regressão linear, 261–263
  - funções de relatório, 258–260
  - funções FIRST e LAST, 261
  - GROUP BY, 231–232
  - GROUP\_ID(), 243–244
  - GROUPING SETS, 239–240
  - GROUPING(), 237–239

- GROUPING\_ID(), 240–242  
 hierárquicas, 224–227  
 hierárquicas, incluindo outras condições, 230–231  
 iniciando em um nó que não é o raiz, 228  
 LAG() e LEAD(), 260–261  
 MODEL, 264–265  
 múltiplas funções agregadas em pivô, 275–276  
 notação posicional e simbólica, 265–266  
 operadores de conjunto, 212–216  
 operadores de conjunto, combinando, 216–218  
 percorrendo a árvore para cima, 229  
 PIVOT e UNPIVOT, 272  
 resumo, 277–278  
 ROLLUP, 233–235  
 subconsulta START WITH, 228–229  
 TRANSLATE(), 218–219  
 tratando de valores nulos e ausentes, 269–271  
 UNPIVOT, 276–277  
 usando pivô em várias colunas, 274–275  
 uso de coluna GROUP BY, 242  
 valores de dimensão atuais usando CURRENTV(), 267–268  
 consultas em árvore. *Consulte também* consultas hierárquicas definidas, 225  
 eliminando nós e ramos, 229–230  
 percorrendo para cima em, 229  
 consultas flashback de tempo, 298–300  
 consultas hierárquicas definidas, 224–227  
 eliminando nós e ramos, 229–230  
 formatando resultados, 227–228  
 incluindo outras condições, 230–231  
 iniciando em um nó que não é o raiz, 228  
 percorrendo a árvore para cima, 229  
 subconsulta START WITH, 228–229
- CONTENT  
 parsing de expressões XML, 641  
 XMLSERIALIZE(), 644  
 conteúdo de tabela, alterando, 279–301  
 adicionando linhas, 280–282  
 consultas flashback, 298–301  
 integridade do banco de dados, 284–286  
 mesclando linhas, 287–290  
 modificando linhas, 282–283  
 removendo linhas, 284  
 resumo, 301  
 RETURNING, 283–284  
 transações de banco de dados, 290–297  
 valores padrão, 286–287  
 conversão implícita, 551–552  
 convertendo data/horário, 159–167, 178  
 CONVERTTOBLOB(), 511–512  
 CONVERTTOCLOB(), 511–512  
 copiando dados de LOB  
 anexando dados em CLOBs, 534–535  
 de arquivo em CLOBs e BLOBs, 540–543  
 de BFILE para CLOBs e BLOBs, 545–548  
 de CLOBs e BLOBs para arquivos, 543–545  
 copiando linhas, 282  
 COPY()  
 copiando dados de CLOBs, 536–537  
 definido, 511–512  
 métodos de LOB, 515  
 corpo, pacote, 393–395  
 corpo, tipo de objeto, 409–410  
 COS(), 127  
 COSH(), 127  
 COUNT(), 145–147, 476–479  
 CREATE, 31, 39–40  
 CREATE PROCEDURE, 53–54  
 CREATE SYNONYM, 315–316  
 CREATE USER, 304–305  
 CREATETEMPORARY(), 511–512, 516  
 criptografia, LOB, 553–556  
 CUBE  
 definido, 235–236  
 GROUPING() com, 238–239  
 usando com funções analíticas, 249–251  
 versus GROUPING SETS, 240, 616
- CUME\_DIST(), 246, 251  
 CURRENT\_DATE(), 177  
 CURRENT\_TIMESTAMP, 183–184  
 CURRENTV(), 267–268  
 curval, 345, 404  
 CURSOR\_ALREADY\_OPEN, 382  
 cursores  
 exemplo product\_cursor.sql, 376–377  
 instrução OPEN-FOR, 378–379  
 loops FOR e, 377–378  
 passos de programação, 374–376  
 visão geral, 373–374  
 cursores irrestritos, 380–381  
 cursores restritos, 373–379  
 customer\_id, 42, 44–46
- 
- ## D
- 
- dados, LOB  
 anexando em CLOB, 534–535  
 apagando de CLOB, 538  
 CLOBs e BLOBs em SQL, 506–509  
 compactação, 557  
 comparando em CLOBs, 535–536  
 copiando. *Consulte* copiando dados de LOB  
 criptografia, 553–556  
 definidos, 506  
 lendo de CLOBs e BLOBs, 532–534  
 pesquisando em CLOB, 538–539  
 removendo duplicados, 557  
 dados de LOB duplicados, 557  
 dados relacionais, gerando XML. *Consulte* XML (Extensible Markup Language)  
 data de nascimento (dob), 43  
 data/horário, 158–194  
 anos de dois dígitos, 168–170  
 armazenando/recuperando, 158–159  
 convertendo, 159–167  
 formato padrão, 167–168  
 funções, 170–174  
 funções de timestamp, 182–187  
 fusos horários, 174–178  
 intervalos de tempo, 187–193  
 resumo, 194  
 timestamp, 178–182

DATE, 41  
 DD, 158  
 declarações, cursor, 374  
 DECODE(), 219–221  
 DEFINE, 105–106  
 definições, visão, 358–359  
 DEL, 93–94  
 DELETE  
   definido, 31  
   instruções contendo subconsultas, 208–209  
   JDBC, 576  
   removendo linhas, 50–51, 284  
 DELETE(), 477, 479–480  
 delete\_product(), 423, 428–429  
 DENSE RANK(), 245–251  
 depósitos, 251  
 Deref(), 419–420  
 desativando constraints, 337  
 desativando triggers, 402  
 DESCRIBE  
   adicionando linhas, 48  
   identificadores de linha, 58–59  
   informações de tipo de objeto, 410–411  
   recuperação de informações de coleção, 459–461  
   recuperação de informações de tabela, 330  
   vendo estrutura de tabela, 92–93  
 descrição, 44–45  
 deslocamentos, fuso horário, 177  
 dias  
   formato padrão, 158  
   funções de data/horário, 171–173  
   INTERVAL DAY TO SECOND, 190–192  
   parâmetros de formato, 162  
 dicas, passando para o otimizador, 626–628  
 diferente (!=), 66–67  
 diferente (<>). *Consulte* <> (diferente)  
 dimensionando colunas, 333  
 dimensionando tipos de elemento, 491  
 dimensões, valores atuais, 267–268  
 display\_product(), 422, 424–425  
 DISTINCT  
   definido, 65  
   *versus* EXISTS, 615

distribuição, 263  
 dob (data de nascimento), 43  
 DOCUMENT, 641, 644  
 driver interno server-side, 564  
 driver Thin, 563–564  
 driver Thin server-side, 564  
 drivers, JDBC  
   registrando, 565  
   tipos de, 563–564  
 drivers OCI (Oracle Call Interface), 564  
 drivers Oracle Call Interface (OCI), 564  
 DROP, 31  
 DROP TABLE, 114  
 DROP USER, 306  
 DUP\_VAL\_ON\_INDEX, 382, 384

## E

ED[IT], 95  
 editando instruções SQL, 35, 93–94  
 editores padrão, 97  
 elementos  
   adicionando em coleção, 481  
   aumentando em varray, 491  
   contando, 476–479  
   dimensionamento de tipo, 491  
   excluindo, 479–480  
   existentes, 480–481  
   modificando em coleções, 466–468  
   obtendo o primeiro, 482  
   obtendo o último, 482–483  
   preenchendo coleções, 462–463  
   recuperando de coleções, 463–464  
   removendo de coleção, 485–486  
   retornando o anterior, 484–485  
   retornando o próximo, 483–484  
   XML, 633–636  
 eliminação de ramo, 229–230  
 eliminando  
   atribuições, 323  
   colunas, 334  
   constraints, 337  
   funções PL/SQL, 393  
   índices, 352  
   pacotes, 396  
   procedures, 390  
   seqüências, 348  
   tabelas, 342  
   triggers, 402  
   visões, 362  
 ELSE, 370–371  
 ELSIF, 370–371  
 END IF, 370–371  
 equijoints  
   definidas, 74  
   instruções SELECT usando várias tabelas, 77–78  
   sintaxe SQL/92, 85–86  
 ERASE()  
   dados de CLOB, 538  
   definido, 511–512  
   métodos de LOB, 516–517  
 erros  
   exceções. *Consulte* exceções subconsultas de uma linha, 200–201  
   vendo em procedures, 390  
 escala, 41  
 especificações, pacote, 393–394, 423  
 esquemas, 30  
 esquemas, loja  
   instruções DDL, 39–48  
   introdução, 38–39  
 esquemas, XML  
   atualizando informações, 658–661  
   criando, 651–653  
   recuperação de informações, 653–658  
 esquemas da loja  
   instruções DDL, 39–48  
   introdução, 38–39  
 estatísticas na otimização baseada em custo, 625  
 estrutura de bloco, 368–370  
 exceções  
   DUP\_VAL\_ON\_INDEX, 384  
   executando SQL usando Java, 581–582  
   INVALID\_NUMBER, 384–385  
   lançadas por métodos. *Consulte* métodos  
   OTHERS, 385  
   PL/SQL, 381–383  
   ZERO\_DIVIDE, 383–384  
 exceções OTHERS, 385  
 exceções predefinidas, 382–383  
 exceções ZERO\_DIVIDE, 383–384  
 exemplo product\_cursor.sql, 376–377

exibindo linhas distintas, 65  
 EXISTS  
   subconsultas correlacionadas, 204–207  
   *versus* DISTINCT, 615  
   *versus* IN, 614–615  
 EXISTS(), 477, 480–481  
 EXIT, 53, 114  
 EXP(), 127  
 EXPLAIN PLAN, 622  
 expressão XQuery, 646–650  
 expressões  
   aritméticas, 59–62  
   CASE. *Consulte* expressões CASE  
   funções de expressão regular, 137–144  
   usando com funções, 126  
   XMLPARSE(), 641  
 expressões CASE  
   consultas avançadas, 221–224  
   usando com GROUPING(), 237–238  
   *versus* várias consultas, 611  
 expressões CASE pesquisadas, 222–224  
 expressões CASE simples, 221–222  
 expressões regulares  
   funções, 141–144  
   metacaracteres, 137–140  
 EXTEND(), 477, 481  
 Extensible Markup Language (XML). *Consulte* XML (Extensible Markup Language)  
 EXTRACT(), 183–185

## F

fechando  
   cursors, 375  
   objeto ResultSet, 574–575  
   objetos JDBC, 582–584  
 ferramentas, ajuste de SQL. *Consulte* ajuste de SQL (Structured Query Language)  
 FETCH, 375  
 FILECLOSE(), 511–512, 516–517  
 FILECLOSEALL(), 511–512, 516–517  
 FILEEXISTS(), 511–512, 517–518  
 FILEGETNAME(), 511–512, 517–518  
 FILEISOPEN(), 511–512, 517–519  
 FILEOPEN(), 511–512, 519

filtrando linhas  
   com HAVING, 153–154  
   com WHERE, 608–609  
   WHERE *versus* HAVING, 612–613  
 finalizando transações, 291  
 FIRST, 244, 261  
 FIRST(), 477, 482  
 first\_name, 42  
 FIRST\_VALUE(), 257–258  
 flashback, arquivos, 362–365  
 flashbacks, consulta, 298–301  
 FLOOR(), 127–129  
 florestas  
   XMLAGG(), 637–640  
   XMLFOREST(), 636–637  
 formatação de século, 161  
 formatando resultados de consulta hierárquica, 227–228  
 formato de data/horário padrão, 167–168  
 formato FOR[MAT], 98–99  
 formato RR, 168–170  
 formato YY, 168  
 formatos, data/horário  
   conversão, 159–167  
   data, 158  
   interpretação de ano de dois dígitos, 168–170  
   padrão, 167–168  
 fragmentos, XML, 640  
 FREETEMPORARY(), 512, 520  
 FROM, 199–200  
 FROM\_TZ(), 183, 185  
 funções, 117–155  
   agregadas, 145–148  
   agrupando linhas, 148–155  
   analíticas. *Consulte* funções analíticas  
   armazenando valores retornados com variáveis de bind, 618  
   caractere, 118–126  
   chamando em pacotes, 394–396  
   com CAST(), 471–472  
   conversão, 130–137  
   conversão de data/horário, 159–167  
   CURRENTV(), 267–268  
   data/horário, 170–174  
   de uma linha, 118  
   DECODE(), 219–221  
   DEREF(), 419–420  
   expressão regular, 137–144  
   get\_product(), 426–427

get\_product\_ref(), 428  
 get\_products(), 423–424  
 GROUP\_ID(), 243–244  
 GROUPING(), 237–239  
 GROUPING\_ID(), 240–242  
 IS OF(), 436–440  
 numéricas, 126–130  
 PL/SQL, 391–393  
 REF(), 418–420  
 resumo, 155  
 suporte para tabela aninhada  
   ANSI, 497–500  
   SYS\_TYPEID(), 444  
 TABLE(), 464–466  
 timestamp, 182–187  
 TRANSLATE(), 218–219  
 TREAT(), 440–444  
 usando objetos em PL/SQL, 422–423  
 XMLAGG(), 637–640  
 XMLATTRIBUTES(), 636  
 XMLCOLATTVAL(), 640  
 XMLCOMMENT(), 642  
 XMLCONCAT(), 641  
 XMLELEMENT(), 633–636  
 XMLFOREST(), 636–637  
 XMLPARSE(), 641  
 XMLPI(), 642  
 XMLQUERY(), 646–650  
 XMLSEQUENCE(), 643  
 XMLSERIALIZE(), 644  
 funções agregadas  
   agrupando linhas, 148–155  
   com ROLLUP, 235  
   definidas, 118  
   RETURNING, 283–284  
   tipos de, 145–148  
   várias em PIVOT, 275–276  
 funções analíticas  
   definidas, 244–245  
   função de classificação hipotética e distribuição, 263  
   funções de classificação, 245–252  
   funções de janela, 253–258  
   funções de percentil inversas, 252–253  
   funções de regressão linear, 261–263  
   funções de relatório, 258–260  
   funções FIRST e LAST, 261  
   LAG() e LEAD(), 260–261  
 funções de caractere  
   definidas, 118  
   tipos de, 118–126

funções de classificação, 244–252  
 funções de conversão, 118, 130–137  
 funções de expressão regular, 118, 137–144  
 funções de janela, 244, 253–258  
 funções de percentil inversas, 244, 252–253  
 funções de regressão linear, 244, 261–263  
 funções de relatório, 244, 258–260  
 funções de uma linha  
   caractere, 118–126  
   conversão, 130–137  
   expressão regular, 137–144  
   numéricas, 126–130  
   tipos de, 118  
 funções hipotéticas, 244, 263  
 funções numéricas, 118, 126–130  
 fusos horários, 163, 174–178, 180–182  
 fusos horários locais, 181–182, 187  
 fusos horários normalizados, 181–182

## G

geração de código de máquina nativo, 404–405  
 GET nome\_arquivo, 95  
 get\_product(), 422, 426–427  
 get\_product\_ref(), 423, 428  
 get\_products(), 422–424  
 GET\_STORAGE\_LIMIT(), 512, 520  
 GETCHUNKSIZE(), 512, 520  
 getConnection(), 565–566  
 GETLENGTH(), 512, 521  
 girando. *Consulte* PIVOT  
 GMT (horário médio de Greenwich), 174–175  
 GRANT, 32  
 GROUP BY, 148–155  
 GROUP BY, cláusulas estendidas  
   CUBE, 235–236  
   definido, 231–232  
   GROUP\_ID(), 243–244  
   GROUPING SETS, 239–240  
   GROUPING(), 237–239  
   GROUPING\_ID(), 240–242  
   ROLLUP, 233–235  
   uso de coluna GROUP BY, 242  
   uso em várias colunas, 242

GROUP\_ID(), 243–244  
 GROUPING SETS  
   definido, 239–240  
   usando com funções analíticas, 249–251  
   *versus* CUBE, 616  
 GROUPING(), 237–239  
 GROUPING\_ID(), 240–242

## H

HAVING  
   agrupando linhas, 153–155  
   subconsultas em, 198–199  
   *versus* WHERE, 612–613  
 herança, tipo, 430–433  
 hierarquias de tipos de objeto, 430–433  
 horário médio de Greenwich (GMT), 174–175

## I

ID  
   GROUP\_ID(), 243–244  
   GROUPING\_ID(), 240–242  
   SYS\_TYPEID(), 444  
 identificadores, linha, 58–59  
 identificadores de objeto (OIDs), 418–420  
 IF, 370–371  
 IGNORE NAV, 271  
 igual (=). *Consulte* = (igual)  
 importando pacotes JDBC, 564–565  
 imprimindo variáveis de bind, 618  
 IN  
   comparando valores de objeto, 420–422  
   definido, 67  
   limitações da join externa, 82  
   parâmetros de procedure, 386  
   subconsultas de várias linhas, 201–202  
   suporte para tabela aninhada  
     ANSI, 493–494  
     usando, 69–70  
     *versus* EXISTS, 614–615  
     *versus* EXISTS em subconsultas correlacionadas, 206  
 IN OUT, 386  
 índices  
   adicionando em tabelas, 612  
   exceções DUP\_VAL\_ON\_IN-DEX, 384

  resumo, 365  
   visão geral, 348–353  
 índices baseados em função, 350–351  
 índices de árvore B, 349–350  
 índices de bitmap, 349, 352–353  
 INF (infinito), 52  
 inicialização do SQL\*Plus na linha de comando, 34  
 iniciando transações, 291  
 INITCAP(), 119–121  
 INSERT  
   adicionando linhas, 48–49, 280–282  
   definido, 31  
   usando JDBC, 575–576  
   usando visão, 356–357  
 insert\_product(), 422, 425  
 INSTR(), 119–121, 512  
   métodos de LOB, 521–522  
   pesquisando dados em CLOB, 538–539  
 instruções, processando, 642  
 instruções Data Control Language (DCL), 32  
 instruções Data Definition Language (DDL)  
   definidas, 31, 39–48  
   executando SQL usando Java, 581  
 instruções DCL (Data Control Language), 32  
 instruções DDL (Data Definition Language)  
   definidas, 31, 39–48  
   executando SQL usando Java, 581  
 instruções de consulta, SQL, 31  
 instruções DML (Data Manipulation Language)  
   alterando conteúdo de tabela. *Consulte* conteúdo de tabela, alterando  
   definidas, 31  
   executando com visões, 356–357  
   triggers, 397–402  
 instruções SQL geradas automaticamente, 114  
 instruções SQL idênticas, 616–617  
 instruções SQL não idênticas, 616  
 instruções TC (Transaction Control), 32  
 instruções Transaction Control (TC), 32

instruções try/catch, 581–582  
 INTEGER, 41  
 integridade, banco de dados, 284–286  
 interfaces, 601  
 interpretação de ano de dois dígitos, 168–170  
 INTERSECT, 212, 216–217  
 INTERVAL DAY TO SECOND, 190–192  
 INTERVAL YEAR TO MONTH, 187–190  
 intervalo, 51  
 intervalo inclusivo, 70  
 intervalos, tempo, 187–193  
 introdução ao Oracle, 29–54  
   bancos de dados relacionais, 30–31  
   BINARY\_FLOAT e BINARY\_DOUBLE tipos, 51–53  
   esquema da loja, 38–39  
   instruções DDL usadas para criar o esquema da loja, 39–48  
   linhas, adicionando/modificando/removendo, 48–51  
   PL/SQL, 53–54  
   resumo, 54  
   saindo do SQL\*Plus, 53  
   SQL, 31–32  
   SQL Developer, 35–38  
   SQL\*Plus, 32–35  
 INVALID\_CURSOR, 382  
 INVALID\_NUMBER, 382, 384–385  
 invocação generalizada, 451–453  
 IS A SET, 498–499  
 IS ANY, 266–267  
 IS EMPTY, 499  
 IS INFINITE, 67  
 IS NAN, 67  
 IS NULL, 67  
 IS OF(), 436–440  
 IS PRESENT, 269–270  
 ISOPEN(), 512, 522–523  
 ISTEMPORARY(), 512, 522–524  
 Itens privados, 393

---

**J**


---

Java  
   compatibilidade com SQL Developer, 36  
   executando SQL usando. *Consulte* SQL (Structured Query Language), executando usando Java  
   tipos, 572–573

JDBC. *Consulte também* SQL (Structured Query Language), executando usando Java  
   drivers Oracle, 563–564  
   extensões Oracle, 595–596  
   fechando objetos, 582–584  
   importando pacotes, 564–565  
   objeto Statement, 570–571  
   registrando drivers, 565  
 joins  
   autojoins, 83–84  
   condições e tipos, 78  
   instruções SELECT usando duas tabelas, 73–75  
   instruções SELECT usando várias tabelas, 77–78  
   joins de tabela *versus* várias consultas, 609–610  
   joins externas, 79–83  
   MERGE, 289  
   não-equi-joins, 78–79  
   planos de execução envolvendo tabela, 624–625  
   referências de coluna qualificada, 610  
   usando sintaxe SQL/92, 84–89  
 joins cruzadas, 89  
 joins de tabela. *Consulte também* joins  
   definidas, 74  
   planos de execução envolvendo, 624–625  
   *versus* várias consultas, 609–610  
 joins externas  
   definidas, 78, 79–83  
   sintaxe SQL/92, 87–88  
 joins externas à direita, 80–82, 87–88  
 joins externas à esquerda, 80–81, 87  
 joins externas completas, 88  
 joins internas, 78, 84–88  
 JUS[TIFY], 98–99

---

**K**


---

KEEP NAV, 271

---

**L**


---

L[IST], 93–94  
 LAG(), 244, 260–261  
 large objects (LOBs). *Consulte* LOBs (large objects)

LAST, 244, 261  
 LAST(), 477, 482–483  
 LAST\_DAY(), 171–172  
 last\_name, 43  
 LAST\_VALUE(), 257–258  
 LEAD(), 244, 260–261  
 leituras fantasmas, 295–296  
 leituras que não podem ser repetidas, 295–296  
 leituras sujas, 295–296  
 LENGTH(), 119, 122  
 LIKE  
   comparando valores de objeto, 421  
   definido, 67–69  
 LIMIT(), 477  
 limites inferiores, loops FOR, 372–373  
 limites superiores, 372–373  
 linguagens de programação  
   PL/SQL. *Consulte* PL/SQL (Procedural Language/SQL)  
   SQL\*Plus. *Consulte* SQL\*Plus  
   XML. *Consulte* XML (Extensible Markup Language)  
 linhas  
   adicionando usando JDBC, 575–576  
   adicionando/modificando/removendo, 48–51  
   agrupando, 148–155  
   alterando conteúdo de tabela. *Consulte* conteúdo de tabela, alterando  
   armazenando com variável de bind, 618–619  
   classificando, 72–74  
   definidas, 30  
   excluindo usando JDBC, 576  
   exibindo distintas, 65  
   filtrando com WHERE, 608–609  
   identificadores, 58–59  
   modificando usando JDBC, 576  
   números, 58–59  
   obtendo a primeiro e a última, 257–258  
   recuperando com cursores. *Consulte* cursores  
   recuperando específicas usando WHERE, 57–58  
   recuperando usando JDBC, 571–575  
   removendo duplicadas com GROUP\_ID(), 243–244



subconsultas de uma linha, 196–201  
 subconsultas de várias linhas, 201–203  
 usando pivô. *Consulte* PIVOT  
 usando TABLE() para tratar coleção como série de, 464–466  
 linhas de saída, 103–104  
 linhas duplicadas, 243–244  
 Linux, variável de ambiente ORACLE\_HOME, 561–562  
 listando variáveis de bind, 618  
 LN(), 127  
 LOADBLOBFROMFILE(), 512, 525–526  
 LOADCLOBFROMFILE(), 512, 525–526  
 LOADFROMFILE(), 512, 523–524  
 LOBMAXSIZE, 512  
 LOBs (large objects)  
   anexando dados em CLOB, 534–535  
   apagando dados de CLOB, 538  
   APPEND(), 513  
   aprimoramentos do Oracle Database 10g, 551–553  
   aprimoramentos do Oracle Database 11g, 553–557  
   arquivos de exemplo, 504–505  
   CLOB temporários, 537–538  
   CLOSE(), 513  
   comparando dados em CLOBs, 535–536  
   COMPARE(), 514–515  
   contendo tabelas, 506  
   copiando dados de arquivo em CLOBs e BLOBs, 540–543  
   copiando dados de BFILE em CLOBs e BLOBs, 545–548  
   copiando dados de CLOBs, 536–537  
   copiando dados de CLOBs e BLOBs em arquivos, 543–545  
   COPY(), 515  
   CREATETEMPORARY(), 516  
   em PL/SQL, 510–512  
   em SQL, 506–510  
   ERASE(), 516–517  
   FILECLOSE() e FILECLOSE-ALL(), 516–517  
   FILEEXISTS(), 517–518  
   FILEGETNAME(), 517–518  
   FILEISOPEN(), 517–519  
   FILEOPEN(), 519  
   FREETEMPORARY(), 520

GET\_STORAGE\_LIMIT(), 520  
 GETCHUNKSIZE(), 520  
 GETLENGTH(), 521  
 gravando em CLOB, 534  
 INSTR(), 521–522  
 introdução, 504  
 ISOPEN(), 522–523  
 ISTEMPORARY(), 522–524  
 lendo dados de CLOBs e BLOBs, 532–534  
 LOADBLOBFROMFILE(), 525–526  
 LOADCLOBFROMFILE(), 525–526  
 LOADFROMFILE(), 523–524  
 OPEN(), 526–527  
 pesquisando dados de CLOB, 538–539  
 READ(), 527–528  
 recuperando localizador, 531–532  
 resumo, 557–558  
 SUBSTR(), 528–529  
 tipos, 504–506  
 tipos LONG e LONG RAW, 549–550  
 TRIM(), 529–530  
 WRITE(), 530  
 WRITEAPPEND(), 531  
 XMLSERIALIZE(), 644  
 LOBs de caractere (CLOBs). *Consulte* CLOBs (LOBs de caractere)  
 LOBs temporários  
   CREATETEMPORARY(), 516  
   FREETEMPORARY(), 520  
   usando CLOBs temporários, 537–538  
 localizadores, LOB, 506, 531–532  
 localizando triggers, 399–400  
 LOCALTIMESTAMP, 183–184  
 LOG(), 127  
 lógica condicional, 370–371  
 LOGIN\_DENIED, 382  
 loops  
   acesso à célula usando FOR, 268–269  
   cursos e, 377–378  
   PL/SQL, 371–373  
 loops FOR  
   acesso à célula usando, 268–269  
   cursos e, 377–378  
   definidos, 371–373  
 loops simples, 371–372

loops WHILE, 371, 372  
 LOWER(), 119, 122–123  
 LPAD(), 119, 123  
 LTRIM(), 119, 123

## M

*Macbeth* (Shakespeare), 504–505  
 maior ou igual (>=). *Consulte* >= (maior ou igual)  
 maior que (>). *Consulte* > (maior que)  
 mais (+). *Consulte* + (mais)  
 MAX(), 145, 147  
 média centralizada, 256–257  
 média móvel, 255–256  
 MEMBER OF, 497–498  
 menor ou igual (<=). *Consulte* <= (menor ou igual)  
 menor que (<). *Consulte* < (menor que)  
 menos (–), 59  
 MERGE, 287–290  
 meses  
   formato padrão, 158  
   funções de data/horário, 170–172  
   INTERVAL YEAR TO MONTH, 187–190  
   parâmetros de formatação, 161–162  
 metacaracteres  
   expressões regulares, 137–140  
 metacaracteres influenciados pela linguagem Perl, 140  
 método de mapeamento, 468–470  
 métodos  
   APPEND(), 513  
   CLOSE(), 513  
   COMPARE(), 514–515  
   COPY(), 515  
   COUNT(), 476–479  
   CREATETEMPORARY(), 516  
   DELETE(), 479–480  
   ERASE(), 516–517  
   EXISTS(), 480–481  
   EXTEND(), 481  
   FILECLOSE() e FILECLOSE-ALL(), 516–517  
   FILEEXISTS(), 517–518  
   FILEGETNAME(), 517–518  
   FILEISOPEN(), 517–519  
   FILEOPEN(), 519  
   FIRST(), 482

FREETEMPORARY(), 520  
 GET\_STORAGE\_LIMIT(), 520  
 GETCHUNKSIZE(), 520  
 getConnection(), 565–566  
 GETLENGTH(), 521  
 INSTR(), 521–522  
 invocação generalizada, 451–453  
 ISOPEN(), 522–523  
 ISTEMPORARY(), 522–524  
 LAST(), 482–483  
 LOADBLOBFROMFILE(), 525–526  
 LOADCLOBFROMFILE(), 525–526  
 LOADFROMFILE(), 523–524  
 NEXT(), 483–484  
 OPEN(), 526–527  
 PRIOR(), 484–485  
 READ(), 527–528  
 SUBSTR(), 528–529  
 TRIM(), 485–486, 529–530  
 WRITE(), 530  
 WRITEAPPEND(), 531  
 métodos get, 568–571, 573–574  
 métodos set, 568–571  
 MIN(), 145, 147  
 MINUS, 212, 216  
 MOD(), 127, 129  
 MODEL  
   acesso à célula usando loop FOR, 268–269  
   acesso a intervalo de célula usando BETWEEN e AND, 266  
   acesso a todas as células usando ANY e IS ANY, 266–267  
   atualizando células existentes, 271–272  
   definido, 264–265  
   notação posicional e simbólica, 265–266  
   tratando de valores nulos ausentes, 269–271  
   valores de dimensão atuais usando CURRENTV(), 267–268  
 modificando elementos, 466–468  
 modo de autocommit, 579–581  
 modos, 386  
 MON, 158  
 MONTHS\_BETWEEN(), 171–172  
 múltiplas consultas  
   *versus* expressões CASE, 611  
   *versus* joins de tabela, 609–610

múltiplos operadores de conjunto, 216–217  
 MULTISSET, 495–497

## N

não é número (NAN), 52  
 não-equi joins, 78–79  
 NCLOBs (National Character Set LOBs)  
   compactação, 557  
   conversão implícita, 551–552  
   criptografia, 554–556  
   definidos, 504–506  
   LOADCLOBFROMFILE(), 525–526  
 NEXT(), 477, 483–484  
 NEXT\_DAY(), 171–173  
 nextval, 345–347, 404  
 nível de isolamento de transação, 295–297  
 NLS\_DATE\_FORMAT, 167–168  
 nó país, 225  
 NO\_DATA\_FOUND, 382  
 NOCACHE, 347  
 nome, 43–45  
 nomes, fuso horário, 177  
 nós  
   definidos, 225  
   eliminando, 229–230  
   que não é raiz, 228  
 nós filhos, 225  
 nós folha, 225  
 nós raiz  
   definidos, 225  
   iniciando em nó que não é, 228  
 NOT  
   definido, 68, 71  
   *versus* NOT EXISTS em subconsultas correlacionadas, 206  
 NOT EXISTS, 204–207  
 NOT IN, 493–494  
 NOT NULL, 334–336  
 NOT\_LOGGED\_ON, 382  
 notação  
   chamando procedures, 388–389  
   posicional e simbólica, 265–266  
 notação mista, 389  
 notação nomeada, 389  
 notação posicional  
   chamando procedures, 388–389  
   consultas avançadas, 265–266

notação simbólica, 265–266  
 NTILE(), 246, 251–252  
 NULLS FIRST, 247  
 NULLS LAST, 247  
 NUMBER, 51  
 NUMBER(), 41  
 número (#), 135  
 números  
   linhas, 58–59  
   tratando de, 577–578  
 números de alteração de sistema (SCNs), 300–301  
 NUMTODSINTERVAL(), 192–193  
 NUMTOYMINTERVAL(), 193  
 NVL(), 64, 119, 124  
 NVL2(), 119, 124

## O

objeto Statement, 570–571  
 objetos de banco de dados, 407–454  
   coluna, 411–414  
   comparando valores, 420–422  
   construtores definidos pelo usuário, 446–450  
   criando tipos, 409–410  
   delete\_product(), 428–429  
   display\_product(), 424–425  
   em PL/SQL, 422–423  
   gerando XML para, 635  
   get\_product(), 426–427  
   get\_product\_ref(), 428  
   get\_products(), 423–424  
   herança de tipo, 430–433  
   insert\_product(), 425  
   introdução, 408  
   invocação generalizada, 451–453  
   IS OF(), 436–440  
   JDBC Statement, 570–571  
   large. *Consulte* LOBs (large objects)  
   OIDs e referências, 418–420  
   OracleDataSource, 568–571  
   product\_lifecycle(), 429–430  
   product\_lifecycle2(), 430–431  
   ResultSet, 571–575  
   resumo, 452–454  
   sobrescrevendo métodos, 450–451  
   subtipo no lugar do supertipo, 433–436



SYS\_TYPEID(), 444  
 tabelas, 414–417  
 tipos NOT INSTANTIABLE, 444–446  
 TREAT(), 440–444  
 update\_product(), 427  
 update\_product\_price(), 426  
 usando DESCRIBE para obter informações de tipo, 410–411  
 wrappers, 578–580  
 objetos de diretório  
   usando BFILES, 509  
   wallets, 553–554  
 OIDs (identificadores de objeto), 418–420  
 ON, 84–85  
 ONLY, 438–439  
 OPEN(), 512, 526–527  
 OPEN-FOR, 378–379  
 operações fechadas, 51  
 operador de concatenação (||), 62  
 operadores  
   aritméticos, 59–62  
   combinando operadores de conjunto, 216–218  
   comparação de valor, 65–67  
   comparando valores de objeto, 420–422  
   joins. *Consulte* joins lógicos, 71  
   operadores de conjunto, 212–216  
   precedência, 72  
   SQL, 67–71  
   subconsultas correlacionadas, 204  
   subconsultas de uma linha, 197–198  
   subconsultas de várias linhas, 201  
   suporte para tabela aninhada ANSI, 492–499  
 operadores de conjunto, 212–218  
 operadores lógicos, 71  
 operandos, 59  
 OR, 71, 83, 93–94  
 Oracle  
   conectando no banco de dados com origem de dados, 567–571  
   interpretação de ano de dois dígitos, 168–170  
   JDBC. *Consulte* JDBC

tipos, 572–573, 663–666  
 XML e. *Consulte* XML (Extensible Markup Language)  
 Oracle Database 10g  
   aprimoramentos em LOB, 551–553  
   aprimoramentos nas coleções, 489–490  
   arrays associativos, 490  
   aumentando elementos em varray, 491  
   dimensionamento de tipo de elemento, 491  
   otimização baseada em custo, 619  
   suporte para tabela aninhada ANSI, 492–500  
   tablespace de tabela de armazenamento de uma tabela aninhada, 491–492  
   varrays em tabelas temporárias, 491  
 Oracle Database 11g  
   aprimoramentos em LOB, 553–557  
   novos recursos PL/SQL, 402–405  
   recurso de invocação generalizada, 451–453  
 Oracle Enterprise Manager Diagnostics Pack, 628  
 ORDER BY, 72–74, 200–201  
 origens de dados, 567–571  
 otimização baseada em custo, 619–626  
 otimização baseada em regra, 619  
 otimizadores  
   comparação de custos de consulta, 619–626  
   dicas, 626–628  
 OUT, 386

## P

pacote oracle.jdbc, 599–603  
 pacote oracle.sql, 596–599  
 pacotes  
   definidos, 393–396  
   importando JDBC, 564–565  
 parâmetros  
   construtores definidos pelo usuário, 446–450  
   convertendo data/horário usando TO\_CHAR(), 161–163  
   procedures, 386–387  
   TO\_CHAR(), 133–134  
 parâmetros de formatação de hora, 162  
 parâmetros de formatação de minuto, 162  
 parâmetros de formatação de semana, 162  
 parâmetros de formatação de trimestre, 161  
 parâmetros de formatação era, 163  
 parênteses (), 62, 196  
 parsing  
   definido, 589–590  
   XMLPARSE(), 641  
 PARTITION BY, 248–249  
 PERCENT\_RANK(), 246, 251  
 pesquisando, 567  
 pesquisando dados de CLOB, 538–539  
 phone, 43  
 PIVOT  
   definido, 272  
   em várias colunas, 274–275  
   exemplo, 272–274  
   várias funções agregadas em, 275–276  
 PL/SQL (Procedural Language/SQL), 367–405  
   armazenando valores retornados de função com variável de bind, 618  
   construtores definidos pelo usuário, 446–450  
   COUNT(), 476–479  
   cursores, 373–374  
   cursores e loops FOR, 377–378  
   cursores restritos, 380–381  
   DELETE(), 479–480  
   delete\_product(), 428–429  
   display\_product(), 424–425  
   em coleções, 472–476  
   estrutura de bloco, 368–370  
   exceções, 381–383  
   exceções DUP\_VAL\_ON\_INDEX, 384  
   exceções INVALID\_NUMBER, 384–385  
   exceções OTHERS, 385  
   exceções ZERO\_DIVIDE, 383–384  
   exemplo product\_cursor.sql, 376–377  
   EXISTS(), 480–481

EXTEND(), 481  
 FIRST(), 482  
 funções, 391–393  
 get\_product(), 426–427  
 get\_product\_ref(), 428  
 get\_products(), 423–424  
 gravando dados XML em arquivo, 644–646  
 insert\_product(), 425  
 instrução OPEN-FOR, 378–379  
 introdução, 53–54  
 LAST(), 482–483  
 LOBs em, 510–512. *Consulte também* LOBs (large objects)  
 lógica condicional, 370–371  
 loops, 371–373  
 NEXT(), 483–484  
 novos recursos do Oracle Database 11g, 402–405  
 objetos de banco de dados em, 422–423  
 pacotes, 393–396  
 passos de cursor, 374–376  
 PRIOR(), 484–485  
 procedures, 386–390  
 product\_lifecycle(), 429–430  
 product\_lifecycle2(), 430–431  
 resumo, 404–405  
 subtipo no lugar do supertipo, 434–435  
 tipos de dados Oracle, 666  
 TREAT() em, 442–444  
 triggers, 397–402  
 TRIM(), 485–486  
 update\_product(), 427  
 update\_product\_price(), 426  
 variáveis e tipos, 370  
 planos de execução, 589–590, 619–626  
 ponteiros, 510  
 ponto de interrogação (?), 592  
 ponto-e-vírgula (;), 34, 369  
 porcentagem (%), 68–69  
 POWER(), 127, 129  
 POWERMULTISET(), 500  
 POWERMULTISET\_BY\_CARDINALITY(), 500  
 precisão, 41  
 preenchendo chave primárias, 347  
 preenchendo coleções, 462–463

preenchendo LOBs  
   BFILES, 510  
   CLOBs e BLOBs, 507  
 preenchendo objeto ResultSet, 571–572  
 PRESENTNNV(), 270–271  
 PRESENTV(), 270  
 price, 44–45  
 PRIOR(), 477, 484–485  
 privilégios  
   auditoria, 323  
   concedendo objeto a atribuições, 320–321  
   consulta flashback, 298  
   objeto, 309–315  
   resumo, 325–326  
   revogando objeto, 316  
   sinônimos, 315–316  
   usuário, 306–309  
   verificando atribuição, 319–320  
   visões, 355  
 privilégios de objeto  
   concedendo a atribuições, 320–321  
   definidos, 309–315  
   revogando, 316  
   sinônimos, 315–316  
 privilégios de sistema. *Consulte* privilégios  
 Procedural Language/SQL (PL/SQL). *Consulte* PL/SQL (Procedural Language/SQL)  
 procedures  
   anexando dados em CLOB, 534–535  
   apagando dados de CLOB, 538  
   chamando em pacotes, 394–396  
   CLOBs temporários, 537–538  
   comparando dados em CLOBs, 535–536  
   copiando dados de CLOBs, 536–537  
   copiando dados de LOB. *Consulte* copiando dados de LOB  
   definidas, 386–390  
   delete\_product(), 428–429  
   display\_product(), 424–425  
   gravando em CLOB, 534  
   insert\_product(), 425  
   lendo dados de CLOBs e BLOBs, 532–534  
   pesquisando dados de CLOB, 538–539  
   product\_lifecycle(), 429–430

product\_lifecycle2(), 430–431  
 recuperando localizador de LOBs, 531–532  
 triggers, 397–402  
 update\_product(), 427  
 update\_product\_price(), 426  
 usando objetos em PL/SQL, 422–423  
 processando data/horário. *Consulte* data/horário  
 processando instruções, 642  
 product\_id, 44–46  
 product\_lifecycle(), 429–430  
 product\_lifecycle2(), 430–431  
 product\_type\_id, 43–45  
 produtos cartesianos, 76–77  
 PROGRAM\_ERROR, 382  
 propriedades de transação ACID (atômicas, consistentes, isoladas e duráveis), 293–294  
 pseudocoluna LEVEL, 226–227  
 pseudocolunas  
   definidas, 58–59  
   LEVEL, 226–227  
   seqüências, 345–347

---

## Q

---

quantity, 46

---

## R

---

R[UN], 93  
 RATIO\_TO\_REPORT(), 259–260  
 READ COMMITTED, 295–296  
 READ ONLY, 334, 358  
 READ UNCOMMITTED, 295–296  
 READ(), 512, 527–528, 532–534  
 recuperação de informações  
   coleções, 459–461  
   comentários de coluna, 341–342  
   comentários de tabela, 341  
   constraints, 338–340  
   de CLOBs, 507–508  
   esquema XML, 653–658  
   funções PL/SQL, 393  
   índices, 351–352  
   pacotes, 396  
   procedures, 389–390  
   seqüências, 344–345

tabelas. *Consulte* recuperação de informações de tabela de banco de dados  
triggers, 400–402  
usando DESCRIBE para obter informações do tipo de objeto, 410–411  
visões, 358–360

recuperação de informações de tabela do banco de dados, 55–89  
apelidos de coluna, 62–63  
apelidos de tabela, 75–76  
aritmética, 59–62  
autojoins, 83–84  
classificando linhas, 72–74  
comparação de valor, 65–67  
concatenação, 63  
condições de join e tipos, 78  
exibição de linha distinta, 65  
identificadores de linha, 58–59  
instruções SELECT usando duas tabelas, 73–75  
instruções SELECT usando uma tabela, 56–58  
instruções SELECT usando várias tabelas, 77–78  
joins externas, 79–83  
joins usando sintaxe SQL/92, 84–89  
não-equijoins, 78–79  
números de linha, 58–59  
operadores SQL, 67–71  
precedência de operador, 72  
produtos cartesianos, 76–77  
resumo, 89  
tabelas e colunas em tabelas, 330–331  
todas as colunas, 57–58  
valores nulos, 63–65  
WHERE, 57–58

recuperando informações. *Consulte* recuperação de informações

REF CURSOR, 379, 618–619  
REF(), 418–420  
referências, objeto, 411, 418–420  
referências de coluna totalmente qualificada, 610  
REGEXP\_COUNT(), 142, 144  
REGEXP\_INSTR(), 141, 143  
REGEXP\_LIKE(), 141–143  
REGEXP\_REPLACE(), 142, 144  
REGEXP\_SUBSTR(), 142, 144  
relatórios, 107–113

removendo linhas, 284  
RENAME, 31  
renomeando tabelas, 341  
REPEATABLE READ, 295–296  
REPLACE(), 119, 124–125  
resultados, consulta hierárquica, 227–228  
RETURNING, 283–284  
revogando atribuições, 322  
revogando privilégios de usuários, 309  
objeto, 316  
REVOKE, 32  
rodapés, 110–111  
ROLLBACK  
definido, 32  
remoção de linha, 50–51  
transações, 290–291  
ROLLUP  
definido, 233–235  
GROUPING(), 237–238  
usando com funções analíticas, 249–251  
*Romeu e Julieta* (Shakespeare), 138  
ROUND()  
definido, 127, 129  
funções de data/horário, 171, 173  
ROW\_NUMBER(), 246, 252  
ROWID, 58–59  
ROWNUM, 58–59  
ROWTYPE\_MISMATCH, 382  
RPAD(), 119, 123  
RTRIM(), 119, 123  
RULES UPDATE, 271–272

## S

saída, combinando coluna, 63  
saindo do SQL \*Plus, 53  
salvando arquivos SQL\*Plus, 94–98  
salvando XML  
arquivo de exemplo, 651  
atualizando informações no esquema de exemplo, 658–661  
em banco de dados, 650  
esquema de exemplo, 651–653  
recuperação de informações do esquema de exemplo, 653–658  
savepoint, 292–293  
SAVEPOINT, 32

SCNs (números de alteração de sistema), 300–301  
scripts, 107–113  
segundos  
INTERVAL DAY TO SECOND, 190–192  
parâmetros de formatação, 162  
SELECT  
adicionando linhas, 49  
executando usando SQL\*Plus, 34–35  
recuperação de informações de uma tabela, 56–58  
usando duas tabelas, 73–75  
usando várias tabelas, 77–78  
SELF, 447  
SELF\_IS\_NULL, 383  
senhas, 39–40, 305–306  
separadores, 162  
seqüências, 342–348  
criando, 342–344  
eliminando, 348  
modificando, 348  
novos recursos do Oracle Database 11g, 403–405  
preenchendo chave primárias, 347  
recuperando informações, 344–345  
resumo, 365  
usando, 345–347  
XMLSEQUENCE(), 643  
sessões  
definidas, 167  
fusos horários, 175–177  
SET LINESIZE, 101  
SET PAGESIZE, 100–101  
SET(), 498  
SETS, GROUPING, 239–240  
Shakespeare, William, 138, 504–505  
SIGN(), 127, 130  
SIN(), 127  
SINH(), 127  
sinônimos, privilégios, 315–316  
sinônimos públicos, 315–316  
sistemas de gerenciamento de banco de dados, 30–31  
sobrecarregando métodos, 448  
sobrescrevendo métodos, 450–451  
somadas, relatórios em, 258–259  
somadas acumuladas, 253–255  
SOME, 66–67  
SOUNDEX(), 119, 125

- SPO[OL], 95
- SQL (Structured Query Language)
  - introdução, 31–32
  - LOBs em, 506–510
  - operadores, 67–71
  - precedência de operador, 72
  - Procedural Language. *Consulte* PL/SQL (Procedural Language/SQL)
  - subtipo no lugar do supertipo, 433–434
  - tipos de dados Oracle, 664–665
- SQL (Structured Query Language), executando usando Java, 559–606
  - adicionando linhas, 575–576
  - `BasicExample1.java`, 584–590
  - `BasicExample2.java`, 593–595
  - `BasicExample3.java`, 603–605
  - conexão de banco de dados, 565–571
  - configurando o computador, 561–563
  - controle de transação de banco de dados, 579–581
  - drivers JDBC da Oracle, 563–564
  - excluindo linhas, 576
  - extensões JDBC da Oracle, 595–596
  - fechando objetos JDBC, 582–584
  - fundamentos, 560
  - importando pacotes JDBC, 564–565
  - instruções DDL, 581
  - modificando linhas, 576
  - objeto JDBC Statement, 570–571
  - pacote `oracle.jdbc`, 599–603
  - pacote `oracle.sql`, 596–599
  - prepared statements, 589–592
  - recuperando linhas, 571–575
  - registrando drivers JDBC da Oracle, 565
  - resumo, 606
  - tratando de exceções, 581–582
  - tratando de números, 577–578
  - tratando de valores nulos de banco de dados, 578–580
- SQL Access Advisor, 629
- SQL Developer, 35–38
- SQL Tuning Advisor, 628–629
- SQL\*Plus, 91–115
  - arquivos, salvando/recuperando/executando, 94–98
  - desconectando/saindo, 114
  - editando instruções SQL, 93–94
  - esquema da loja, 38–39
  - estrutura de tabela, 92–93
  - formatação de coluna, 98–100
  - instruções DDL usadas para criar o esquema da loja, 39–48
  - instruções geradas automaticamente, 114
  - introdução, 32–35
  - limpando formato de coluna, 101
  - linhas, adicionando/modificando/removendo, 48–51
  - obtendo ajuda, 113–114
  - recuperação de informações. *Consulte* recuperação de informações de tabela de banco de dados
  - relatórios simples, 107–113
  - resumo, 115
  - saindo, 53
  - tamanho da linha, configurando, 101
  - tamanho da página, configurando, 100–101
  - tipos BINARY\_FLOAT e BINARY\_DOUBLE, 51–53
  - variáveis, 102–107
- SQL/92
  - definida, 75
  - expressões CASE, 221
  - joins usando, 84–89
- SQLPrepared Statements, 589–592
- SQRT(), 127, 130
- START WITH, 226, 228–229
- STDDEV(), 145, 148
- STORAGE\_ERROR, 383
- strings
  - convertendo data/horário usando TO\_CHAR(), 160–164
  - convertendo data/horário usando TO\_DATE(), 164–167
  - convertendo para TIMESTAMP WITH LOCAL TIME ZONE, 187
  - XMLSERIALIZE(), 644
- Structured Query Language (SQL). *Consulte* SQL (Structured Query Language)
- subconsultas, 195–209
  - aninhadas, 207–208
  - correlacionadas, 203–207
  - de uma linha, 196–201
  - instruções UPDATE e DELETE contendo, 208–209
  - resumo, 209
  - START WITH, 228–229
  - tipos de, 196
  - UPDATE, 283
  - várias colunas, 203
  - várias linhas, 201–203
- subconsultas escalares, 198
- sublinhado (\_), 68–69
- SUBMULTISET, 470, 494–495
- subprogramas armazenados, 391
- SUBSCRIPT\_BEYOND\_COUNT, 383
- SUBSCRIPT\_OUTSIDE\_LIMIT, 383
- SUBSTR(), 119, 125–126, 512, 528–529
- subtipos
  - definidos, 432
  - invocação generalizada, 451–453
  - IS OF(), 436–440
  - sobrescrevendo métodos, 450–451
  - usando no lugar do supertipo, 433–436
- SUM(), 145, 148
- supertipos
  - definidos, 432
  - invocação generalizada, 451–453
  - sobrescrevendo métodos, 450–451
  - subtipo, usando no lugar de, 433–436
  - tipos de objeto NOT INSTANTIABLE, 444–446
- SYS\_EXTRACT\_UTC(), 183, 185–186
- SYS\_INVALID\_ROWID, 383
- SYS\_TYPEID(), 436, 444
- SYSDATE, 34–35, 171, 173
- SYSTIMESTAMP, 183–184

---

**T**

- tabela customers, 42–43
- tabela de planos, 620–621
- tabela dual, 60

- tabela `employees`, 46–47
- tabela `product_types`, 43–44
- tabela `products`, 43–44
- tabela `purchases`, 44–46
- tabela `salary_grades`, 47–48
- tabelas, 328–342
  - adicionando comentários, 341–342
  - adicionando índices em, 612
  - alterando, 330–334
  - aninhadas. *Consulte* tabelas aninhadas
  - `BINARY_FLOAT` e `BINARY_DOUBLE`, 52–53
  - comparando valores de objeto, 420–422
  - constraints, 334–340
  - contendo LOBs, 506
  - criando, 328–330
  - definindo coluna com tipo de coleção, 458
  - eliminando, 342
  - estrutura do SQL\*Plus, 92–93
  - examinando, 40–48
  - objetos de coluna, 411–414
  - obtendo informações sobre, 330–331
  - OIDs e referências de objeto, 418–420
  - recuperação de informações. *Consulte* recuperação de informações de tabela de banco de dados
  - renomeando, 341
  - resumo, 365
  - substituindo nomes usando variáveis, 104–105
  - tabela de plano, 620–621
  - tabelas de objeto, 414–417
  - tablespace de tabela de armazenamento, 491–492
  - truncando, 342
  - varrays em, temporárias, 491
- tabelas aninhadas
  - coleções de vários níveis, 486–489
  - comparando conteúdo com método de mapeamento, 468–470
  - convertendo com `CAST()`, 471–472
  - definidas, 456
  - manipulando, 474–476
  - modificando elementos, 467–468
  - preenchendo com elementos, 462–463
  - recuperação de informações, 460–461
  - recuperando elementos de, 464
  - suporte para ANSI, 492–500
  - tablespace de tabela de armazenamento, 491–492
  - tipos, 456–458
    - usando `TABLE()` com, 466
- tabelas de base, 353
- tabelas de plano centrais, 621
- tabelas filhas, 285–286
- tabelas pais, 285–286
- tabelas temporárias, 491
- `TABLE()`, 464–466
- tablespaces, 304, 491–492
- tamanho de página, configurando, 100–101
- `TAN()`, 127
- `TANH()`, 127
- Tempo Universal Coordenado (UTC), 174–175
- tempo. *Consulte* data/horário
- `THEN`, 370–371
- `TIMEOUT_ON_RESOURCE`, 383
- timestamp
  - arquivos de flashback, 362–365
  - funções, 182–187
  - usando, 178–182
- tipo `SIMPLE_INTEGER`, 403
- tipos
  - alterando coluna, 333
  - classes e tipos de banco de dados Oracle compatíveis, 596
  - Oracle, 663–666
  - Oracle e Java, 572–573
  - PL/SQL, 370
  - `SIMPLE_INTEGER`, 403
  - tratando de números com Java, 577–578
- tipos, coleção
  - convertendo com `CAST()`, 471–472
  - definidos, 456–458
  - definindo colunas com, 458
  - dimensionamento de tipo de elemento, 491
- tipos, LOB
  - BFILES. *Consulte* BFILES (tipos binários FILE)
  - BLOBs (binary LOBs). *Consulte* BLOBs (binary LOBs)
  - CLOBs. *Consulte* CLOBs (Character LOBs)
  - criando, 504–506
  - NCLOBs. *Consulte* NCLOBs (National Character Set LOBs)
  - tipos LONG e LONG RAW, 549–550
- tipos, objeto
  - criando, 409–410
  - herança, 430–433
  - invocação generalizada, 451–453
  - IS OF(), 436–440
  - NOT INSTANTIABLE, 444–446
  - sobrescrevendo métodos, 450–451
  - subtipo no lugar do supertipo, 433–436
  - `SYS_TYPEID()`, 444
  - usando `DESCRIBE` para obter informações, 410–411
- tipos de dados. *Consulte* tipos
- tipos LONG, 549–550
- tipos LONG RAW, 549–550
- tipos NOT INSTANTIABLE, 444–446
- tipos RAW, 549
- `TO_CHAR()`
  - combinando com `TO_DATE()`, 166–167
  - convertendo data/horário, 159–164
  - definido, 132–135
- `TO_DATE()`, 60, 159–160, 164–167
- `TO_NUMBER()`, 135–136
- `TO_TIMESTAMP()`, 183, 186
- `TO_TIMESTAMP_TZ()`, 183, 186
- `TOO_MANY_ROWS`, 383
- transações, banco de dados, 290–297, 579–581
- transações atômicas, 293
- transações concorrentes, 294
- transações consistentes, 293
- transações duráveis, 293
- transações isoladas, 293
- transações `SERIALIZABLE`, 295–297
- `TRANSLATE()`, 218–219
- tratamento de exceções. *Consulte* exceções
- tratamento de números, 577–578
- tratamento de valores nulos de banco de dados, 578–580

tratamento de valores nulos e ausentes, 269–271  
 tratando de erros em tempo de execução. *Consulte* exceções  
 TREAT(), 436, 440–444  
 trechos de dados, 520  
 trigger em nível de instrução, 397  
 trigger em nível de linhas, 397  
 triggers  
   atributo :new, 552–553  
   definidos, 397–402  
 trilha, visões, auditoria, 324  
 TRIM()  
   definido, 119, 123, 477, 485–486, 512  
   métodos de LOB, 529–530  
 TRUNC()  
   definido, 127, 130  
   funções de data/horário, 171, 174  
 TRUNCATE, 31, 342  
 TTITLE, 110–111

## U

UNDEFINE, 106–107  
 unidade lógica de trabalho, 290  
 UNION  
   combinando operadores de conjunto, 216–217  
   definido, 212  
   usando, 215  
   *versus* UNION ALL, 613–614  
 UNION ALL  
   definido, 212  
   usando, 214–215  
   *versus* UNION, 613–614  
 UNIQUE, 334, 337  
 Unix, variável de ambiente ORACLE\_HOME, 561–562  
 UNPIVOT, 272, 276–277  
 UPDATE  
   definido, 31  
   instruções contendo subconsultas, 208–209  
   modificando linhas, 50, 282–283  
   usando JDBC, 576  
 update\_product(), 422, 427  
 update\_product\_price(), 422, 426  
 UPPER(), 119, 122–123  
 URL, conexão de banco de dados, 566–567  
 USING, 85–86, 289

usuários  
   atribuições concedidas a, 318–319  
   auditoria, 323–324  
   concedendo privilégios de objeto, 310  
   criando, 39–40  
   privilégios, 306–309  
   resumo, 325–326  
   visão geral, 304–306  
 UTC (Tempo Universal Coordenado), 174–175

## V

VARCHAR(), 41  
 valores  
   armazenando, retornados com variável de bind, 618  
   arrays associativos, 490  
   comparação, 65–67  
   comparação de objetos de banco de dados, 420–422  
   convertendo com GROUPING, 237–238  
   declarando variáveis para armazenar coluna, 374  
   lendo coluna de objeto ResultSet, 572–573  
   nulos. *Consulte* valores nulos obtendo, atuais, 267–268 obtendo a primeiro e a última linha usando, 257–258  
   tratando de nulos e ausentes, 269–271  
   usando padrão, 286–287  
 valores ausentes, tratando de, 269–271  
 valores duplicados, 384  
 valores especiais, 52–53  
 valores nulos  
   controlando classificação, 247  
   especificando para colunas, 281  
   NOT IN e, 70  
   recuperação de informações de tabela de banco de dados, 63–65  
   tratando de, 269–271, 578–580  
 valores padrão, 286–287, 334  
 VALUE(), 415  
 VALUE\_ERROR, 383  
 VARIANCE, 145, 148  
 várias funções agregadas, 275–276  
 variáveis  
   bind, 616–619  
   cursos, 374  
   de ambiente ORACLE\_HOME, 561–562  
   em scripts, 108–110  
   PL/SQL, 370  
   SQL\*Plus, 102–107  
   variáveis de ambiente, 561–563  
   variáveis de bind, 616–619  
   variáveis de substituição, 102–107  
   variáveis definidas  
     definidas, 102, 105–107  
     em scripts, 109  
   variáveis temporárias, 102–105, 108  
   variável de ambiente CLASSPATH, 562–563  
   variável de ambiente JAVA\_HOME, 562  
   variável de ambiente LD\_LIBRARY\_PATH, 563  
   variável de ambiente ORACLE\_HOME, 561–562  
   variável de ambiente PATH, 562  
 varrays  
   aumentando elementos em, 491  
   coleções de vários níveis, 486–489  
   convertendo com CAST(), 471–472  
   definidos, 456  
   em tabelas temporárias, 491  
   manipulando, 472–474  
   modificando elementos, 466–467  
   preenchendo com elementos, 462  
   recuperação de informações, 459–460  
   recuperando elementos de, 463  
   tipos, 456–458  
   usando TABLE() com, 465–466  
 velocidade de desempenho, 51  
 verificações em tempo de execução, 440–444  
 vetores de bit, 240  
 visões, 353–362  
   criando/usando, 354–361  
   definidos, 353–354  
   eliminando, 362  
   erros em procedures, 390

estrutura de tabela, 92–93  
 modificando, 361–362  
 recuperação de informações de  
 tabela aninhada, 461  
 recuperação de informações de  
 varray, 459–460  
 resumo, 365  
 visões complexos, 359–361  
 visões do usuário  
   tabelas aninhadas, 461  
   varrays, 459–460  
 visões em linha, 199–200  
 visões simples, 355

---

## W

wallets, 553–554  
 WHEN NOT MATCHED, 289  
 WHERE  
   agrupando linhas, 154–155  
   consultas hierárquicas, 230–  
   231  
   filtrando linhas com, 608–609  
   modificação de linha, 50

recuperando linhas específicas,  
 57–58  
 subconsultas em, 196–197  
 usando operadores SQL, 67–71  
   *versus* HAVING, 612–613  
 Windows XP, variável de ambien-  
 te ORACLE\_HOME, 561  
 WOR[D\_WRAPPED], 98–99  
 WRA[PPED], 98–99  
 WRITE(), 512, 530, 534  
 WRITEAPPEND(), 512, 531

---

## X

x, 93–94  
 XML (Extensible Markup Language), 631–661  
   arquivo de exemplo , 651  
   atualizando informações no es-  
   quema de exemplo, 658–661  
   esquema de exemplo, 651–653  
   exemplo de PL/SQL, 644–646  
   gerando a partir de dados rela-  
   cionais, 632

introdução, 632  
 recuperação de informações  
 do esquema de exemplo,  
 653–658  
 resumo, 661  
 salvando em banco de dados,  
 650  
 XMLAGG(), 637–640  
 XMLATTRIBUTES(), 636  
 XMLCOLATTVAL(), 640  
 XMLCOMMENT(), 642  
 XMLCONCAT(), 641  
 XMLELEMENT(), 633–636  
 XMLFOREST(), 636–637  
 XMLPARSE(), 641  
 XMLPI(), 642  
 XMLQUERY(), 646–650  
 XMLSEQUENCE(), 643  
 XMLSERIALIZE(), 644

---

## Y

YYYY, 158